

# Introduction



An **operating system** is software that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. An amazing aspect of operating systems is how they vary in accomplishing these tasks in a wide variety of computing environments. Operating systems are everywhere, from cars and home appliances that include "Internet of Things" devices, to smart phones, personal computers, enterprise computers, and cloud computing environments.

In order to explore the role of an operating system in a modern computing environment, it is important first to understand the organization and architecture of computer hardware. This includes the CPU, memory, and I/O devices, as well as storage. A fundamental responsibility of an operating system is to allocate these resources to programs.

Because an operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions. In this chapter, we provide a general overview of the major components of a contemporary computer system as well as the functions provided by the operating system. Additionally, we cover several topics to help set the stage for the remainder of the text: data structures used in operating systems, computing environments, and open-source and free operating systems.

## CHAPTER OBJECTIVES

- Describe the general organization of a computer system and the role of interrupts.
- Describe the components in a modern multiprocessor computer system.
- Illustrate the transition from user mode to kernel mode.
- Discuss how operating systems are used in various computing environments.
- Provide examples of free and open-source operating systems.

## 1.1 What Operating Systems Do

We begin our discussion by looking at the operating system's role in the overall computer system. A computer system can be divided roughly into four components: the *hardware*, the *operating system*, the *application programs*, and a *user* (Figure 1.1).

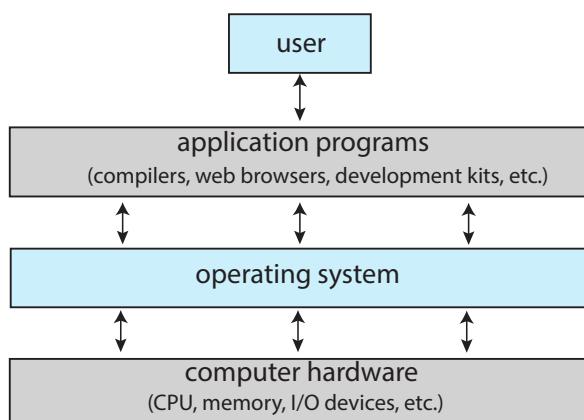
The **hardware**—the central processing unit (CPU), the memory, and the input/output (I/O) devices—provides the basic computing resources for the system. The **application programs**—such as word processors, spreadsheets, compilers, and web browsers—define the ways in which these resources are used to solve users' computing problems. The operating system controls the hardware and coordinates its use among the various application programs for the various users.

We can also view a computer system as consisting of hardware, software, and data. The operating system provides the means for proper use of these resources in the operation of the computer system. An operating system is similar to a government. Like a government, it performs no useful function by itself. It simply provides an *environment* within which other programs can do useful work.

To understand more fully the operating system's role, we next explore operating systems from two viewpoints: that of the user and that of the system.

### 1.1.1 User View

The user's view of the computer varies according to the interface being used. Many computer users sit with a laptop or in front of a PC consisting of a monitor, keyboard, and mouse. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for **ease of use**, with some attention paid to performance and security and none paid to **resource utilization**—how various hardware and software resources are shared.



**Figure 1.1** Abstract view of the components of a computer system.

Increasingly, many users interact with mobile devices such as smartphones and tablets—devices that are replacing desktop and laptop computer systems for some users. These devices are typically connected to networks through cellular or other wireless technologies. The user interface for mobile computers generally features a **touch screen**, where the user interacts with the system by pressing and swiping fingers across the screen rather than using a physical keyboard and mouse. Many mobile devices also allow users to interact through a **voice recognition** interface, such as Apple's **Siri**.

Some computers have little or no user view. For example, **embedded computers** in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating systems and applications are designed primarily to run without user intervention.

### 1.1.2 System View

From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a **resource allocator**. A computer system has many resources that may be required to solve a problem: CPU time, memory space, storage space, I/O devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly.

A slightly different view of an operating system emphasizes the need to control the various I/O devices and user programs. An operating system is a control program. A **control program** manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

### 1.1.3 Defining Operating Systems

By now, you can probably see that the term *operating system* covers many roles and functions. That is the case, at least in part, because of the myriad designs and uses of computers. Computers are present within toasters, cars, ships, spacecraft, homes, and businesses. They are the basis for game machines, cable TV tuners, and industrial control systems.

To explain this diversity, we can turn to the history of computers. Although computers have a relatively short history, they have evolved rapidly. Computing started as an experiment to determine what could be done and quickly moved to fixed-purpose systems for military uses, such as code breaking and trajectory plotting, and governmental uses, such as census calculation. Those early computers evolved into general-purpose, multifunction mainframes, and that's when operating systems were born. In the 1960s, **Moore's Law** predicted that the number of transistors on an integrated circuit would double every 18 months, and that prediction has held true. Computers gained in functionality and shrank in size, leading to a vast number of uses and a vast number and variety of operating systems. (See Appendix A for more details on the history of operating systems.)

How, then, can we define what an operating system is? In general, we have no completely adequate definition of an operating system. Operating systems

exist because they offer a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of computer systems is to execute programs and to make solving user problems easier. Computer hardware is constructed toward this goal. Since bare hardware alone is not particularly easy to use, application programs are developed. These programs require certain common operations, such as those controlling the I/O devices. The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.

In addition, we have no universally accepted definition of what is part of the operating system. A simple viewpoint is that it includes everything a vendor ships when you order “the operating system.” The features included, however, vary greatly across systems. Some systems take up less than a megabyte of space and lack even a full-screen editor, whereas others require gigabytes of space and are based entirely on graphical windowing systems. A more common definition, and the one that we usually follow, is that the operating system is the one program running at all times on the computer—usually called the **kernel**. Along with the kernel, there are two other types of programs: **system programs**, which are associated with the operating system but are not necessarily part of the kernel, and application programs, which include all programs not associated with the operation of the system.

The matter of what constitutes an operating system became increasingly important as personal computers became more widespread and operating systems grew increasingly sophisticated. In 1998, the United States Department of Justice filed suit against Microsoft, in essence claiming that Microsoft included too much functionality in its operating systems and thus prevented application vendors from competing. (For example, a web browser was an integral part of Microsoft’s operating systems.) As a result, Microsoft was found guilty of using its operating-system monopoly to limit competition.

Today, however, if we look at operating systems for mobile devices, we see that once again the number of features constituting the operating system is increasing. Mobile operating systems often include not only a core kernel but also **middleware**—a set of software frameworks that provide additional services to application developers. For example, each of the two most prominent mobile operating systems—Apple’s iOS and Google’s Android—features

### WHY STUDY OPERATING SYSTEMS?

Although there are many practitioners of computer science, only a small percentage of them will be involved in the creation or modification of an operating system. Why, then, study operating systems and how they work? Simply because, as almost all code runs on top of an operating system, knowledge of how operating systems work is crucial to proper, efficient, effective, and secure programming. Understanding the fundamentals of operating systems, how they drive computer hardware, and what they provide to applications is not only essential to those who program them but also highly useful to those who write programs on them and use them.

a core kernel along with middleware that supports databases, multimedia, and graphics (to name only a few).

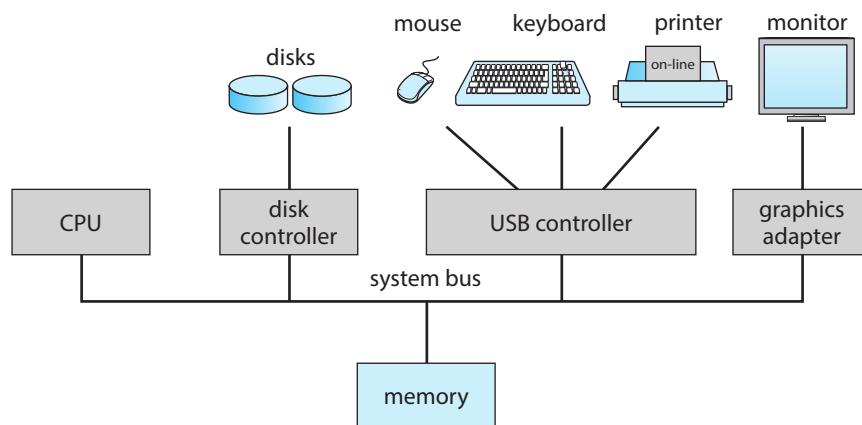
In summary, for our purposes, the operating system includes the always-running kernel, middleware frameworks that ease application development and provide features, and system programs that aid in managing the system while it is running. Most of this text is concerned with the kernel of general-purpose operating systems, but other components are discussed as needed to fully explain operating system design and operation.

## 1.2 Computer-System Organization

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common **bus** that provides access between components and shared memory (Figure 1.2). Each device controller is in charge of a specific type of device (for example, a disk drive, audio device, or graphics display). Depending on the controller, more than one device may be attached. For instance, one system USB port can connect to a USB hub, to which several devices can connect. A device controller maintains some local buffer storage and a set of special-purpose registers. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage.

Typically, operating systems have a **device driver** for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device. The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

In the following subsections, we describe some basics of how such a system operates, focusing on three key aspects of the system. We start with interrupts, which alert the CPU to events that require attention. We then discuss storage structure and I/O structure.



**Figure 1.2** A typical PC computer system.

### 1.2.1 Interrupts

Consider a typical computer operation: a program performing I/O. To start an I/O operation, the device driver loads the appropriate registers in the device controller. The device controller, in turn, examines the contents of these registers to determine what action to take (such as “read a character from the keyboard”). The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver that it has finished its operation. The device driver then gives control to other parts of the operating system, possibly returning the data or a pointer to the data if the operation was a read. For other operations, the device driver returns status information such as “write completed successfully” or “device busy”. But how does the controller inform the device driver that it has finished its operation? This is accomplished via an [interrupt](#).

#### 1.2.1.1 Overview

Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. (There may be many buses within a computer system, but the system bus is the main communications path between the major components.) Interrupts are used for many other purposes as well and are a key part of how operating systems and hardware interact.

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located. The interrupt service routine executes; on completion, the CPU resumes the interrupted computation. A timeline of this operation is shown in Figure 1.3. To run the animation associated with this figure please click [here](#).

Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. The interrupt must transfer control to the appropriate interrupt service routine. The straightforward method for managing this transfer would be to invoke a generic routine to examine the interrupt information. The routine, in turn,

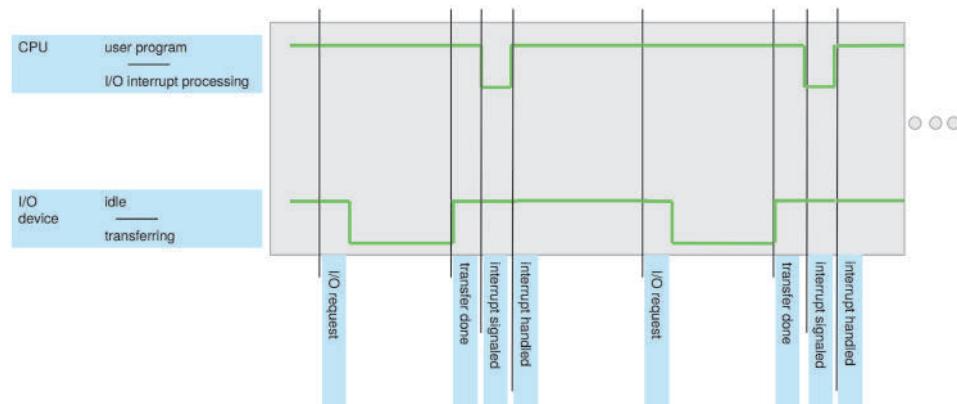


Figure 1.3 Interrupt timeline for a single program doing output.

would call the interrupt-specific handler. However, interrupts must be handled quickly, as they occur very frequently. A table of pointers to interrupt routines can be used instead to provide the necessary speed. The interrupt routine is called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is stored in low memory (the first hundred or so locations). These locations hold the addresses of the interrupt service routines for the various devices. This array, or **interrupt vector**, of addresses is then indexed by a unique number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device. Operating systems as different as Windows and UNIX dispatch interrupts in this manner.

The interrupt architecture must also save the state information of whatever was interrupted, so that it can restore this information after servicing the interrupt. If the interrupt routine needs to modify the processor state—for instance, by modifying register values—it must explicitly save the current state and then restore that state before returning. After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.

### 1.2.1.2 Implementation

The basic interrupt mechanism works as follows. The CPU hardware has a wire called the **interrupt-request line** that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt-request line, it reads the interrupt number and jumps to the **interrupt-handler routine** by using that interrupt number as an index into the interrupt vector. It then starts execution at the address associated with that index. The interrupt handler saves any state it will be changing during its operation, determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a `return_from_interrupt` instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller *raises* an interrupt by asserting a signal on the interrupt request line, the CPU *catches* the interrupt and *dispatches* it to the interrupt handler, and the handler *clears* the interrupt by servicing the device. Figure 1.4 summarizes the interrupt-driven I/O cycle.

The basic interrupt mechanism just described enables the CPU to respond to an asynchronous event, as when a device controller becomes ready for service. In a modern operating system, however, we need more sophisticated interrupt-handling features.

1. We need the ability to defer interrupt handling during critical processing.
2. We need an efficient way to dispatch to the proper interrupt handler for a device.
3. We need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency.

In modern computer hardware, these three features are provided by the CPU and the **interrupt-controller hardware**.

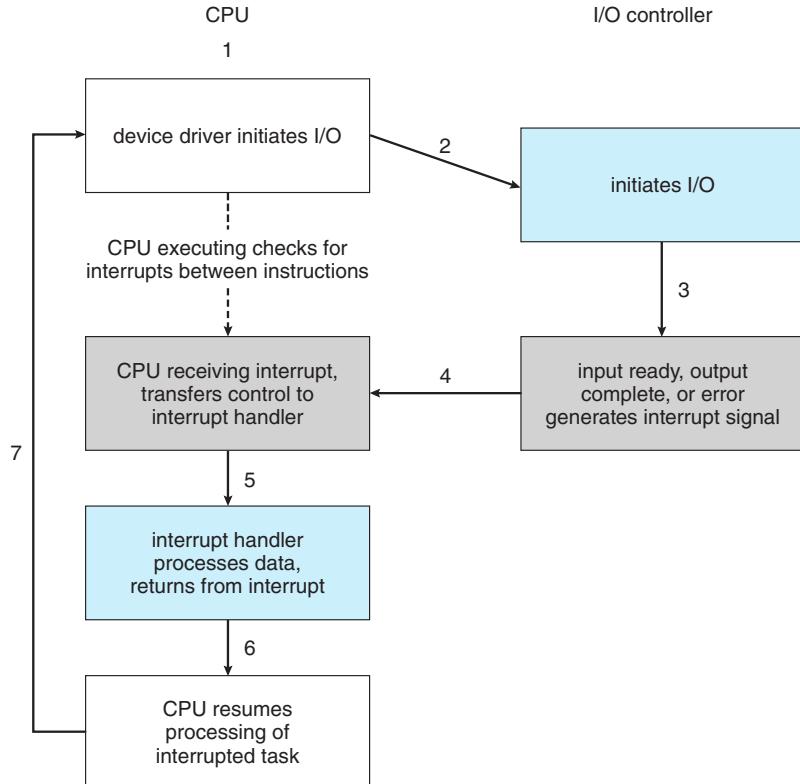


Figure 1.4 Interrupt-driven I/O cycle.

Most CPUs have two interrupt request lines. One is the **nonmaskable interrupt**, which is reserved for events such as unrecoverable memory errors. The second interrupt line is **maskable**: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service.

Recall that the purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service. In practice, however, computers have more devices (and, hence, interrupt handlers) than they have address elements in the interrupt vector. A common way to solve this problem is to use **interrupt chaining**, in which each element in the interrupt vector points to the head of a list of interrupt handlers. When an interrupt is raised, the handlers on the corresponding list are called one by one, until one is found that can service the request. This structure is a compromise between the overhead of a huge interrupt table and the inefficiency of dispatching to a single interrupt handler.

Figure 1.5 illustrates the design of the interrupt vector for Intel processors. The events from 0 to 31, which are nonmaskable, are used to signal various error conditions. The events from 32 to 255, which are maskable, are used for purposes such as device-generated interrupts.

The interrupt mechanism also implements a system of **interrupt priority levels**. These levels enable the CPU to defer the handling of low-priority inter-

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Figure 1.5 Intel processor event-vector table.

rupts without masking all interrupts and makes it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt.

In summary, interrupts are used throughout modern operating systems to handle asynchronous events (and for other purposes we will discuss throughout the text). Device controllers and hardware faults raise interrupts. To enable the most urgent work to be done first, modern computers use a system of interrupt priorities. Because interrupts are used so heavily for time-sensitive processing, efficient interrupt handling is required for good system performance.

### 1.2.2 Storage Structure

The CPU can load instructions only from memory, so any programs must first be loaded into memory to run. General-purpose computers run most of their programs from rewritable memory, called main memory (also called **random-access memory**, or **RAM**). Main memory commonly is implemented in a semiconductor technology called **dynamic random-access memory (DRAM)**.

Computers use other forms of memory as well. For example, the first program to run on computer power-on is a **bootstrap program**, which then loads the operating system. Since RAM is **volatile**—loses its content when power is turned off or otherwise lost—we cannot trust it to hold the bootstrap program. Instead, for this and some other purposes, the computer uses electrically erasable programmable read-only memory (EEPROM) and other forms of **firmware**—storage that is infrequently written to and is nonvolatile. EEPROM

### STORAGE DEFINITIONS AND NOTATION

The basic unit of computer storage is the **bit**. A bit can contain one of two values, 0 and 1. All other storage in a computer is based on collections of bits. Given enough bits, it is amazing how many things a computer can represent: numbers, letters, images, movies, sounds, documents, and programs, to name a few. A **byte** is 8 bits, and on most computers it is the smallest convenient chunk of storage. For example, most computers don't have an instruction to move a bit but do have one to move a byte. A less common term is **word**, which is a given computer architecture's native unit of data. A word is made up of one or more bytes. For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words. A computer executes many operations in its native word size rather than a byte at a time.

Computer storage, along with most computer throughput, is generally measured and manipulated in bytes and collections of bytes. A **kilobyte**, or **KB**, is 1,024 bytes; a **megabyte**, or **MB**, is  $1,024^2$  bytes; a **gigabyte**, or **GB**, is  $1,024^3$  bytes; a **terabyte**, or **TB**, is  $1,024^4$  bytes; and a **petabyte**, or **PB**, is  $1,024^5$  bytes. Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes. Networking measurements are an exception to this general rule; they are given in bits (because networks move data a bit at a time).

can be changed but cannot be changed frequently. In addition, it is low speed, and so it contains mostly static programs and data that aren't frequently used. For example, the iPhone uses EEPROM to store serial numbers and hardware information about the device.

All forms of memory provide an array of bytes. Each byte has its own address. Interaction is achieved through a sequence of **load** or **store** instructions to specific memory addresses. The **load** instruction moves a byte or word from main memory to an internal register within the CPU, whereas the **store** instruction moves the content of a register to main memory. Aside from explicit loads and stores, the CPU automatically loads instructions from main memory for execution from the location stored in the program counter.

A typical instruction–execution cycle, as executed on a system with a **von Neumann architecture**, first fetches an instruction from memory and stores that instruction in the **instruction register**. The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory. Notice that the memory unit sees only a stream of memory addresses. It does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, or some other means) or what they are for (instructions or data). Accordingly, we can ignore **how** a memory address is generated by a program. We are interested only in the sequence of memory addresses generated by the running program.

Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is not possible on most systems for two reasons:

1. Main memory is usually too small to store all needed programs and data permanently.
2. Main memory, as mentioned, is volatile—it loses its contents when power is turned off or otherwise lost.

Thus, most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently.

The most common secondary-storage devices are **hard-disk drives (HDDs)** and **nonvolatile memory (NVM) devices**, which provide storage for both programs and data. Most programs (system and application) are stored in secondary storage until they are loaded into memory. Many programs then use secondary storage as both the source and the destination of their processing. Secondary storage is also much slower than main memory. Hence, the proper management of secondary storage is of central importance to a computer system, as we discuss in Chapter 11.

In a larger sense, however, the storage structure that we have described—consisting of registers, main memory, and secondary storage—is only one of many possible storage system designs. Other possible components include cache memory, CD-ROM or blu-ray, magnetic tapes, and so on. Those that are slow enough and large enough that they are used only for special purposes—to store backup copies of material stored on other devices, for example—are called **tertiary storage**. Each storage system provides the basic functions of storing a datum and holding that datum until it is retrieved at a later time. The main differences among the various storage systems lie in speed, size, and volatility.

The wide variety of storage systems can be organized in a hierarchy (Figure 1.6) according to storage capacity and access time. As a general rule, there is a

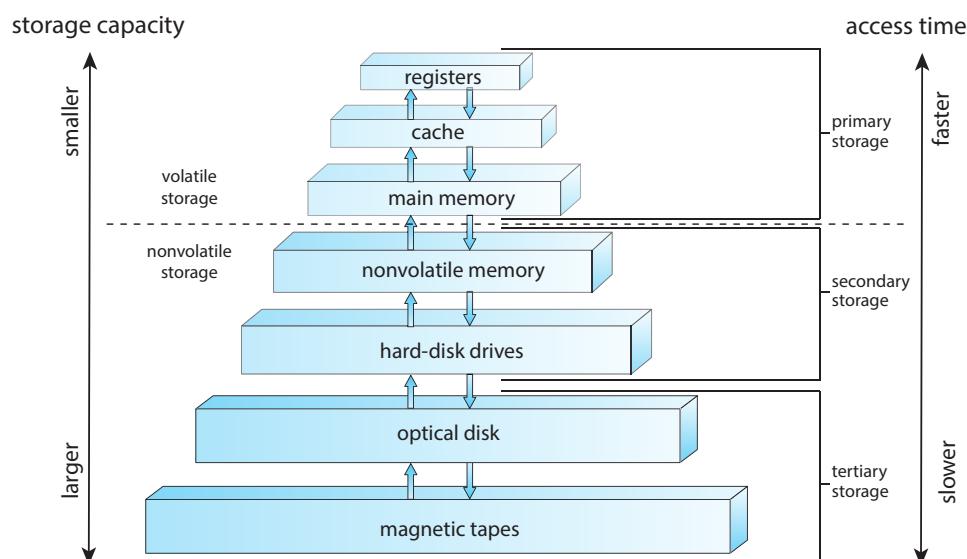


Figure 1.6 Storage-device hierarchy.

trade-off between size and speed, with smaller and faster memory closer to the CPU. As shown in the figure, in addition to differing in speed and capacity, the various storage systems are either volatile or nonvolatile. Volatile storage, as mentioned earlier, loses its contents when the power to the device is removed, so data must be written to nonvolatile storage for safekeeping.

The top four levels of memory in the figure are constructed using **semiconductor memory**, which consists of semiconductor-based electronic circuits. NVM devices, at the fourth level, have several variants but in general are faster than hard disks. The most common form of NVM device is flash memory, which is popular in mobile devices such as smartphones and tablets. Increasingly, flash memory is being used for long-term storage on laptops, desktops, and servers as well.

Since storage plays an important role in operating-system structure, we will refer to it frequently in the text. In general, we will use the following terminology:

- Volatile storage will be referred to simply as **memory**. If we need to emphasize a particular type of storage device (for example, a register), we will do so explicitly.
- Nonvolatile storage retains its contents when power is lost. It will be referred to as **NVS**. The vast majority of the time we spend on NVS will be on secondary storage. This type of storage can be classified into two distinct types:
  - **Mechanical**. A few examples of such storage systems are HDDs, optical disks, holographic storage, and magnetic tape. If we need to emphasize a particular type of mechanical storage device (for example, magnetic tape), we will do so explicitly.
  - **Electrical**. A few examples of such storage systems are flash memory, FRAM, NRAM, and SSD. Electrical storage will be referred to as **NVM**. If we need to emphasize a particular type of electrical storage device (for example, SSD), we will do so explicitly.

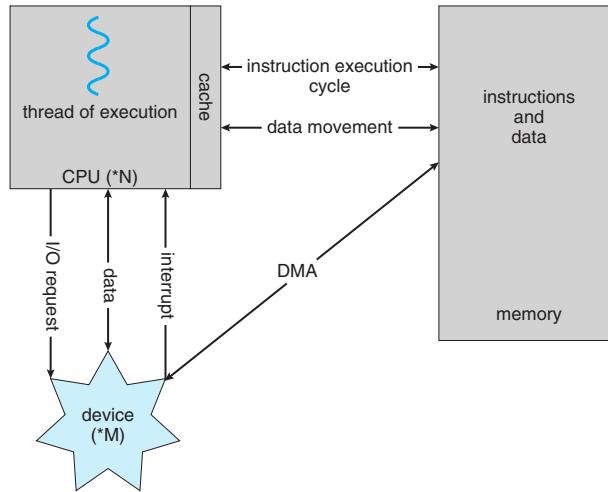
Mechanical storage is generally larger and less expensive per byte than electrical storage. Conversely, electrical storage is typically costly, smaller, and faster than mechanical storage.

The design of a complete storage system must balance all the factors just discussed: it must use only as much expensive memory as necessary while providing as much inexpensive, nonvolatile storage as possible. Caches can be installed to improve performance where a large disparity in access time or transfer rate exists between two components.

### 1.2.3 I/O Structure

A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices.

Recall from the beginning of this section that a general-purpose computer system consists of multiple devices, all of which exchange data via a common



**Figure 1.7** How a modern computer system works.

bus. The form of interrupt-driven I/O described in Section 1.2.1 is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as NVS I/O. To solve this problem, **direct memory access (DMA)** is used. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from the device and main memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices. While the device controller is performing these operations, the CPU is available to accomplish other work.

Some high-end systems use switch rather than bus architecture. On these systems, multiple components can talk to other components concurrently, rather than competing for cycles on a shared bus. In this case, DMA is even more effective. Figure 1.7 shows the interplay of all components of a computer system.

## 1.3 Computer-System Architecture

In Section 1.2, we introduced the general structure of a typical computer system. A computer system can be organized in a number of different ways, which we can categorize roughly according to the number of general-purpose processors used.

### 1.3.1 Single-Processor Systems

Many years ago, most computer systems used a single processor containing one CPU with a single processing core. The **core** is the component that executes instructions and registers for storing data locally. The one main CPU with its core is capable of executing a general-purpose instruction set, including instructions from processes. These systems have other special-purpose proces-

sors as well. They may come in the form of device-specific processors, such as disk, keyboard, and graphics controllers.

All of these special-purpose processors run a limited instruction set and do not run processes. Sometimes, they are managed by the operating system, in that the operating system sends them information about their next task and monitors their status. For example, a disk-controller microprocessor receives a sequence of requests from the main CPU core and implements its own disk queue and scheduling algorithm. This arrangement relieves the main CPU of the overhead of disk scheduling. PCs contain a microprocessor in the keyboard to convert the keystrokes into codes to be sent to the CPU. In other systems or circumstances, special-purpose processors are low-level components built into the hardware. The operating system cannot communicate with these processors; they do their jobs autonomously. The use of special-purpose microprocessors is common and does not turn a single-processor system into a multiprocessor. If there is only one general-purpose CPU with a single processing core, then the system is a single-processor system. According to this definition, however, very few contemporary computer systems are single-processor systems.

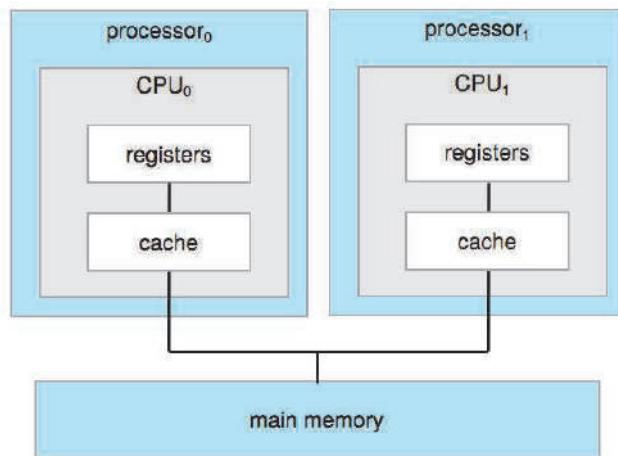
### 1.3.2 Multiprocessor Systems

On modern computers, from mobile devices to servers, **multiprocessor systems** now dominate the landscape of computing. Traditionally, such systems have two (or more) processors, each with a single-core CPU. The processors share the computer bus and sometimes the clock, memory, and peripheral devices. The primary advantage of multiprocessor systems is increased throughput. That is, by increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with  $N$  processors is not  $N$ , however; it is less than  $N$ . When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors.

The most common multiprocessor systems use **symmetric multiprocessing (SMP)**, in which each peer CPU processor performs all tasks, including operating-system functions and user processes. Figure 1.8 illustrates a typical SMP architecture with two processors, each with its own CPU. Notice that each CPU processor has its own set of registers, as well as a private—or local—cache. However, all processors share physical memory over the system bus.

The benefit of this model is that many processes can run simultaneously— $N$  processes can run if there are  $N$  CPUs—without causing performance to deteriorate significantly. However, since the CPUs are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies. These inefficiencies can be avoided if the processors share certain data structures. A multiprocessor system of this form will allow processes and resources—such as memory—to be shared dynamically among the various processors and can lower the workload variance among the processors. Such a system must be written carefully, as we shall see in Chapter 5 and Chapter 6.

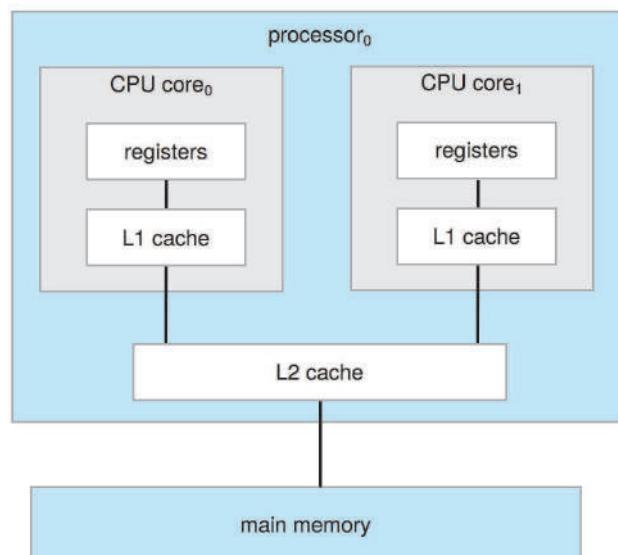
The definition of *multiprocessor* has evolved over time and now includes **multicore** systems, in which multiple computing cores reside on a single chip. Multicore systems can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication.



**Figure 1.8** Symmetric multiprocessing architecture.

In addition, one chip with multiple cores uses significantly less power than multiple single-core chips, an important issue for mobile devices as well as laptops.

In Figure 1.9, we show a dual-core design with two cores on the same processor chip. In this design, each core has its own register set, as well as its own local cache, often known as a level 1, or L1, cache. Notice, too, that a level 2 (L2) cache is local to the chip but is shared by the two processing cores. Most architectures adopt this approach, combining local and shared caches, where local, lower-level caches are generally smaller and faster than higher-level shared



**Figure 1.9** A dual-core design with two cores on the same chip.

**DEFINITIONS OF COMPUTER SYSTEM COMPONENTS**

- **CPU**—The hardware that executes instructions.
- **Processor**—A physical chip that contains one or more CPUs.
- **Core**—The basic computation unit of the CPU.
- **Multicore**—Including multiple computing cores on the same CPU.
- **Multiprocessor**—Including multiple processors.

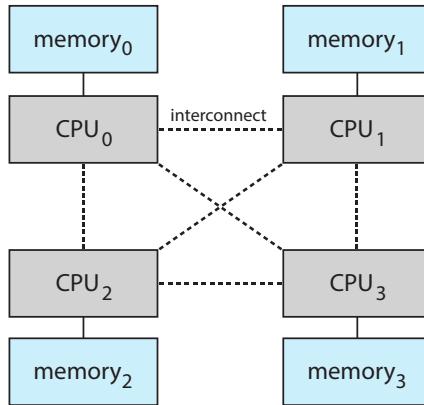
Although virtually all systems are now multicore, we use the general term *CPU* when referring to a single computational unit of a computer system and *core* as well as *multicore* when specifically referring to one or more cores on a CPU.

caches. Aside from architectural considerations, such as cache, memory, and bus contention, a multicore processor with  $N$  cores appears to the operating system as  $N$  standard CPUs. This characteristic puts pressure on operating-system designers—and application programmers—to make efficient use of these processing cores, an issue we pursue in Chapter 4. Virtually all modern operating systems—including Windows, macOS, and Linux, as well as Android and iOS mobile systems—support multicore SMP systems.

Adding additional CPUs to a multiprocessor system will increase computing power; however, as suggested earlier, the concept does not scale very well, and once we add too many CPUs, contention for the system bus becomes a bottleneck and performance begins to degrade. An alternative approach is instead to provide each CPU (or group of CPUs) with its own local memory that is accessed via a small, fast local bus. The CPUs are connected by a **shared system interconnect**, so that all CPUs share one physical address space. This approach—known as **non-uniform memory access**, or **NUMA**—is illustrated in Figure 1.10. The advantage is that, when a CPU accesses its local memory, not only is it fast, but there is also no contention over the system interconnect. Thus, NUMA systems can scale more effectively as more processors are added.

A potential drawback with a NUMA system is increased latency when a CPU must access remote memory across the system interconnect, creating a possible performance penalty. In other words, for example,  $\text{CPU}_0$  cannot access the local memory of  $\text{CPU}_3$  as quickly as it can access its own local memory, slowing down performance. Operating systems can minimize this NUMA penalty through careful CPU scheduling and memory management, as discussed in Section 5.5.2 and Section 10.5.4. Because NUMA systems can scale to accommodate a large number of processors, they are becoming increasingly popular on servers as well as high-performance computing systems.

Finally, **blade servers** are systems in which multiple processor boards, I/O boards, and networking boards are placed in the same chassis. The difference between these and traditional multiprocessor systems is that each blade-processor board boots independently and runs its own operating system. Some blade-server boards are multiprocessor as well, which blurs the lines between



**Figure 1.10** NUMA multiprocessing architecture.

types of computers. In essence, these servers consist of multiple independent multiprocessor systems.

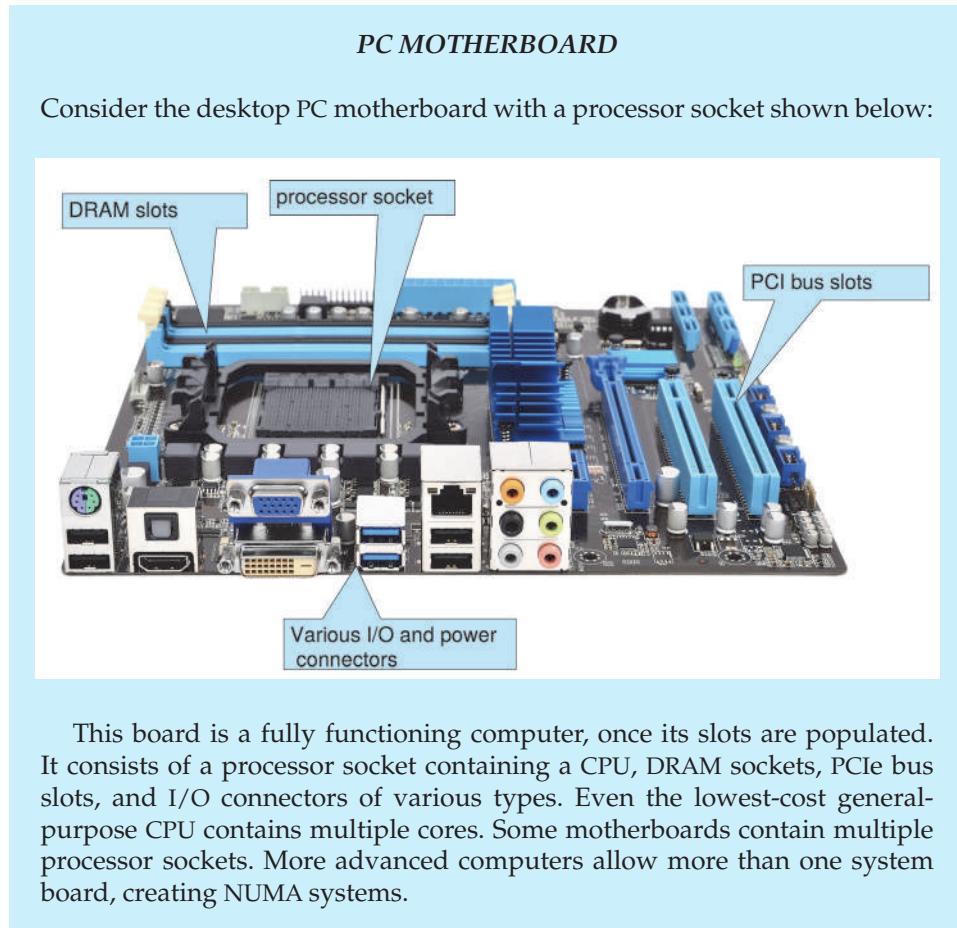
### 1.3.3 Clustered Systems

Another type of multiprocessor system is a **clustered system**, which gathers together multiple CPUs. Clustered systems differ from the multiprocessor systems described in Section 1.3.2 in that they are composed of two or more individual systems—or nodes—joined together; each node is typically a multicore system. Such systems are considered **loosely coupled**. We should note that the definition of *clustered* is not concrete; many commercial and open-source packages wrestle to define what a clustered system is and why one form is better than another. The generally accepted definition is that clustered computers share storage and are closely linked via a local-area network LAN (as described in Chapter 19) or a faster interconnect, such as InfiniBand.

Clustering is usually used to provide **high-availability service**—that is, a service that will continue even if one or more systems in the cluster fail. Generally, we obtain high availability by adding a level of redundancy in the system. A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the network). If the monitored machine fails, the monitoring machine can take ownership of its storage and restart the applications that were running on the failed machine. The users and clients of the applications see only a brief interruption of service.

High availability provides increased reliability, which is crucial in many applications. The ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation**. Some systems go beyond graceful degradation and are called **fault tolerant**, because they can suffer a failure of any single component and still continue operation. Fault tolerance requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected.

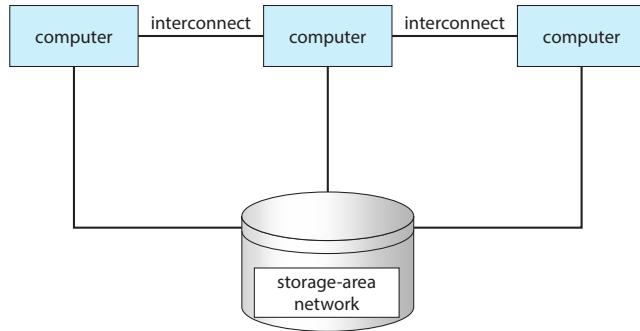
Clustering can be structured asymmetrically or symmetrically. In **asymmetric clustering**, one machine is in **hot-standby mode** while the other is running the applications. The hot-standby host machine does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active



server. In **symmetric clustering**, two or more hosts are running applications and are monitoring each other. This structure is obviously more efficient, as it uses all of the available hardware. However, it does require that more than one application be available to run.

Since a cluster consists of several computer systems connected via a network, clusters can also be used to provide **high-performance computing** environments. Such systems can supply significantly greater computational power than single-processor or even SMP systems because they can run an application concurrently on all computers in the cluster. The application must have been written specifically to take advantage of the cluster, however. This involves a technique known as **parallelization**, which divides a program into separate components that run in parallel on individual cores in a computer or computers in a cluster. Typically, these applications are designed so that once each computing node in the cluster has solved its portion of the problem, the results from all the nodes are combined into a final solution.

Other forms of clusters include parallel clusters and clustering over a wide-area network (WAN) (as described in Chapter 19). Parallel clusters allow multiple hosts to access the same data on shared storage. Because most oper-



**Figure 1.11** General structure of a clustered system.

ating systems lack support for simultaneous data access by multiple hosts, parallel clusters usually require the use of special versions of software and special releases of applications. For example, Oracle Real Application Cluster is a version of Oracle’s database that has been designed to run on a parallel cluster. Each machine runs Oracle, and a layer of software tracks access to the shared disk. Each machine has full access to all data in the database. To provide this shared access, the system must also supply access control and locking to ensure that no conflicting operations occur. This function, commonly known as a **distributed lock manager (DLM)**, is included in some cluster technology.

Cluster technology is changing rapidly. Some cluster products support thousands of systems in a cluster, as well as clustered nodes that are separated by miles. Many of these improvements are made possible by **storage-area networks (SANs)**, as described in Section 11.7.4, which allow many systems to attach to a pool of storage. If the applications and their data are stored on the SAN, then the cluster software can assign the application to run on any host that is attached to the SAN. If the host fails, then any other host can take over. In a database cluster, dozens of hosts can share the same database, greatly increasing performance and reliability. Figure 1.11 depicts the general structure of a clustered system.

## 1.4 Operating-System Operations

Now that we have discussed basic information about computer-system organization and architecture, we are ready to talk about operating systems. An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly, since they are organized along many different lines. There are, however, many commonalities, which we consider in this section.

For a computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run. As noted earlier, this initial program, or bootstrap program, tends to be simple. Typically, it is stored within the computer hardware in firmware. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and how to

## HADOOP

Hadoop is an open-source software framework that is used for distributed processing of large data sets (known as **big data**) in a clustered system containing simple, low-cost hardware components. Hadoop is designed to scale from a single system to a cluster containing thousands of computing nodes. Tasks are assigned to a node in the cluster, and Hadoop arranges communication between nodes to manage parallel computations to process and coalesce results. Hadoop also detects and manages failures in nodes, providing an efficient and highly reliable distributed computing service.

Hadoop is organized around the following three components:

1. A distributed file system that manages data and files across distributed computing nodes.
2. The YARN (“Yet Another Resource Negotiator”) framework, which manages resources within the cluster as well as scheduling tasks on nodes in the cluster.
3. The **MapReduce** system, which allows parallel processing of data across nodes in the cluster.

Hadoop is designed to run on Linux systems, and Hadoop applications can be written using several programming languages, including scripting languages such as PHP, Perl, and Python. Java is a popular choice for developing Hadoop applications, as Hadoop has several Java libraries that support MapReduce. More information on MapReduce and Hadoop can be found at [https://hadoop.apache.org/docs/r1.2.1/mapred\\_tutorial.html](https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html) and <https://hadoop.apache.org>

start executing that system. To accomplish this goal, the bootstrap program must locate the operating-system kernel and load it into memory.

Once the kernel is loaded and executing, it can start providing services to the system and its users. Some services are provided outside of the kernel by system programs that are loaded into memory at boot time to become **system daemons**, which run the entire time the kernel is running. On Linux, the first system program is “`systemd`,” and it starts many other daemons. Once this phase is complete, the system is fully booted, and the system waits for some event to occur.

If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt. In Section 1.2.1 we described hardware interrupts. Another form of interrupt is a **trap** (or an **exception**), which is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed by executing a special operation called a **system call**.

### 1.4.1 Multiprogramming and Multitasking

One of the most important aspects of operating systems is the ability to run multiple programs, as a single program cannot, in general, keep either the CPU or the I/O devices busy at all times. Furthermore, users typically *want* to run more than one program at a time as well. **Multiprogramming** increases CPU utilization, as well as keeping users satisfied, by organizing programs so that the CPU always has one to execute. In a multiprogrammed system, a program in execution is termed a **process**.

The idea is as follows: The operating system keeps several processes in memory simultaneously (Figure 1.12). The operating system picks and begins to execute one of these processes. Eventually, the process may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and executes, another process. When *that* process needs to wait, the CPU switches to *another* process, and so on. Eventually, the first process finishes waiting and gets the CPU back. As long as at least one process needs to execute, the CPU is never idle.

This idea is common in other life situations. A lawyer does not work for only one client at a time, for example. While one case is waiting to go to trial or have papers typed, the lawyer can work on another case. If she has enough clients, the lawyer will never be idle for lack of work. (Idle lawyers tend to become politicians, so there is a certain social value in keeping lawyers busy.)

**Multitasking** is a logical extension of multiprogramming. In multitasking systems, the CPU executes multiple processes by switching among them, but the switches occur frequently, providing the user with a fast **response time**. Consider that when a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output goes to a display for the user, and input comes from a user keyboard, mouse, or touch screen. Since interactive I/O typically runs at “people speeds,” it may take a long time to complete. Input, for example, may be

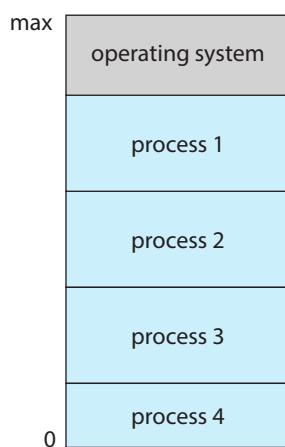


Figure 1.12 Memory layout for a multiprogramming system.

bounded by the user’s typing speed; seven characters per second is fast for people but incredibly slow for computers. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to another process.

Having several processes in memory at the same time requires some form of memory management, which we cover in Chapter 9 and Chapter 10. In addition, if several processes are ready to run at the same time, the system must choose which process will run next. Making this decision is **CPU scheduling**, which is discussed in Chapter 5. Finally, running multiple processes concurrently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management. We discuss these considerations throughout the text.

In a multitasking system, the operating system must ensure reasonable response time. A common method for doing so is **virtual memory**, a technique that allows the execution of a process that is not completely in memory (Chapter 10). The main advantage of this scheme is that it enables users to run programs that are larger than actual **physical memory**. Further, it abstracts main memory into a large, uniform array of storage, separating **logical memory** as viewed by the user from physical memory. This arrangement frees programmers from concern over memory-storage limitations.

Multiprogramming and multitasking systems must also provide a file system (Chapter 13, Chapter 14, and Chapter 15). The file system resides on a secondary storage; hence, storage management must be provided (Chapter 11). In addition, a system must protect resources from inappropriate use (Chapter 17). To ensure orderly execution, the system must also provide mechanisms for process synchronization and communication (Chapter 6 and Chapter 7), and it may ensure that processes do not get stuck in a deadlock, forever waiting for one another (Chapter 8).

### 1.4.2 Dual-Mode and Multimode Operation

Since the operating system and its users share the hardware and software resources of the computer system, a properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs—or the operating system itself—to execute incorrectly. In order to ensure the proper execution of the system, we must be able to distinguish between the execution of operating-system code and user-defined code. The approach taken by most computer systems is to provide hardware support that allows differentiation among various modes of execution.

At the very least, we need two separate *modes* of operation: **user mode** and **kernel mode** (also called **supervisor mode**, **system mode**, or **privileged mode**). A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill

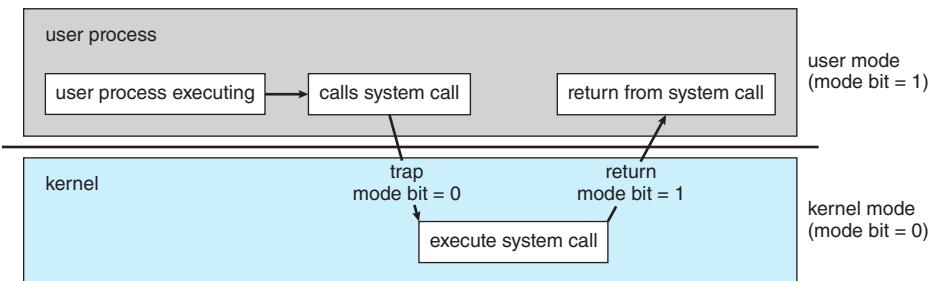


Figure 1.13 Transition from user to kernel mode.

the request. This is shown in Figure 1.13. As we shall see, this architectural enhancement is useful for many other aspects of system operation as well.

At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions**. The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

The instruction to switch to kernel mode is an example of a privileged instruction. Some other examples include I/O control, timer management, and interrupt management. Many additional privileged instructions are discussed throughout the text.

The concept of modes can be extended beyond two modes. For example, Intel processors have four separate **protection rings**, where ring 0 is kernel mode and ring 3 is user mode. (Although rings 1 and 2 could be used for various operating-system services, in practice they are rarely used.) ARMv8 systems have seven modes. CPUs that support virtualization (Section 18.1) frequently have a separate mode to indicate when the **virtual machine manager (VMM)** is in control of the system. In this mode, the VMM has more privileges than user processes but fewer than the kernel. It needs that level of privilege so it can create and manage virtual machines, changing the CPU state to do so.

We can now better understand the life cycle of instruction execution in a computer system. Initial control resides in the operating system, where instructions are executed in kernel mode. When control is given to a user application, the mode is set to user mode. Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call. Most contemporary operating systems—such as Microsoft Windows, Unix, and Linux—

take advantage of this dual-mode feature and provide greater protection for the operating system.

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. A system call is invoked in a variety of ways, depending on the functionality provided by the underlying processor. In all forms, it is the method used by a process to request action by the operating system. A system call usually takes the form of a trap to a specific location in the interrupt vector. This trap can be executed by a generic trap instruction, although some systems have a specific `syscall` instruction to invoke a system call.

When a system call is executed, it is typically treated by the hardware as a software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode. The system-call service routine is a part of the operating system. The kernel examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting. Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers). The kernel verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call. We describe system calls more fully in Section 2.3.

Once hardware protection is in place, it detects errors that violate modes. These errors are normally handled by the operating system. If a user program fails in some way—such as by making an attempt either to execute an illegal instruction or to access memory that is not in the user's address space—then the hardware traps to the operating system. The trap transfers control through the interrupt vector to the operating system, just as an interrupt does. When a program error occurs, the operating system must terminate the program abnormally. This situation is handled by the same code as a user-requested abnormal termination. An appropriate error message is given, and the memory of the program may be dumped. The memory dump is usually written to a file so that the user or programmer can examine it and perhaps correct it and restart the program.

### 1.4.3 Timer

We must ensure that the operating system maintains control over the CPU. We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system. To accomplish this goal, we can use a **timer**. A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A **variable timer** is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond.

Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or

**LINUX TIMERS**

On Linux systems, the kernel configuration parameter `HZ` specifies the frequency of timer interrupts. An `HZ` value of 250 means that the timer generates 250 interrupts per second, or one interrupt every 4 milliseconds. The value of `HZ` depends upon how the kernel is configured, as well the machine type and architecture on which it is running. A related kernel variable is `jiffies`, which represent the number of timer interrupts that have occurred since the system was booted. A programming project in Chapter 2 further explores timing in the Linux kernel.

may give the program more time. Clearly, instructions that modify the content of the timer are privileged.

## 1.5 Resource Management

As we have seen, an operating system is a **resource manager**. The system's CPU, memory space, file-storage space, and I/O devices are among the resources that the operating system must manage.

### 1.5.1 Process Management

A program can do nothing unless its instructions are executed by a CPU. A program in execution, as mentioned, is a process. A program such as a compiler is a process, and a word-processing program being run by an individual user on a PC is a process. Similarly, a social media app on a mobile device is a process. For now, you can consider a process to be an instance of a program in execution, but later you will see that the concept is more general. As described in Chapter 3, it is possible to provide system calls that allow processes to create subprocesses to execute concurrently.

A process needs certain resources—including CPU time, memory, files, and I/O devices—to accomplish its task. These resources are typically allocated to the process while it is running. In addition to the various physical and logical resources that a process obtains when it is created, various initialization data (input) may be passed along. For example, consider a process running a web browser whose function is to display the contents of a web page on a screen. The process will be given the URL as an input and will execute the appropriate instructions and system calls to obtain and display the desired information on the screen. When the process terminates, the operating system will reclaim any reusable resources.

We emphasize that a program by itself is not a process. A program is a *passive* entity, like the contents of a file stored on disk, whereas a process is an *active* entity. A single-threaded process has one **program counter** specifying the next instruction to execute. (Threads are covered in Chapter 4.) The execution of such a process must be sequential. The CPU executes one instruction of the process after another, until the process completes. Further, at any time, one instruction at most is executed on behalf of the process. Thus, although

two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. A multithreaded process has multiple program counters, each pointing to the next instruction to execute for a given thread.

A process is the unit of work in a system. A system consists of a collection of processes, some of which are operating-system processes (those that execute system code) and the rest of which are user processes (those that execute user code). All these processes can potentially execute concurrently—by multiplexing on a single CPU core—or in parallel across multiple CPU cores.

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Scheduling processes and threads on the CPUs
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication

We discuss process-management techniques in Chapter 3 through Chapter 7.

### **1.5.2 Memory Management**

As discussed in Section 1.2.2, the main memory is central to the operation of a modern computer system. Main memory is a large array of bytes, ranging in size from hundreds of thousands to billions. Each byte has its own address. Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. The CPU reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle (on a von Neumann architecture). As noted earlier, the main memory is generally the only large storage device that the CPU is able to address and access directly. For example, for the CPU to process data from disk, those data must first be transferred to main memory by CPU-generated I/O calls. In the same way, instructions must be in memory for the CPU to execute them.

For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute addresses. Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed.

To improve both the utilization of the CPU and the speed of the computer's response to its users, general-purpose computers must keep several programs in memory, creating a need for memory management. Many different memory-management schemes are used. These schemes reflect various approaches, and the effectiveness of any given algorithm depends on the situation. In selecting a memory-management scheme for a specific system, we must take into account many factors—especially the *hardware* design of the system. Each algorithm requires its own hardware support.

The operating system is responsible for the following activities in connection with memory management:

- Keeping track of which parts of memory are currently being used and which process is using them
- Allocating and deallocating memory space as needed
- Deciding which processes (or parts of processes) and data to move into and out of memory

Memory-management techniques are discussed in Chapter 9 and Chapter 10.

### 1.5.3 File-System Management

To make the computer system convenient for users, the operating system provides a uniform, logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**. The operating system maps files onto physical media and accesses these files via the storage devices.

File management is one of the most visible components of an operating system. Computers can store information on several different types of physical media. Secondary storage is the most common, but tertiary storage is also possible. Each of these media has its own characteristics and physical organization. Most are controlled by a device, such as a disk drive, that also has its own unique characteristics. These properties include access speed, capacity, data-transfer rate, and access method (sequential or random).

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free-form (for example, text files), or they may be formatted rigidly (for example, fixed fields such as an mp3 music file). Clearly, the concept of a file is an extremely general one.

The operating system implements the abstract concept of a file by managing mass storage media and the devices that control them. In addition, files are normally organized into directories to make them easier to use. Finally, when multiple users have access to files, it may be desirable to control which user may access a file and how that user may access it (for example, read, write, append).

The operating system is responsible for the following activities in connection with file management:

- Creating and deleting files
- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto mass storage
- Backing up files on stable (nonvolatile) storage media

File-management techniques are discussed in Chapter 13, Chapter 14, and Chapter 15.

#### **1.5.4 Mass-Storage Management**

As we have already seen, the computer system must provide secondary storage to back up main memory. Most modern computer systems use HDDs and NVM devices as the principal on-line storage media for both programs and data. Most programs—including compilers, web browsers, word processors, and games—are stored on these devices until loaded into memory. The programs then use the devices as both the source and the destination of their processing. Hence, the proper management of secondary storage is of central importance to a computer system. The operating system is responsible for the following activities in connection with secondary storage management:

- Mounting and unmounting
- Free-space management
- Storage allocation
- Disk scheduling
- Partitioning
- Protection

Because secondary storage is used frequently and extensively, it must be used efficiently. The entire speed of operation of a computer may hinge on the speeds of the secondary storage subsystem and the algorithms that manipulate that subsystem.

At the same time, there are many uses for storage that is slower and lower in cost (and sometimes higher in capacity) than secondary storage. Backups of disk data, storage of seldom-used data, and long-term archival storage are some examples. Magnetic tape drives and their tapes and CD DVD and Blu-ray drives and platters are typical tertiary storage devices.

Tertiary storage is not crucial to system performance, but it still must be managed. Some operating systems take on this task, while others leave tertiary-storage management to application programs. Some of the functions that operating systems can provide include mounting and unmounting media in devices, allocating and freeing the devices for exclusive use by processes, and migrating data from secondary to tertiary storage.

Techniques for secondary storage and tertiary storage management are discussed in Chapter 11.

#### **1.5.5 Cache Management**

**Caching** is an important principle of computer systems. Here's how it works. Information is normally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system—the cache—on a temporary basis. When we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache.

Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

**Figure 1.14** Characteristics of various types of storage.

If it is not, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon.

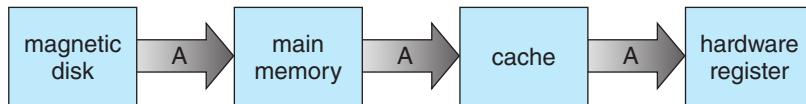
In addition, internal programmable registers provide a high-speed cache for main memory. The programmer (or compiler) implements the register-allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory.

Other caches are implemented totally in hardware. For instance, most systems have an instruction cache to hold the instructions expected to be executed next. Without this cache, the CPU would have to wait several cycles while an instruction was fetched from main memory. For similar reasons, most systems have one or more high-speed data caches in the memory hierarchy. We are not concerned with these hardware-only caches in this text, since they are outside the control of the operating system.

Because caches have limited size, **cache management** is an important design problem. Careful selection of the cache size and of a replacement policy can result in greatly increased performance, as you can see by examining Figure 1.14. Replacement algorithms for software-controlled caches are discussed in Chapter 10.

The movement of information between levels of a storage hierarchy may be either explicit or implicit, depending on the hardware design and the controlling operating-system software. For instance, data transfer from cache to CPU and registers is usually a hardware function, with no operating-system intervention. In contrast, transfer of data from disk to memory is usually controlled by the operating system.

In a hierarchical storage structure, the same data may appear in different levels of the storage system. For example, suppose that an integer A that is to be incremented by 1 is located in file B, and file B resides on hard disk. The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory. This operation is followed by copying A to the cache and to an internal register. Thus, the copy of A appears in several places: on the hard disk, in main memory, in the cache, and in an internal register (see Figure 1.15). Once the increment takes place in the internal register, the value of A differs in the various storage systems. The value of A



**Figure 1.15** Migration of integer A from disk to register.

becomes the same only after the new value of A is written from the internal register back to the hard disk.

In a computing environment where only one process executes at a time, this arrangement poses no difficulties, since an access to integer A will always be to the copy at the highest level of the hierarchy. However, in a multitasking environment, where the CPU is switched back and forth among various processes, extreme care must be taken to ensure that, if several processes wish to access A, then each of these processes will obtain the most recently updated value of A.

The situation becomes more complicated in a multiprocessor environment where, in addition to maintaining internal registers, each of the CPUs also contains a local cache (refer back to Figure 1.8). In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all execute in parallel, we must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency**, and it is usually a hardware issue (handled below the operating-system level).

In a distributed environment, the situation becomes even more complex. In this environment, several copies (or replicas) of the same file can be kept on different computers. Since the various replicas may be accessed and updated concurrently, some distributed systems ensure that, when a replica is updated in one place, all other replicas are brought up to date as soon as possible. There are various ways to achieve this guarantee, as we discuss in Chapter 19.

### 1.5.6 I/O System Management

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, in UNIX, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the **I/O subsystem**. The I/O subsystem consists of several components:

- A memory-management component that includes buffering, caching, and spooling
- A general device-driver interface
- Drivers for specific hardware devices

Only the device driver knows the peculiarities of the specific device to which it is assigned.

We discussed earlier in this chapter how interrupt handlers and device drivers are used in the construction of efficient I/O subsystems. In Chapter 12, we discuss how the I/O subsystem interfaces to the other system components, manages devices, transfers data, and detects I/O completion.

## 1.6 Security and Protection

If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, mechanisms ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system. For example, memory-addressing hardware ensures that a process can execute only within its own address space. The timer ensures that no process can gain control of the CPU without eventually relinquishing control. Device-control registers are not accessible to users, so the integrity of the various peripheral devices is protected.

**Protection**, then, is any mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means to specify the controls to be imposed and to enforce the controls.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning. Furthermore, an unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides a means to distinguish between authorized and unauthorized usage, as we discuss in Chapter 17.

A system can have adequate protection but still be prone to failure and allow inappropriate access. Consider a user whose authentication information (her means of identifying herself to the system) is stolen. Her data could be copied or deleted, even though file and memory protection are working. It is the job of **security** to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial-of-service attacks (which use all of a system's resources and so keep legitimate users out of the system), identity theft, and theft of service (unauthorized use of a system). Prevention of some of these attacks is considered an operating-system function on some systems, while other systems leave it to policy or additional software. Due to the alarming rise in security incidents, operating-system security features are a fast-growing area of research and implementation. We discuss security in Chapter 16.

Protection and security require the system to be able to distinguish among all its users. Most operating systems maintain a list of user names and associated **user identifier** (**user IDs**). In Windows parlance, this is a **security ID** (**SID**). These numerical IDs are unique, one per user. When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all of the user's processes and threads. When an ID needs to be readable by a user, it is translated back to the user name via the user name list.

In some circumstances, we wish to distinguish among sets of users rather than individual users. For example, the owner of a file on a UNIX system may be allowed to issue all operations on that file, whereas a selected set of users may be allowed only to read the file. To accomplish this, we need to define a group name and the set of users belonging to that group. Group functionality can be implemented as a system-wide list of group names and **group identifier**. A user can be in one or more groups, depending on operating-system design

decisions. The user's group IDs are also included in every associated process and thread.

In the course of normal system use, the user ID and group ID for a user are sufficient. However, a user sometimes needs to **escalate privileges** to gain extra permissions for an activity. The user may need access to a device that is restricted, for example. Operating systems provide various methods to allow privilege escalation. On UNIX, for instance, the *setuid* attribute on a program causes that program to run with the user ID of the owner of the file, rather than the current user's ID. The process runs with this **effective UID** until it turns off the extra privileges or terminates.

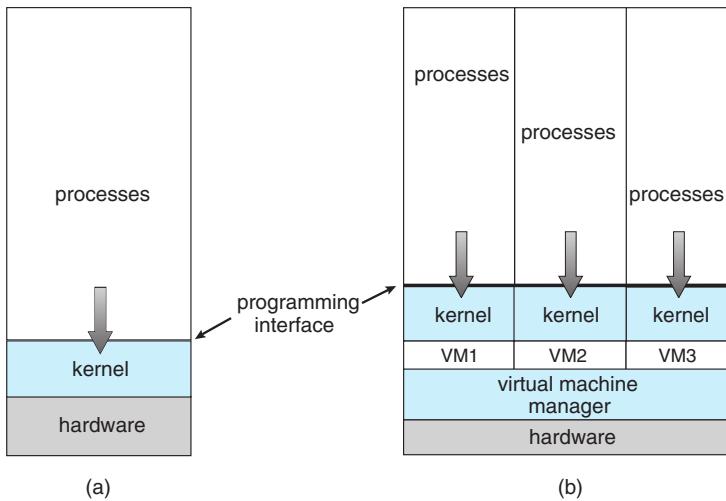
## 1.7 Virtualization

**Virtualization** is a technology that allows us to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate environment is running on its own private computer. These environments can be viewed as different individual operating systems (for example, Windows and UNIX) that may be running at the same time and may interact with each other. A user of a **virtual machine** can switch among the various operating systems in the same way a user can switch among the various processes running concurrently in a single operating system.

Virtualization allows operating systems to run as applications within other operating systems. At first blush, there seems to be little reason for such functionality. But the virtualization industry is vast and growing, which is a testament to its utility and importance.

Broadly speaking, virtualization software is one member of a class that also includes emulation. **Emulation**, which involves simulating computer hardware in software, is typically used when the source CPU type is different from the target CPU type. For example, when Apple switched from the IBM Power CPU to the Intel x86 CPU for its desktop and laptop computers, it included an emulation facility called "Rosetta," which allowed applications compiled for the IBM CPU to run on the Intel CPU. That same concept can be extended to allow an entire operating system written for one platform to run on another. Emulation comes at a heavy price, however. Every machine-level instruction that runs natively on the source system must be translated to the equivalent function on the target system, frequently resulting in several target instructions. If the source and target CPUs have similar performance levels, the emulated code may run much more slowly than the native code.

With virtualization, in contrast, an operating system that is natively compiled for a particular CPU architecture runs within another operating system also native to that CPU. Virtualization first came about on IBM mainframes as a method for multiple users to run tasks concurrently. Running multiple virtual machines allowed (and still allows) many users to run tasks on a system designed for a single user. Later, in response to problems with running multiple Microsoft Windows applications on the Intel x86 CPU, VMware created a new virtualization technology in the form of an application that ran on Windows. That application ran one or more **guest** copies of Windows or other native x86 operating systems, each running its own applications. (See Figure 1.16.)



**Figure 1.16** A computer running (a) a single operating system and (b) three virtual machines.

Windows was the **host** operating system, and the VMware application was the **virtual machine manager (VMM)**. The VMM runs the guest operating systems, manages their resource use, and protects each guest from the others.

Even though modern operating systems are fully capable of running multiple applications reliably, the use of virtualization continues to grow. On laptops and desktops, a VMM allows the user to install multiple operating systems for exploration or to run applications written for operating systems other than the native host. For example, an Apple laptop running macOS on the x86 CPU can run a Windows 10 guest to allow execution of Windows applications. Companies writing software for multiple operating systems can use virtualization to run all of those operating systems on a single physical server for development, testing, and debugging. Within data centers, virtualization has become a common method of executing and managing computing environments. VMMs like VMware ESX and Citrix XenServer no longer run on host operating systems but rather *are* the host operating systems, providing services and resource management to virtual machine processes.

With this text, we provide a Linux virtual machine that allows you to run Linux—as well as the development tools we provide—on your personal system regardless of your host operating system. Full details of the features and implementation of virtualization can be found in Chapter 18.

## 1.8 Distributed Systems

A distributed system is a collection of physically separate, possibly heterogeneous computer systems that are networked to provide users with access to the various resources that the system maintains. Access to a shared resource increases computation speed, functionality, data availability, and reliability. Some operating systems generalize network access as a form of file access, with the details of networking contained in the network interface's device driver.

Others make users specifically invoke network functions. Generally, systems contain a mix of the two modes—for example FTP and NFS. The protocols that create a distributed system can greatly affect that system’s utility and popularity.

A **network**, in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality. Networks vary by the protocols used, the distances between nodes, and the transport media. **TCP/IP** is the most common network protocol, and it provides the fundamental architecture of the Internet. Most operating systems support TCP/IP, including all general-purpose ones. Some systems support proprietary protocols to suit their needs. For an operating system, it is necessary only that a network protocol have an interface device—a network adapter, for example—with a device driver to manage it, as well as software to handle data. These concepts are discussed throughout this book.

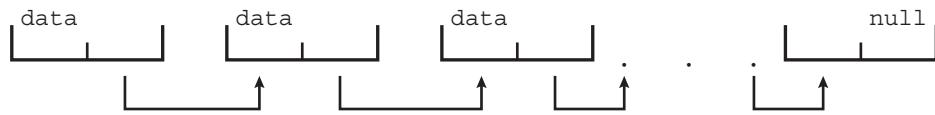
Networks are characterized based on the distances between their nodes. A **local-area network (LAN)** connects computers within a room, a building, or a campus. A **wide-area network (WAN)** usually links buildings, cities, or countries. A global company may have a WAN to connect its offices worldwide, for example. These networks may run one protocol or several protocols. The continuing advent of new technologies brings about new forms of networks. For example, a **metropolitan-area network (MAN)** could link buildings within a city. BlueTooth and 802.11 devices use wireless technology to communicate over a distance of several feet, in essence creating a **personal-area network (PAN)** between a phone and a headset or a smartphone and a desktop computer.

The media to carry networks are equally varied. They include copper wires, fiber strands, and wireless transmissions between satellites, microwave dishes, and radios. When computing devices are connected to cellular phones, they create a network. Even very short-range infrared communication can be used for networking. At a rudimentary level, whenever computers communicate, they use or create a network. These networks also vary in their performance and reliability.

Some operating systems have taken the concept of networks and distributed systems further than the notion of providing network connectivity. A **network operating system** is an operating system that provides features such as file sharing across the network, along with a communication scheme that allows different processes on different computers to exchange messages. A computer running a network operating system acts autonomously from all other computers on the network, although it is aware of the network and is able to communicate with other networked computers. A distributed operating system provides a less autonomous environment. The different computers communicate closely enough to provide the illusion that only a single operating system controls the network. We cover computer networks and distributed systems in Chapter 19.

## 1.9 Kernel Data Structures

We turn next to a topic central to operating-system implementation: the way data are structured in the system. In this section, we briefly describe several fundamental data structures used extensively in operating systems. Readers



**Figure 1.17** Singly linked list.

who require further details on these structures, as well as others, should consult the bibliography at the end of the chapter.

### 1.9.1 Lists, Stacks, and Queues

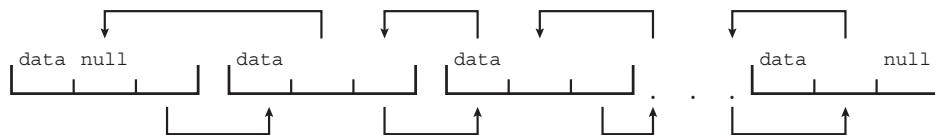
An array is a simple data structure in which each element can be accessed directly. For example, main memory is constructed as an array. If the data item being stored is larger than one byte, then multiple bytes can be allocated to the item, and the item is addressed as “item number  $\times$  item size.” But what about storing an item whose size may vary? And what about removing an item if the relative positions of the remaining items must be preserved? In such situations, arrays give way to other data structures.

After arrays, lists are perhaps the most fundamental data structures in computer science. Whereas each item in an array can be accessed directly, the items in a list must be accessed in a particular order. That is, a **list** represents a collection of data values as a sequence. The most common method for implementing this structure is a **linked list**, in which items are linked to one another. Linked lists are of several types:

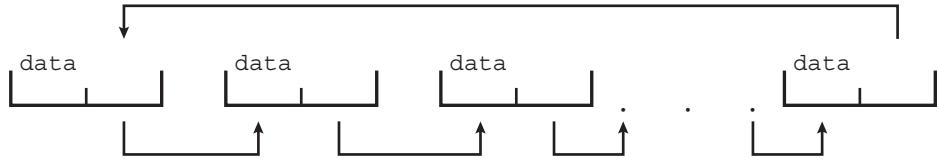
- In a **singly linked list**, each item points to its successor, as illustrated in Figure 1.17.
- In a **doubly linked list**, a given item can refer either to its predecessor or to its successor, as illustrated in Figure 1.18.
- In a **circularly linked list**, the last element in the list refers to the first element, rather than to null, as illustrated in Figure 1.19.

Linked lists accommodate items of varying sizes and allow easy insertion and deletion of items. One potential disadvantage of using a list is that performance for retrieving a specified item in a list of size  $n$  is linear— $O(n)$ , as it requires potentially traversing all  $n$  elements in the worst case. Lists are sometimes used directly by kernel algorithms. Frequently, though, they are used for constructing more powerful data structures, such as stacks and queues.

A **stack** is a sequentially ordered data structure that uses the last in, first out (**LIFO**) principle for adding and removing items, meaning that the last item



**Figure 1.18** Doubly linked list.



**Figure 1.19** Circularly linked list.

placed onto a stack is the first item removed. The operations for inserting and removing items from a stack are known as *push* and *pop*, respectively. An operating system often uses a stack when invoking function calls. Parameters, local variables, and the return address are pushed onto the stack when a function is called; returning from the function call pops those items off the stack.

A **queue**, in contrast, is a sequentially ordered data structure that uses the first in, first out (**FIFO**) principle: items are removed from a queue in the order in which they were inserted. There are many everyday examples of queues, including shoppers waiting in a checkout line at a store and cars waiting in line at a traffic signal. Queues are also quite common in operating systems—jobs that are sent to a printer are typically printed in the order in which they were submitted, for example. As we shall see in Chapter 5, tasks that are waiting to be run on an available CPU are often organized in queues.

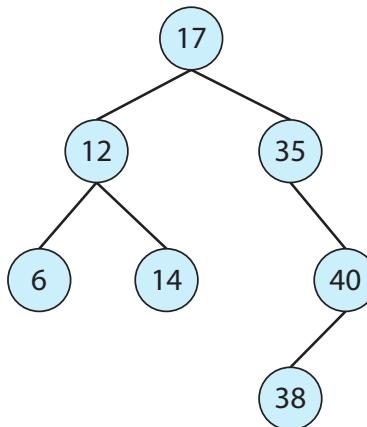
### 1.9.2 Trees

A **tree** is a data structure that can be used to represent data hierarchically. Data values in a tree structure are linked through parent–child relationships. In a **general tree**, a parent may have an unlimited number of children. In a **binary tree**, a parent may have at most two children, which we term the *left child* and the *right child*. A **binary search tree** additionally requires an ordering between the parent’s two children in which *left\_child*  $\leq$  *right\_child*. Figure 1.20 provides an example of a binary search tree. When we search for an item in a binary search tree, the worst-case performance is  $O(n)$  (consider how this can occur). To remedy this situation, we can use an algorithm to create a **balanced binary search tree**. Here, a tree containing  $n$  items has at most  $\lg n$  levels, thus ensuring worst-case performance of  $O(\lg n)$ . We shall see in Section 5.7.1 that Linux uses a balanced binary search tree (known as a **red-black tree**) as part its CPU-scheduling algorithm.

### 1.9.3 Hash Functions and Maps

A **hash function** takes data as its input, performs a numeric operation on the data, and returns a numeric value. This numeric value can then be used as an index into a table (typically an array) to quickly retrieve the data. Whereas searching for a data item through a list of size  $n$  can require up to  $O(n)$  comparisons, using a hash function for retrieving data from a table can be as good as  $O(1)$ , depending on implementation details. Because of this performance, hash functions are used extensively in operating systems.

One potential difficulty with hash functions is that two unique inputs can result in the same output value—that is, they can link to the same table

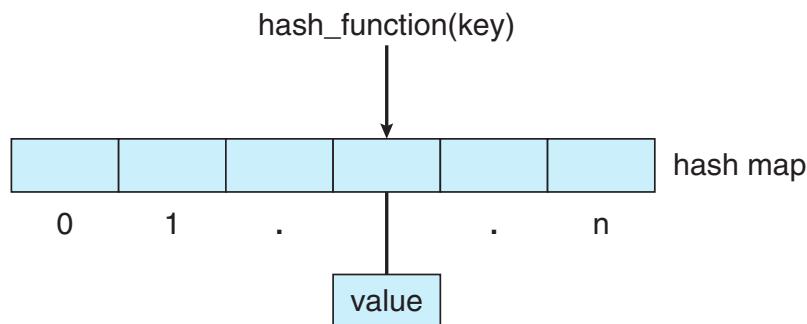
**Figure 1.20** Binary search tree.

location. We can accommodate this *hash collision* by having a linked list at the table location that contains all of the items with the same hash value. Of course, the more collisions there are, the less efficient the hash function is.

One use of a hash function is to implement a **hash map**, which associates (or *maps*) [key:value] pairs using a hash function. Once the mapping is established, we can apply the hash function to the key to obtain the value from the hash map (Figure 1.21). For example, suppose that a user name is mapped to a password. Password authentication then proceeds as follows: a user enters her user name and password. The hash function is applied to the user name, which is then used to retrieve the password. The retrieved password is then compared with the password entered by the user for authentication.

#### 1.9.4 Bitmaps

A **bitmap** is a string of  $n$  binary digits that can be used to represent the status of  $n$  items. For example, suppose we have several resources, and the availability of each resource is indicated by the value of a binary digit: 0 means that the resource is available, while 1 indicates that it is unavailable (or vice versa). The

**Figure 1.21** Hash map.

### LINUX KERNEL DATA STRUCTURES

The data structures used in the Linux kernel are available in the kernel source code. The *include* file <linux/list.h> provides details of the linked-list data structure used throughout the kernel. A queue in Linux is known as a `kfifo`, and its implementation can be found in the `kfifo.c` file in the `kernel` directory of the source code. Linux also provides a balanced binary search tree implementation using *red-black trees*. Details can be found in the include file <linux/rbtree.h>.

value of the  $i^{th}$  position in the bitmap is associated with the  $i^{th}$  resource. As an example, consider the bitmap shown below:

001011101

Resources 2, 4, 5, 6, and 8 are unavailable; resources 0, 1, 3, and 7 are available.

The power of bitmaps becomes apparent when we consider their space efficiency. If we were to use an eight-bit Boolean value instead of a single bit, the resulting data structure would be eight times larger. Thus, bitmaps are commonly used when there is a need to represent the availability of a large number of resources. Disk drives provide a nice illustration. A medium-sized disk drive might be divided into several thousand individual units, called **disk blocks**. A bitmap can be used to indicate the availability of each disk block.

In summary, data structures are pervasive in operating system implementations. Thus, we will see the structures discussed here, along with others, throughout this text as we explore kernel algorithms and their implementations.

## 1.10 Computing Environments

So far, we have briefly described several aspects of computer systems and the operating systems that manage them. We turn now to a discussion of how operating systems are used in a variety of computing environments.

### 1.10.1 Traditional Computing

As computing has matured, the lines separating many of the traditional computing environments have blurred. Consider the “typical office environment.” Just a few years ago, this environment consisted of PCs connected to a network, with servers providing file and print services. Remote access was awkward, and portability was achieved by use of laptop computers.

Today, web technologies and increasing WAN bandwidth are stretching the boundaries of traditional computing. Companies establish **portals**, which provide web accessibility to their internal servers. **Network computers** (or **thin clients**)—which are essentially terminals that understand web-based computing—are used in place of traditional workstations where more security or easier maintenance is desired. Mobile computers can synchronize with PCs to allow very portable use of company information. Mobile devices can also

connect to **wireless networks** and cellular data networks to use the company's web portal (as well as the myriad other web resources).

At home, most users once had a single computer with a slow modem connection to the office, the Internet, or both. Today, network-connection speeds once available only at great cost are relatively inexpensive in many places, giving home users more access to more data. These fast data connections are allowing home computers to serve up web pages and to run networks that include printers, client PCs, and servers. Many homes use **firewall** to protect their networks from security breaches. Firewalls limit the communications between devices on a network.

In the latter half of the 20th century, computing resources were relatively scarce. (Before that, they were nonexistent!) For a period of time, systems were either batch or interactive. Batch systems processed jobs in bulk, with predetermined input from files or other data sources. Interactive systems waited for input from users. To optimize the use of the computing resources, multiple users shared time on these systems. These time-sharing systems used a timer and scheduling algorithms to cycle processes rapidly through the CPU, giving each user a share of the resources.

Traditional time-sharing systems are rare today. The same scheduling technique is still in use on desktop computers, laptops, servers, and even mobile computers, but frequently all the processes are owned by the same user (or a single user and the operating system). User processes, and system processes that provide services to the user, are managed so that each frequently gets a slice of computer time. Consider the windows created while a user is working on a PC, for example, and the fact that they may be performing different tasks at the same time. Even a web browser can be composed of multiple processes, one for each website currently being visited, with time sharing applied to each web browser process.

### 1.10.2 Mobile Computing

**Mobile computing** refers to computing on handheld smartphones and tablet computers. These devices share the distinguishing physical features of being portable and lightweight. Historically, compared with desktop and laptop computers, mobile systems gave up screen size, memory capacity, and overall functionality in return for handheld mobile access to services such as e-mail and web browsing. Over the past few years, however, features on mobile devices have become so rich that the distinction in functionality between, say, a consumer laptop and a tablet computer may be difficult to discern. In fact, we might argue that the features of a contemporary mobile device allow it to provide functionality that is either unavailable or impractical on a desktop or laptop computer.

Today, mobile systems are used not only for e-mail and web browsing but also for playing music and video, reading digital books, taking photos, and recording and editing high-definition video. Accordingly, tremendous growth continues in the wide range of applications that run on such devices. Many developers are now designing applications that take advantage of the unique features of mobile devices, such as global positioning system (GPS) chips, accelerometers, and gyroscopes. An embedded GPS chip allows a mobile device to use satellites to determine its precise location on Earth. That functionality is

especially useful in designing applications that provide navigation—for example, telling users which way to walk or drive or perhaps directing them to nearby services, such as restaurants. An accelerometer allows a mobile device to detect its orientation with respect to the ground and to detect certain other forces, such as tilting and shaking. In several computer games that employ accelerometers, players interface with the system not by using a mouse or a keyboard but rather by tilting, rotating, and shaking the mobile device! Perhaps more a practical use of these features is found in *augmented-reality* applications, which overlay information on a display of the current environment. It is difficult to imagine how equivalent applications could be developed on traditional laptop or desktop computer systems.

To provide access to on-line services, mobile devices typically use either IEEE standard 802.11 wireless or cellular data networks. The memory capacity and processing speed of mobile devices, however, are more limited than those of PCs. Whereas a smartphone or tablet may have 256 GB in storage, it is not uncommon to find 8 TB in storage on a desktop computer. Similarly, because power consumption is such a concern, mobile devices often use processors that are smaller, are slower, and offer fewer processing cores than processors found on traditional desktop and laptop computers.

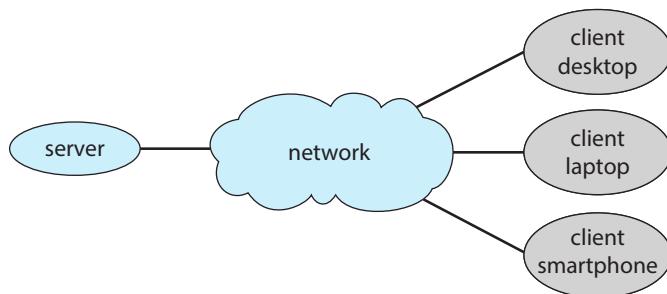
Two operating systems currently dominate mobile computing: **Apple iOS** and **Google Android**. iOS was designed to run on Apple iPhone and iPad mobile devices. Android powers smartphones and tablet computers available from many manufacturers. We examine these two mobile operating systems in further detail in Chapter 2.

### 1.10.3 Client-Server Computing

Contemporary network architecture features arrangements in which **server systems** satisfy requests generated by **client systems**. This form of specialized distributed system, called a **client-server** system, has the general structure depicted in Figure 1.22.

Server systems can be broadly categorized as compute servers and file servers:

- The **compute-server system** provides an interface to which a client can send a request to perform an action (for example, read data). In response, the server executes the action and sends the results to the client. A server



**Figure 1.22** General structure of a client–server system.

running a database that responds to client requests for data is an example of such a system.

- The **file-serve system** provides a file-system interface where clients can create, update, read, and delete files. An example of such a system is a web server that delivers files to clients running web browsers. The actual contents of the files can vary greatly, ranging from traditional web pages to rich multimedia content such as high-definition video.

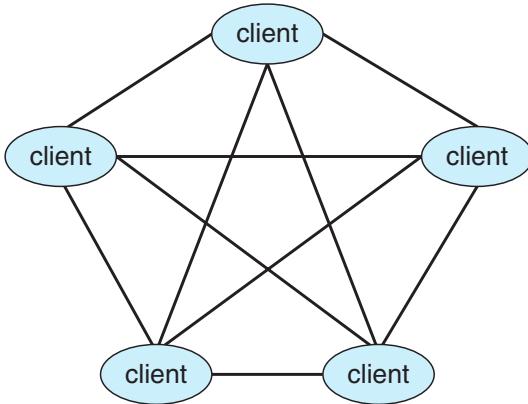
#### 1.10.4 Peer-to-Peer Computing

Another structure for a distributed system is the peer-to-peer (P2P) system model. In this model, clients and servers are not distinguished from one another. Instead, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service. Peer-to-peer systems offer an advantage over traditional client–server systems. In a client–server system, the server is a bottleneck; but in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.

To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network. Determining what services are available is accomplished in one of two general ways:

- When a node joins a network, it registers its service with a centralized lookup service on the network. Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service. The remainder of the communication takes place between the client and the service provider.
- An alternative scheme uses no centralized lookup service. Instead, a peer acting as a client must discover what node provides a desired service by broadcasting a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer making the request. To support this approach, a *discovery protocol* must be provided that allows peers to discover services provided by other peers in the network. Figure 1.23 illustrates such a scenario.

Peer-to-peer networks gained widespread popularity in the late 1990s with several file-sharing services, such as Napster and Gnutella, that enabled peers to exchange files with one another. The Napster system used an approach similar to the first type described above: a centralized server maintained an index of all files stored on peer nodes in the Napster network, and the actual exchange of files took place between the peer nodes. The Gnutella system used a technique similar to the second type: a client broadcast file requests to other nodes in the system, and nodes that could service the request responded directly to the client. Peer-to-peer networks can be used to exchange copyrighted materials (music, for example) anonymously, and there are laws governing the distribution of copyrighted material. Notably, Napster ran into legal trouble for copyright infringement, and its services were shut down in 2001. For this reason, the future of exchanging files remains uncertain.



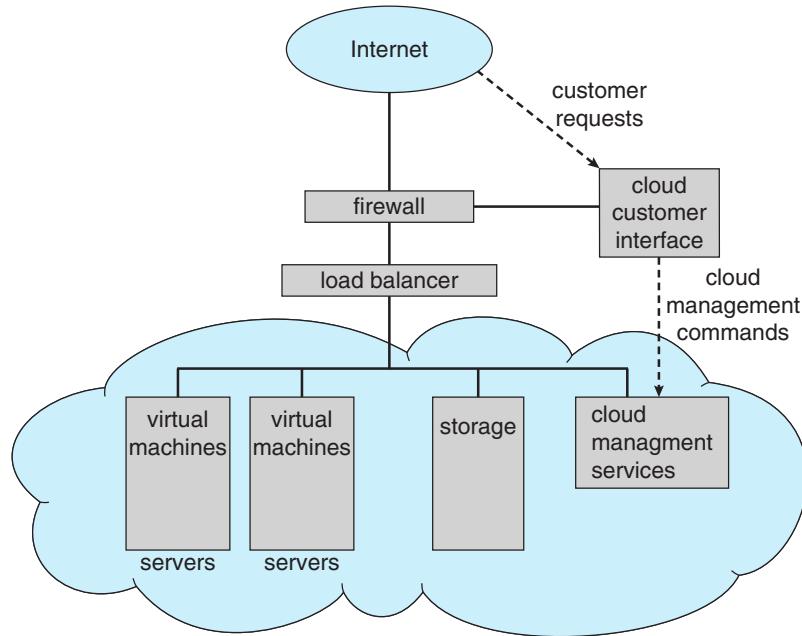
**Figure 1.23** Peer-to-peer system with no centralized service.

Skype is another example of peer-to-peer computing. It allows clients to make voice calls and video calls and to send text messages over the Internet using a technology known as **voice over IP (VoIP)**. Skype uses a hybrid peer-to-peer approach. It includes a centralized login server, but it also incorporates decentralized peers and allows two peers to communicate.

### 1.10.5 Cloud Computing

**Cloud computing** is a type of computing that delivers computing, storage, and even applications as a service across a network. In some ways, it's a logical extension of virtualization, because it uses virtualization as a base for its functionality. For example, the Amazon Elastic Compute Cloud (**ec2**) facility has thousands of servers, millions of virtual machines, and petabytes of storage available for use by anyone on the Internet. Users pay per month based on how much of those resources they use. There are actually many types of cloud computing, including the following:

- **Public cloud**—a cloud available via the Internet to anyone willing to pay for the services
- **Private cloud**—a cloud run by a company for that company's own use
- **Hybrid cloud**—a cloud that includes both public and private cloud components
- Software as a service (**SaaS**)—one or more applications (such as word processors or spreadsheets) available via the Internet
- Platform as a service (**PaaS**)—a software stack ready for application use via the Internet (for example, a database server)
- Infrastructure as a service (**IaaS**)—servers or storage available over the Internet (for example, storage available for making backup copies of production data)



**Figure 1.24** Cloud computing.

These cloud-computing types are not discrete, as a cloud computing environment may provide a combination of several types. For example, an organization may provide both SaaS and IaaS as publicly available services.

Certainly, there are traditional operating systems within many of the types of cloud infrastructure. Beyond those are the VMMs that manage the virtual machines in which the user processes run. At a higher level, the VMMs themselves are managed by cloud management tools, such as VMware vCloud Director and the open-source Eucalyptus toolset. These tools manage the resources within a given cloud and provide interfaces to the cloud components, making a good argument for considering them a new type of operating system.

Figure 1.24 illustrates a public cloud providing IaaS. Notice that both the cloud services and the cloud user interface are protected by a firewall.

### 1.10.6 Real-Time Embedded Systems

Embedded computers are the most prevalent form of computers in existence. These devices are found everywhere, from car engines and manufacturing robots to optical drives and microwave ovens. They tend to have very specific tasks. The systems they run on are usually primitive, and so the operating systems provide limited features. Usually, they have little or no user interface, preferring to spend their time monitoring and managing hardware devices, such as automobile engines and robotic arms.

These embedded systems vary considerably. Some are general-purpose computers, running standard operating systems—such as Linux—with special-purpose applications to implement the functionality. Others are hardware devices with a special-purpose embedded operating system providing just the functionality desired. Yet others are hardware devices

with application-specific integrated circuits (**ASICs**) that perform their tasks without an operating system.

The use of embedded systems continues to expand. The power of these devices, both as standalone units and as elements of networks and the web, is sure to increase as well. Even now, entire houses can be computerized, so that a central computer—either a general-purpose computer or an embedded system—can control heating and lighting, alarm systems, and even coffee makers. Web access can enable a home owner to tell the house to heat up before she arrives home. Someday, the refrigerator will be able to notify the grocery store when it notices the milk is gone.

Embedded systems almost always run **real-time operating systems**. A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application. Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are real-time systems. Some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems.

A real-time system has well-defined, fixed time constraints. Processing *must* be done within the defined constraints, or the system will fail. For instance, it would not do for a robot arm to be instructed to halt *after* it had smashed into the car it was building. A real-time system functions correctly only if it returns the correct result within its time constraints. Contrast this system with a traditional laptop system where it is desirable (but not mandatory) to respond quickly.

In Chapter 5, we consider the scheduling facility needed to implement real-time functionality in an operating system, and in Chapter 20 we describe the real-time components of Linux.

## 1.11 Free and Open-Source Operating Systems

The study of operating systems has been made easier by the availability of a vast number of free software and open-source releases. Both **free operating systems** and **open-source operating systems** are available in source-code format rather than as compiled binary code. Note, though, that free software and open-source software are two different ideas championed by different groups of people (see <http://gnu.org/philosophy/open-source-misses-the-point.html/> for a discussion on the topic). Free software (sometimes referred to as *free/libre software*) not only makes source code available but also is licensed to allow no-cost use, redistribution, and modification. Open-source software does not necessarily offer such licensing. Thus, although all free software is open source, some open-source software is not “free.” GNU/Linux is the most famous open-source operating system, with some distributions free and others open source only (<http://www.gnu.org/distros/>). Microsoft Windows is a well-known example of the opposite **closed-source** approach. Windows is **proprietary** software—Microsoft owns it, restricts its use, and carefully protects its source code. Apple’s macOS operating system comprises a hybrid

approach. It contains an open-source kernel named Darwin but includes proprietary, closed-source components as well.

Starting with the source code allows the programmer to produce binary code that can be executed on a system. Doing the opposite—[reverse engineering](#) the source code from the binaries—is quite a lot of work, and useful items such as comments are never recovered. Learning operating systems by examining the source code has other benefits as well. With the source code in hand, a student can modify the operating system and then compile and run the code to try out those changes, which is an excellent learning tool. This text includes projects that involve modifying operating-system source code, while also describing algorithms at a high level to be sure all important operating-system topics are covered. Throughout the text, we provide pointers to examples of open-source code for deeper study.

There are many benefits to open-source operating systems, including a community of interested (and usually unpaid) programmers who contribute to the code by helping to write it, debug it, analyze it, provide support, and suggest changes. Arguably, open-source code is more secure than closed-source code because many more eyes are viewing the code. Certainly, open-source code has bugs, but open-source advocates argue that bugs tend to be found and fixed faster owing to the number of people using and viewing the code. Companies that earn revenue from selling their programs often hesitate to open-source their code, but Red Hat and a myriad of other companies are doing just that and showing that commercial companies benefit, rather than suffer, when they open-source their code. Revenue can be generated through support contracts and the sale of hardware on which the software runs, for example.

### 1.11.1 History

In the early days of modern computing (that is, the 1950s), software generally came with source code. The original hackers (computer enthusiasts) at MIT's Tech Model Railroad Club left their programs in drawers for others to work on. "Homebrew" user groups exchanged code during their meetings. Company-specific user groups, such as Digital Equipment Corporation's DECUS, accepted contributions of source-code programs, collected them onto tapes, and distributed the tapes to interested members. In 1970, Digital's operating systems were distributed as source code with no restrictions or copyright notice.

Computer and software companies eventually sought to limit the use of their software to authorized computers and paying customers. Releasing only the binary files compiled from the source code, rather than the source code itself, helped them to achieve this goal, as well as protecting their code and their ideas from their competitors. Although the Homebrew user groups of the 1970s exchanged code during their meetings, the operating systems for hobbyist machines (such as CPM) were proprietary. By 1980, proprietary software was the usual case.

### 1.11.2 Free Operating Systems

To counter the move to limit software use and redistribution, Richard Stallman in 1984 started developing a free, UNIX-compatible operating system called GNU(which is a recursive acronym for “GNU’s Not Unix!”). To Stallman, “free” refers to freedom of use, not price. The free-software movement does not object

to trading a copy for an amount of money but holds that users are entitled to four certain freedoms: (1) to freely run the program, (2) to study and change the source code, and to give or sell copies either (3) with or (4) without changes. In 1985, Stallman published the GNU Manifesto, which argues that all software should be free. He also formed the **Free Software Foundation (FSF)** with the goal of encouraging the use and development of free software.

The FSF uses the copyrights on its programs to implement “copyleft,” a form of licensing invented by Stallman. Copylefting a work gives anyone that possesses a copy of the work the four essential freedoms that make the work free, with the condition that redistribution must preserve these freedoms. The **GNU General Public License (GPL)** is a common license under which free software is released. Fundamentally, the GPL requires that the source code be distributed with any binaries and that all copies (including modified versions) be released under the same GPL license. The Creative Commons “Attribution Sharealike” license is also a copyleft license; “sharealike” is another way of stating the idea of copyleft.

### 1.11.3 GNU/Linux

As an example of a free and open-source operating system, consider **GNU/Linux**. By 1991, the GNU operating system was nearly complete. The GNU Project had developed compilers, editors, utilities, libraries, and games — whatever parts it could not find elsewhere. However, the GNU kernel never became ready for prime time. In 1991, a student in Finland, Linus Torvalds, released a rudimentary UNIX-like kernel using the GNU compilers and tools and invited contributions worldwide. The advent of the Internet meant that anyone interested could download the source code, modify it, and submit changes to Torvalds. Releasing updates once a week allowed this so-called “Linux” operating system to grow rapidly, enhanced by several thousand programmers. In 1991, Linux was not free software, as its license permitted only noncommercial redistribution. In 1992, however, Torvalds rereleased Linux under the GPL, making it free software (and also, to use a term coined later, “open source”).

The resulting GNU/Linux operating system (with the kernel properly called Linux but the full operating system including GNU tools called GNU/Linux) has spawned hundreds of unique **distributions**, or custom builds, of the system. Major distributions include Red Hat, SUSE, Fedora, Debian, Slackware, and Ubuntu. Distributions vary in function, utility, installed applications, hardware support, user interface, and purpose. For example, Red Hat Enterprise Linux is geared to large commercial use. PCLinuxOS is a **live CD**—an operating system that can be booted and run from a CD-ROM without being installed on a system’s boot disk. A variant of PCLinuxOS—called PCLinuxOS Supergamer DVD—is a **live DVD** that includes graphics drivers and games. A gamer can run it on any compatible system simply by booting from the DVD. When the gamer is finished, a reboot of the system resets it to its installed operating system.

You can run Linux on a Windows (or other) system using the following simple, free approach:

1. Download the free Virtualbox VMM tool from

<https://www.virtualbox.org/>

and install it on your system.

2. Choose to install an operating system from scratch, based on an installation image like a CD, or choose pre-built operating-system images that can be installed and run more quickly from a site like

<http://virtualboxes.org/images/>

These images are preinstalled with operating systems and applications and include many flavors of GNU/Linux.

3. Boot the virtual machine within Virtualbox.

An alternative to using Virtualbox is to use the free program Qemu (<http://wiki.qemu.org/Download/>), which includes the `qemu-img` command for converting Virtualbox images to Qemu images to easily import them.

With this text, we provide a virtual machine image of GNU/Linux running the Ubuntu release. This image contains the GNU/Linux source code as well as tools for software development. We cover examples involving the GNU/Linux image throughout this text, as well as in a detailed case study in Chapter 20.

#### 1.11.4 BSD UNIX

**BSD UNIX** has a longer and more complicated history than Linux. It started in 1978 as a derivative of AT&T's UNIX. Releases from the University of California at Berkeley (UCB) came in source and binary form, but they were not open source because a license from AT&T was required. BSD UNIX's development was slowed by a lawsuit by AT&T, but eventually a fully functional, open-source version, 4.4BSD-lite, was released in 1994.

Just as with Linux, there are many distributions of BSD UNIX, including FreeBSD, NetBSD, OpenBSD, and DragonflyBSD. To explore the source code of FreeBSD, simply download the virtual machine image of the version of interest and boot it within Virtualbox, as described above for Linux. The source code comes with the distribution and is stored in `/usr/src/`. The kernel source code is in `/usr/src/sys`. For example, to examine the virtual memory implementation code in the FreeBSD kernel, see the files in `/usr/src/sys/vm`. Alternatively, you can simply view the source code online at <https://svnweb.freebsd.org/>.

As with many open-source projects, this source code is contained in and controlled by a **version control system**—in this case, “subversion” (<https://subversion.apache.org/source-code>). Version control systems allow a user to “pull” an entire source code tree to his computer and “push” any changes back into the repository for others to then pull. These systems also provide other features, including an entire history of each file and a conflict resolution feature in case the same file is changed concurrently. Another

version control system is [git](#), which is used for GNU/Linux, as well as other programs (<http://www.git-scm.com>).

Darwin, the core kernel component of macOS, is based on BSD UNIX and is open-sourced as well. That source code is available from <http://www.opensource.apple.com/>. Every macOS release has its open-source components posted at that site. The name of the package that contains the kernel begins with “xnu.” Apple also provides extensive developer tools, documentation, and support at <http://developer.apple.com>.

### THE STUDY OF OPERATING SYSTEMS

There has never been a more interesting time to study operating systems, and it has never been easier. The open-source movement has overtaken operating systems, causing many of them to be made available in both source and binary (executable) format. The list of operating systems available in both formats includes Linux, BSD UNIX, Solaris, and part of macOS. The availability of source code allows us to study operating systems from the inside out. Questions that we could once answer only by looking at documentation or the behavior of an operating system we can now answer by examining the code itself.

Operating systems that are no longer commercially viable have been open-sourced as well, enabling us to study how systems operated in a time of fewer CPU, memory, and storage resources. An extensive but incomplete list of open-source operating-system projects is available from [http://dmoz.org/Computers/Software/Operating\\_Systems/Open\\_Source/](http://dmoz.org/Computers/Software/Operating_Systems/Open_Source/).

In addition, the rise of virtualization as a mainstream (and frequently free) computer function makes it possible to run many operating systems on top of one core system. For example, VMware (<http://www.vmware.com>) provides a free “player” for Windows on which hundreds of free “virtual appliances” can run. Virtualbox (<http://www.virtualbox.com>) provides a free, open-source virtual machine manager on many operating systems. Using such tools, students can try out hundreds of operating systems without dedicated hardware.

In some cases, simulators of specific hardware are also available, allowing the operating system to run on “native” hardware, all within the confines of a modern computer and modern operating system. For example, a DECSYSTEM-20 simulator running on macOS can boot TOPS-20, load the source tapes, and modify and compile a new TOPS-20 kernel. An interested student can search the Internet to find the original papers that describe the operating system, as well as the original manuals.

The advent of open-source operating systems has also made it easier to make the move from student to operating-system developer. With some knowledge, some effort, and an Internet connection, a student can even create a new operating-system distribution. Not so many years ago, it was difficult or impossible to get access to source code. Now, such access is limited only by how much interest, time, and disk space a student has.

### 1.11.5 Solaris

**Solaris** is the commercial UNIX-based operating system of Sun Microsystems. Originally, Sun's **SunOS** operating system was based on BSD UNIX. Sun moved to AT&T's System V UNIX as its base in 1991. In 2005, Sun open-sourced most of the Solaris code as the OpenSolaris project. The purchase of Sun by Oracle in 2009, however, left the state of this project unclear.

Several groups interested in using OpenSolaris have expanded its features, and their working set is Project Illumos, which has expanded from the OpenSolaris base to include more features and to be the basis for several products. Illumos is available at <http://wiki.illumos.org>.

### 1.11.6 Open-Source Systems as Learning Tools

The free-software movement is driving legions of programmers to create thousands of open-source projects, including operating systems. Sites like <http://freshmeat.net/> and <http://distrowatch.com/> provide portals to many of these projects. As we stated earlier, open-source projects enable students to use source code as a learning tool. They can modify programs and test them, help find and fix bugs, and otherwise explore mature, full-featured operating systems, compilers, tools, user interfaces, and other types of programs. The availability of source code for historic projects, such as Multics, can help students to understand those projects and to build knowledge that will help in the implementation of new projects.

Another advantage of working with open-source operating systems is their diversity. GNU/Linux and BSD UNIX are both open-source operating systems, for instance, but each has its own goals, utility, licensing, and purpose. Sometimes, licenses are not mutually exclusive and cross-pollination occurs, allowing rapid improvements in operating-system projects. For example, several major components of OpenSolaris have been ported to BSD UNIX. The advantages of free software and open sourcing are likely to increase the number and quality of open-source projects, leading to an increase in the number of individuals and companies that use these projects.

## 1.12 Summary

- An operating system is software that manages the computer hardware, as well as providing an environment for application programs to run.
- Interrupts are a key way in which hardware interacts with the operating system. A hardware device triggers an interrupt by sending a signal to the CPU to alert the CPU that some event requires attention. The interrupt is managed by the interrupt handler.
- For a computer to do its job of executing programs, the programs must be in main memory, which is the only large storage area that the processor can access directly.
- The main memory is usually a volatile storage device that loses its contents when power is turned off or lost.

- Nonvolatile storage is an extension of main memory and is capable of holding large quantities of data permanently.
- The most common nonvolatile storage device is a hard disk, which can provide storage of both programs and data.
- The wide variety of storage systems in a computer system can be organized in a hierarchy according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases.
- Modern computer architectures are multiprocessor systems in which each CPU contains several computing cores.
- To best utilize the CPU, modern operating systems employ multiprogramming, which allows several jobs to be in memory at the same time, thus ensuring that the CPU always has a job to execute.
- Multitasking is an extension of multiprogramming wherein CPU scheduling algorithms rapidly switch between processes, providing users with a fast response time.
- To prevent user programs from interfering with the proper operation of the system, the system hardware has two modes: user mode and kernel mode.
- Various instructions are privileged and can be executed only in kernel mode. Examples include the instruction to switch to kernel mode, I/O control, timer management, and interrupt management.
- A process is the fundamental unit of work in an operating system. Process management includes creating and deleting processes and providing mechanisms for processes to communicate and synchronize with each other.
- An operating system manages memory by keeping track of what parts of memory are being used and by whom. It is also responsible for dynamically allocating and freeing memory space.
- Storage space is managed by the operating system; this includes providing file systems for representing files and directories and managing space on mass-storage devices.
- Operating systems provide mechanisms for protecting and securing the operating system and users. Protection measures control the access of processes or users to the resources made available by the computer system.
- Virtualization involves abstracting a computer's hardware into several different execution environments.
- Data structures that are used in an operating system include lists, stacks, queues, trees, and maps.
- Computing takes place in a variety of environments, including traditional computing, mobile computing, client–server systems, peer-to-peer systems, cloud computing, and real-time embedded systems.

- Free and open-source operating systems are available in source-code format. Free software is licensed to allow no-cost use, redistribution, and modification. GNU/Linux, FreeBSD, and Solaris are examples of popular open-source systems.

## Practice Exercises

- 1.1 What are the three main purposes of an operating system?
- 1.2 We have stressed the need for an operating system to make efficient use of the computing hardware. When is it appropriate for the operating system to forsake this principle and to “waste” resources? Why is such a system not really wasteful?
- 1.3 What is the main difficulty that a programmer must overcome in writing an operating system for a real-time environment?
- 1.4 Keeping in mind the various definitions of *operating system*, consider whether the operating system should include applications such as web browsers and mail programs. Argue both that it should and that it should not, and support your answers.
- 1.5 How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security)?
- 1.6 Which of the following instructions should be privileged?
  - a. Set value of timer.
  - b. Read the clock.
  - c. Clear memory.
  - d. Issue a trap instruction.
  - e. Turn off interrupts.
  - f. Modify entries in device-status table.
  - g. Switch from user to kernel mode.
  - h. Access I/O device.
- 1.7 Some early computers protected the operating system by placing it in a memory partition that could not be modified by either the user job or the operating system itself. Describe two difficulties that you think could arise with such a scheme.
- 1.8 Some CPUs provide for more than two modes of operation. What are two possible uses of these multiple modes?
- 1.9 Timers could be used to compute the current time. Provide a short description of how this could be accomplished.
- 1.10 Give two reasons why caches are useful. What problems do they solve? What problems do they cause? If a cache can be made as large as the

device for which it is caching (for instance, a cache as large as a disk), why not make it that large and eliminate the device?

- 1.11** Distinguish between the client–server and peer-to-peer models of distributed systems.

## Further Reading

Many general textbooks cover operating systems, including [Stallings (2017)] and [Tanenbaum (2014)]. [Hennessy and Patterson (2012)] provide coverage of I/O systems and buses and of system architecture in general. [Kurose and Ross (2017)] provides a general overview of computer networks.

[Russinovich et al. (2017)] give an overview of Microsoft Windows and covers considerable technical detail about the system internals and components. [McDougall and Mauro (2007)] cover the internals of the Solaris operating system. The macOS and iOS internals are discussed in [Levin (2013)]. [Levin (2015)] covers the internals of Android. [Love (2010)] provides an overview of the Linux operating system and great detail about data structures used in the Linux kernel. The Free Software Foundation has published its philosophy at <http://www.gnu.org/philosophy/free-software-for-freedom.html>.

## Bibliography

- [Hennessy and Patterson (2012)]** J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Fifth Edition, Morgan Kaufmann (2012).
- [Kurose and Ross (2017)]** J. Kurose and K. Ross, *Computer Networking—A Top-Down Approach*, Seventh Edition, Addison-Wesley (2017).
- [Levin (2013)]** J. Levin, *Mac OS X and iOS Internals to the Apple's Core*, Wiley (2013).
- [Levin (2015)]** J. Levin, *Android Internals—A Confectioner's Cookbook. Volume I* (2015).
- [Love (2010)]** R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).
- [McDougall and Mauro (2007)]** R. McDougall and J. Mauro, *Solaris Internals*, Second Edition, Prentice Hall (2007).
- [Russinovich et al. (2017)]** M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals—Part 1*, Seventh Edition, Microsoft Press (2017).
- [Stallings (2017)]** W. Stallings, *Operating Systems, Internals and Design Principles (9th Edition)* Ninth Edition, Prentice Hall (2017).
- [Tanenbaum (2014)]** A. S. Tanenbaum, *Modern Operating Systems*, Prentice Hall (2014).

## Chapter 1 Exercises

- 1.12 How do clustered systems differ from multiprocessor systems? What is required for two machines belonging to a cluster to cooperate to provide a highly available service?
- 1.13 Consider a computing cluster consisting of two nodes running a database. Describe two ways in which the cluster software can manage access to the data on the disk. Discuss the benefits and disadvantages of each.
- 1.14 What is the purpose of interrupts? How does an interrupt differ from a trap? Can traps be generated intentionally by a user program? If so, for what purpose?
- 1.15 Explain how the Linux kernel variables HZ and jiffies can be used to determine the number of seconds the system has been running since it was booted.
- 1.16 Direct memory access is used for high-speed I/O devices in order to avoid increasing the CPU's execution load.
  - a. How does the CPU interface with the device to coordinate the transfer?
  - b. How does the CPU know when the memory operations are complete?
  - c. The CPU is allowed to execute other programs while the DMA controller is transferring data. Does this process interfere with the execution of the user programs? If so, describe what forms of interference are caused.
- 1.17 Some computer systems do not provide a privileged mode of operation in hardware. Is it possible to construct a secure operating system for these computer systems? Give arguments both that it is and that it is not possible.
- 1.18 Many SMP systems have different levels of caches; one level is local to each processing core, and another level is shared among all processing cores. Why are caching systems designed this way?
- 1.19 Rank the following storage systems from slowest to fastest:
  - a. Hard-disk drives
  - b. Registers
  - c. Optical disk
  - d. Main memory
  - e. Nonvolatile memory
  - f. Magnetic tapes
  - g. Cache

**EX-2 Exercises**

- 1.20** Consider an SMP system similar to the one shown in Figure 1.8. Illustrate with an example how data residing in memory could in fact have a different value in each of the local caches.
- 1.21** Discuss, with examples, how the problem of maintaining coherence of cached data manifests itself in the following processing environments:
  - a. Single-processor systems
  - b. Multiprocessor systems
  - c. Distributed systems
- 1.22** Describe a mechanism for enforcing memory protection in order to prevent a program from modifying the memory associated with other programs.
- 1.23** Which network configuration—LAN or WAN—would best suit the following environments?
  - a. A campus student union
  - b. Several campus locations across a statewide university system
  - c. A neighborhood
- 1.24** Describe some of the challenges of designing operating systems for mobile devices compared with designing operating systems for traditional PCs.
- 1.25** What are some advantages of peer-to-peer systems over client–server systems?
- 1.26** Describe some distributed applications that would be appropriate for a peer-to-peer system.
- 1.27** Identify several advantages and several disadvantages of open-source operating systems. Identify the types of people who would find each aspect to be an advantage or a disadvantage.

# Operating- System Structures



An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. These goals form the basis for choices among various algorithms and strategies.

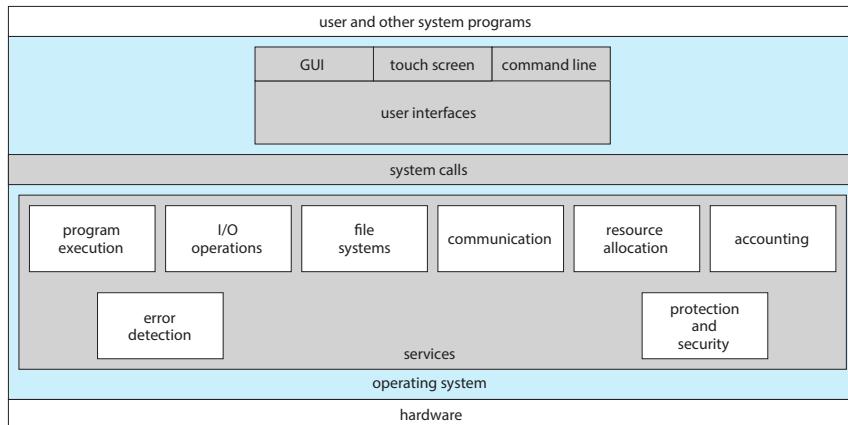
We can view an operating system from several vantage points. One view focuses on the services that the system provides; another, on the interface that it makes available to users and programmers; a third, on its components and their interconnections. In this chapter, we explore all three aspects of operating systems, showing the viewpoints of users, programmers, and operating system designers. We consider what services an operating system provides, how they are provided, how they are debugged, and what the various methodologies are for designing such systems. Finally, we describe how operating systems are created and how a computer starts its operating system.

## CHAPTER OBJECTIVES

- Identify services provided by an operating system.
- Illustrate how system calls are used to provide operating system services.
- Compare and contrast monolithic, layered, microkernel, modular, and hybrid strategies for designing operating systems.
- Illustrate the process for booting an operating system.
- Apply tools for monitoring operating system performance.
- Design and implement kernel modules for interacting with a Linux kernel.

## 2.1 Operating-System Services

An operating system provides an environment for the execution of programs. It makes certain services available to programs and to the users of those programs. The specific services provided, of course, differ from one operating



**Figure 2.1** A view of operating system services.

system to another, but we can identify common classes. Figure 2.1 shows one view of the various operating-system services and how they interrelate. Note that these services also make the programming task easier for the programmer.

One set of operating system services provides functions that are helpful to the user.

- **User interface.** Almost all operating systems have a **user interface (UI)**. This interface can take several forms. Most commonly, a **graphical user interface (GUI)** is used. Here, the interface is a window system with a mouse that serves as a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Mobile systems such as phones and tablets provide a **touch-screen interface**, enabling users to slide their fingers across the screen or press buttons on the screen to select choices. Another option is a **command-line interface (CLI)**, which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options). Some systems provide two or all three of these variations.
- **Program execution.** The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).
- **I/O operations.** A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as reading from a network interface or writing to a file system). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.
- **File-system manipulation.** The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some operating systems include permissions management to allow or deny access to files or directories based on file ownership. Many operating systems provide a variety of file systems, sometimes to allow

personal choice and sometimes to provide specific features or performance characteristics.

- **Communications.** There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a network. Communications may be implemented via **shared memory**, in which two or more processes read and write to a shared section of memory, or **message passing**, in which packets of information in predefined formats are moved between processes by the operating system.
- **Error detection.** The operating system needs to be detecting and correcting errors constantly. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow or an attempt to access an illegal memory location). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing. Sometimes, it has no choice but to halt the system. At other times, it might terminate an error-causing process or return an error code to a process for the process to detect and possibly correct.

Another set of operating-system functions exists not for helping the user but rather for ensuring the efficient operation of the system itself. Systems with multiple processes can gain efficiency by sharing the computer resources among the different processes.

- **Resource allocation.** When there are multiple processes running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code. For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that take into account the speed of the CPU, the process that must be executed, the number of processing cores on the CPU, and other factors. There may also be routines to allocate printers, USB storage drives, and other peripheral devices.
- **Logging.** We want to keep track of which programs use how much and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics. Usage statistics may be a valuable tool for system administrators who wish to reconfigure the system to improve computing services.
- **Protection and security.** The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate himself

or herself to the system, usually by means of a password, to gain access to system resources. It extends to defending external I/O devices, including network adapters, from invalid access attempts and recording all such connections for detection of break-ins. If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

## 2.2 User and Operating-System Interface

We mentioned earlier that there are several ways for users to interface with the operating system. Here, we discuss three fundamental approaches. One provides a command-line interface, or **command interpreter**, that allows users to directly enter commands to be performed by the operating system. The other two allow users to interface with the operating system via a graphical user interface, or GUI.

### 2.2.1 Command Interpreters

Most operating systems, including Linux, UNIX, and Windows, treat the command interpreter as a special program that is running when a process is initiated or when a user first logs on (on interactive systems). On systems with multiple command interpreters to choose from, the interpreters are known as **shells**. For example, on UNIX and Linux systems, a user may choose among several different shells, including the *C shell*, *Bourne-Again shell*, *Korn shell*, and others. Third-party shells and free user-written shells are also available. Most shells provide similar functionality, and a user's choice of which shell to use is generally based on personal preference. Figure 2.2 shows the Bourne-Again (or bash) shell command interpreter being used on macOS.

The main function of the command interpreter is to get and execute the next user-specified command. Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. The various shells available on UNIX systems operate in this way. These commands can be implemented in two general ways.

In one approach, the command interpreter itself contains the code to execute the command. For example, a command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call. In this case, the number of commands that can be given determines the size of the command interpreter, since each command requires its own implementing code.

An alternative approach—used by UNIX, among other operating systems—implements most commands through system programs. In this case, the command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memory and executed. Thus, the UNIX command to delete a file

```
rm file.txt
```

would search for a file called `rm`, load the file into memory, and execute it with the parameter `file.txt`. The logic associated with the `rm` command would be

```

1. root@r6181-d5-us01:~ (ssh)
Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbgs$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52,  3 users,  load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem      Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                      50G   19G   28G  41% /
tmpfs           127G  520K  127G  1% /dev/shm
/dev/sda1        477M   71M   381M  16% /boot
/dev/dssd0000    1.0T  480G  545G  47% /dssd_xfs
tcp://192.168.150.1:3334/orangetfs
                      12T  5.7T  6.4T  47% /mnt/orangetfs
/dev/gpfs-test   23T  1.1T  22T   5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root     97653 11.2  6.6 42665344 17520636 ?  S<Ll Jul13 166:23 /usr/lpp/mmfs/bin/mmfsd
root     69849  6.6  0.0      0  0 ?  S  Jul12 181:54 [vpthread-1-1]
root     69850  6.4  0.0      0  0 ?  S  Jul12 177:42 [vpthread-1-2]
root     3829  3.0  0.0      0  0 ?  S  Jun27 730:04 [rp_thread 7:0]
root     3826  3.0  0.0      0  0 ?  S  Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfsd
-rwx----- 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfsd
[root@r6181-d5-us01 ~]#

```

**Figure 2.2** The bash shell command interpreter in macOS.

defined completely by the code in the file `rm`. In this way, programmers can add new commands to the system easily by creating new files with the proper program logic. The command-interpreter program, which can be small, does not have to be changed for new commands to be added.

### 2.2.2 Graphical User Interface

A second strategy for interfacing with the operating system is through a user-friendly graphical user interface, or GUI. Here, rather than entering commands directly via a command-line interface, users employ a mouse-based window-and-menu system characterized by a **desktop** metaphor. The user moves the mouse to position its pointer on images, or **icons**, on the screen (the desktop) that represent programs, files, directories, and system functions. Depending on the mouse pointer’s location, clicking a button on the mouse can invoke a program, select a file or directory—known as a **folder**—or pull down a menu that contains commands.

Graphical user interfaces first appeared due in part to research taking place in the early 1970s at Xerox PARC research facility. The first GUI appeared on the Xerox Alto computer in 1973. However, graphical interfaces became more widespread with the advent of Apple Macintosh computers in the 1980s. The user interface for the Macintosh operating system has undergone various changes over the years, the most significant being the adoption of the *Aqua* interface that appeared with macOS. Microsoft’s first version of Windows—Version 1.0—was based on the addition of a GUI interface to the MS-DOS operating system. Later versions of Windows have made significant changes in the appearance of the GUI along with several enhancements in its functionality.

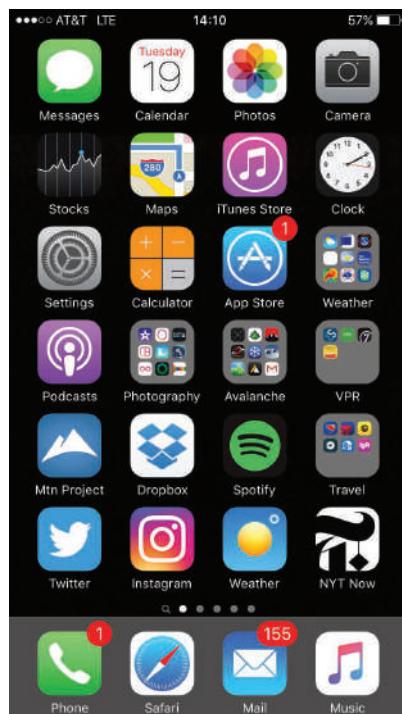
Traditionally, UNIX systems have been dominated by command-line interfaces. Various GUI interfaces are available, however, with significant development in GUI designs from various open-source projects, such as *K Desktop Environment* (or *KDE*) and the *GNOME* desktop by the GNU project. Both the KDE and GNOME desktops run on Linux and various UNIX systems and are available under open-source licenses, which means their source code is readily available for reading and for modification under specific license terms.

### 2.2.3 Touch-Screen Interface

Because either a command-line interface or a mouse-and-keyboard system is impractical for most mobile systems, smartphones and handheld tablet computers typically use a touch-screen interface. Here, users interact by making **gestures** on the touch screen—for example, pressing and swiping fingers across the screen. Although earlier smartphones included a physical keyboard, most smartphones and tablets now simulate a keyboard on the touch screen. Figure 2.3 illustrates the touch screen of the Apple iPhone. Both the iPad and the iPhone use the **Springboard** touch-screen interface.

### 2.2.4 Choice of Interface

The choice of whether to use a command-line or GUI interface is mostly one of personal preference. **System administrators** who manage computers and **power users** who have deep knowledge of a system frequently use the



**Figure 2.3** The iPhone touch screen.

command-line interface. For them, it is more efficient, giving them faster access to the activities they need to perform. Indeed, on some systems, only a subset of system functions is available via the GUI, leaving the less common tasks to those who are command-line knowledgeable. Further, command-line interfaces usually make repetitive tasks easier, in part because they have their own programmability. For example, if a frequent task requires a set of command-line steps, those steps can be recorded into a file, and that file can be run just like a program. The program is not compiled into executable code but rather is interpreted by the command-line interface. These **shell scripts** are very common on systems that are command-line oriented, such as UNIX and Linux.

In contrast, most Windows users are happy to use the Windows GUI environment and almost never use the shell interface. Recent versions of the Windows operating system provide both a standard GUI for desktop and traditional laptops and a touch screen for tablets. The various changes undergone by the Macintosh operating systems also provide a nice study in contrast. Historically, Mac OS has not provided a command-line interface, always requiring its users to interface with the operating system using its GUI. However, with the release of macOS (which is in part implemented using a UNIX kernel), the operating system now provides both an Aqua GUI and a command-line interface. Figure 2.4 is a screenshot of the macOS GUI.

Although there are apps that provide a command-line interface for iOS and Android mobile systems, they are rarely used. Instead, almost all users of mobile systems interact with their devices using the touch-screen interface.

The user interface can vary from system to system and even from user to user within a system; however, it typically is substantially removed from the actual system structure. The design of a useful and intuitive user interface is therefore not a direct function of the operating system. In this book, we concentrate on the fundamental problems of providing adequate service to

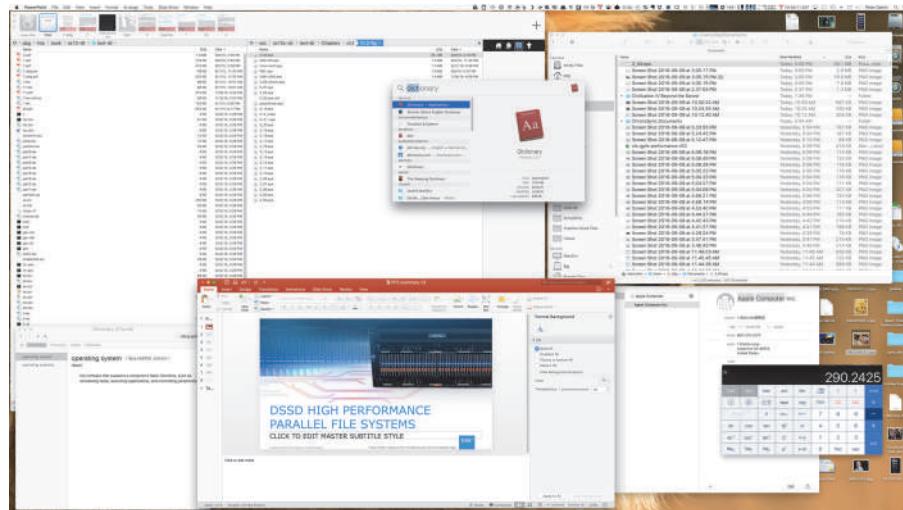


Figure 2.4 The macOS GUI.

user programs. From the point of view of the operating system, we do not distinguish between user programs and system programs.

## 2.3 System Calls

**System calls** provide an interface to the services made available by an operating system. These calls are generally available as functions written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions.

### 2.3.1 Example

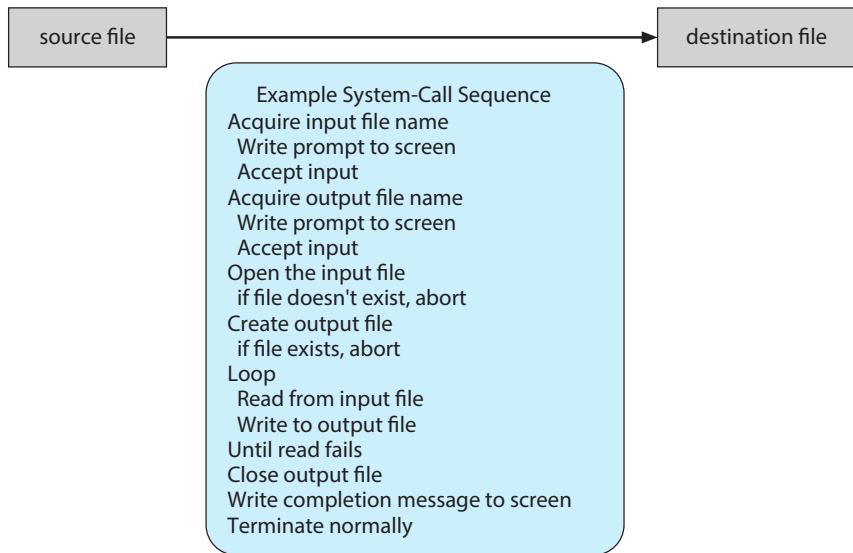
Before we discuss how an operating system makes system calls available, let's first use an example to illustrate how system calls are used: writing a simple program to read data from one file and copy them to another file. The first input that the program will need is the names of the two files: the input file and the output file. These names can be specified in many ways, depending on the operating-system design. One approach is to pass the names of the two files as part of the command—for example, the UNIX cp command:

```
cp in.txt out.txt
```

This command copies the input file `in.txt` to the output file `out.txt`. A second approach is for the program to ask the user for the names. In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the characters that define the two files. On mouse-based and icon-based systems, a menu of file names is usually displayed in a window. The user can then use the mouse to select the source name, and a window can be opened for the destination name to be specified. This sequence requires many I/O system calls.

Once the two file names have been obtained, the program must open the input file and create and open the output file. Each of these operations requires another system call. Possible error conditions for each system call must be handled. For example, when the program tries to open the input file, it may find that there is no file of that name or that the file is protected against access. In these cases, the program should output an error message (another sequence of system calls) and then terminate abnormally (another system call). If the input file exists, then we must create a new output file. We may find that there is already an output file with the same name. This situation may cause the program to abort (a system call), or we may delete the existing file (another system call) and create a new one (yet another system call). Another option, in an interactive system, is to ask the user (via a sequence of system calls to output the prompting message and to read the response from the terminal) whether to replace the existing file or to abort the program.

When both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Each read and write must return status information regarding various possible error conditions. On input, the program may find that the end of the file has been



**Figure 2.5** Example of how system calls are used.

reached or that there was a hardware failure in the read (such as a parity error). The write operation may encounter various errors, depending on the output device (for example, no more available disk space).

Finally, after the entire file is copied, the program may close both files (two system calls), write a message to the console or window (more system calls), and finally terminate normally (the final system call). This system-call sequence is shown in Figure 2.5.

### 2.3.2 Application Programming Interface

As you can see, even simple programs may make heavy use of the operating system. Frequently, systems execute thousands of system calls per second. Most programmers never see this level of detail, however. Typically, application developers design programs according to an **application programming interface (API)**. The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect. Three of the most common APIs available to application programmers are the Windows API for Windows systems, the POSIX API for POSIX-based systems (which include virtually all versions of UNIX, Linux, and macOS), and the Java API for programs that run on the Java virtual machine. A programmer accesses an API via a library of code provided by the operating system. In the case of UNIX and Linux for programs written in the C language, the library is called **libc**. Note that—unless specified—the system-call names used throughout this text are generic examples. Each operating system has its own name for each system call.

Behind the scenes, the functions that make up an API typically invoke the actual system calls on behalf of the application programmer. For example, the Windows function `CreateProcess()` (which, unsurprisingly, is used to create

**EXAMPLE OF STANDARD API**

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

<code>#include &lt;unistd.h&gt;</code>		
	<code>ssize_t</code>	<code>read(int fd, void *buf, size_t count)</code>
<code>return</code>	<code>function</code>	<code>parameters</code>
<code>value</code>	<code>name</code>	

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

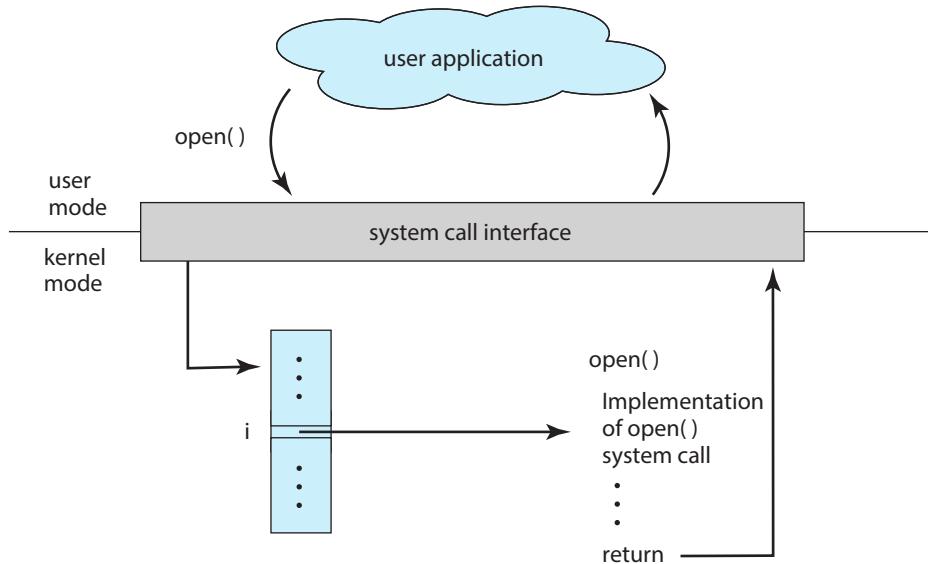
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns `-1`.

a new process) actually invokes the `NTCreateProcess()` system call in the Windows kernel.

Why would an application programmer prefer programming according to an API rather than invoking actual system calls? There are several reasons for doing so. One benefit concerns program portability. An application programmer designing a program using an API can expect her program to compile and run on any system that supports the same API (although, in reality, architectural differences often make this more difficult than it may appear). Furthermore, actual system calls can often be more detailed and difficult to work with than the API available to an application programmer. Nevertheless, there often exists a strong correlation between a function in the API and its associated system call within the kernel. In fact, many of the POSIX and Windows APIs are similar to the native system calls provided by the UNIX, Linux, and Windows operating systems.

Another important factor in handling system calls is the **run-time environment (RTE)**—the full suite of software needed to execute applications written in a given programming language, including its compilers or interpreters as well as other software, such as libraries and loaders. The RTE provides a



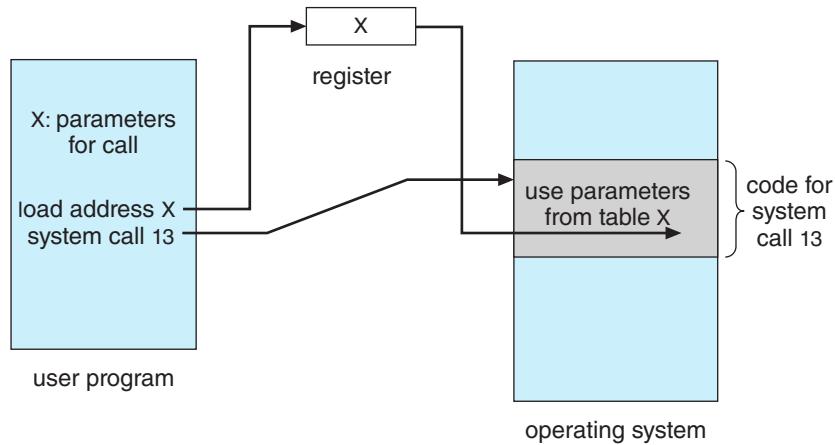
**Figure 2.6** The handling of a user application invoking the `open()` system call.

**system-call interface** that serves as the link to system calls made available by the operating system. The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system. Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers. The system-call interface then invokes the intended system call in the operating-system kernel and returns the status of the system call.

The caller need know nothing about how the system call is implemented or what it does during execution. Rather, the caller need only obey the API and understand what the operating system will do as a result of the execution of that system call. Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the RTE. The relationship among an API, the system-call interface, and the operating system is shown in Figure 2.6, which illustrates how the operating system handles a user application invoking the `open()` system call.

System calls occur in different ways, depending on the computer in use. Often, more information is required than simply the identity of the desired system call. The exact type and amount of information vary according to the particular operating system and call. For example, to get input, we may need to specify the file or device to use as the source, as well as the address and length of the memory buffer into which the input should be read. Of course, the device or file and length may be implicit in the call.

Three general methods are used to pass parameters to the operating system. The simplest approach is to pass the parameters in registers. In some cases, however, there may be more parameters than registers. In these cases, the parameters are generally stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register (Figure 2.7). Linux uses a combination of these approaches. If there are five or fewer parameters,



**Figure 2.7** Passing of parameters as a table.

registers are used. If there are more than five parameters, the block method is used. Parameters also can be placed, or **pushed**, onto a **stack** by the program and **popped** off the stack by the operating system. Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.

### 2.3.3 Types of System Calls

System calls can be grouped roughly into six major categories: **process control**, **file management**, **device management**, **information maintenance**, **communications**, and **protection**. Below, we briefly discuss the types of system calls that may be provided by an operating system. Most of these system calls support, or are supported by, concepts and functions that are discussed in later chapters. Figure 2.8 summarizes the types of system calls normally provided by an operating system. As mentioned, in this text, we normally refer to the system calls by generic names. Throughout the text, however, we provide examples of the actual counterparts to the system calls for UNIX, Linux, and Windows systems.

#### 2.3.3.1 Process Control

A running program needs to be able to halt its execution either normally (`end()`) or abnormally (`abort()`). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated. The dump is written to a special log file on disk and may be examined by a **debugger**—a system program designed to aid the programmer in finding and correcting errors, or **bugs**—to determine the cause of the problem. Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. The command interpreter then reads the next command. In an interactive system, the command interpreter simply continues with the next command; it is assumed that the user will issue an appropriate command to respond to

- Process control
  - create process, terminate process
  - load, execute
  - get process attributes, set process attributes
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices
- Protection
  - get file permissions
  - set file permissions

---

Figure 2.8 Types of system calls.

**EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS**

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	<b>Windows</b>	<b>Unix</b>
<b>Process control</b>	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
<b>File management</b>	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
<b>Device management</b>	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
<b>Information maintenance</b>	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
<b>Communications</b>	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
<b>Protection</b>	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

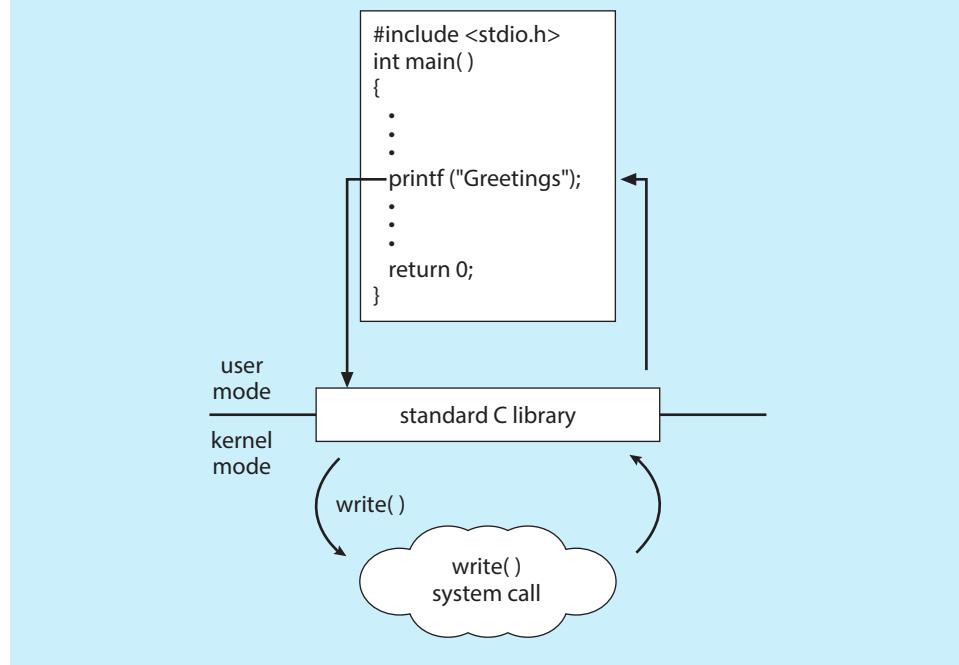
any error. In a GUI system, a pop-up window might alert the user to the error and ask for guidance. Some systems may allow for special recovery actions in case an error occurs. If the program discovers an error in its input and wants to terminate abnormally, it may also want to define an error level. More severe errors can be indicated by a higher-level error parameter. It is then possible to combine normal and abnormal termination by defining a normal termination as an error at level 0. The command interpreter or a following program can use this error level to determine the next action automatically.

A process executing one program may want to load() and execute() another program. This feature allows the command interpreter to execute a program as directed by, for example, a user command or the click of a mouse. An interesting question is where to return control when the loaded program terminates. This question is related to whether the existing program is lost, saved, or allowed to continue execution concurrently with the new program.

If control returns to the existing program when the new program terminates, we must save the memory image of the existing program; thus, we have

### THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:

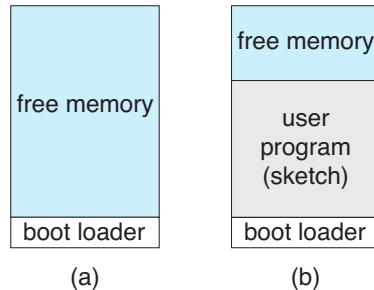


effectively created a mechanism for one program to call another program. If both programs continue concurrently, we have created a new process to be multiprogrammed. Often, there is a system call specifically for this purpose (`create_process()`).

If we create a new process, or perhaps even a set of processes, we should be able to control its execution. This control requires the ability to determine and reset the attributes of a process, including the process's priority, its maximum allowable execution time, and so on (`get_process_attributes()` and `set_process_attributes()`). We may also want to terminate a process that we created (`terminate_process()`) if we find that it is incorrect or is no longer needed.

Having created new processes, we may need to wait for them to finish their execution. We may want to wait for a certain amount of time to pass (`wait_time()`). More probably, we will want to wait for a specific event to occur (`wait_event()`). The processes should then signal when that event has occurred (`signal_event()`).

Quite often, two or more processes may share data. To ensure the integrity of the data being shared, operating systems often provide system calls allowing

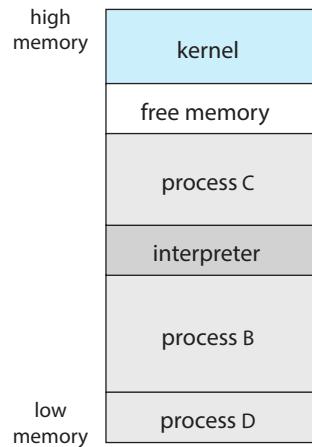


**Figure 2.9** Arduino execution. (a) At system startup. (b) Running a sketch.

a process to **lock** shared data. Then, no other process can access the data until the lock is released. Typically, such system calls include `acquire_lock()` and `release_lock()`. System calls of these types, dealing with the coordination of concurrent processes, are discussed in great detail in Chapter 6 and Chapter 7.

There are so many facets of and variations in process control that we next use two examples—one involving a single-tasking system and the other a multitasking system—to clarify these concepts. The Arduino is a simple hardware platform consisting of a microcontroller along with input sensors that respond to a variety of events, such as changes to light, temperature, and barometric pressure, to just name a few. To write a program for the Arduino, we first write the program on a PC and then upload the compiled program (known as a **sketch**) from the PC to the Arduino’s flash memory via a USB connection. The standard Arduino platform does not provide an operating system; instead, a small piece of software known as a **boot loader** loads the sketch into a specific region in the Arduino’s memory (Figure 2.9). Once the sketch has been loaded, it begins running, waiting for the events that it is programmed to respond to. For example, if the Arduino’s temperature sensor detects that the temperature has exceeded a certain threshold, the sketch may have the Arduino start the motor for a fan. An Arduino is considered a single-tasking system, as only one sketch can be present in memory at a time; if another sketch is loaded, it replaces the existing sketch. Furthermore, the Arduino provides no user interface beyond hardware input sensors.

FreeBSD (derived from Berkeley UNIX) is an example of a multitasking system. When a user logs on to the system, the shell of the user’s choice is run, awaiting commands and running programs the user requests. However, since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed (Figure 2.10). To start a new process, the shell executes a `fork()` system call. Then, the selected program is loaded into memory via an `exec()` system call, and the program is executed. Depending on how the command was issued, the shell then either waits for the process to finish or runs the process “in the background.” In the latter case, the shell immediately waits for another command to be entered. When a process is running in the background, it cannot receive input directly from the keyboard, because the shell is using this resource. I/O is therefore done through files or through a GUI interface. Meanwhile, the user is free to ask the shell to run other programs, to monitor the progress of the running process, to change that program’s priority, and so on. When the process is done, it executes an `exit()`



**Figure 2.10** FreeBSD running multiple programs.

system call to terminate, returning to the invoking process a status code of 0 or a nonzero error code. This status or error code is then available to the shell or other programs. Processes are discussed in Chapter 3 with a program example using the `fork()` and `exec()` system calls.

### 2.3.3.2 File Management

The file system is discussed in more detail in Chapter 13 through Chapter 15. Here, we identify several common system calls dealing with files.

We first need to be able to `create()` and `delete()` files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to `open()` it and to use it. We may also `read()`, `write()`, or `reposition()` (rewind or skip to the end of the file, for example). Finally, we need to `close()` the file, indicating that we are no longer using it.

We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system. In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to set them if necessary. File attributes include the file name, file type, protection codes, accounting information, and so on. At least two system calls, `get_file_attributes()` and `set_file_attributes()`, are required for this function. Some operating systems provide many more calls, such as calls for file `move()` and `copy()`. Others might provide an API that performs those operations using code and other system calls, and others might provide system programs to perform the tasks. If the system programs are callable by other programs, then each can be considered an API by other system programs.

### 2.3.3.3 Device Management

A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available.

The various resources controlled by the operating system can be thought of as devices. Some of these devices are physical devices (for example, disk drives), while others can be thought of as abstract or virtual devices (for example, files). A system with multiple users may require us to first `request()` a device, to ensure exclusive use of it. After we are finished with the device, we `release()` it. These functions are similar to the `open()` and `close()` system calls for files. Other operating systems allow unmanaged access to devices. The hazard then is the potential for device contention and perhaps deadlock, which are described in Chapter 8.

Once the device has been requested (and allocated to us), we can `read()`, `write()`, and (possibly) `reposition()` the device, just as we can with files. In fact, the similarity between I/O devices and files is so great that many operating systems, including UNIX, merge the two into a combined file–device structure. In this case, a set of system calls is used on both files and devices. Sometimes, I/O devices are identified by special file names, directory placement, or file attributes.

The user interface can also make files and devices appear to be similar, even though the underlying system calls are dissimilar. This is another example of the many design decisions that go into building an operating system and user interface.

#### 2.3.3.4 Information Maintenance

Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most systems have a system call to return the current `time()` and `date()`. Other system calls may return information about the system, such as the version number of the operating system, the amount of free memory or disk space, and so on.

Another set of system calls is helpful in debugging a program. Many systems provide system calls to `dump()` memory. This provision is useful for debugging. The program `strace`, which is available on Linux systems, lists each system call as it is executed. Even microprocessors provide a CPU mode, known as **single step**, in which a trap is executed by the CPU after every instruction. The trap is usually caught by a debugger.

Many operating systems provide a time profile of a program to indicate the amount of time that the program executes at a particular location or set of locations. A time profile requires either a tracing facility or regular timer interrupts. At every occurrence of the timer interrupt, the value of the program counter is recorded. With sufficiently frequent timer interrupts, a statistical picture of the time spent on various parts of the program can be obtained.

In addition, the operating system keeps information about all its processes, and system calls are used to access this information. Generally, calls are also used to get and set the process information (`get_process_attributes()` and `set_process_attributes()`). In Section 3.1.3, we discuss what information is normally kept.

#### 2.3.3.5 Communication

There are two common models of interprocess communication: the message-passing model and the shared-memory model. In the **message-passing model**, the communicating processes exchange messages with one another to trans-

fer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened. The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network. Each computer in a network has a **host name** by which it is commonly known. A host also has a network identifier, such as an IP address. Similarly, each process has a **process name**, and this name is translated into an identifier by which the operating system can refer to the process. The `get_hostid()` and `get_processid()` system calls do this translation. The identifiers are then passed to the general-purpose `open()` and `close()` calls provided by the file system or to specific `open_connection()` and `close_connection()` system calls, depending on the system's model of communication. The recipient process usually must give its permission for communication to take place with an `accept_connection()` call. Most processes that will be receiving connections are special-purpose **daemons**, which are system programs provided for that purpose. They execute a `wait_for_connection()` call and are awakened when a connection is made. The source of the communication, known as the **client**, and the receiving daemon, known as a **server**, then exchange messages by using `read_message()` and `write_message()` system calls. The `close_connection()` call terminates the communication.

In the **shared-memory model**, processes use `shared_memory_create()` and `shared_memory_attach()` system calls to create and gain access to regions of memory owned by other processes. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data is determined by the processes and is not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously. Such mechanisms are discussed in Chapter 6. In Chapter 4, we look at a variation of the process scheme—threads—in which some memory is shared by default.

Both of the models just discussed are common in operating systems, and most systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. It is also easier to implement than is shared memory for intercomputer communication. Shared memory allows maximum speed and convenience of communication, since it can be done at memory transfer speeds when it takes place within a computer. Problems exist, however, in the areas of protection and synchronization between the processes sharing memory.

#### 2.3.3.6 Protection

Protection provides a mechanism for controlling access to the resources provided by a computer system. Historically, protection was a concern only on multiprogrammed computer systems with several users. However, with the advent of networking and the Internet, all computer systems, from servers to mobile handheld devices, must be concerned with protection.

Typically, system calls providing protection include `set_permission()` and `get_permission()`, which manipulate the permission settings of

resources such as files and disks. The `allow_user()` and `deny_user()` system calls specify whether particular users can—or cannot—be allowed access to certain resources. We cover protection in Chapter 17 and the much larger issue of security—which involves using protection against external threats—in Chapter 16.

## 2.4 System Services

Another aspect of a modern system is its collection of system services. Recall Figure 1.1, which depicted the logical computer hierarchy. At the lowest level is hardware. Next is the operating system, then the system services, and finally the application programs. **System services**, also known as **system utilities**, provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls. Others are considerably more complex. They can be divided into these categories:

- **File management.** These programs create, delete, copy, rename, print, list, and generally access and manipulate files and directories.
- **Status information.** Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a **registry**, which is used to store and retrieve configuration information.
- **File modification.** Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.
- **Programming-language support.** Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and Python) are often provided with the operating system or available as a separate download.
- **Program loading and execution.** Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.
- **Communications.** These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse web pages, to send e-mail messages, to log in remotely, or to transfer files from one machine to another.
- **Background services.** All general-purpose systems have methods for launching certain system-program processes at boot time. Some of these processes terminate after completing their tasks, while others continue to

run until the system is halted. Constantly running system-program processes are known as **services**, **subsystems**, or daemons. One example is the network daemon discussed in Section 2.3.3.5. In that example, a system needed a service to listen for network connections in order to connect those requests to the correct processes. Other examples include process schedulers that start processes according to a specified schedule, system error monitoring services, and print servers. Typical systems have dozens of daemons. In addition, operating systems that run important activities in user context rather than in kernel context may use daemons to run these activities.

Along with system programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations. Such **application programs** include web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games.

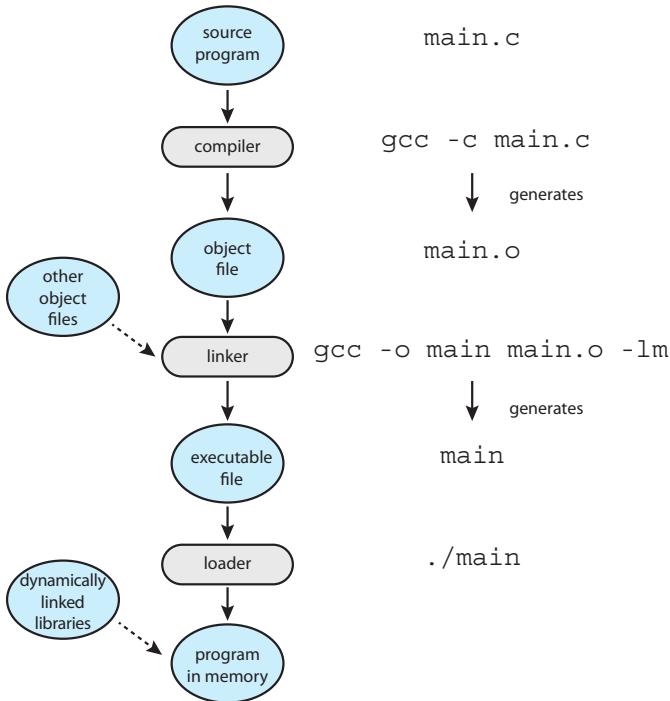
The view of the operating system seen by most users is defined by the application and system programs, rather than by the actual system calls. Consider a user's PC. When a user's computer is running the macOS operating system, the user might see the GUI, featuring a mouse-and-windows interface. Alternatively, or even in one of the windows, the user might have a command-line UNIX shell. Both use the same set of system calls, but the system calls look different and act in different ways. Further confusing the user view, consider the user dual-booting from macOS into Windows. Now the same user on the same hardware has two entirely different interfaces and two sets of applications using the same physical resources. On the same hardware, then, a user can be exposed to multiple user interfaces sequentially or concurrently.

## 2.5 Linkers and Loaders

Usually, a program resides on disk as a binary executable file—for example, `a.out` or `prog.exe`. To run on a CPU, the program must be brought into memory and placed in the context of a process. In this section, we describe the steps in this procedure, from compiling a program to placing it in memory, where it becomes eligible to run on an available CPU core. The steps are highlighted in Figure 2.11.

Source files are compiled into object files that are designed to be loaded into any physical memory location, a format known as an **relocatable object file**. Next, the **linker** combines these relocatable object files into a single binary **executable** file. During the linking phase, other object files or libraries may be included as well, such as the standard C or math library (specified with the flag `-lm`).

A **loader** is used to load the binary executable file into memory, where it is eligible to run on a CPU core. An activity associated with linking and loading is **relocation**, which assigns final addresses to the program parts and adjusts code and data in the program to match those addresses so that, for example, the code can call library functions and access its variables as it executes. In Figure 2.11, we see that to run the loader, all that is necessary is to enter the name of the executable file on the command line. When a program name is entered on the



**Figure 2.11** The role of the linker and loader.

command line on UNIX systems—for example, `./main`—the shell first creates a new process to run the program using the `fork()` system call. The shell then invokes the loader with the `exec()` system call, passing `exec()` the name of the executable file. The loader then loads the specified program into memory using the address space of the newly created process. (When a GUI interface is used, double-clicking on the icon associated with the executable file invokes the loader using a similar mechanism.)

The process described thus far assumes that all libraries are linked into the executable file and loaded into memory. In reality, most systems allow a program to dynamically link libraries as the program is loaded. Windows, for instance, supports dynamically linked libraries (**DLLs**). The benefit of this approach is that it avoids linking and loading libraries that may end up not being used into an executable file. Instead, the library is conditionally linked and is loaded if it is required during program run time. For example, in Figure 2.11, the math library is not linked into the executable file `main`. Rather, the linker inserts relocation information that allows it to be dynamically linked and loaded as the program is loaded. We shall see in Chapter 9 that it is possible for multiple processes to share dynamically linked libraries, resulting in a significant savings in memory use.

Object files and executable files typically have standard formats that include the compiled machine code and a symbol table containing metadata about functions and variables that are referenced in the program. For UNIX and Linux systems, this standard format is known as ELF (for **Executable and Linkable Format**). There are separate ELF formats for relocatable and

### ELF FORMAT

Linux provides various commands to identify and evaluate ELF files. For example, the `file` command determines a file type. If `main.o` is an object file, and `main` is an executable file, the command

```
file main.o
```

will report that `main.o` is an ELF relocatable file, while the command

```
file main
```

will report that `main` is an ELF executable. ELF files are divided into a number of sections and can be evaluated using the `readelf` command.

executable files. One piece of information in the ELF file for executable files is the program's *entry point*, which contains the address of the first instruction to be executed when the program runs. Windows systems use the **Portable Executable** (PE) format, and macOS uses the **Mach-O** format.

## 2.6 Why Applications Are Operating-System Specific

Fundamentally, applications compiled on one operating system are not executable on other operating systems. If they were, the world would be a better place, and our choice of what operating system to use would depend on utility and features rather than which applications were available.

Based on our earlier discussion, we can now see part of the problem—each operating system provides a unique set of system calls. System calls are part of the set of services provided by operating systems for use by applications. Even if system calls were somehow uniform, other barriers would make it difficult for us to execute application programs on different operating systems. But if you have used multiple operating systems, you may have used some of the same applications on them. How is that possible?

An application can be made available to run on multiple operating systems in one of three ways:

1. The application can be written in an interpreted language (such as Python or Ruby) that has an interpreter available for multiple operating systems. The interpreter reads each line of the source program, executes equivalent instructions on the native instruction set, and calls native operating system calls. Performance suffers relative to that for native applications, and the interpreter provides only a subset of each operating system's features, possibly limiting the feature sets of the associated applications.
2. The application can be written in a language that includes a virtual machine containing the running application. The virtual machine is part of the language's full RTE. One example of this method is Java. Java has an RTE that includes a loader, byte-code verifier, and other components that load the Java application into the Java virtual machine. This RTE has been

**ported**, or developed, for many operating systems, from mainframes to smartphones, and in theory any Java app can run within the RTE wherever it is available. Systems of this kind have disadvantages similar to those of interpreters, discussed above.

3. The application developer can use a standard language or API in which the compiler generates binaries in a machine- and operating-system-specific language. The application must be ported to each operating system on which it will run. This porting can be quite time consuming and must be done for each new version of the application, with subsequent testing and debugging. Perhaps the best-known example is the POSIX API and its set of standards for maintaining source-code compatibility between different variants of UNIX-like operating systems.

In theory, these three approaches seemingly provide simple solutions for developing applications that can run across different operating systems. However, the general lack of application mobility has several causes, all of which still make developing cross-platform applications a challenging task. At the application level, the libraries provided with the operating system contain APIs to provide features like GUI interfaces, and an application designed to call one set of APIs (say, those available from iOS on the Apple iPhone) will not work on an operating system that does not provide those APIs (such as Android). Other challenges exist at lower levels in the system, including the following.

- Each operating system has a binary format for applications that dictates the layout of the header, instructions, and variables. Those components need to be at certain locations in specified structures within an executable file so the operating system can open the file and load the application for proper execution.
- CPUs have varying instruction sets, and only applications containing the appropriate instructions can execute correctly.
- Operating systems provide system calls that allow applications to request various activities, such as creating files and opening network connections. Those system calls vary among operating systems in many respects, including the specific operands and operand ordering used, how an application invokes the system calls, their numbering and number, their meanings, and their return of results.

There are some approaches that have helped address, though not completely solve, these architectural differences. For example, Linux—and almost every UNIX system—has adopted the ELF format for binary executable files. Although ELF provides a common standard across Linux and UNIX systems, the ELF format is not tied to any specific computer architecture, so it does not guarantee that an executable file will run across different hardware platforms.

APIs, as mentioned above, specify certain functions at the application level. At the architecture level, an **application binary interface** (ABI) is used to define how different components of binary code can interface for a given operating system on a given architecture. An ABI specifies low-level details, including address width, methods of passing parameters to system calls, the organization

of the run-time stack, the binary format of system libraries, and the size of data types, just to name a few. Typically, an ABI is specified for a given architecture (for example, there is an ABI for the ARMv8 processor). Thus, an ABI is the architecture-level equivalent of an API. If a binary executable file has been compiled and linked according to a particular ABI, it should be able to run on different systems that support that ABI. However, because a particular ABI is defined for a certain operating system running on a given architecture, ABIs do little to provide cross-platform compatibility.

In sum, all of these differences mean that unless an interpreter, RTE, or binary executable file is written for and compiled on a specific operating system on a specific CPU type (such as Intel x86 or ARMv8), the application will fail to run. Imagine the amount of work that is required for a program such as the Firefox browser to run on Windows, macOS, various Linux releases, iOS, and Android, sometimes on various CPU architectures.

## 2.7 Operating-System Design and Implementation

In this section, we discuss problems we face in designing and implementing an operating system. There are, of course, no complete solutions to such problems, but there are approaches that have proved successful.

### 2.7.1 Design Goals

The first problem in designing a system is to define goals and specifications. At the highest level, the design of the system will be affected by the choice of hardware and the type of system: traditional desktop/laptop, mobile, distributed, or real time.

Beyond this highest design level, the requirements may be much harder to specify. The requirements can, however, be divided into two basic groups: **user goals** and **system goals**.

Users want certain obvious properties in a system. The system should be convenient to use, easy to learn and to use, reliable, safe, and fast. Of course, these specifications are not particularly useful in the system design, since there is no general agreement on how to achieve them.

A similar set of requirements can be defined by the developers who must design, create, maintain, and operate the system. The system should be easy to design, implement, and maintain; and it should be flexible, reliable, error free, and efficient. Again, these requirements are vague and may be interpreted in various ways.

There is, in short, no unique solution to the problem of defining the requirements for an operating system. The wide range of systems in existence shows that different requirements can result in a large variety of solutions for different environments. For example, the requirements for Wind River VxWorks, a real-time operating system for embedded systems, must have been substantially different from those for Windows Server, a large multiaccess operating system designed for enterprise applications.

Specifying and designing an operating system is a highly creative task. Although no textbook can tell you how to do it, general principles have been

developed in the field of **software engineering**, and we turn now to a discussion of some of these principles.

### 2.7.2 Mechanisms and Policies

One important principle is the separation of **policy** from **mechanism**. Mechanisms determine *how* to do something; policies determine *what* will be done. For example, the timer construct (see Section 1.4.3) is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision.

The separation of policy and mechanism is important for flexibility. Policies are likely to change across places or over time. In the worst case, each change in policy would require a change in the underlying mechanism. A general mechanism flexible enough to work across a range of policies is preferable. A change in policy would then require redefinition of only certain parameters of the system. For instance, consider a mechanism for giving priority to certain types of programs over others. If the mechanism is properly separated from policy, it can be used either to support a policy decision that I/O-intensive programs should have priority over CPU-intensive ones or to support the opposite policy.

Microkernel-based operating systems (discussed in Section 2.8.3) take the separation of mechanism and policy to one extreme by implementing a basic set of primitive building blocks. These blocks are almost policy free, allowing more advanced mechanisms and policies to be added via user-created kernel modules or user programs themselves. In contrast, consider Windows, an enormously popular commercial operating system available for over three decades. Microsoft has closely encoded both mechanism and policy into the system to enforce a global look and feel across all devices that run the Windows operating system. All applications have similar interfaces, because the interface itself is built into the kernel and system libraries. Apple has adopted a similar strategy with its macOS and iOS operating systems.

We can make a similar comparison between commercial and open-source operating systems. For instance, contrast Windows, discussed above, with Linux, an open-source operating system that runs on a wide range of computing devices and has been available for over 25 years. The “standard” Linux kernel has a specific CPU scheduling algorithm (covered in Section 5.7.1), which is a mechanism that supports a certain policy. However, anyone is free to modify or replace the scheduler to support a different policy.

Policy decisions are important for all resource allocation. Whenever it is necessary to decide whether or not to allocate a resource, a policy decision must be made. Whenever the question is *how* rather than *what*, it is a mechanism that must be determined.

### 2.7.3 Implementation

Once an operating system is designed, it must be implemented. Because operating systems are collections of many programs, written by many people over a long period of time, it is difficult to make general statements about how they are implemented.

Early operating systems were written in assembly language. Now, most are written in higher-level languages such as C or C++, with small amounts

of the system written in assembly language. In fact, more than one higher-level language is often used. The lowest levels of the kernel might be written in assembly language and C. Higher-level routines might be written in C and C++, and system libraries might be written in C++ or even higher-level languages. Android provides a nice example: its kernel is written mostly in C with some assembly language. Most Android system libraries are written in C or C++, and its application frameworks—which provide the developer interface to the system—are written mostly in Java. We cover Android’s architecture in more detail in Section 2.8.5.2.

The advantages of using a higher-level language, or at least a systems-implementation language, for implementing operating systems are the same as those gained when the language is used for application programs: the code can be written faster, is more compact, and is easier to understand and debug. In addition, improvements in compiler technology will improve the generated code for the entire operating system by simple recompilation. Finally, an operating system is far easier to port to other hardware if it is written in a higher-level language. This is particularly important for operating systems that are intended to run on several different hardware systems, such as small embedded devices, Intel x86 systems, and ARM chips running on phones and tablets.

The only possible disadvantages of implementing an operating system in a higher-level language are reduced speed and increased storage requirements. This, however, is not a major issue in today’s systems. Although an expert assembly-language programmer can produce efficient small routines, for large programs a modern compiler can perform complex analysis and apply sophisticated optimizations that produce excellent code. Modern processors have deep pipelining and multiple functional units that can handle the details of complex dependencies much more easily than can the human mind.

As is true in other systems, major performance improvements in operating systems are more likely to be the result of better data structures and algorithms than of excellent assembly-language code. In addition, although operating systems are large, only a small amount of the code is critical to high performance; the interrupt handlers, I/O manager, memory manager, and CPU scheduler are probably the most critical routines. After the system is written and is working correctly, bottlenecks can be identified and can be refactored to operate more efficiently.

## 2.8 Operating-System Structure

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components, or modules, rather than have one single system. Each of these modules should be a well-defined portion of the system, with carefully defined interfaces and functions. You may use a similar approach when you structure your programs: rather than placing all of your code in the `main()` function, you instead separate logic into a number of functions, clearly articulate parameters and return values, and then call those functions from `main()`.

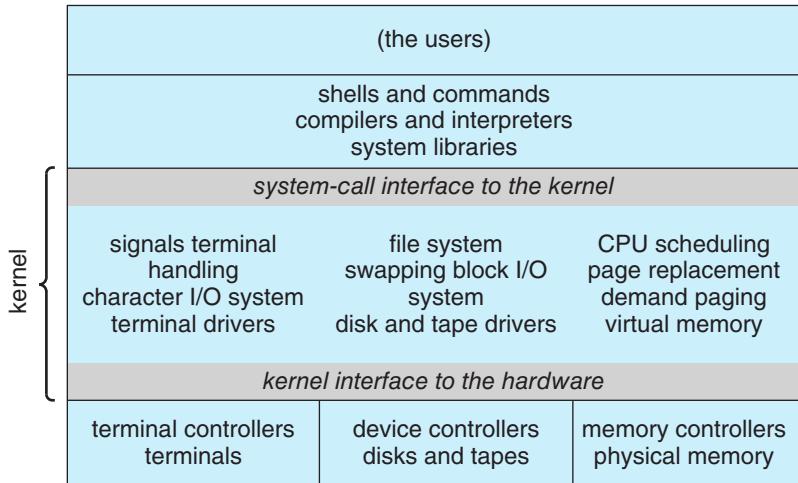


Figure 2.12 Traditional UNIX system structure.

We briefly discussed the common components of operating systems in Chapter 1. In this section, we discuss how these components are interconnected and melded into a kernel.

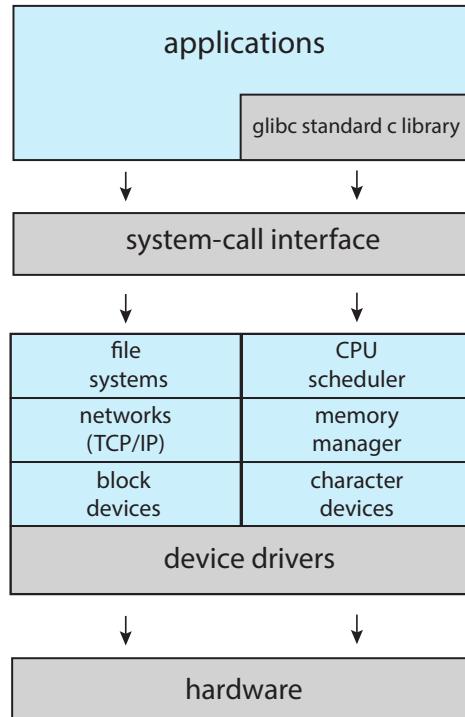
### 2.8.1 Monolithic Structure

The simplest structure for organizing an operating system is no structure at all. That is, place all of the functionality of the kernel into a single, static binary file that runs in a single address space. This approach—known as a **monolithic** structure—is a common technique for designing operating systems.

An example of such limited structuring is the original UNIX operating system, which consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved. We can view the traditional UNIX operating system as being layered to some extent, as shown in Figure 2.12. Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Taken in sum, that is an enormous amount of functionality to be combined into one single address space.

The Linux operating system is based on UNIX and is structured similarly, as shown in Figure 2.13. Applications typically use the glibc standard C library when communicating with the system call interface to the kernel. The Linux kernel is monolithic in that it runs entirely in kernel mode in a single address space, but as we shall see in Section 2.8.4, it does have a modular design that allows the kernel to be modified during run time.

Despite the apparent simplicity of monolithic kernels, they are difficult to implement and extend. Monolithic kernels do have a distinct performance advantage, however: there is very little overhead in the system-call interface, and communication within the kernel is fast. Therefore, despite the drawbacks



**Figure 2.13** Linux system structure.

of monolithic kernels, their speed and efficiency explains why we still see evidence of this structure in the UNIX, Linux, and Windows operating systems.

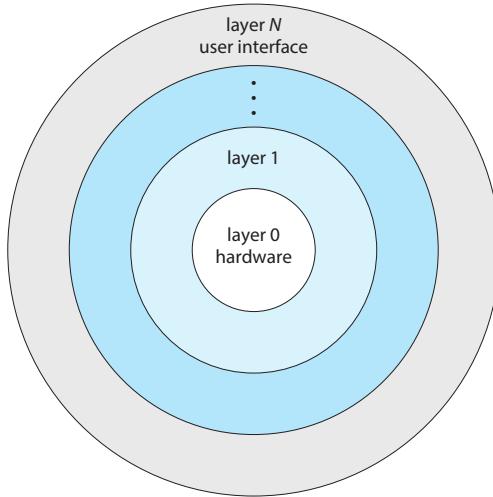
### 2.8.2 Layered Approach

The monolithic approach is often known as a **tightly coupled** system because changes to one part of the system can have wide-ranging effects on other parts. Alternatively, we could design a **loosely coupled** system. Such a system is divided into separate, smaller components that have specific and limited functionality. All these components together comprise the kernel. The advantage of this modular approach is that changes in one component affect only that component, and no others, allowing system implementers more freedom in creating and changing the inner workings of the system.

A system can be made modular in many ways. One method is the **layered approach**, in which the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer  $N$ ) is the user interface. This layering structure is depicted in Figure 2.14.

An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data. A typical operating-system layer—say, layer  $M$ —consists of data structures and a set of functions that can be invoked by higher-level layers. Layer  $M$ , in turn, can invoke operations on lower-level layers.

The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations)



**Figure 2.14** A layered operating system.

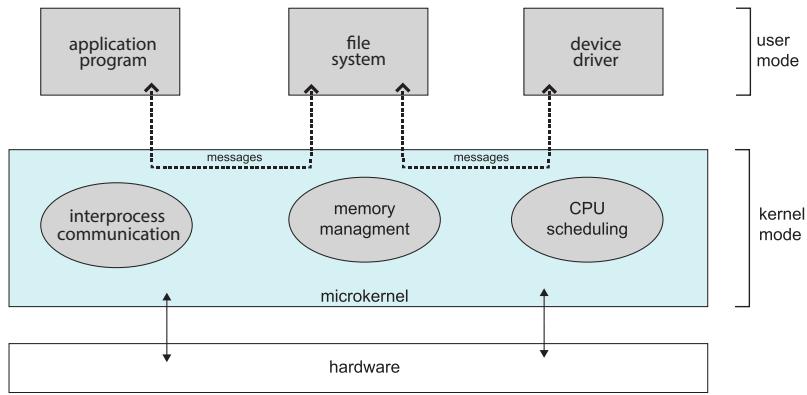
and services of only lower-level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system are simplified.

Each layer is implemented only with operations provided by lower-level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

Layered systems have been successfully used in computer networks (such as TCP/IP) and web applications. Nevertheless, relatively few operating systems use a pure layered approach. One reason involves the challenges of appropriately defining the functionality of each layer. In addition, the overall performance of such systems is poor due to the overhead of requiring a user program to traverse through multiple layers to obtain an operating-system service. *Some* layering is common in contemporary operating systems, however. Generally, these systems have fewer layers with more functionality, providing most of the advantages of modularized code while avoiding the problems of layer definition and interaction.

### 2.8.3 Microkernels

We have already seen that the original UNIX system had a monolithic structure. As UNIX expanded, the kernel became large and difficult to manage. In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the **microkernel** approach. This method structures the operating system by removing



**Figure 2.15** Architecture of a typical microkernel.

all nonessential components from the kernel and implementing them as user-level programs that reside in separate address spaces. The result is a smaller kernel. There is little consensus regarding which services should remain in the kernel and which should be implemented in user space. Typically, however, microkernels provide minimal process and memory management, in addition to a communication facility. Figure 2.15 illustrates the architecture of a typical microkernel.

The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space. Communication is provided through message passing, which was described in Section 2.3.3.5. For example, if the client program wishes to access a file, it must interact with the file server. The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.

One benefit of the microkernel approach is that it makes extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel. The resulting operating system is easier to port from one hardware design to another. The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched.

Perhaps the best-known illustration of a microkernel operating system is *Darwin*, the kernel component of the macOS and iOS operating systems. Darwin, in fact, consists of two kernels, one of which is the Mach microkernel. We will cover the macOS and iOS systems in further detail in Section 2.8.5.1.

Another example is QNX, a real-time operating system for embedded systems. The QNX Neutrino microkernel provides services for message passing and process scheduling. It also handles low-level network communication and hardware interrupts. All other services in QNX are provided by standard processes that run outside the kernel in user mode.

Unfortunately, the performance of microkernels can suffer due to increased system-function overhead. When two user-level services must communicate, messages must be copied between the services, which reside in separate

address spaces. In addition, the operating system may have to switch from one process to the next to exchange the messages. The overhead involved in copying messages and switching between processes has been the largest impediment to the growth of microkernel-based operating systems. Consider the history of Windows NT: The first release had a layered microkernel organization. This version's performance was low compared with that of Windows 95. Windows NT 4.0 partially corrected the performance problem by moving layers from user space to kernel space and integrating them more closely. By the time Windows XP was designed, Windows architecture had become more monolithic than microkernel. Section 2.8.5.1 will describe how macOS addresses the performance issues of the Mach microkernel.

#### 2.8.4 Modules

Perhaps the best current methodology for operating-system design involves using **loadable kernel modules** (**LKMs**). Here, the kernel has a set of core components and can link in additional services via modules, either at boot time or during run time. This type of design is common in modern implementations of UNIX, such as Linux, macOS, and Solaris, as well as Windows.

The idea of the design is for the kernel to provide core services, while other services are implemented dynamically, as the kernel is running. Linking services dynamically is preferable to adding new features directly to the kernel, which would require recompiling the kernel every time a change was made. Thus, for example, we might build CPU scheduling and memory management algorithms directly into the kernel and then add support for different file systems by way of loadable modules.

The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system, because any module can call any other module. The approach is also similar to the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.

Linux uses loadable kernel modules, primarily for supporting device drivers and file systems. LKMs can be “inserted” into the kernel as the system is started (or *booted*) or during run time, such as when a USB device is plugged into a running machine. If the Linux kernel does not have the necessary driver, it can be dynamically loaded. LKMs can be removed from the kernel during run time as well. For Linux, LKMs allow a dynamic and modular kernel, while maintaining the performance benefits of a monolithic system. We cover creating LKMs in Linux in several programming exercises at the end of this chapter.

#### 2.8.5 Hybrid Systems

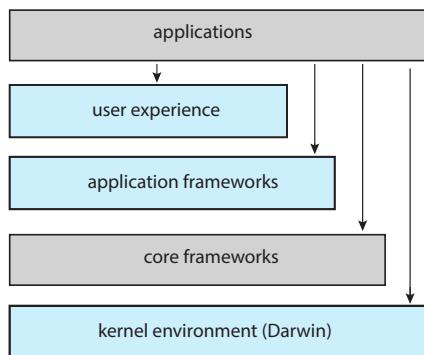
In practice, very few operating systems adopt a single, strictly defined structure. Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues. For example, Linux is monolithic, because having the operating system in a single address space provides very efficient performance. However, it is also modular, so that new functionality can be dynamically added to the kernel. Windows is largely

monolithic as well (again primarily for performance reasons), but it retains some behavior typical of microkernel systems, including providing support for separate subsystems (known as operating-system *personalities*) that run as user-mode processes. Windows systems also provide support for dynamically loadable kernel modules. We provide case studies of Linux and Windows 10 in Chapter 20 and Chapter 21, respectively. In the remainder of this section, we explore the structure of three hybrid systems: the Apple macOS operating system and the two most prominent mobile operating systems—iOS and Android.

### 2.8.5.1 macOS and iOS

Apple's macOS operating system is designed to run primarily on desktop and laptop computer systems, whereas iOS is a mobile operating system designed for the iPhone smartphone and iPad tablet computer. Architecturally, macOS and iOS have much in common, and so we present them together, highlighting what they share as well as how they differ from each other. The general architecture of these two systems is shown in Figure 2.16. Highlights of the various layers include the following:

- **User experience layer.** This layer defines the software interface that allows users to interact with the computing devices. macOS uses the *Aqua* user interface, which is designed for a mouse or trackpad, whereas iOS uses the *Springboard* user interface, which is designed for touch devices.
- **Application frameworks layer.** This layer includes the *Cocoa* and *Cocoa Touch* frameworks, which provide an API for the Objective-C and Swift programming languages. The primary difference between Cocoa and Cocoa Touch is that the former is used for developing macOS applications, and the latter by iOS to provide support for hardware features unique to mobile devices, such as touch screens.
- **Core frameworks.** This layer defines frameworks that support graphics and media including, Quicktime and OpenGL.



**Figure 2.16** Architecture of Apple's macOS and iOS operating systems.

- **Kernel environment.** This environment, also known as **Darwin**, includes the Mach microkernel and the BSD UNIX kernel. We will elaborate on Darwin shortly.

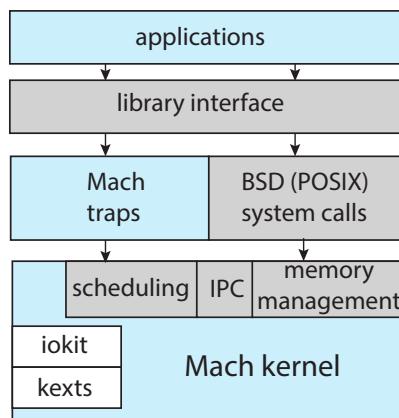
As shown in Figure 2.16, applications can be designed to take advantage of user-experience features or to bypass them and interact directly with either the application framework or the core framework. Additionally, an application can forego frameworks entirely and communicate directly with the kernel environment. (An example of this latter situation is a C program written with no user interface that makes POSIX system calls.)

Some significant distinctions between macOS and iOS include the following:

- Because macOS is intended for desktop and laptop computer systems, it is compiled to run on Intel architectures. iOS is designed for mobile devices and thus is compiled for ARM-based architectures. Similarly, the iOS kernel has been modified somewhat to address specific features and needs of mobile systems, such as power management and aggressive memory management. Additionally, iOS has more stringent security settings than macOS.
- The iOS operating system is generally much more restricted to developers than macOS and may even be closed to developers. For example, iOS restricts access to POSIX and BSD APIs on iOS, whereas they are openly available to developers on macOS.

We now focus on Darwin, which uses a hybrid structure. Darwin is a layered system that consists primarily of the Mach microkernel and the BSD UNIX kernel. Darwin's structure is shown in Figure 2.17.

Whereas most operating systems provide a single system-call interface to the kernel—such as through the standard C library on UNIX and Linux systems—Darwin provides *two* system-call interfaces: Mach system calls (known as



**Figure 2.17** The structure of Darwin.

**traps**) and BSD system calls (which provide POSIX functionality). The interface to these system calls is a rich set of libraries that includes not only the standard C library but also libraries that provide networking, security, and programming language support (to name just a few).

Beneath the system-call interface, Mach provides fundamental operating-system services, including memory management, CPU scheduling, and inter-process communication (IPC) facilities such as message passing and remote procedure calls (RPCs). Much of the functionality provided by Mach is available through **kernel abstractions**, which include tasks (a Mach process), threads, memory objects, and ports (used for IPC). As an example, an application may create a new process using the BSD POSIX `fork()` system call. Mach will, in turn, use a task kernel abstraction to represent the process in the kernel.

In addition to Mach and BSD, the kernel environment provides an I/O kit for development of device drivers and dynamically loadable modules (which macOS refers to as **kernel extensions**, or **kexts**).

In Section 2.8.3, we described how the overhead of message passing between different services running in user space compromises the performance of microkernels. To address such performance problems, Darwin combines Mach, BSD, the I/O kit, and any kernel extensions into a single address space. Thus, Mach is not a pure microkernel in the sense that various subsystems run in user space. Message passing within Mach still does occur, but no copying is necessary, as the services have access to the same address space.

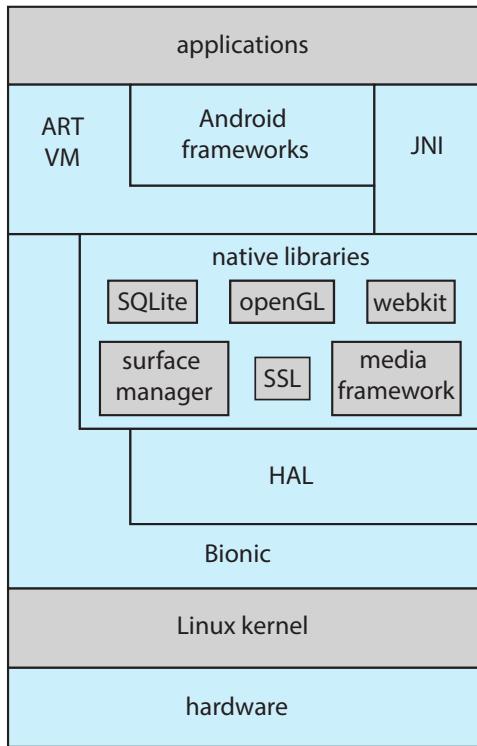
Apple has released the Darwin operating system as open source. As a result, various projects have added extra functionality to Darwin, such as the X-11 windowing system and support for additional file systems. Unlike Darwin, however, the Cocoa interface, as well as other proprietary Apple frameworks available for developing macOS applications, are closed.

### 2.8.5.2 Android

The Android operating system was designed by the Open Handset Alliance (led primarily by Google) and was developed for Android smartphones and tablet computers. Whereas iOS is designed to run on Apple mobile devices and is close-sourced, Android runs on a variety of mobile platforms and is open-sourced, partly explaining its rapid rise in popularity. The structure of Android appears in Figure 2.18.

Android is similar to iOS in that it is a layered stack of software that provides a rich set of frameworks supporting graphics, audio, and hardware features. These features, in turn, provide a platform for developing mobile applications that run on a multitude of Android-enabled devices.

Software designers for Android devices develop applications in the Java language, but they do not generally use the standard Java API. Google has designed a separate Android API for Java development. Java applications are compiled into a form that can execute on the Android RunTime ART, a virtual machine designed for Android and optimized for mobile devices with limited memory and CPU processing capabilities. Java programs are first compiled to a Java bytecode `.class` file and then translated into an executable `.dex` file. Whereas many Java virtual machines perform just-in-time (JIT) compilation to improve application efficiency, ART performs **ahead-of-time (AOT)** compila-



**Figure 2.18** Architecture of Google's Android.

tion. Here, .dex files are compiled into native machine code when they are installed on a device, from which they can execute on the ART. AOT compilation allows more efficient application execution as well as reduced power consumption, features that are crucial for mobile systems.

Android developers can also write Java programs that use the Java native interface—or JNI—which allows developers to bypass the virtual machine and instead write Java programs that can access specific hardware features. Programs written using JNI are generally not portable from one hardware device to another.

The set of native libraries available for Android applications includes frameworks for developing web browsers (webkit), database support (SQLite), and network support, such as secure sockets (SSLs).

Because Android can run on an almost unlimited number of hardware devices, Google has chosen to abstract the physical hardware through the hardware abstraction layer, or HAL. By abstracting all hardware, such as the camera, GPS chip, and other sensors, the HAL provides applications with a consistent view independent of specific hardware. This feature, of course, allows developers to write programs that are portable across different hardware platforms.

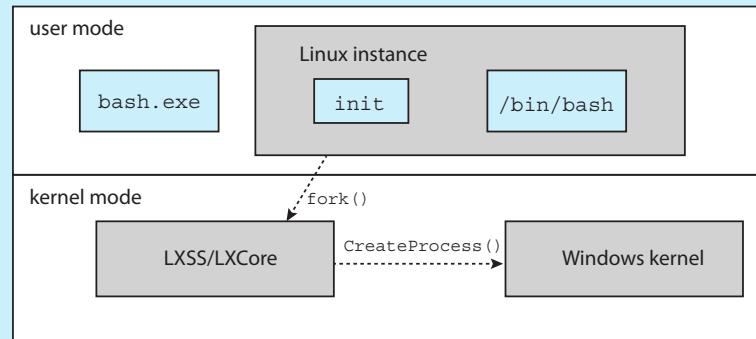
The standard C library used by Linux systems is the GNU C library (*glibc*). Google instead developed the **Bionic** standard C library for Android. Not only does Bionic have a smaller memory footprint than *glibc*, but it also has been designed for the slower CPUs that characterize mobile devices. (In addition, Bionic allows Google to bypass GPL licensing of *glibc*.)

At the bottom of Android's software stack is the Linux kernel. Google has modified the Linux kernel used in Android in a variety of areas to support the special needs of mobile systems, such as power management. It has also made changes in memory management and allocation and has added a new form of IPC known as *Binder* (which we will cover in Section 3.8.2.1).

### WINDOWS SUBSYSTEM FOR LINUX

Windows uses a hybrid architecture that provides subsystems to emulate different operating-system environments. These user-mode subsystems communicate with the Windows kernel to provide actual services. Windows 10 adds a Windows subsystem for Linux ([WSL](#)), which allows native Linux applications (specified as ELF binaries) to run on Windows 10. The typical operation is for a user to start the Windows application `bash.exe`, which presents the user with a `bash` shell running Linux. Internally, the WSL creates a [Linux instance](#) consisting of the `init` process, which in turn creates the `bash` shell running the native Linux application `/bin/bash`. Each of these processes runs in a Windows [Pico](#) process. This special process loads the native Linux binary into the process's own address space, thus providing an environment in which a Linux application can execute.

Pico processes communicate with the kernel services LXCore and LXSS to translate Linux system calls, if possible using native Windows system calls. When the Linux application makes a system call that has no Windows equivalent, the LXSS service must provide the equivalent functionality. When there is a one-to-one relationship between the Linux and Windows system calls, LXSS forwards the Linux system call directly to the equivalent call in the Windows kernel. In some situations, Linux and Windows have system calls that are similar but not identical. When this occurs, LXSS will provide some of the functionality and will invoke the similar Windows system call to provide the remainder of the functionality. The Linux `fork()` provides an illustration of this: The Windows `CreateProcess()` system call is similar to `fork()` but does not provide exactly the same functionality. When `fork()` is invoked in WSL, the LXSS service does some of the initial work of `fork()` and then calls `CreateProcess()` to do the remainder of the work. The figure below illustrates the basic behavior of WSL.



## 2.9 Building and Booting an Operating System

It is possible to design, code, and implement an operating system specifically for one specific machine configuration. More commonly, however, operating systems are designed to run on any of a class of machines with a variety of peripheral configurations.

### 2.9.1 Operating-System Generation

Most commonly, a computer system, when purchased, has an operating system already installed. For example, you may purchase a new laptop with Windows or macOS preinstalled. But suppose you wish to replace the preinstalled operating system or add additional operating systems. Or suppose you purchase a computer without an operating system. In these latter situations, you have a few options for placing the appropriate operating system on the computer and configuring it for use.

If you are generating (or building) an operating system from scratch, you must follow these steps:

1. Write the operating system source code (or obtain previously written source code).
2. Configure the operating system for the system on which it will run.
3. Compile the operating system.
4. Install the operating system.
5. Boot the computer and its new operating system.

Configuring the system involves specifying which features will be included, and this varies by operating system. Typically, parameters describing how the system is configured is stored in a configuration file of some type, and once this file is created, it can be used in several ways.

At one extreme, a system administrator can use it to modify a copy of the operating-system source code. Then the operating system is completely compiled (known as a **system build**). Data declarations, initializations, and constants, along with compilation, produce an output-object version of the operating system that is tailored to the system described in the configuration file.

At a slightly less tailored level, the system description can lead to the selection of precompiled object modules from an existing library. These modules are linked together to form the generated operating system. This process allows the library to contain the device drivers for all supported I/O devices, but only those needed are selected and linked into the operating system. Because the system is not recompiled, system generation is faster, but the resulting system may be overly general and may not support different hardware configurations.

At the other extreme, it is possible to construct a system that is completely modular. Here, selection occurs at execution time rather than at compile or link time. System generation involves simply setting the parameters that describe the system configuration.

The major differences among these approaches are the size and generality of the generated system and the ease of modifying it as the hardware configuration changes. For embedded systems, it is not uncommon to adopt the first approach and create an operating system for a specific, static hardware configuration. However, most modern operating systems that support desktop and laptop computers as well as mobile devices have adopted the second approach. That is, the operating system is still generated for a specific hardware configuration, but the use of techniques such as loadable kernel modules provides modular support for dynamic changes to the system.

We now illustrate how to build a Linux system from scratch, where it is typically necessary to perform the following steps:

1. Download the Linux source code from <http://www.kernel.org>.
2. Configure the kernel using the “`make menuconfig`” command. This step generates the `.config` configuration file.
3. Compile the main kernel using the “`make`” command. The `make` command compiles the kernel based on the configuration parameters identified in the `.config` file, producing the file `vmlinuz`, which is the kernel image.
4. Compile the kernel modules using the “`make modules`” command. Just as with compiling the kernel, module compilation depends on the configuration parameters specified in the `.config` file.
5. Use the command “`make modules_install`” to install the kernel modules into `vmlinuz`.
6. Install the new kernel on the system by entering the “`make install`” command.

When the system reboots, it will begin running this new operating system.

Alternatively, it is possible to modify an existing system by installing a Linux virtual machine. This will allow the host operating system (such as Windows or macOS) to run Linux. (We introduced virtualization in Section 1.7 and cover the topic more fully in Chapter 18.)

There are a few options for installing Linux as a virtual machine. One alternative is to build a virtual machine from scratch. This option is similar to building a Linux system from scratch; however, the operating system does not need to be compiled. Another approach is to use a Linux virtual machine appliance, which is an operating system that has already been built and configured. This option simply requires downloading the appliance and installing it using virtualization software such as VirtualBox or VMware. For example, to build the operating system used in the virtual machine provided with this text, the authors did the following:

1. Downloaded the Ubuntu ISO image from <https://www.ubuntu.com/>
2. Instructed the virtual machine software VirtualBox to use the ISO as the bootable medium and booted the virtual machine
3. Answered the installation questions and then installed and booted the operating system as a virtual machine

### 2.9.2 System Boot

After an operating system is generated, it must be made available for use by the hardware. But how does the hardware know where the kernel is or how to load that kernel? The process of starting a computer by loading the kernel is known as **booting** the system. On most systems, the boot process proceeds as follows:

1. A small piece of code known as the **bootstrap program** or **boot loader** locates the kernel.
2. The kernel is loaded into memory and started.
3. The kernel initializes hardware.
4. The root file system is mounted.

In this section, we briefly describe the boot process in more detail.

Some computer systems use a multistage boot process: When the computer is first powered on, a small boot loader located in nonvolatile firmware known as **BIOS** is run. This initial boot loader usually does nothing more than load a second boot loader, which is located at a fixed disk location called the **boot block**. The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution. More typically, it is simple code (as it must fit in a single disk block) and knows only the address on disk and the length of the remainder of the bootstrap program.

Many recent computer systems have replaced the BIOS-based boot process with **UEFI** (Unified Extensible Firmware Interface). UEFI has several advantages over BIOS, including better support for 64-bit systems and larger disks. Perhaps the greatest advantage is that UEFI is a single, complete boot manager and therefore is faster than the multistage BIOS boot process.

Whether booting from BIOS or UEFI, the bootstrap program can perform a variety of tasks. In addition to loading the file containing the kernel program into memory, it also runs diagnostics to determine the state of the machine—for example, inspecting memory and the CPU and discovering devices. If the diagnostics pass, the program can continue with the booting steps. The bootstrap can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system and mounts the root file system. It is only at this point is the system said to be **running**.

**GRUB** is an open-source bootstrap program for Linux and UNIX systems. Boot parameters for the system are set in a GRUB configuration file, which is loaded at startup. GRUB is flexible and allows changes to be made at boot time, including modifying kernel parameters and even selecting among different kernels that can be booted. As an example, the following are kernel parameters from the special Linux file `/proc/cmdline`, which is used at boot time:

```
BOOT_IMAGE=/boot/vmlinuz-4.4.0-59-generic
root=UUID=5f2e2232-4e47-4fe8-ae94-45ea749a5c92
```

`BOOT_IMAGE` is the name of the kernel image to be loaded into memory, and `root` specifies a unique identifier of the root file system.

To save space as well as decrease boot time, the Linux kernel image is a compressed file that is extracted after it is loaded into memory. During the boot process, the boot loader typically creates a temporary RAM file system, known as `initramfs`. This file system contains necessary drivers and kernel modules that must be installed to support the *real* root file system (which is not in main memory). Once the kernel has started and the necessary drivers are installed, the kernel switches the root file system from the temporary RAM location to the appropriate root file system location. Finally, Linux creates the `systemd` process, the initial process in the system, and then starts other services (for example, a web server and/or database). Ultimately, the system will present the user with a login prompt. In Section 11.5.2, we describe the boot process for Windows.

It is worthwhile to note that the booting mechanism is not independent from the boot loader. Therefore, there are specific versions of the GRUB boot loader for BIOS and UEFI, and the firmware must know as well which specific bootloader is to be used.

The boot process for mobile systems is slightly different from that for traditional PCs. For example, although its kernel is Linux-based, Android does not use GRUB and instead leaves it up to vendors to provide boot loaders. The most common Android boot loader is LK (for “little kernel”). Android systems use the same compressed kernel image as Linux, as well as an initial RAM file system. However, whereas Linux discards the `initramfs` once all necessary drivers have been loaded, Android maintains `initramfs` as the root file system for the device. Once the kernel has been loaded and the root file system mounted, Android starts the `init` process and creates a number of services before displaying the home screen.

Finally, boot loaders for most operating systems—including Windows, Linux, and macOS, as well as both iOS and Android—provide booting into **recovery mode** or **single-user mode** for diagnosing hardware issues, fixing corrupt file systems, and even reinstalling the operating system. In addition to hardware failures, computer systems can suffer from software errors and poor operating-system performance, which we consider in the following section.

## 2.10 Operating-System Debugging

We have mentioned debugging from time to time in this chapter. Here, we take a closer look. Broadly, **debugging** is the activity of finding and fixing errors in a system, both in hardware and in software. Performance problems are considered bugs, so debugging can also include **performance tuning**, which seeks to improve performance by removing processing **bottlenecks**. In this section, we explore debugging process and kernel errors and performance problems. Hardware debugging is outside the scope of this text.

### 2.10.1 Failure Analysis

If a process fails, most operating systems write the error information to a **log file** to alert system administrators or users that the problem occurred. The operating system can also take a **core dump**—a capture of the memory of the process—and store it in a file for later analysis. (Memory was referred to as the

“core” in the early days of computing.) Running programs and core dumps can be probed by a debugger, which allows a programmer to explore the code and memory of a process at the time of failure.

Debugging user-level process code is a challenge. Operating-system kernel debugging is even more complex because of the size and complexity of the kernel, its control of the hardware, and the lack of user-level debugging tools. A failure in the kernel is called a **crash**. When a crash occurs, error information is saved to a log file, and the memory state is saved to a **crash dump**.

Operating-system debugging and process debugging frequently use different tools and techniques due to the very different nature of these two tasks. Consider that a kernel failure in the file-system code would make it risky for the kernel to try to save its state to a file on the file system before rebooting. A common technique is to save the kernel’s memory state to a section of disk set aside for this purpose that contains no file system. If the kernel detects an unrecoverable error, it writes the entire contents of memory, or at least the kernel-owned parts of the system memory, to the disk area. When the system reboots, a process runs to gather the data from that area and write it to a crash dump file within a file system for analysis. Obviously, such strategies would be unnecessary for debugging ordinary user-level processes.

### 2.10.2 Performance Monitoring and Tuning

We mentioned earlier that performance tuning seeks to improve performance by removing processing bottlenecks. To identify bottlenecks, we must be able to monitor system performance. Thus, the operating system must have some means of computing and displaying measures of system behavior. Tools may be characterized as providing either *per-process* or *system-wide* observations. To make these observations, tools may use one of two approaches—*counters* or *tracing*. We explore each of these in the following sections.

#### 2.10.2.1 Counters

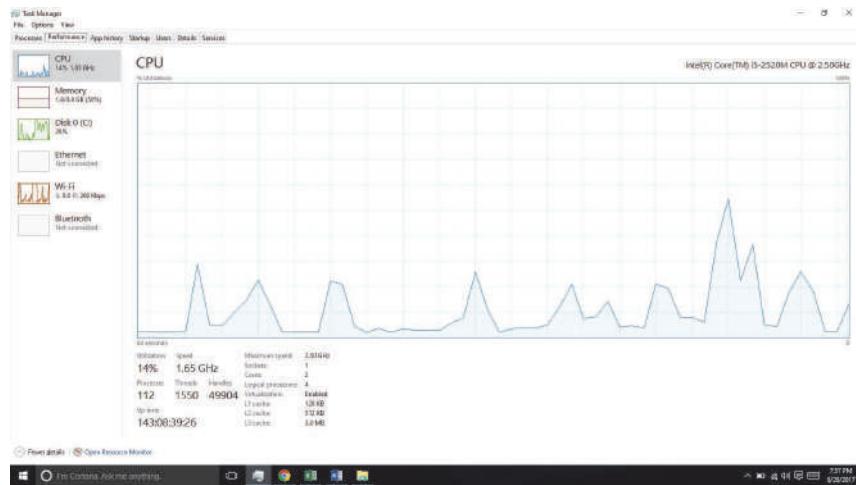
Operating systems keep track of system activity through a series of counters, such as the number of system calls made or the number of operations performed to a network device or disk. The following are examples of Linux tools that use counters:

##### Per-Process

- `ps`—reports information for a single process or selection of processes
- `top`—reports real-time statistics for current processes

##### System-Wide

- `vmstat`—reports memory-usage statistics
- `netstat`—reports statistics for network interfaces
- `iostat`—reports I/O usage for disks



**Figure 2.19** The Windows 10 task manager.

Most of the counter-based tools on Linux systems read statistics from the /proc file system. /proc is a “pseudo” file system that exists only in kernel memory and is used primarily for querying various per-process as well as kernel statistics. The /proc file system is organized as a directory hierarchy, with the process (a unique integer value assigned to each process) appearing as a subdirectory below /proc. For example, the directory entry /proc/2155 would contain per-process statistics for the process with an ID of 2155. There are /proc entries for various kernel statistics as well. In both this chapter and Chapter 3, we provide programming projects where you will create and access the /proc file system.

Windows systems provide the **Windows Task Manager**, a tool that includes information for current applications as well as processes, CPU and memory usage, and networking statistics. A screen shot of the task manager in Windows 10 appears in Figure 2.19.

### 2.10.3 Tracing

Whereas counter-based tools simply inquire on the current value of certain statistics that are maintained by the kernel, tracing tools collect data for a specific event—such as the steps involved in a system-call invocation.

The following are examples of Linux tools that trace events:

#### Per-Process

- **strace**—traces system calls invoked by a process
- **gdb**—a source-level debugger

#### System-Wide

- **perf**—a collection of Linux performance tools
- **tcpdump**—collects network packets

**Kernighan's Law**

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

Making operating systems easier to understand, debug, and tune as they run is an active area of research and practice. A new generation of kernel-enabled performance analysis tools has made significant improvements in how this goal can be achieved. Next, we discuss BCC, a toolkit for dynamic kernel tracing in Linux.

#### 2.10.4 BCC

Debugging the interactions between user-level and kernel code is nearly impossible without a toolset that understands both sets of code and can instrument their interactions. For that toolset to be truly useful, it must be able to debug any area of a system, including areas that were not written with debugging in mind, and do so without affecting system reliability. This toolset must also have a minimal performance impact—ideally it should have no impact when not in use and a proportional impact during use. The BCC toolkit meets these requirements and provides a dynamic, secure, low-impact debugging environment.

BCC (BPF Compiler Collection) is a rich toolkit that provides tracing features for Linux systems. BCC is a front-end interface to the eBPF (extended Berkeley Packet Filter) tool. The BPF technology was developed in the early 1990s for filtering traffic across a computer network. The “extended” BPF (eBPF) added various features to BPF. eBPF programs are written in a subset of C and are compiled into eBPF instructions, which can be dynamically inserted into a running Linux system. The eBPF instructions can be used to capture specific events (such as a certain system call being invoked) or to monitor system performance (such as the time required to perform disk I/O). To ensure that eBPF instructions are well behaved, they are passed through a **verifie** before being inserted into the running Linux kernel. The verifier checks to make sure that the instructions do not affect system performance or security.

Although eBPF provides a rich set of features for tracing within the Linux kernel, it traditionally has been very difficult to develop programs using its C interface. BCC was developed to make it easier to write tools using eBPF by providing a front-end interface in Python. A BCC tool is written in Python and it embeds C code that interfaces with the eBPF instrumentation, which in turn interfaces with the kernel. The BCC tool also compiles the C program into eBPF instructions and inserts it into the kernel using either probes or tracepoints, two techniques that allow tracing events in the Linux kernel.

The specifics of writing custom BCC tools are beyond the scope of this text, but the BCC package (which is installed on the Linux virtual machine we provide) provides a number of existing tools that monitor several areas

of activity in a running Linux kernel. As an example, the BCC `disksnoop` tool traces disk I/O activity. Entering the command

```
./disksnoop.py
```

generates the following example output:

TIME(s)	T	BYTES	LAT(ms)
1946.29186700	R	8	0.27
1946.33965000	R	8	0.26
1948.34585000	W	8192	0.96
1950.43251000	R	4096	0.56
1951.74121000	R	4096	0.35

This output tells us the timestamp when the I/O operation occurred, whether the I/O was a Read or Write operation, and how many bytes were involved in the I/O. The final column reflects the duration (expressed as latency or LAT) in milliseconds of the I/O.

Many of the tools provided by BCC can be used for specific applications, such as MySQL databases, as well as Java and Python programs. Probes can also be placed to monitor the activity of a specific process. For example, the command

```
./opensnoop -p 1225
```

will trace `open()` system calls performed only by the process with an identifier of 1225.

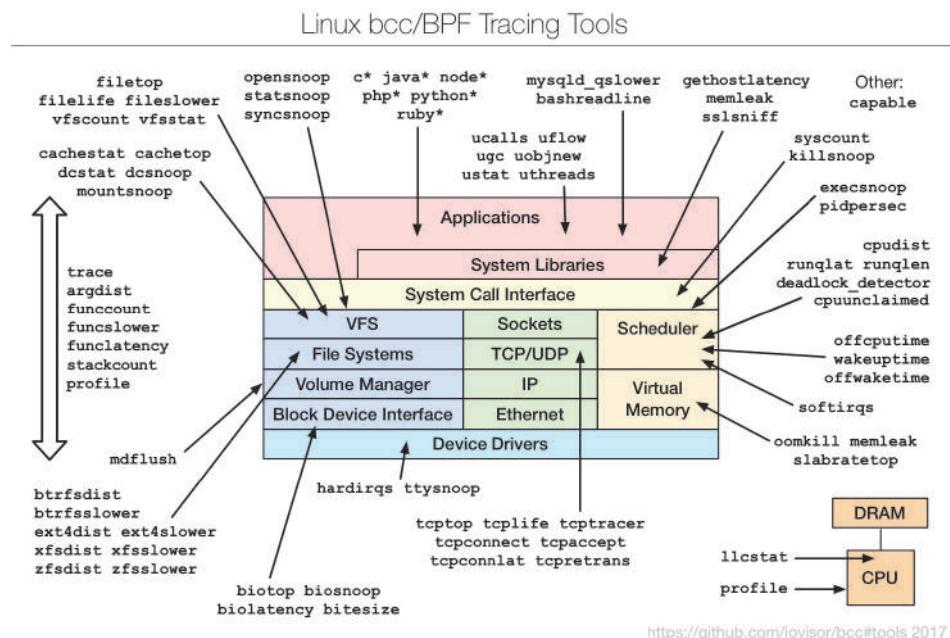


Figure 2.20 The BCC and eBPF tracing tools.

What makes BCC especially powerful is that its tools can be used on live production systems that are running critical applications without causing harm to the system. This is particularly useful for system administrators who must monitor system performance to identify possible bottlenecks or security exploits. Figure 2.20 illustrates the wide range of tools currently provided by BCC and eBPF and their ability to trace essentially any area of the Linux operating system. BCC is a rapidly changing technology with new features constantly being added.

## 2.11 Summary

- An operating system provides an environment for the execution of programs by providing services to users and programs.
- The three primary approaches for interacting with an operating system are (1) command interpreters, (2) graphical user interfaces, and (3) touch-screen interfaces.
- System calls provide an interface to the services made available by an operating system. Programmers use a system call's application programming interface (API) for accessing system-call services.
- System calls can be divided into six major categories: (1) process control, (2) file management, (3) device management, (4) information maintenance, (5) communications, and (6) protection.
- The standard C library provides the system-call interface for UNIX and Linux systems.
- Operating systems also include a collection of system programs that provide utilities to users.
- A linker combines several relocatable object modules into a single binary executable file. A loader loads the executable file into memory, where it becomes eligible to run on an available CPU.
- There are several reasons why applications are operating-system specific. These include different binary formats for program executables, different instruction sets for different CPUs, and system calls that vary from one operating system to another.
- An operating system is designed with specific goals in mind. These goals ultimately determine the operating system's policies. An operating system implements these policies through specific mechanisms.
- A monolithic operating system has no structure; all functionality is provided in a single, static binary file that runs in a single address space. Although such systems are difficult to modify, their primary benefit is efficiency.
- A layered operating system is divided into a number of discrete layers, where the bottom layer is the hardware interface and the highest layer is the user interface. Although layered software systems have had some suc-

cess, this approach is generally not ideal for designing operating systems due to performance problems.

- The microkernel approach for designing operating systems uses a minimal kernel; most services run as user-level applications. Communication takes place via message passing.
- A modular approach for designing operating systems provides operating-system services through modules that can be loaded and removed during run time. Many contemporary operating systems are constructed as hybrid systems using a combination of a monolithic kernel and modules.
- A boot loader loads an operating system into memory, performs initialization, and begins system execution.
- The performance of an operating system can be monitored using either counters or tracing. Counters are a collection of system-wide or per-process statistics, while tracing follows the execution of a program through the operating system.

## Practice Exercises

- 2.1 What is the purpose of system calls?
- 2.2 What is the purpose of the command interpreter? Why is it usually separate from the kernel?
- 2.3 What system calls have to be executed by a command interpreter or shell in order to start a new process on a UNIX system?
- 2.4 What is the purpose of system programs?
- 2.5 What is the main advantage of the layered approach to system design? What are the disadvantages of the layered approach?
- 2.6 List five services provided by an operating system, and explain how each creates convenience for users. In which cases would it be impossible for user-level programs to provide these services? Explain your answer.
- 2.7 Why do some systems store the operating system in firmware, while others store it on disk?
- 2.8 How could a system be designed to allow a choice of operating systems from which to boot? What would the bootstrap program need to do?

## Further Reading

[Bryant and O'Hallaron (2015)] provide an overview of computer systems, including the role of the linker and loader. [Atlidakis et al. (2016)] discuss POSIX system calls and how they relate to modern operating systems. [Levin (2013)] covers the internals of both macOS and iOS, and [Levin (2015)] describes details of the Android system. Windows 10 internals are covered in [Russinovich et al. (2017)]. BSD UNIX is described in [McKusick et al. (2015)]. [Love (2010)] and

[Mauerer (2008)] thoroughly discuss the Linux kernel. Solaris is fully described in [McDougall and Mauro (2007)].

Linux source code is available at <http://www.kernel.org>. The Ubuntu ISO image is available from <https://www.ubuntu.com/>.

Comprehensive coverage of Linux kernel modules can be found at <http://www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf>. [Ward (2015)] and <http://www.ibm.com/developerworks/linux/library/l-linuxboot/> describe the Linux boot process using GRUB. Performance tuning—with a focus on Linux and Solaris systems—is covered in [Gregg (2014)]. Details for the BCC toolkit can be found at <https://github.com/iovisor/bcc/#tools>.

## Bibliography

- [Atlidakis et al. (2016)] V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh, “POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing” (2016), pages 19:1–19:17.
- [Bryant and O’Hallaron (2015)] R. Bryant and D. O’Hallaron, *Computer Systems: A Programmer’s Perspective*, Third Edition (2015).
- [Gregg (2014)] B. Gregg, *Systems Performance—Enterprise and the Cloud*, Pearson (2014).
- [Levin (2013)] J. Levin, *Mac OS X and iOS Internals to the Apple’s Core*, Wiley (2013).
- [Levin (2015)] J. Levin, *Android Internals—A Confectioner’s Cookbook. Volume I* (2015).
- [Love (2010)] R. Love, *Linux Kernel Development*, Third Edition, Developer’s Library (2010).
- [Mauerer (2008)] W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons (2008).
- [McDougall and Mauro (2007)] R. McDougall and J. Mauro, *Solaris Internals*, Second Edition, Prentice Hall (2007).
- [McKusick et al. (2015)] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIX Operating System—Second Edition*, Pearson (2015).
- [Russinovich et al. (2017)] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals—Part 1*, Seventh Edition, Microsoft Press (2017).
- [Ward (2015)] B. Ward, *How LINUX Works—What Every Superuser Should Know*, Second Edition, No Starch Press (2015).

## Chapter 2 Exercises

- 2.9 The services and functions provided by an operating system can be divided into two main categories. Briefly describe the two categories, and discuss how they differ.
- 2.10 Describe three general methods for passing parameters to the operating system.
- 2.11 Describe how you could obtain a statistical profile of the amount of time a program spends executing different sections of its code. Discuss the importance of obtaining such a statistical profile.
- 2.12 What are the advantages and disadvantages of using the same system-call interface for manipulating both files and devices?
- 2.13 Would it be possible for the user to develop a new command interpreter using the system-call interface provided by the operating system?
- 2.14 Describe why Android uses ahead-of-time (AOT) rather than just-in-time (JIT) compilation.
- 2.15 What are the two models of interprocess communication? What are the strengths and weaknesses of the two approaches?
- 2.16 Contrast and compare an application programming interface (API) and an application binary interface (ABI).
- 2.17 Why is the separation of mechanism and policy desirable?
- 2.18 It is sometimes difficult to achieve a layered approach if two components of the operating system are dependent on each other. Identify a scenario in which it is unclear how to layer two system components that require tight coupling of their functionalities.
- 2.19 What is the main advantage of the microkernel approach to system design? How do user programs and system services interact in a microkernel architecture? What are the disadvantages of using the microkernel approach?
- 2.20 What are the advantages of using loadable kernel modules?
- 2.21 How are iOS and Android similar? How are they different?
- 2.22 Explain why Java programs running on Android systems do not use the standard Java API and virtual machine.
- 2.23 The experimental Synthesis operating system has an assembler incorporated in the kernel. To optimize system-call performance, the kernel assembles routines within kernel space to minimize the path that the system call must take through the kernel. This approach is the antithesis of the layered approach, in which the path through the kernel is extended to make building the operating system easier. Discuss the pros and cons of the Synthesis approach to kernel design and system-performance optimization.

## Programming Problems

- 2.24** In Section 2.3, we described a program that copies the contents of one file to a destination file. This program works by first prompting the user for the name of the source and destination files. Write this program using either the POSIX or Windows API. Be sure to include all necessary error checking, including ensuring that the source file exists.

Once you have correctly designed and tested the program, if you used a system that supports it, run the program using a utility that traces system calls. Linux systems provide the `strace` utility, and macOS systems use the `dtruss` command. (The `dtruss` command, which actually is a front end to `dtrace`, requires admin privileges, so it must be run using `sudo`.) These tools can be used as follows (assume that the name of the executable file is `FileCopy`):

**Linux:**

```
strace ./FileCopy
```

**macOS:**

```
sudo dtruss ./FileCopy
```

Since Windows systems do not provide such a tool, you will have to trace through the Windows version of this program using a debugger.

## Programming Projects

### Introduction to Linux Kernel Modules

In this project, you will learn how to create a kernel module and load it into the Linux kernel. You will then modify the kernel module so that it creates an entry in the `/proc` file system. The project can be completed using the Linux virtual machine that is available with this text. Although you may use any text editor to write these C programs, you will have to use the *terminal* application to compile the programs, and you will have to enter commands on the command line to manage the modules in the kernel.

As you'll discover, the advantage of developing kernel modules is that it is a relatively easy method of interacting with the kernel, thus allowing you to write programs that directly invoke kernel functions. It is important for you to keep in mind that you are indeed writing *kernel code* that directly interacts with the kernel. That normally means that any errors in the code could crash the system! However, since you will be using a virtual machine, any failures will at worst only require rebooting the system.

## I. Kernel Modules Overview

The first part of this project involves following a series of steps for creating and inserting a module into the Linux kernel.

You can list all kernel modules that are currently loaded by entering the command

```
lsmod
```

This command will list the current kernel modules in three columns: name, size, and where the module is being used.

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

/* This function is called when the module is loaded. */
int simple_init(void)
{
    printk(KERN_INFO "Loading Kernel Module\n");

    return 0;
}

/* This function is called when the module is removed. */
void simple_exit(void)
{
    printk(KERN_INFO "Removing Kernel Module\n");
}

/* Macros for registering module entry and exit points. */
module_init(simple_init);
module_exit(simple_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple Module");
MODULE_AUTHOR("SGG");
```

**Figure 2.21** Kernel module simple.c.

The program in Figure 2.21 (named `simple.c` and available with the source code for this text) illustrates a very basic kernel module that prints appropriate messages when it is loaded and unloaded.

The function `simple_init()` is the **module entry point**, which represents the function that is invoked when the module is loaded into the kernel. Similarly, the `simple_exit()` function is the **module exit point**—the function that is called when the module is removed from the kernel.

The module entry point function must return an integer value, with 0 representing success and any other value representing failure. The module exit point function returns void. Neither the module entry point nor the module exit point is passed any parameters. The two following macros are used for registering the module entry and exit points with the kernel:

```
module_init(simple_init)
module_exit(simple_exit)
```

Notice in the figure how the module entry and exit point functions make calls to the `printk()` function. `printk()` is the kernel equivalent of `printf()`, but its output is sent to a kernel log buffer whose contents can be read by the `dmesg` command. One difference between `printf()` and `printk()` is that `printk()` allows us to specify a priority flag, whose values are given in the `<linux/printk.h>` include file. In this instance, the priority is `KERN_INFO`, which is defined as an *informational* message.

The final lines—`MODULE_LICENSE()`, `MODULE_DESCRIPTION()`, and `MODULE_AUTHOR()`—represent details regarding the software license, description of the module, and author. For our purposes, we do not require this information, but we include it because it is standard practice in developing kernel modules.

This kernel module `simple.c` is compiled using the `Makefile` accompanying the source code with this project. To compile the module, enter the following on the command line:

```
make
```

The compilation produces several files. The file `simple.ko` represents the compiled kernel module. The following step illustrates inserting this module into the Linux kernel.

## II. Loading and Removing Kernel Modules

Kernel modules are loaded using the `insmod` command, which is run as follows:

```
sudo insmod simple.ko
```

To check whether the module has loaded, enter the `lsmod` command and search for the module `simple`. Recall that the module entry point is invoked when the module is inserted into the kernel. To check the contents of this message in the kernel log buffer, enter the command

```
dmesg
```

You should see the message "Loading Module."

Removing the kernel module involves invoking the `rmmod` command (notice that the `.ko` suffix is unnecessary):

```
sudo rmmod simple
```

Be sure to check with the `dmesg` command to ensure the module has been removed.

Because the kernel log buffer can fill up quickly, it often makes sense to clear the buffer periodically. This can be accomplished as follows:

```
sudo dmesg -c
```

Proceed through the steps described above to create the kernel module and to load and unload the module. Be sure to check the contents of the kernel log buffer using `dmesg` to ensure that you have followed the steps properly.

As kernel modules are running within the kernel, it is possible to obtain values and call functions that are available only in the kernel and not to regular user applications. For example, the Linux include file `<linux/hash.h>` defines several hashing functions for use within the kernel. This file also defines the constant value `GOLDEN_RATIO_PRIME` (which is defined as an `unsigned long`). This value can be printed out as follows:

```
printk(KERN_INFO "%lu\n", GOLDEN_RATIO_PRIME);
```

As another example, the include file `<linux/gcd.h>` defines the following function

```
unsigned long gcd(unsigned long a, unsigned b);
```

which returns the greatest common divisor of the parameters `a` and `b`.

Once you are able to correctly load and unload your module, complete the following additional steps:

1. Print out the value of `GOLDEN_RATIO_PRIME` in the `simple_init()` function.
2. Print out the greatest common divisor of 3,300 and 24 in the `simple_exit()` function.

As compiler errors are not often helpful when performing kernel development, it is important to compile your program often by running `make` regularly. Be sure to load and remove the kernel module and check the kernel log buffer using `dmesg` to ensure that your changes to `simple.c` are working properly.

In Section 1.4.3, we described the role of the timer as well as the timer interrupt handler. In Linux, the rate at which the timer ticks (the `tick rate`) is the value `HZ` defined in `<asm/param.h>`. The value of `HZ` determines the frequency of the timer interrupt, and its value varies by machine type and architecture. For example, if the value of `HZ` is 100, a timer interrupt occurs 100 times per second, or every 10 milliseconds. Additionally, the kernel keeps track of the global variable `jiffies`, which maintains the number of timer interrupts that have occurred since the system was booted. The `jiffies` variable is declared in the file `<linux/jiffies.h>`.

1. Print out the values of `jiffies` and `HZ` in the `simple_init()` function.
2. Print out the value of `jiffies` in the `simple_exit()` function.

Before proceeding to the next set of exercises, consider how you can use the different values of `jiffies` in `simple_init()` and `simple_exit()` to determine the number of seconds that have elapsed since the time the kernel module was loaded and then removed.

### III. The /proc File System

The `/proc` file system is a “pseudo” file system that exists only in kernel memory and is used primarily for querying various kernel and per-process statistics.

---

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

#define BUFFER_SIZE 128
#define PROC_NAME "hello"

ssize_t proc_read(struct file *file, char __user *usr_buf,
    size_t count, loff_t *pos);

static struct file_operations proc_ops = {
    .owner = THIS_MODULE,
    .read = proc_read,
};

/* This function is called when the module is loaded. */
int proc_init(void)
{
    /* creates the /proc/hello entry */
    proc_create(PROC_NAME, 0666, NULL, &proc_ops);

    return 0;
}

/* This function is called when the module is removed. */
void proc_exit(void)
{
    /* removes the /proc/hello entry */
    remove_proc_entry(PROC_NAME, NULL);
}
```

---

**Figure 2.22** The `/proc` file-system kernel module, Part 1

This exercise involves designing kernel modules that create additional entries in the `/proc` file system involving both kernel statistics and information related

to specific processes. The entire program is included in Figure 2.22 and Figure 2.23.

We begin by describing how to create a new entry in the /proc file system. The following program example (named `hello.c` and available with the source code for this text) creates a /proc entry named /proc/hello. If a user enters the command

```
cat /proc/hello
```

the infamous Hello World message is returned.

```
/* This function is called each time /proc/hello is read */
ssize_t proc_read(struct file *file, char __user *usr_buf,
    size_t count, loff_t *pos)
{
    int rv = 0;
    char buffer[BUFFER_SIZE];
    static int completed = 0;

    if (completed) {
        completed = 0;
        return 0;
    }

    completed = 1;

    rv = sprintf(buffer, "Hello World\n");

    /* copies kernel space buffer to user space usr_buf */
    copy_to_user(usr_buf, buffer, rv);

    return rv;
}
module_init(proc_init);
module_exit(proc_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Hello Module");
MODULE_AUTHOR("SGG");
```

**Figure 2.23** The /proc file system kernel module, Part 2

In the module entry point `proc_init()`, we create the new /proc/hello entry using the `proc_create()` function. This function is passed `proc_ops`, which contains a reference to a `struct file_operations`. This struct initial-

izes the `.owner` and `.read` members. The value of `.read` is the name of the function `proc_read()` that is to be called whenever `/proc/hello` is read.

Examining this `proc_read()` function, we see that the string "Hello World\n" is written to the variable `buffer` where `buffer` exists in kernel memory. Since `/proc/hello` can be accessed from user space, we must copy the contents of `buffer` to user space using the kernel function `copy_to_user()`. This function copies the contents of kernel memory `buffer` to the variable `usr_buf`, which exists in user space.

Each time the `/proc/hello` file is read, the `proc_read()` function is called repeatedly until it returns 0, so there must be logic to ensure that this function returns 0 once it has collected the data (in this case, the string "Hello World\n") that is to go into the corresponding `/proc/hello` file.

Finally, notice that the `/proc/hello` file is removed in the module exit point `proc_exit()` using the function `remove_proc_entry()`.

## IV. Assignment

This assignment will involve designing two kernel modules:

1. Design a kernel module that creates a `/proc` file named `/proc/jiffies` that reports the current value of `jiffies` when the `/proc/jiffies` file is read, such as with the command

```
cat /proc/jiffies
```

Be sure to remove `/proc/jiffies` when the module is removed.

2. Design a kernel module that creates a `proc` file named `/proc/seconds` that reports the number of elapsed seconds since the kernel module was loaded. This will involve using the value of `jiffies` as well as the `HZ` rate. When a user enters the command

```
cat /proc/seconds
```

your kernel module will report the number of seconds that have elapsed since the kernel module was first loaded. Be sure to remove `/proc/seconds` when the module is removed.

# Processes



Early computers allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, contemporary computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various programs; and these needs resulted in the notion of a **process**, which is a program in execution. A process is the unit of work in a modern computing system.

The more complex the operating system is, the more it is expected to do on behalf of its users. Although its main concern is the execution of user programs, it also needs to take care of various system tasks that are best done in user space, rather than within the kernel. A system therefore consists of a collection of processes, some executing user code, others executing operating system code. Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them. In this chapter, you will read about what processes are, how they are represented in an operating system, and how they work.

## CHAPTER OBJECTIVES

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that use pipes and POSIX shared memory to perform interprocess communication.
- Describe client–server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.

## 3.1 Process Concept

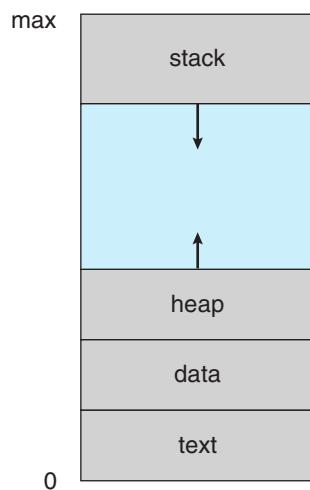
A question that arises in discussing operating systems involves what to call all the CPU activities. Early computers were batch systems that executed **jobs**, followed by the emergence of time-shared systems that ran **user programs**, or **tasks**. Even on a single-user system, a user may be able to run several programs at one time: a word processor, a web browser, and an e-mail package. And even if a computer can execute only one program at a time, such as on an embedded device that does not support multitasking, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them **processes**.

Although we personally prefer the more contemporary term *process*, the term *job* has historical significance, as much of operating system theory and terminology was developed during a time when the major activity of operating systems was job processing. Therefore, in some appropriate instances we use *job* when describing the role of the operating system. As an example, it would be misleading to avoid the use of commonly accepted terms that include the word *job* (such as *job scheduling*) simply because *process* has superseded *job*.

### 3.1.1 The Process

Informally, as mentioned earlier, a process is a program in execution. The status of the current activity of a process is represented by the value of the **program counter** and the contents of the processor's registers. The memory layout of a process is typically divided into multiple sections, and is shown in Figure 3.1. These sections include:

- **Text section**—the executable code
- **Data section**—global variables



**Figure 3.1** Layout of a process in memory.

- **Heap section**—memory that is dynamically allocated during program run time
- **Stack section**—temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

Notice that the sizes of the text and data sections are fixed, as their sizes do not change during program run time. However, the stack and heap sections can shrink and grow dynamically during program execution. Each time a function is called, an **activation record** containing function parameters, local variables, and the return address is pushed onto the stack; when control is returned from the function, the activation record is popped from the stack. Similarly, the heap will grow as memory is dynamically allocated, and will shrink when memory is returned to the system. Although the stack and heap sections grow *toward* one another, the operating system must ensure they do not *overlap* one another.

We emphasize that a program by itself is not a process. A program is a *passive* entity, such as a file containing a list of instructions stored on disk (often called an **executable file**). In contrast, a process is an *active* entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in `prog.exe` or `a.out`).

Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary. It is also common to have a process that spawns many processes as it runs. We discuss such matters in Section 3.4.

Note that a process can itself be an execution environment for other code. The Java programming environment provides a good example. In most circumstances, an executable Java program is executed within the Java virtual machine (JVM). The JVM executes as a process that interprets the loaded Java code and takes actions (via native machine instructions) on behalf of that code. For example, to run the compiled Java program `Program.class`, we would enter

```
java Program
```

The command `java` runs the JVM as an ordinary process, which in turns executes the Java program `Program` in the virtual machine. The concept is the same as simulation, except that the code, instead of being written for a different instruction set, is written in the Java language.

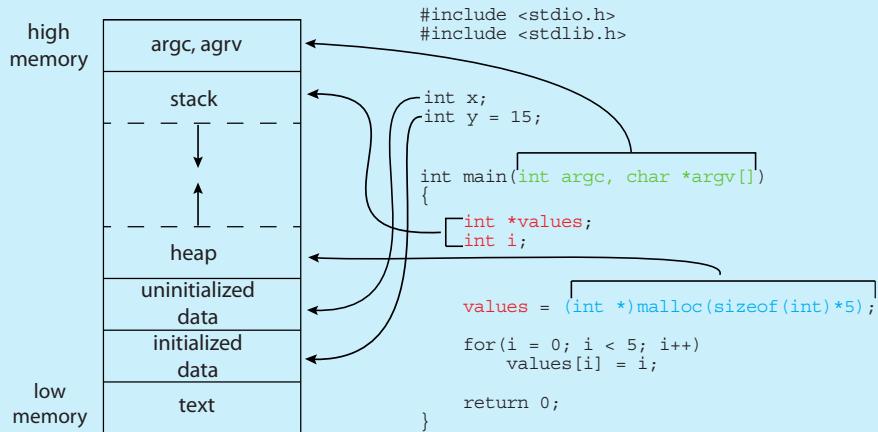
### 3.1.2 Process State

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

### MEMORY LAYOUT OF A C PROGRAM

The figure shown below illustrates the layout of a C program in memory, highlighting how the different sections of a process relate to an actual C program. This figure is similar to the general concept of a process in memory as shown in Figure 3.1, with a few differences:

- The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.
- A separate section is provided for the `argc` and `argv` parameters passed to the `main()` function.

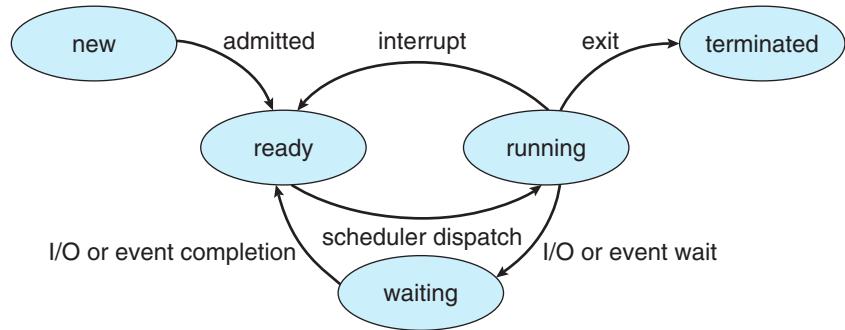


The GNU `size` command can be used to determine the size (in bytes) of some of these sections. Assuming the name of the executable file of the above C program is `memory`, the following is the output generated by entering the command `size memory`:

text	data	bss	dec	hex	filename
1158	284	8	1450	5aa	memory

The `data` field refers to uninitialized data, and `bss` refers to initialized data. (`bss` is a historical term referring to *block started by symbol*.) The `dec` and `hex` values are the sum of the three sections represented in decimal and hexadecimal, respectively.

- **New.** The process is being created.
- **Running.** Instructions are being executed.
- **Waiting.** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready.** The process is waiting to be assigned to a processor.



**Figure 3.2** Diagram of process state.

- **Terminated.** The process has finished execution.

These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be *running* on any processor core at any instant. Many processes may be *ready* and *waiting*, however. The state diagram corresponding to these states is presented in Figure 3.2.

### 3.1.3 Process Control Block

Each process is represented in the operating system by a **process control block (PCB)**—also called a **task control block**. A PCB is shown in Figure 3.3. It contains many pieces of information associated with a specific process, including these:

- **Process state.** The state may be new, ready, running, waiting, halted, and so on.
- **Program counter.** The counter indicates the address of the next instruction to be executed for this process.



**Figure 3.3** Process control block (PCB).

- **CPU registers.** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.
- **CPU-scheduling information.** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. (Chapter 5 describes process scheduling.)
- **Memory-management information.** This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system (Chapter 9).
- **Accounting information.** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information.** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

In brief, the PCB simply serves as the repository for all the data needed to start, or restart, a process, along with some accounting data.

#### 3.1.4 Threads

The process model discussed so far has implied that a process is a program that performs a single **thread** of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time. Thus, the user cannot simultaneously type in characters and run the spell checker. Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. This feature is especially beneficial on multicore systems, where multiple threads can run in parallel. A multithreaded word processor could, for example, assign one thread to manage user input while another thread runs the spell checker. On systems that support threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads. Chapter 4 explores threads in detail.

## 3.2 Process Scheduling

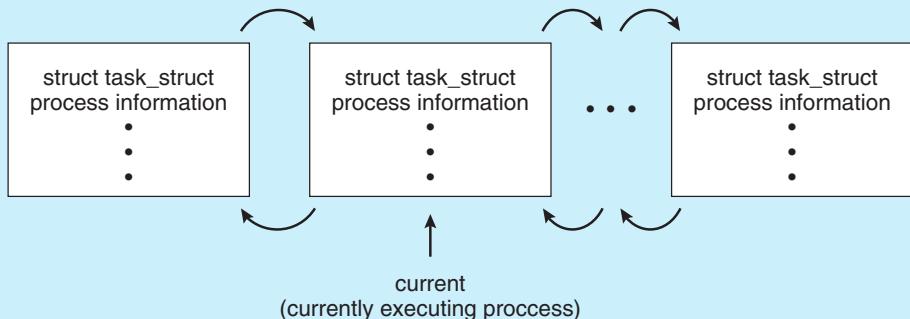
The objective of multiprogramming is to have some process running at all times so as to maximize CPU utilization. The objective of time sharing is to switch a CPU core among processes so frequently that users can interact with each program while it is running. To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on a core. Each CPU core can run one process at a time.

### PROCESS REPRESENTATION IN LINUX

The process control block in the Linux operating system is represented by the C structure `task_struct`, which is found in the `<include/linux/sched.h>` include file in the kernel source-code directory. This structure contains all the necessary information for representing a process, including the state of the process, scheduling and memory-management information, list of open files, and pointers to the process's parent and a list of its children and siblings. (A process's **parent** is the process that created it; its **children** are any processes that it creates. Its **siblings** are children with the same parent process.) Some of these fields include:

```
long state;           /* state of the process */
struct sched_entity se; /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space */
```

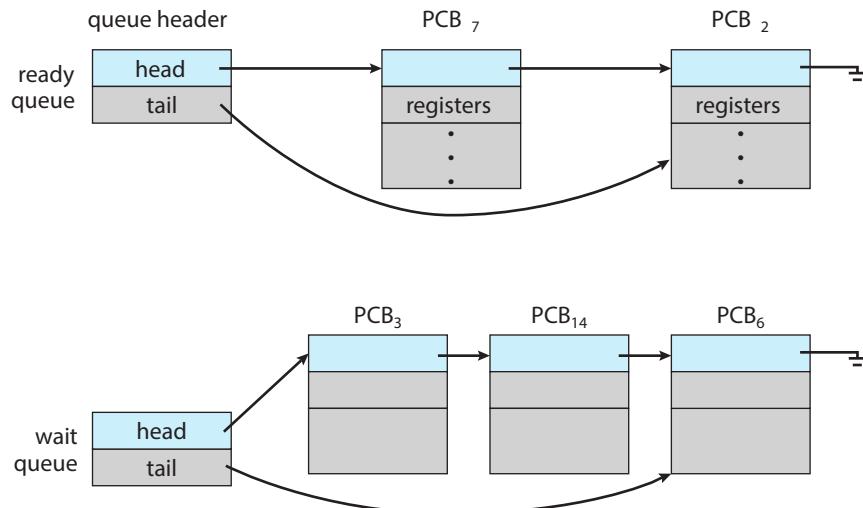
For example, the state of a process is represented by the field `long state` in this structure. Within the Linux kernel, all active processes are represented using a doubly linked list of `task_struct`. The kernel maintains a pointer—`current`—to the process currently executing on the system, as shown below:



As an illustration of how the kernel might manipulate one of the fields in the `task_struct` for a specified process, let's assume the system would like to change the state of the process currently running to the value `new_state`. If `current` is a pointer to the process currently executing, its state is changed with the following:

```
current->state = new_state;
```

For a system with a single CPU core, there will never be more than one process running at a time, whereas a multicore system can run multiple processes at one time. If there are more processes than cores, excess processes will have



**Figure 3.4** The ready queue and wait queues.

to wait until a core is free and can be rescheduled. The number of processes currently in memory is known as the **degree of multiprogramming**.

Balancing the objectives of multiprogramming and time sharing also requires taking the general behavior of a process into account. In general, most processes can be described as either I/O bound or CPU bound. An **I/O-bound process** is one that spends more of its time doing I/O than it spends doing computations. A **CPU-bound process**, in contrast, generates I/O requests infrequently, using more of its time doing computations.

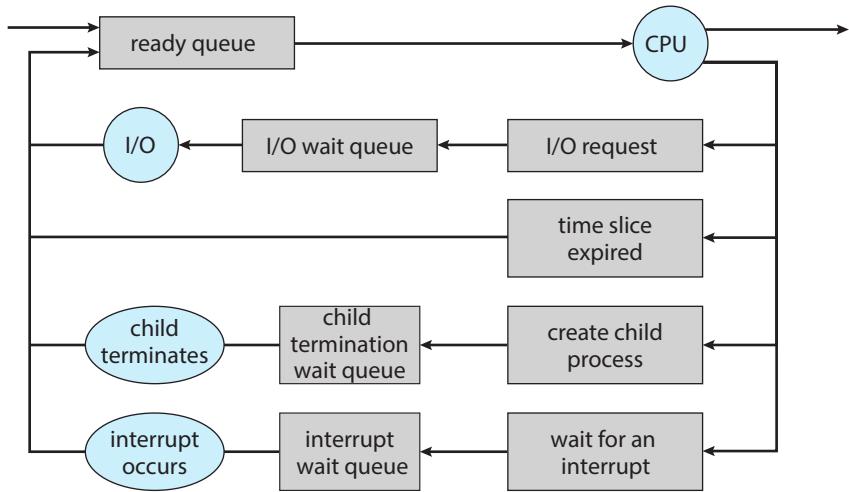
### 3.2.1 Scheduling Queues

As processes enter the system, they are put into a **ready queue**, where they are ready and waiting to execute on a CPU's core. This queue is generally stored as a linked list; a ready-queue header contains pointers to the first PCB in the list, and each PCB includes a pointer field that points to the next PCB in the ready queue.

The system also includes other queues. When a process is allocated a CPU core, it executes for a while and eventually terminates, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. Suppose the process makes an I/O request to a device such as a disk. Since devices run significantly slower than processors, the process will have to wait for the I/O to become available. Processes that are waiting for a certain event to occur — such as completion of I/O — are placed in a **wait queue** (Figure 3.4).

A common representation of process scheduling is a **queueing diagram**, such as that in Figure 3.5. Two types of queues are present: the ready queue and a set of wait queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits there until it is selected for execution, or **dispatched**. Once the process is allocated a CPU core and is executing, one of several events could occur:



**Figure 3.5** Queueing-diagram representation of process scheduling.

- The process could issue an I/O request and then be placed in an I/O wait queue.
- The process could create a new child process and then be placed in a wait queue while it awaits the child's termination.
- The process could be removed forcibly from the core, as a result of an interrupt or having its time slice expire, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

### 3.2.2 CPU Scheduling

A process migrates among the ready queue and various wait queues throughout its lifetime. The role of the **CPU scheduler** is to select from among the processes that are in the ready queue and allocate a CPU core to one of them. The CPU scheduler must select a new process for the CPU frequently. An I/O-bound process may execute for only a few milliseconds before waiting for an I/O request. Although a CPU-bound process will require a CPU core for longer durations, the scheduler is unlikely to grant the core to a process for an extended period. Instead, it is likely designed to forcibly remove the CPU from a process and schedule another process to run. Therefore, the CPU scheduler executes at least once every 100 milliseconds, although typically much more frequently.

Some operating systems have an intermediate form of scheduling, known as **swapping**, whose key idea is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is known as *swapping* because a process can be “swapped out”

from memory to disk, where its current status is saved, and later “swapped in” from disk back to memory, where its status is restored. Swapping is typically only necessary when memory has been overcommitted and must be freed up. Swapping is discussed in Chapter 9.

### 3.2.3 Context Switch

As mentioned in Section 1.2.1, interrupts cause the operating system to change a CPU core from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems. When an interrupt occurs, the system needs to save the current **context** of the process running on the CPU core so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state (see Figure 3.2), and memory-management information. Generically, we perform a **state save** of the current state of the CPU core, be it in kernel or user mode, and then a **state restore** to resume operations.

Switching the CPU core to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch** and is illustrated in Figure 3.6. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. Switching speed varies from machine to machine, depending on the

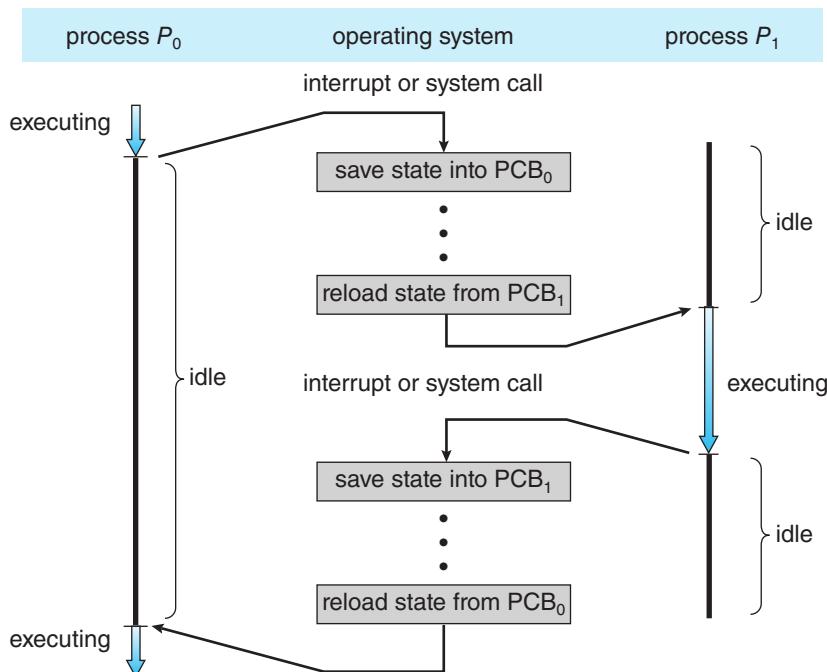


Figure 3.6 Diagram showing context switch from process to process.

### MULTITASKING IN MOBILE SYSTEMS

Because of the constraints imposed on mobile devices, early versions of iOS did not provide user-application multitasking; only one application ran in the foreground while all other user applications were suspended. Operating-system tasks were multitasked because they were written by Apple and well behaved. However, beginning with iOS 4, Apple provided a limited form of multitasking for user applications, thus allowing a single foreground application to run concurrently with multiple background applications. (On a mobile device, the **foreground** application is the application currently open and appearing on the display. The **background** application remains in memory, but does not occupy the display screen.) The iOS 4 programming API provided support for multitasking, thus allowing a process to run in the background without being suspended. However, it was limited and only available for a few application types. As hardware for mobile devices began to offer larger memory capacities, multiple processing cores, and greater battery life, subsequent versions of iOS began to support richer functionality for multitasking with fewer restrictions. For example, the larger screen on iPad tablets allowed running two foreground apps at the same time, a technique known as **split-screen**.

Since its origins, Android has supported multitasking and does not place constraints on the types of applications that can run in the background. If an application requires processing while in the background, the application must use a **service**, a separate application component that runs on behalf of the background process. Consider a streaming audio application: if the application moves to the background, the service continues to send audio data to the audio device driver on behalf of the background application. In fact, the service will continue to run even if the background application is suspended. Services do not have a user interface and have a small memory footprint, thus providing an efficient technique for multitasking in a mobile environment.

memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a several microseconds.

Context-switch times are highly dependent on hardware support. For instance, some processors provide multiple sets of registers. A context switch here simply requires changing the pointer to the current register set. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the greater the amount of work that must be done during a context switch. As we will see in Chapter 9, advanced memory-management techniques may require that extra data be switched with each context. For instance, the address space of the current process must be preserved as the space of the next task is prepared for use. How the address space is preserved, and what amount of work is needed to preserve it, depend on the memory-management method of the operating system.

### 3.3 Operations on Processes

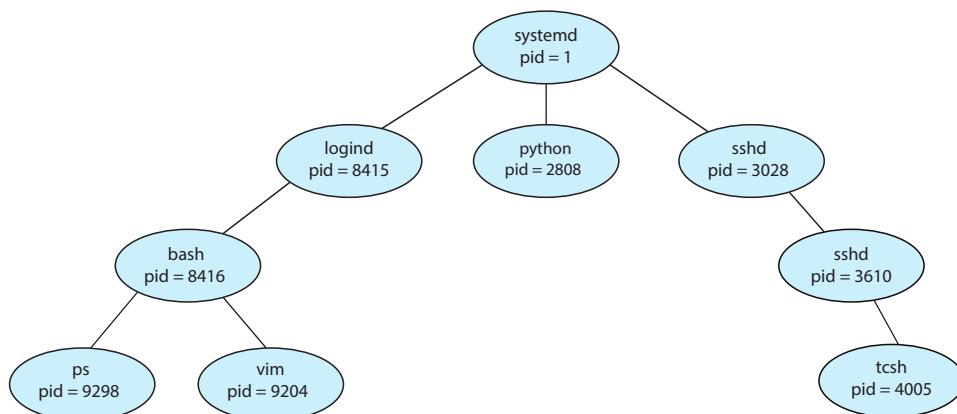
The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination. In this section, we explore the mechanisms involved in creating processes and illustrate process creation on UNIX and Windows systems.

#### 3.3.1 Process Creation

During the course of execution, a process may create several new processes. As mentioned earlier, the creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a **tree** of processes.

Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number. The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.

Figure 3.7 illustrates a typical process tree for the Linux operating system, showing the name of each process and its pid. (We use the term *process* rather loosely in this situation, as Linux prefers the term *task* instead.) The `systemd` process (which always has a pid of 1) serves as the root parent process for all user processes, and is the first user process created when the system boots. Once the system has booted, the `systemd` process creates processes which provide additional services such as a web or print server, an ssh server, and the like. In Figure 3.7, we see two children of `systemd`—`logind` and `sshd`. The `logind` process is responsible for managing clients that directly log onto the system. In this example, a client has logged on and is using the `bash` shell, which has been assigned pid 8416. Using the `bash` command-line interface, this user has created the process `ps` as well as the `vim` editor. The `sshd` process is responsible for managing clients that connect to the system by using ssh (which is short for *secure shell*).



**Figure 3.7** A tree of processes on a typical Linux system.

### *THE init AND systemd PROCESSES*

Traditional UNIX systems identify the process `init` as the root of all child processes. `init` (also known as **System V init**) is assigned a pid of 1, and is the first process created when the system is booted. On a process tree similar to what is shown in Figure 3.7, `init` is at the root.

Linux systems initially adopted the System V `init` approach, but recent distributions have replaced it with `systemd`. As described in Section 3.3.1, `systemd` serves as the system's initial process, much the same as System V `init`; however it is much more flexible, and can provide more services, than `init`.

On UNIX and Linux systems, we can obtain a listing of processes by using the `ps` command. For example, the command

```
ps -el
```

will list complete information for all processes currently active in the system. A process tree similar to the one shown in Figure 3.7 can be constructed by recursively tracing parent processes all the way to the `systemd` process. (In addition, Linux systems provide the `pstree` command, which displays a tree of all processes in the system.)

In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.

In addition to supplying various physical and logical resources, the parent process may pass along initialization data (input) to the child process. For example, consider a process whose function is to display the contents of a file—say, `hw1.c`—on the screen of a terminal. When the process is created, it will get, as an input from its parent process, the name of the file `hw1.c`. Using that file name, it will open the file and write the contents out. It may also get the name of the output device. Alternatively, some operating systems pass resources to child processes. On such a system, the new process may get two open files, `hw1.c` and the terminal device, and may simply transfer the datum between the two.

When a process creates a new process, two possibilities for execution exist:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two address-space possibilities for the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

To illustrate these differences, let's first consider the UNIX operating system. In UNIX, as we've seen, each process is identified by its process identifier, which is a unique integer. A new process is created by the `fork()` system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the `fork()`, with one difference: the return code for the `fork()` is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

After a `fork()` system call, one of the two processes typically uses the `exec()` system call to replace the process's memory space with a new program. The `exec()` system call loads a binary file into memory (destroying the memory image of the program containing the `exec()` system call) and starts

---

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

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

---

**Figure 3.8** Creating a separate process using the UNIX `fork()` system call.

its execution. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a `wait()` system call to move itself off the ready queue until the termination of the child. Because the call to `exec()` overlays the process's address space with a new program, `exec()` does not return control unless an error occurs.

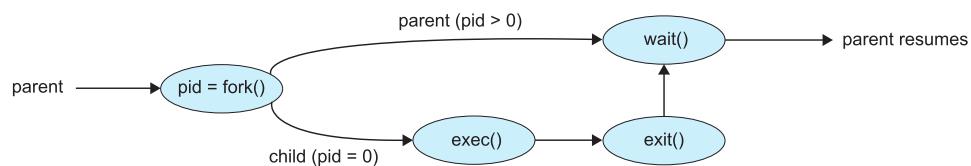
The C program shown in Figure 3.8 illustrates the UNIX system calls previously described. We now have two different processes running copies of the same program. The only difference is that the value of the variable `pid` for the child process is zero, while that for the parent is an integer value greater than zero (in fact, it is the actual `pid` of the child process). The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files. The child process then overlays its address space with the UNIX command `/bin/ls` (used to get a directory listing) using the `execvp()` system call (`execvp()` is a version of the `exec()` system call). The parent waits for the child process to complete with the `wait()` system call. When the child process completes (by either implicitly or explicitly invoking `exit()`), the parent process resumes from the call to `wait()`, where it completes using the `exit()` system call. This is also illustrated in Figure 3.9.

Of course, there is nothing to prevent the child from *not* invoking `exec()` and instead continuing to execute as a copy of the parent process. In this scenario, the parent and child are concurrent processes running the same code instructions. Because the child is a copy of the parent, each process has its own copy of any data.

As an alternative example, we next consider process creation in Windows. Processes are created in the Windows API using the `CreateProcess()` function, which is similar to `fork()` in that a parent creates a new child process. However, whereas `fork()` has the child process inheriting the address space of its parent, `CreateProcess()` requires loading a specified program into the address space of the child process at process creation. Furthermore, whereas `fork()` is passed no parameters, `CreateProcess()` expects no fewer than ten parameters.

The C program shown in Figure 3.10 illustrates the `CreateProcess()` function, which creates a child process that loads the application `mspaint.exe`. We opt for many of the default values of the ten parameters passed to `CreateProcess()`. Readers interested in pursuing the details of process creation and management in the Windows API are encouraged to consult the bibliographical notes at the end of this chapter.

The two parameters passed to the `CreateProcess()` function are instances of the `STARTUPINFO` and `PROCESS_INFORMATION` structures. `STARTUPINFO` specifies many properties of the new process, such as window



**Figure 3.9** Process creation using the `fork()` system call.

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

/* allocate memory */
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));

/* create child process */
if (!CreateProcess(NULL, /* use command line */
    "C:\\\\WINDOWS\\\\system32\\\\mspaint.exe", /* command */
    NULL, /* don't inherit process handle */
    NULL, /* don't inherit thread handle */
    FALSE, /* disable handle inheritance */
    0, /* no creation flags */
    NULL, /* use parent's environment block */
    NULL, /* use parent's existing directory */
    &si,
    &pi))
{
    fprintf(stderr, "Create Process Failed");
    return -1;
}
/* parent will wait for the child to complete */
WaitForSingleObject(pi.hProcess, INFINITE);
printf("Child Complete");

/* close handles */
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
}
```

---

**Figure 3.10** Creating a separate process using the Windows API.

size and appearance and handles to standard input and output files. The PROCESS\_INFORMATION structure contains a handle and the identifiers to the newly created process and its thread. We invoke the ZeroMemory() function to allocate memory for each of these structures before proceeding with CreateProcess().

The first two parameters passed to CreateProcess() are the application name and command-line parameters. If the application name is NULL (as it is in this case), the command-line parameter specifies the application to load.

In this instance, we are loading the Microsoft Windows `mspaint.exe` application. Beyond these two initial parameters, we use the default parameters for inheriting process and thread handles as well as specifying that there will be no creation flags. We also use the parent's existing environment block and starting directory. Last, we provide two pointers to the `STARTUPINFO` and `PROCESS_INFORMATION` structures created at the beginning of the program. In Figure 3.8, the parent process waits for the child to complete by invoking the `wait()` system call. The equivalent of this in Windows is `WaitForSingleObject()`, which is passed a handle of the child process—`pi.hProcess`—and waits for this process to complete. Once the child process exits, control returns from the `WaitForSingleObject()` function in the parent process.

### 3.3.2 Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the `exit()` system call. At that point, the process may return a status value (typically an integer) to its waiting parent process (via the `wait()` system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated and reclaimed by the operating system.

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, `TerminateProcess()` in Windows). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, a user—or a misbehaving application—could arbitrarily kill another user's processes. Note that a parent needs to know the identities of its children if it is to terminate them. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

To illustrate process execution and termination, consider that, in Linux and UNIX systems, we can terminate a process by using the `exit()` system call, providing an exit status as a parameter:

```
/* exit with status 1 */
exit(1);
```

In fact, under normal termination, `exit()` will be called either directly (as shown above) or indirectly, as the C run-time library (which is added to UNIX executable files) will include a call to `exit()` by default.

A parent process may wait for the termination of a child process by using the `wait()` system call. The `wait()` system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

```
pid_t pid;
int status;

pid = wait(&status);
```

When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls `wait()`, because the process table contains the process's exit status. A process that has terminated, but whose parent has not yet called `wait()`, is known as a **zombie** process. All processes transition to this state when they terminate, but generally they exist as zombies only briefly. Once the parent calls `wait()`, the process identifier of the zombie process and its entry in the process table are released.

Now consider what would happen if a parent did not invoke `wait()` and instead terminated, thereby leaving its child processes as **orphans**. Traditional UNIX systems addressed this scenario by assigning the `init` process as the new parent to orphan processes. (Recall from Section 3.3.1 that `init` serves as the root of the process hierarchy in UNIX systems.) The `init` process periodically invokes `wait()`, thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

Although most Linux systems have replaced `init` with `systemd`, the latter process can still serve the same role, although Linux also allows processes other than `systemd` to inherit orphan processes and manage their termination.

### 3.3.2.1 Android Process Hierarchy

Because of resource constraints such as limited memory, mobile operating systems may have to terminate existing processes to reclaim limited system resources. Rather than terminating an arbitrary process, Android has identified an ***importance hierarchy*** of processes, and when the system must terminate a process to make resources available for a new, or more important, process, it terminates processes in order of increasing importance. From most to least important, the hierarchy of process classifications is as follows:

- **Foreground process**—The current process visible on the screen, representing the application the user is currently interacting with
- **Visible process**—A process that is not directly visible on the foreground but that is performing an activity that the foreground process is referring to (that is, a process performing an activity whose status is displayed on the foreground process)

# Main Memory



In Chapter 5, we showed how the CPU can be shared by a set of processes. As a result of CPU scheduling, we can improve both the utilization of the CPU and the speed of the computer's response to its users. To realize this increase in performance, however, we must keep many processes in memory—that is, we must share memory.

In this chapter, we discuss various ways to manage memory. The memory-management algorithms vary from a primitive bare-machine approach to a strategy that uses paging. Each approach has its own advantages and disadvantages. Selection of a memory-management method for a specific system depends on many factors, especially on the hardware design of the system. As we shall see, most algorithms require hardware support, leading many systems to have closely integrated hardware and operating-system memory management.

## CHAPTER OBJECTIVES

- Explain the difference between a logical and a physical address and the role of the memory management unit (MMU) in translating addresses.
- Apply first-, best-, and worst-fit strategies for allocating memory contiguously.
- Explain the distinction between internal and external fragmentation.
- Translate logical to physical addresses in a paging system that includes a translation look-aside buffer (TLB).
- Describe hierarchical paging, hashed paging, and inverted page tables.
- Describe address translation for IA-32, x86-64, and ARMv8 architectures.

### 9.1 Background

As we saw in Chapter 1, memory is central to the operation of a modern computer system. Memory consists of a large array of bytes, each with its own address. The CPU fetches instructions from memory according to the value of

the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore *how* a program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program.

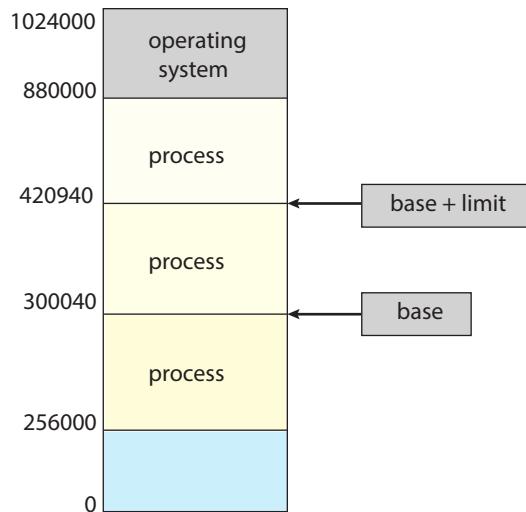
We begin our discussion by covering several issues that are pertinent to managing memory: basic hardware, the binding of symbolic (or virtual) memory addresses to actual physical addresses, and the distinction between logical and physical addresses. We conclude the section with a discussion of dynamic linking and shared libraries.

### 9.1.1 Basic Hardware

Main memory and the registers built into each processing core are the only general-purpose storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them.

Registers that are built into each CPU core are generally accessible within one cycle of the CPU clock. Some CPU cores can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick. The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Completing a memory access may take many cycles of the CPU clock. In such cases, the processor normally needs to **stall**, since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory, typically on the CPU chip for fast access. Such a **cache** was described in Section 1.5.5. To manage a cache built into the CPU, the hardware automatically speeds up memory access without any operating-system control. (Recall from Section 5.5.2 that during a memory stall, a multithreaded core can switch from the stalled hardware thread to another hardware thread.)

Not only are we concerned with the relative speed of accessing physical memory, but we also must ensure correct operation. For proper system operation, we must protect the operating system from access by user processes, as well as protect user processes from one another. This protection must be provided by the hardware, because the operating system doesn't usually intervene between the CPU and its memory accesses (because of the resulting performance penalty). Hardware implements this protection in several different ways, as we show throughout the chapter. Here, we outline one possible implementation.



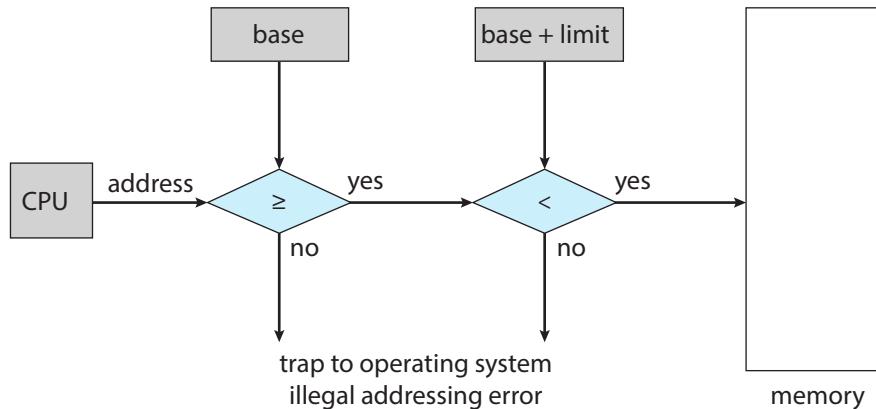
**Figure 9.1** A base and a limit register define a logical address space.

We first need to make sure that each process has a separate memory space. Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution. To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit, as illustrated in Figure 9.1. The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range. For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error (Figure 9.2). This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers. This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.

The operating system, executing in kernel mode, is given unrestricted access to both operating-system memory and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, to perform I/O to and from user memory, and to provide many other services. Consider, for example, that an operating system for a



**Figure 9.2** Hardware address protection with base and limit registers.

multiprocessing system must execute context switches, storing the state of one process from the registers into main memory before loading the next process's context from main memory into the registers.

### 9.1.2 Address Binding

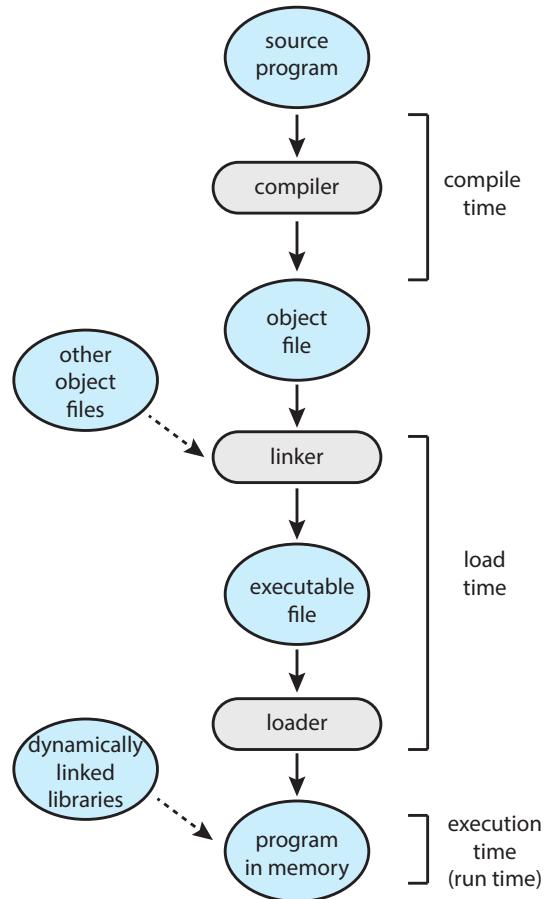
Usually, a program resides on a disk as a binary executable file. To run, the program must be brought into memory and placed within the context of a process (as described in Section 2.5), where it becomes eligible for execution on an available CPU. As the process executes, it accesses instructions and data from memory. Eventually, the process terminates, and its memory is reclaimed for use by other processes.

Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer may start at 00000, the first address of the user process need not be 00000. You will see later how the operating system actually places a process in physical memory.

In most cases, a user program goes through several steps—some of which may be optional—before being executed (Figure 9.3). Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as the variable count). A compiler typically **binds** these symbolic addresses to relocatable addresses (such as “14 bytes from the beginning of this module”). The linker or loader (see Section 2.5) in turn binds the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.

Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

- **Compile time.** If you know at compile time where the process will reside in memory, then **absolute code** can be generated. For example, if you know that a user process will reside starting at location *R*, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.



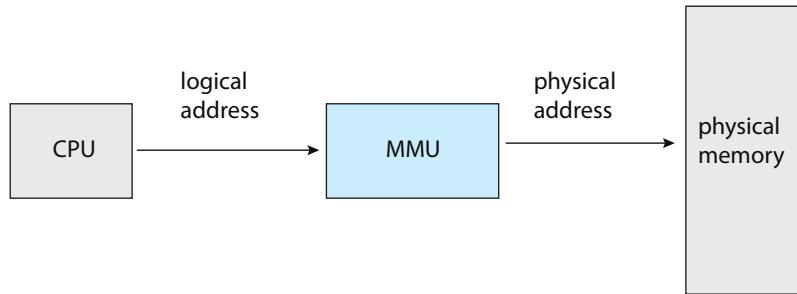
**Figure 9.3** Multistep processing of a user program.

- **Load time.** If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code**. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.
- **Execution time.** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work, as will be discussed in Section 9.1.3. Most operating systems use this method.

A major portion of this chapter is devoted to showing how these various bindings can be implemented effectively in a computer system and to discussing appropriate hardware support.

### 9.1.3 Logical Versus Physical Address Space

An address generated by the CPU is commonly referred to as a **logical address**, whereas an address seen by the memory unit—that is, the one loaded into

**Figure 9.4** Memory management unit (MMU).

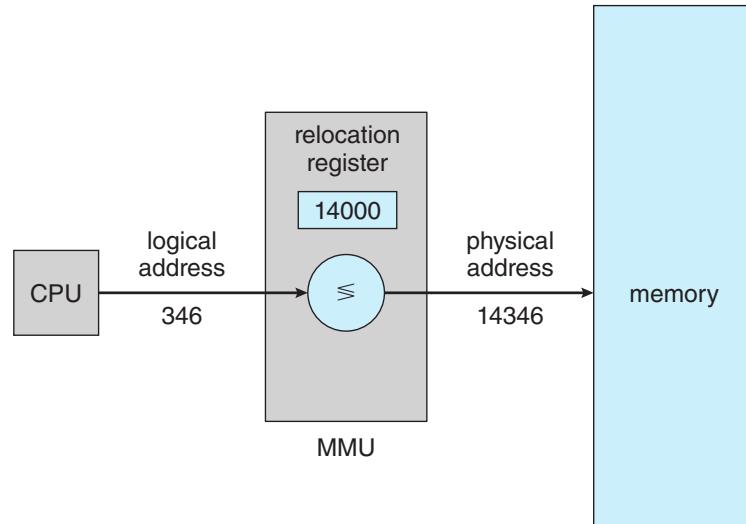
the **memory-address register** of the memory—is commonly referred to as a **physical address**.

Binding addresses at either compile or load time generates identical logical and physical addresses. However, the execution-time address-binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address**. We use ***logical address*** and ***virtual address*** interchangeably in this text. The set of all logical addresses generated by a program is a **logical address space**. The set of all physical addresses corresponding to these logical addresses is a **physical address space**. Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.

The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU)** (Figure 9.4). We can choose from many different methods to accomplish such mapping, as we discuss in Section 9.2 through Section 9.3. For the time being, we illustrate this mapping with a simple MMU scheme that is a generalization of the base-register scheme described in Section 9.1.1. The base register is now called a **relocation register**. The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory (see Figure 9.5). For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.

The user program never accesses the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses—all as the number 346. Only when it is used as a memory address (in an indirect load or store, perhaps) is it relocated relative to the base register. The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses. This form of execution-time binding was discussed in Section 9.1.2. The final location of a referenced memory address is not determined until the reference is made.

We now have two different types of addresses: logical addresses (in the range 0 to *max*) and physical addresses (in the range  $R + 0$  to  $R + \text{max}$  for a base value  $R$ ). The user program generates only logical addresses and thinks that the process runs in memory locations from 0 to *max*. However, these logical addresses must be mapped to physical addresses before they are used. The



**Figure 9.5** Dynamic relocation using a relocation register.

concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

#### 9.1.4 Dynamic Loading

In our discussion so far, it has been necessary for the entire program and all data of a process to be in physical memory for the process to execute. The size of a process has thus been limited to the size of physical memory. To obtain better memory-space utilization, we can use **dynamic loading**. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.

The advantage of dynamic loading is that a routine is loaded only when it is needed. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In such a situation, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.

Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a method. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

#### 9.1.5 Dynamic Linking and Shared Libraries

**Dynamically linked libraries (DLLs)** are system libraries that are linked to user programs when the programs are run (refer back to Figure 9.3). Some operating systems support only **static linking**, in which system libraries are treated

like any other object module and are combined by the loader into the binary program image. Dynamic linking, in contrast, is similar to dynamic loading. Here, though, linking, rather than loading, is postponed until execution time. This feature is usually used with system libraries, such as the standard C language library. Without this facility, each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable image. This requirement not only increases the size of an executable image but also may waste main memory. A second advantage of DLLs is that these libraries can be shared among multiple processes, so that only one instance of the DLL in main memory. For this reason, DLLs are also known as **shared libraries**, and are used extensively in Windows and Linux systems.

When a program references a routine that is in a dynamic library, the loader locates the DLL, loading it into memory if necessary. It then adjusts addresses that reference functions in the dynamic library to the location in memory where the DLL is stored.

Dynamically linked libraries can be extended to library updates (such as bug fixes). In addition, a library may be replaced by a new version, and all programs that reference the library will automatically use the new version. Without dynamic linking, all such programs would need to be relinked to gain access to the new library. So that programs will not accidentally execute new, incompatible versions of libraries, version information is included in both the program and the library. More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use. Versions with minor changes retain the same version number, whereas versions with major changes increment the number. Thus, only programs that are compiled with the new library version are affected by any incompatible changes incorporated in it. Other programs linked before the new library was installed will continue using the older library.

Unlike dynamic loading, dynamic linking and shared libraries generally require help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses. We elaborate on this concept, as well as how DLLs can be shared by multiple processes, when we discuss paging in Section 9.3.4.

## 9.2 Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. This section explains one early method, contiguous memory allocation.

The memory is usually divided into two partitions: one for the operating system and one for the user processes. We can place the operating system in either low memory addresses or high memory addresses. This decision depends on many factors, such as the location of the interrupt vector. However, many operating systems (including Linux and Windows) place the operating system in high memory, and therefore we discuss only that situation.

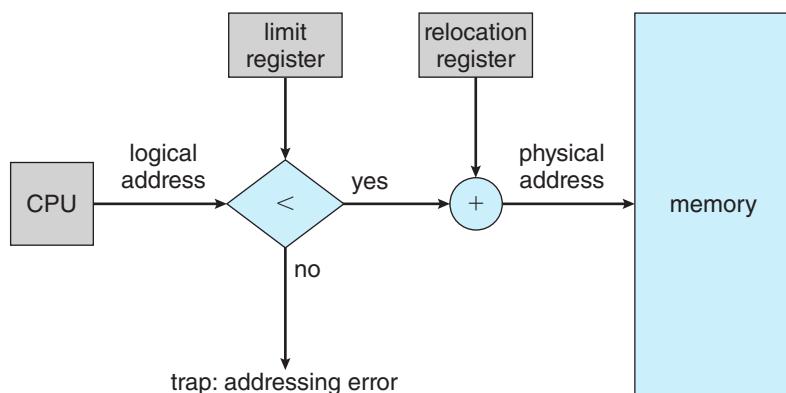
We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are waiting to be brought into memory. In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process. Before discussing this memory allocation scheme further, though, we must address the issue of memory protection.

### 9.2.1 Memory Protection

We can prevent a process from accessing memory that it does not own by combining two ideas previously discussed. If we have a system with a relocation register (Section 9.1.3), together with a limit register (Section 9.1.1), we accomplish our goal. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600). Each logical address must fall within the range specified by the limit register. The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory (Figure 9.6).

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically. This flexibility is desirable in many situations. For example, the operating system contains code and buffer space for device drivers. If a device driver is not currently in use, it makes little sense to keep it in memory; instead, it can be loaded into memory only when it is needed. Likewise, when the device driver is no longer needed, it can be removed and its memory allocated for other needs.



**Figure 9.6** Hardware support for relocation and limit registers.

### 9.2.2 Memory Allocation

Now we are ready to turn to memory allocation. One of the simplest methods of allocating memory is to assign processes to variably sized partitions in memory, where each partition may contain exactly one process. In this **variable-partition** scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a **hole**. Eventually, as you will see, memory contains a set of holes of various sizes.

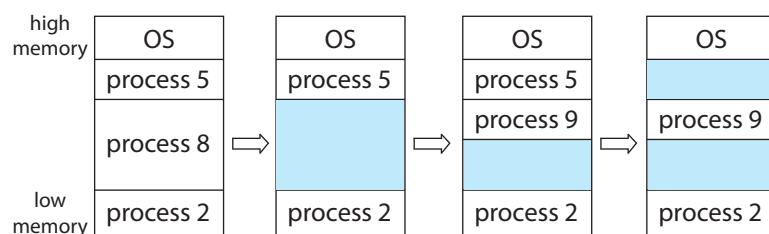
Figure 9.7 depicts this scheme. Initially, the memory is fully utilized, containing processes 5, 8, and 2. After process 8 leaves, there is one contiguous hole. Later on, process 9 arrives and is allocated memory. Then process 5 departs, resulting in two noncontiguous holes.

As processes enter the system, the operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory, where it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then provide to another process.

What happens when there isn't sufficient memory to satisfy the demands of an arriving process? One option is to simply reject the process and provide an appropriate error message. Alternatively, we can place such processes into a wait queue. When memory is later released, the operating system checks the wait queue to determine if it will satisfy the memory demands of a waiting process.

In general, as mentioned, the memory blocks available comprise a *set* of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

This procedure is a particular instance of the general **dynamic storage-allocation problem**, which concerns how to satisfy a request of size  $n$  from a list of free holes. There are many solutions to this problem. The **first-fit**, **best-fit**, and **worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes.



**Figure 9.7** Variable partition.

- **First fit.** Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best fit.** Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst fit.** Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

### 9.2.3 Fragmentation

Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**. As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Whether we are using the first-fit or best-fit strategy can affect the amount of fragmentation. (First fit is better for some systems, whereas best fit is better for others.) Another factor is which end of a free block is allocated. (Which is the leftover piece—the one on the top or the one on the bottom?) No matter which algorithm is used, however, external fragmentation will be a problem.

Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem. Statistical analysis of first fit, for instance, reveals that, even with some optimization, given  $N$  allocated blocks, another  $0.5 N$  blocks will be lost to fragmentation. That is, one-third of memory may be unusable! This property is known as the **50-percent rule**.

Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation**—unused memory that is internal to a partition.

One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done. It is possible only if relocation is dynamic and is done at execution time. If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address. When compaction is possible, we must determine its cost. The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.

Another possible solution to the external-fragmentation problem is to permit the logical address space of processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available. This is the strategy used in **paging**, the most common memory-management technique for computer systems. We describe paging in the following section.

Fragmentation is a general problem in computing that can occur wherever we must manage blocks of data. We discuss the topic further in the storage management chapters (Chapter 11 through Chapter 15).

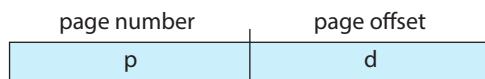
## 9.3 Paging

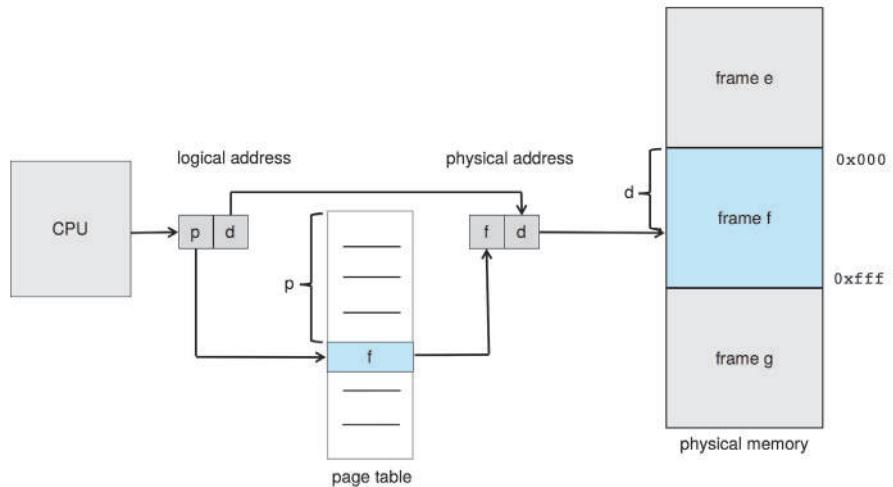
Memory management discussed thus far has required the physical address space of a process to be contiguous. We now introduce **paging**, a memory-management scheme that permits a process's physical address space to be non-contiguous. Paging avoids external fragmentation and the associated need for compaction, two problems that plague contiguous memory allocation. Because it offers numerous advantages, paging in its various forms is used in most operating systems, from those for large servers through those for mobile devices. Paging is implemented through cooperation between the operating system and the computer hardware.

### 9.3.1 Basic Method

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames. This rather simple idea has great functionality and wide ramifications. For example, the logical address space is now totally separate from the physical address space, so a process can have a logical 64-bit address space even though the system has less than  $2^{64}$  bytes of physical memory.

Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d)**:





**Figure 9.8** Paging hardware.

The page number is used as an index into a per-process **page table**. This is illustrated in Figure 9.8. The page table contains the base address of each frame in physical memory, and the offset is the location in the frame being referenced. Thus, the base address of the frame is combined with the page offset to define the physical memory address. The paging model of memory is shown in Figure 9.9.

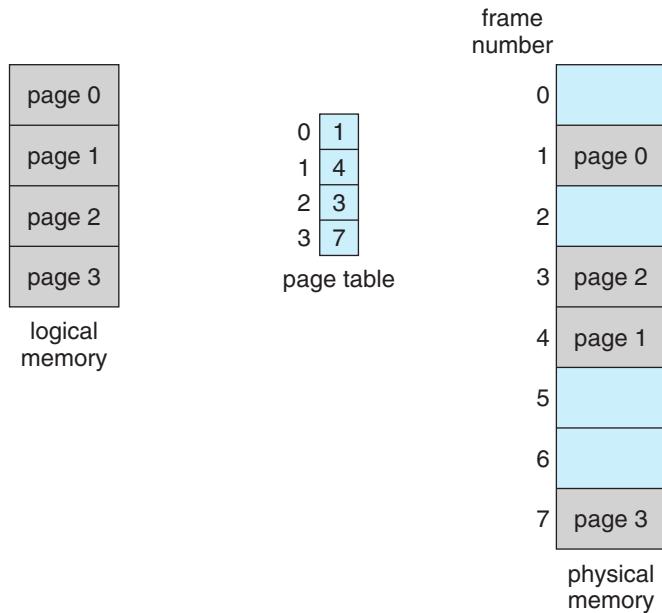
The following outlines the steps taken by the MMU to translate a logical address generated by the CPU to a physical address:

1. Extract the page number  $p$  and use it as an index into the page table.
2. Extract the corresponding frame number  $f$  from the page table.
3. Replace the page number  $p$  in the logical address with the frame number  $f$ .

As the offset  $d$  does not change, it is not replaced, and the frame number and offset now comprise the physical address.

The page size (like the frame size) is defined by the hardware. The size of a page is a power of 2, typically varying between 4 KB and 1 GB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of the logical address space is  $2^m$ , and a page size is  $2^n$  bytes, then the high-order  $m-n$  bits of a logical address designate the page number, and the  $n$  low-order bits designate the page offset. Thus, the logical address is as follows:

page number	page offset
$p$	$d$
$m-n$	$n$



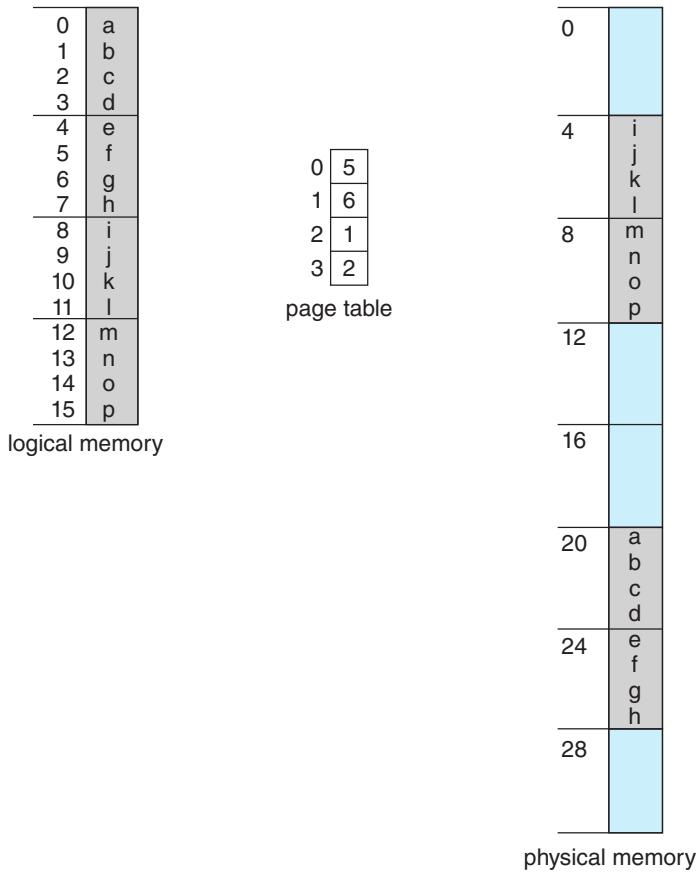
**Figure 9.9** Paging model of logical and physical memory.

where  $p$  is an index into the page table and  $d$  is the displacement within the page.

As a concrete (although minuscule) example, consider the memory in Figure 9.10. Here, in the logical address,  $n = 2$  and  $m = 4$ . Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the programmer's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [=  $(5 \times 4) + 0$ ]. Logical address 3 (page 0, offset 3) maps to physical address 23 [=  $(5 \times 4) + 3$ ]. Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [=  $(6 \times 4) + 0$ ]. Logical address 13 maps to physical address 9.

You may have noticed that paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.

When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units. If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full. For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of  $2,048 - 1,086 = 962$  bytes. In the worst case, a process would need  $n$  pages plus 1 byte. It would be allocated  $n + 1$  frames, resulting in internal fragmentation of almost an entire frame.



**Figure 9.10** Paging example for a 32-byte memory with 4-byte pages.

If process size is independent of page size, we expect internal fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. However, overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases. Also, disk I/O is more efficient when the amount of data being transferred is larger (Chapter 11). Generally, page sizes have grown over time as processes, data sets, and main memory have become larger. Today, pages are typically either 4 KB or 8 KB in size, and some systems support even larger page sizes. Some CPUs and operating systems even support multiple page sizes. For instance, on x86-64 systems, Windows 10 supports page sizes of 4 KB and 2 MB. Linux also supports two page sizes: a default page size (typically 4 KB) and an architecture-dependent larger page size called **huge pages**.

Frequently, on a 32-bit CPU, each page-table entry is 4 bytes long, but that size can vary as well. A 32-bit entry can point to one of  $2^{32}$  physical page frames. If the frame size is 4 KB ( $2^{12}$ ), then a system with 4-byte entries can address  $2^{44}$  bytes (or 16 TB) of physical memory. We should note here that the size of physical memory in a paged memory system is typically different from the maximum logical size of a process. As we further explore paging, we will

### OBTAINING THE PAGE SIZE ON LINUX SYSTEMS

On a Linux system, the page size varies according to architecture, and there are several ways of obtaining the page size. One approach is to use the system call `getpagesize()`. Another strategy is to enter the following command on the command line:

```
getconf PAGESIZE
```

Each of these techniques returns the page size as a number of bytes.

introduce other information that must be kept in the page-table entries. That information reduces the number of bits available to address page frames. Thus, a system with 32-bit page-table entries may address less physical memory than the possible maximum.

When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires  $n$  pages, at least  $n$  frames must be available in memory. If  $n$  frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on (Figure 9.11).

An important aspect of paging is the clear separation between the programmer's view of memory and the actual physical memory. The programmer views memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds



**Figure 9.11** Free frames (a) before allocation and (b) after allocation.

other programs. The difference between the programmer's view of memory and the actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the programmer and is controlled by the operating system. Notice that the user process by definition is unable to access memory it does not own. It has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.

Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory—which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a single, system-wide data structure called a **frame table**. The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process (or processes).

In addition, the operating system must be aware that user processes operate in user space, and all logical addresses must be mapped to produce physical addresses. If a user makes a system call (to do I/O, for example) and provides an address as a parameter (a buffer, for instance), that address must be mapped to produce the correct physical address. The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents. This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually. It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. Paging therefore increases the context-switch time.

### 9.3.2 Hardware Support

As page tables are per-process data structures, a pointer to the page table is stored with the other register values (like the instruction pointer) in the process control block of each process. When the CPU scheduler selects a process for execution, it must reload the user registers and the appropriate hardware page-table values from the stored user page table.

The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated high-speed hardware registers, which makes the page-address translation very efficient. However, this approach increases context-switch time, as each one of these registers must be exchanged during a context switch.

The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary CPUs, however, support much larger page tables (for example,  $2^{20}$  entries). For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a **page-table base register (PTBR)** points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.

#### 9.3.2.1 Translation Look-Aside Buffer

Although storing the page table in main memory can yield faster context switches, it may also result in slower memory access times. Suppose we want to access location  $i$ . We must first index into the page table, using the value in

the PTBR offset by the page number for  $i$ . This task requires one memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory. With this scheme, *two* memory accesses are needed to access data (one for the page-table entry and one for the actual data). Thus, memory access is slowed by a factor of 2, a delay that is considered intolerable under most circumstances.

The standard solution to this problem is to use a special, small, fast-lookup hardware cache called a **translation look-aside buffer (TLB)**. The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; a TLB lookup in modern hardware is part of the instruction pipeline, essentially adding no performance penalty. To be able to execute the search within a pipeline step, however, the TLB must be kept small. It is typically between 32 and 1,024 entries in size. Some CPUs implement separate instruction and data address TLBs. That can double the number of TLB entries available, because those lookups occur in different pipeline steps. We can see in this development an example of the evolution of CPU technology: systems have evolved from having no TLBs to having multiple levels of TLBs, just as they have multiple levels of caches.

The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, the MMU first checks if its page number is present in the TLB. If the page number is found, its frame number is immediately available and is used to access memory. As just mentioned, these steps are executed as part of the instruction pipeline within the CPU, adding no performance penalty compared with a system that does not implement paging.

If the page number is not in the TLB (known as a **TLB miss**), address translation proceeds following the steps illustrated in Section 9.3.1, where a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory (Figure 9.12). In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.

If the TLB is already full of entries, an existing entry must be selected for replacement. Replacement policies range from least recently used (LRU) through round-robin to random. Some CPUs allow the operating system to participate in LRU entry replacement, while others handle the matter themselves. Furthermore, some TLBs allow certain entries to be **wired down**, meaning that they cannot be removed from the TLB. Typically, TLB entries for key kernel code are wired down.

Some TLBs store **address-space identifier (ASIDs)** in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process. When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, the attempt is treated as a TLB miss. In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously. If the TLB does not support separate ASIDs, then every time a new page table is selected (for instance, with each context switch), the TLB must be **flushed** (or erased) to ensure that the next executing process does not use the

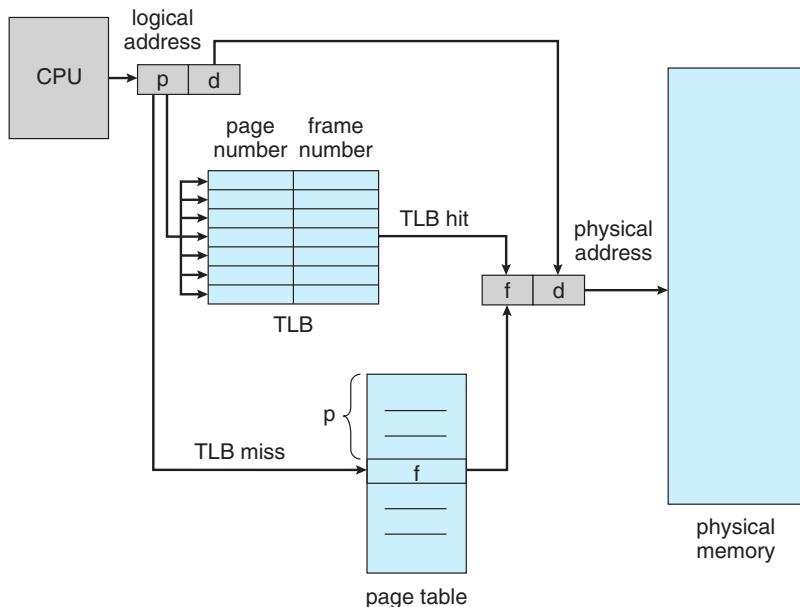


Figure 9.12 Paging hardware with TLB.

wrong translation information. Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**. An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 10 nanoseconds to access memory, then a mapped-memory access takes 10 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (10 nanoseconds) and then access the desired byte in memory (10 nanoseconds), for a total of 20 nanoseconds. (We are assuming that a page-table lookup takes only one memory access, but it can take more, as we shall see.) To find the **effective memory-access time**, we weight the case by its probability:

$$\begin{aligned}\text{effective access time} &= 0.80 \times 10 + 0.20 \times 20 \\ &= 12 \text{ nanoseconds}\end{aligned}$$

In this example, we suffer a 20-percent slowdown in average memory-access time (from 10 to 12 nanoseconds). For a 99-percent hit ratio, which is much more realistic, we have

$$\begin{aligned}\text{effective access time} &= 0.99 \times 10 + 0.01 \times 20 \\ &= 10.1 \text{ nanoseconds}\end{aligned}$$

This increased hit rate produces only a 1 percent slowdown in access time.

As noted earlier, CPUs today may provide multiple levels of TLBs. Calculating memory access times in modern CPUs is therefore much more complicated than shown in the example above. For instance, the Intel Core i7 CPU has a 128-entry L1 instruction TLB and a 64-entry L1 data TLB. In the case of a miss at L1, it takes the CPU six cycles to check for the entry in the L2 512-entry TLB. A miss in L2 means that the CPU must either walk through the page-table entries in memory to find the associated frame address, which can take hundreds of cycles, or interrupt to the operating system to have it do the work.

A complete performance analysis of paging overhead in such a system would require miss-rate information about each TLB tier. We can see from the general information above, however, that hardware features can have a significant effect on memory performance and that operating-system improvements (such as paging) can result in and, in turn, be affected by hardware changes (such as TLBs). We will further explore the impact of the hit ratio on the TLB in Chapter 10.

TLBs are a hardware feature and therefore would seem to be of little concern to operating systems and their designers. But the designer needs to understand the function and features of TLBs, which vary by hardware platform. For optimal operation, an operating-system design for a given platform must implement paging according to the platform's TLB design. Likewise, a change in the TLB design (for example, between different generations of Intel CPUs) may necessitate a change in the paging implementation of the operating systems that use it.

### 9.3.3 Protection

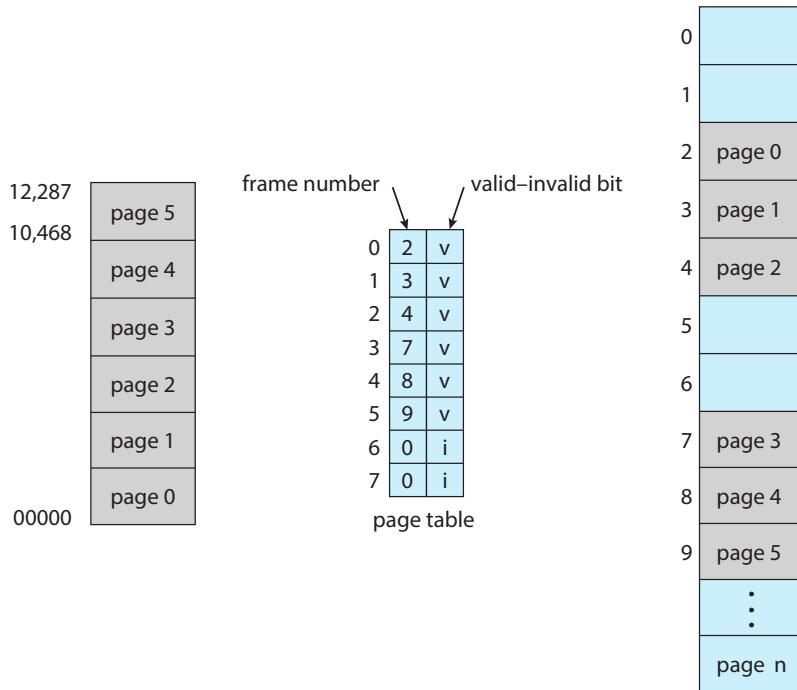
Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table.

One bit can define a page to be read–write or read-only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).

We can easily expand this approach to provide a finer level of protection. We can create hardware to provide read-only, read–write, or execute-only protection; or, by providing separate protection bits for each kind of access, we can allow any combination of these accesses. Illegal attempts will be trapped to the operating system.

One additional bit is generally attached to each entry in the page table: a **valid–invalid** bit. When this bit is set to *valid*, the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to *invalid*, the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid–invalid bit. The operating system sets this bit for each page to allow or disallow access to the page.

Suppose, for example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468. Given a page size of 2 KB, we have the situation shown in Figure 9.13. Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, will find that the



**Figure 9.13** Valid (v) or invalid (i) bit in a page table.

valid–invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).

Notice that this scheme has created a problem. Because the program extends only to address 10468, any reference beyond that address is illegal. However, references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are invalid. This problem is a result of the 2-KB page size and reflects the internal fragmentation of paging.

Rarely does a process use all its address range. In fact, many processes use only a small fraction of the address space available to them. It would be wasteful in these cases to create a page table with entries for every page in the address range. Most of this table would be unused but would take up valuable memory space. Some systems provide hardware, in the form of a **page-table length register (PTLR)**, to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap to the operating system.

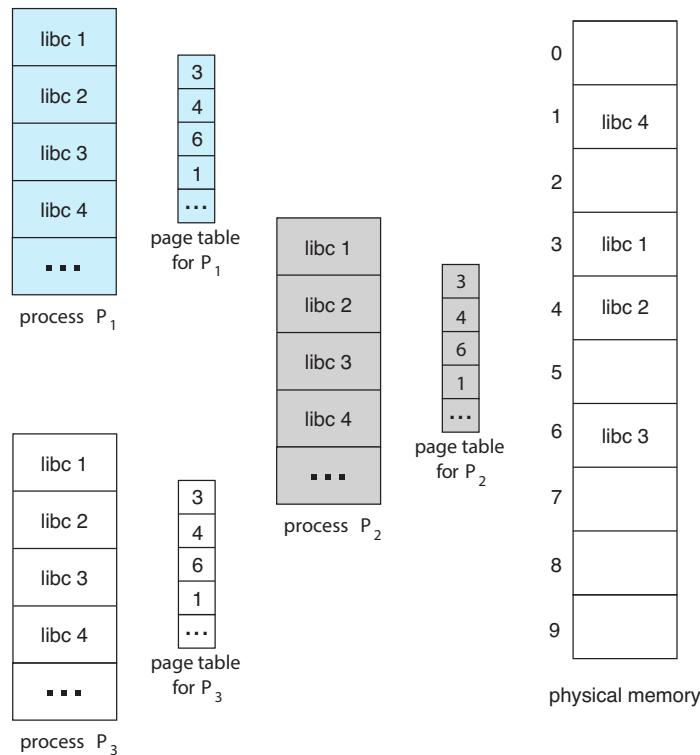
#### 9.3.4 Shared Pages

An advantage of paging is the possibility of *sharing* common code, a consideration that is particularly important in an environment with multiple processes. Consider the standard C library, which provides a portion of the system call interface for many versions of UNIX and Linux. On a typical Linux system, most user processes require the standard C library `libc`. One option is to have

each process load its own copy of `libc` into its address space. If a system has 40 user processes, and the `libc` library is 2 MB, this would require 80 MB of memory.

If the code is **reentrant code**, however, it can be shared, as shown in Figure 9.14. Here, we see three processes sharing the pages for the standard C library `libc`. (Although the figure shows the `libc` library occupying four pages, in reality, it would occupy more.) Reentrant code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will, of course, be different. Only one copy of the standard C library need be kept in physical memory, and the page table for each user process maps onto the same physical copy of `libc`. Thus, to support 40 processes, we need only one copy of the library, and the total space now required is 2 MB instead of 80 MB—a significant saving!

In addition to run-time libraries such as `libc`, other heavily used programs can also be shared—compilers, window systems, database systems, and so on. The shared libraries discussed in Section 9.1.5 are typically implemented with shared pages. To be sharable, the code must be reentrant. The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property.



**Figure 9.14** Sharing of standard C library in a paging environment.

The sharing of memory among processes on a system is similar to the sharing of the address space of a task by threads, described in Chapter 4. Furthermore, recall that in Chapter 3 we described shared memory as a method of interprocess communication. Some operating systems implement shared memory using shared pages.

Organizing memory according to pages provides numerous benefits in addition to allowing several processes to share the same physical pages. We cover several other benefits in Chapter 10.

## 9.4 Structure of the Page Table

In this section, we explore some of the most common techniques for structuring the page table, including hierarchical paging, hashed page tables, and inverted page tables.

### 9.4.1 Hierarchical Paging

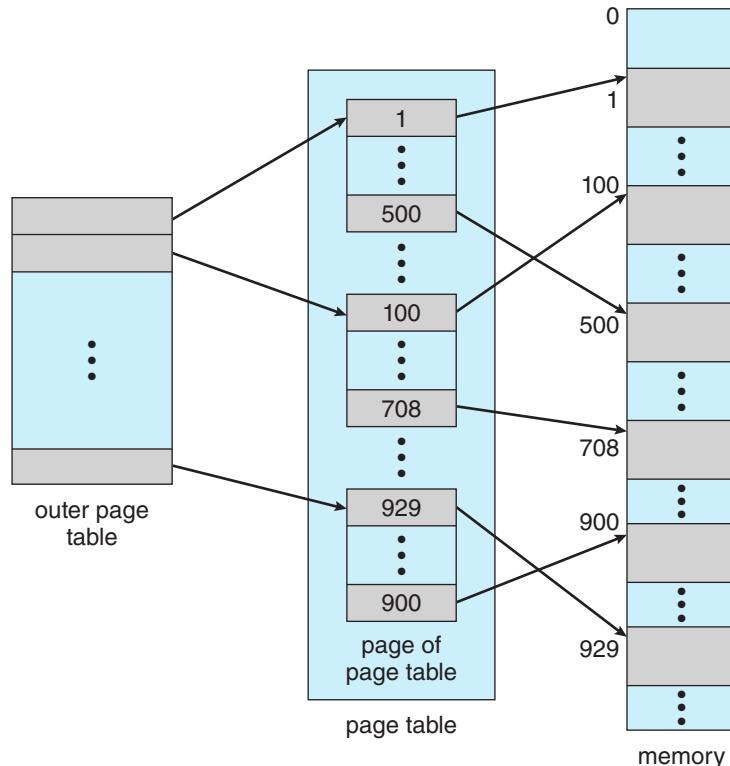
Most modern computer systems support a large logical address space ( $2^{32}$  to  $2^{64}$ ). In such an environment, the page table itself becomes excessively large. For example, consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB ( $2^{12}$ ), then a page table may consist of over 1 million entries ( $2^{20} = 2^{32}/2^{12}$ ). Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces. We can accomplish this division in several ways.

One way is to use a two-level paging algorithm, in which the page table itself is also paged (Figure 9.15). For example, consider again the system with a 32-bit logical address space and a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:

page number		page offset
$p_1$	$p_2$	$d$
10	10	12

where  $p_1$  is an index into the outer page table and  $p_2$  is the displacement within the page of the inner page table. The address-translation method for this architecture is shown in Figure 9.16. Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped** page table.

For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate. To illustrate this point, let's suppose that the page size in such a system is 4 KB ( $2^{12}$ ). In this case, the page table consists of up to  $2^{52}$  entries. If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain  $2^{10}$  4-byte entries. The addresses look like this:



**Figure 9.15** A two-level page-table scheme.

outer page	inner page	offset
$p_1$	$p_2$	$d$

42            10            12

The outer page table consists of  $2^{42}$  entries, or  $2^{44}$  bytes. The obvious way to avoid such a large table is to divide the outer page table into smaller pieces. (This approach is also used on some 32-bit processors for added flexibility and efficiency.)

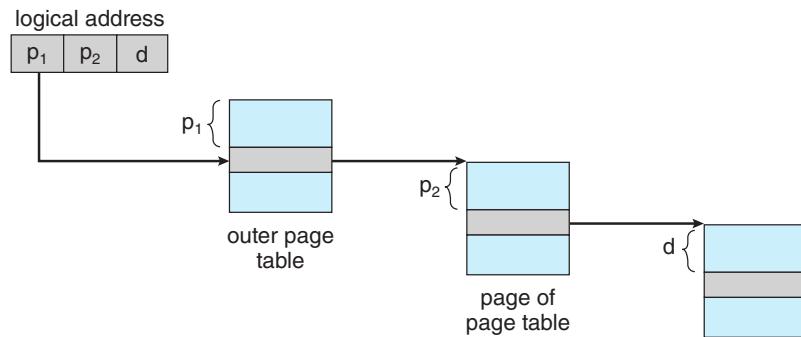
We can divide the outer page table in various ways. For example, we can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages ( $2^{10}$  entries, or  $2^{12}$  bytes). In this case, a 64-bit address space is still daunting:

2nd outer page	outer page	inner page	offset
$p_1$	$p_2$	$p_3$	$d$

32            10            10            12

The outer page table is still  $2^{34}$  bytes (16 GB) in size.

The next step would be a four-level paging scheme, where the second-level outer page table itself is also paged, and so forth. The 64-bit UltraSPARC would require seven levels of paging—a prohibitive number of memory accesses—



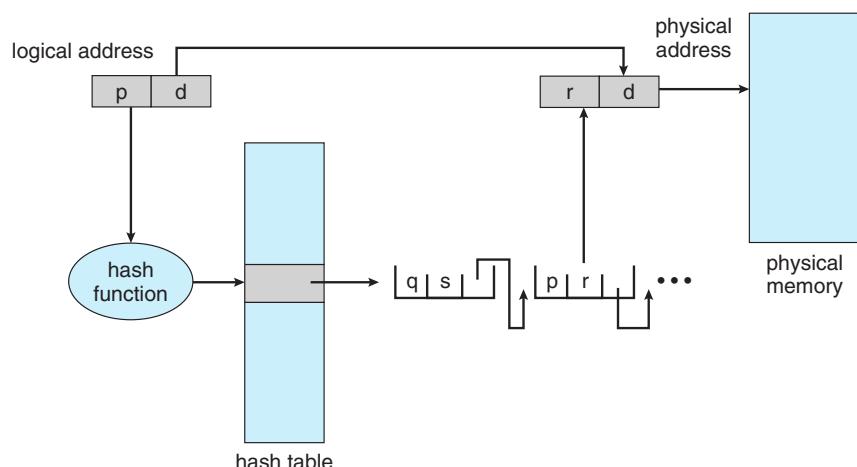
**Figure 9.16** Address translation for a two-level 32-bit paging architecture.

to translate each logical address. You can see from this example why, for 64-bit architectures, hierarchical page tables are generally considered inappropriate.

#### 9.4.2 Hashed Page Tables

One approach for handling address spaces larger than 32 bits is to use a **hashed page table**, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.

The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in Figure 9.17.



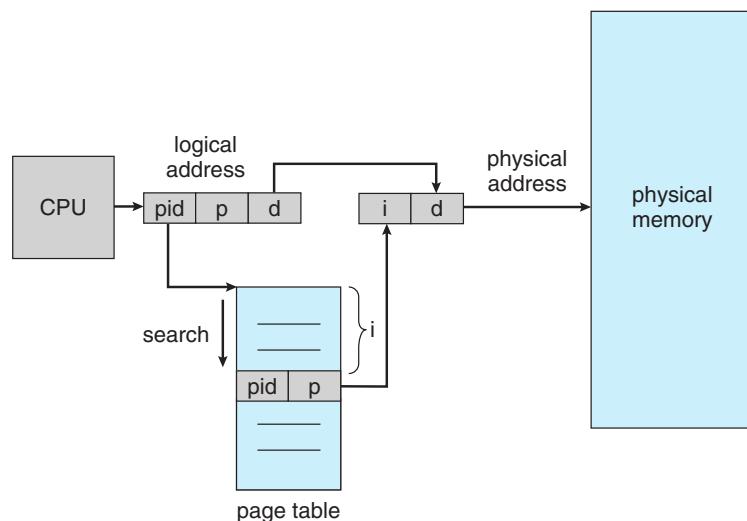
**Figure 9.17** Hashed page table.

A variation of this scheme that is useful for 64-bit address spaces has been proposed. This variation uses **clustered page tables**, which are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical-page frames. Clustered page tables are particularly useful for **sparse** address spaces, where memory references are noncontiguous and scattered throughout the address space.

### 9.4.3 Inverted Page Tables

Usually, each process has an associated page table. The page table has one entry for each page that the process is using (or one slot for each virtual address, regardless of the latter's validity). This table representation is a natural one, since processes reference pages through the pages' virtual addresses. The operating system must then translate this reference into a physical memory address. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is located and to use that value directly. One of the drawbacks of this method is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.

To solve this problem, we can use an **inverted page table**. An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory. Figure 9.18 shows the operation of an inverted page table. Compare it with Figure 9.8, which depicts a standard page table in operation. Inverted page tables often require that an address-space identifier (Section 9.3.2) be stored in each entry of the page table, since the table usually contains several



**Figure 9.18** Inverted page table.

different address spaces mapping physical memory. Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame. Examples of systems using inverted page tables include the 64-bit UltraSPARC and PowerPC.

To illustrate this method, we describe a simplified version of the inverted page table used in the IBM RT. IBM was the first major company to use inverted page tables, starting with the IBM System 38 and continuing through the RS/6000 and the current IBM Power CPUs. For the IBM RT, each virtual address in the system consists of a triple:

$$\langle \text{process-id}, \text{page-number}, \text{offset} \rangle.$$

Each inverted page-table entry is a pair  $\langle \text{process-id}, \text{page-number} \rangle$  where the process-id assumes the role of the address-space identifier. When a memory reference occurs, part of the virtual address, consisting of  $\langle \text{process-id}, \text{page-number} \rangle$ , is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found—say, at entry  $i$ —then the physical address  $\langle i, \text{offset} \rangle$  is generated. If no match is found, then an illegal address access has been attempted.

Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs. Because the inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched before a match is found. This search would take far too long. To alleviate this problem, we use a hash table, as described in Section 9.4.2, to limit the search to one—or at most a few—page-table entries. Of course, each access to the hash table adds a memory reference to the procedure, so one virtual memory reference requires at least two real memory reads—one for the hash-table entry and one for the page table. (Recall that the TLB is searched first, before the hash table is consulted, offering some performance improvement.)

One interesting issue with inverted page tables involves shared memory. With standard paging, each process has its own page table, which allows multiple virtual addresses to be mapped to the same physical address. This method cannot be used with inverted page tables; because there is only one virtual page entry for every physical page, one physical page cannot have two (or more) shared virtual addresses. Therefore, with inverted page tables, only one mapping of a virtual address to the shared physical address may occur at any given time. A reference by another process sharing the memory will result in a page fault and will replace the mapping with a different virtual address.

#### 9.4.4 Oracle SPARC Solaris

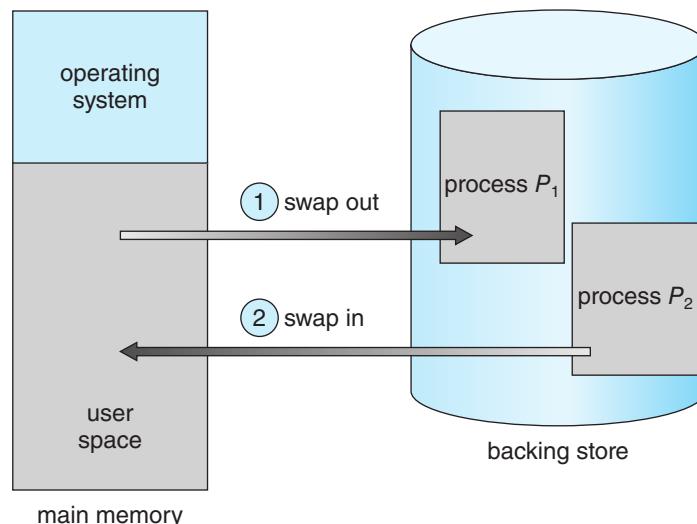
Consider as a final example a modern 64-bit CPU and operating system that are tightly integrated to provide low-overhead virtual memory. **Solaris** running on the **SPARC** CPU is a fully 64-bit operating system and as such has to solve the problem of virtual memory without using up all of its physical memory by keeping multiple levels of page tables. Its approach is a bit complex but solves the problem efficiently using hashed page tables. There are two hash tables—one for the kernel and one for all user processes. Each maps memory addresses from virtual to physical memory. Each hash-table entry represents a contiguous area of mapped virtual memory, which is more efficient than

having a separate hash-table entry for each page. Each entry has a base address and a span indicating the number of pages the entry represents.

Virtual-to-physical translation would take too long if each address required searching through a hash table, so the CPU implements a TLB that holds translation table entries (TTEs) for fast hardware lookups. A cache of these TTEs resides in a translation storage buffer (TSB), which includes an entry per recently accessed page. When a virtual address reference occurs, the hardware searches the TLB for a translation. If none is found, the hardware walks through the in-memory TSB looking for the TTE that corresponds to the virtual address that caused the lookup. This **TLB walk** functionality is found on many modern CPUs. If a match is found in the TSB, the CPU copies the TSB entry into the TLB, and the memory translation completes. If no match is found in the TSB, the kernel is interrupted to search the hash table. The kernel then creates a TTE from the appropriate hash table and stores it in the TSB for automatic loading into the TLB by the CPU memory-management unit. Finally, the interrupt handler returns control to the MMU, which completes the address translation and retrieves the requested byte or word from main memory.

## 9.5 Swapping

Process instructions and the data they operate on must be in memory to be executed. However, a process, or a portion of a process, can be **swapped** temporarily out of memory to a **Backing store** and then brought back into memory for continued execution (Figure 9.19). Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.



**Figure 9.19** Standard swapping of two processes using a disk as a backing store.

### 9.5.1 Standard Swapping

Standard swapping involves moving entire processes between main memory and a backing store. The backing store is commonly fast secondary storage. It must be large enough to accommodate whatever parts of processes need to be stored and retrieved, and it must provide direct access to these memory images. When a process or part is swapped to the backing store, the data structures associated with the process must be written to the backing store. For a multithreaded process, all per-thread data structures must be swapped as well. The operating system must also maintain metadata for processes that have been swapped out, so they can be restored when they are swapped back in to memory.

The advantage of standard swapping is that it allows physical memory to be oversubscribed, so that the system can accommodate more processes than there is actual physical memory to store them. Idle or mostly idle processes are good candidates for swapping; any memory that has been allocated to these inactive processes can then be dedicated to active processes. If an inactive process that has been swapped out becomes active once again, it must then be swapped back in. This is illustrated in Figure 9.19.

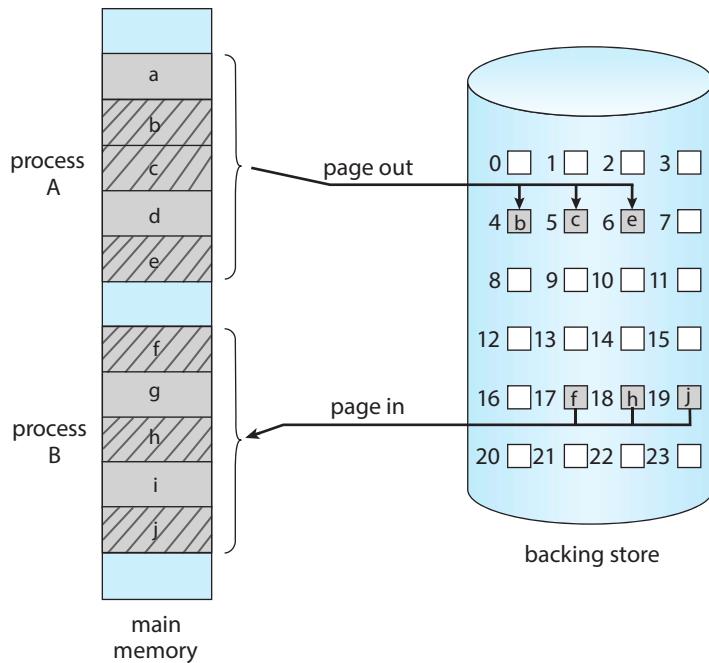
### 9.5.2 Swapping with Paging

Standard swapping was used in traditional UNIX systems, but it is generally no longer used in contemporary operating systems, because the amount of time required to move entire processes between memory and the backing store is prohibitive. (An exception to this is Solaris, which still uses standard swapping, however only under dire circumstances when available memory is extremely low.)

Most systems, including Linux and Windows, now use a variation of swapping in which pages of a process—rather than an entire process—can be swapped. This strategy still allows physical memory to be oversubscribed, but does not incur the cost of swapping entire processes, as presumably only a small number of pages will be involved in swapping. In fact, the term *swapping* now generally refers to standard swapping, and *paging* refers to swapping with paging. A *page out* operation moves a page from memory to the backing store; the reverse process is known as a *page in*. Swapping with paging is illustrated in Figure 9.20 where a subset of pages for processes A and B are being paged-out and paged-in respectively. As we shall see in Chapter 10, swapping with paging works well in conjunction with virtual memory.

### 9.5.3 Swapping on Mobile Systems

Most operating systems for PCs and servers support swapping pages. In contrast, mobile systems typically do not support swapping in any form. Mobile devices generally use flash memory rather than more spacious hard disks for nonvolatile storage. The resulting space constraint is one reason why mobile operating-system designers avoid swapping. Other reasons include the limited number of writes that flash memory can tolerate before it becomes unreliable and the poor throughput between main memory and flash memory in these devices.

**Figure 9.20** Swapping with paging.

Instead of using swapping, when free memory falls below a certain threshold, Apple’s iOS *asks* applications to voluntarily relinquish allocated memory. Read-only data (such as code) are removed from main memory and later reloaded from flash memory if necessary. Data that have been modified (such as the stack) are never removed. However, any applications that fail to free up sufficient memory may be terminated by the operating system.

Android adopts a strategy similar to that used by iOS. It may terminate a process if insufficient free memory is available. However, before terminating a process, Android writes its **application state** to flash memory so that it can be quickly restarted.

Because of these restrictions, developers for mobile systems must carefully allocate and release memory to ensure that their applications do not use too much memory or suffer from memory leaks.

#### SYSTEM PERFORMANCE UNDER SWAPPING

Although swapping pages is more efficient than swapping entire processes, when a system is undergoing *any* form of swapping, it is often a sign there are more active processes than available physical memory. There are generally two approaches for handling this situation: (1) terminate some processes, or (2) get more physical memory!

## 9.6 Example: Intel 32- and 64-bit Architectures

The architecture of Intel chips has dominated the personal computer landscape for decades. The 16-bit Intel 8086 appeared in the late 1970s and was soon followed by another 16-bit chip—the Intel 8088—which was notable for being the chip used in the original IBM PC. Intel later produced a series of 32-bit chips—the IA-32—which included the family of 32-bit Pentium processors. More recently, Intel has produced a series of 64-bit chips based on the x86-64 architecture. Currently, all the most popular PC operating systems run on Intel chips, including Windows, macOS, and Linux (although Linux, of course, runs on several other architectures as well). Notably, however, Intel's dominance has not spread to mobile systems, where the ARM architecture currently enjoys considerable success (see Section 9.7).

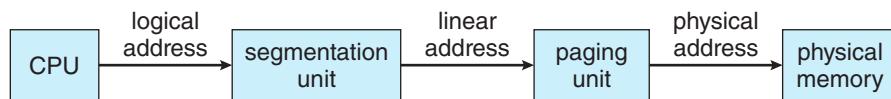
In this section, we examine address translation for both IA-32 and x86-64 architectures. Before we proceed, however, it is important to note that because Intel has released several versions—as well as variations—of its architectures over the years, we cannot provide a complete description of the memory-management structure of all its chips. Nor can we provide all of the CPU details, as that information is best left to books on computer architecture. Rather, we present the major memory-management concepts of these Intel CPUs.

### 9.6.1 IA-32 Architecture

Memory management in IA-32 systems is divided into two components—segmentation and paging—and works as follows: The CPU generates logical addresses, which are given to the segmentation unit. The segmentation unit produces a linear address for each logical address. The linear address is then given to the paging unit, which in turn generates the physical address in main memory. Thus, the segmentation and paging units form the equivalent of the memory-management unit (MMU). This scheme is shown in Figure 9.21.

#### 9.6.1.1 IA-32 Segmentation

The IA-32 architecture allows a segment to be as large as 4 GB, and the maximum number of segments per process is 16 K. The logical address space of a process is divided into two partitions. The first partition consists of up to 8 K segments that are private to that process. The second partition consists of up to 8 K segments that are shared among all the processes. Information about the first partition is kept in the **local descriptor table (LDT)**; information about the second partition is kept in the **global descriptor table (GDT)**. Each entry in the LDT and GDT consists of an 8-byte segment descriptor with detailed information about a particular segment, including the base location and limit of that segment.



**Figure 9.21** Logical to physical address translation in IA-32.

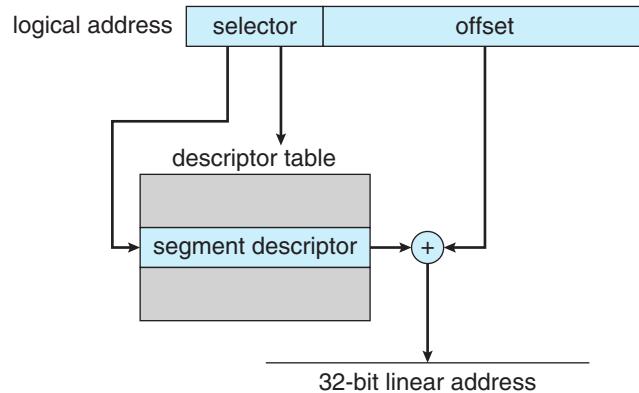


Figure 9.22 IA-32 segmentation.

The logical address is a pair (selector, offset), where the selector is a 16-bit number:

<i>s</i>	<i>g</i>	<i>p</i>
13	1	2

Here, *s* designates the segment number, *g* indicates whether the segment is in the GDT or LDT, and *p* deals with protection. The offset is a 32-bit number specifying the location of the byte within the segment in question.

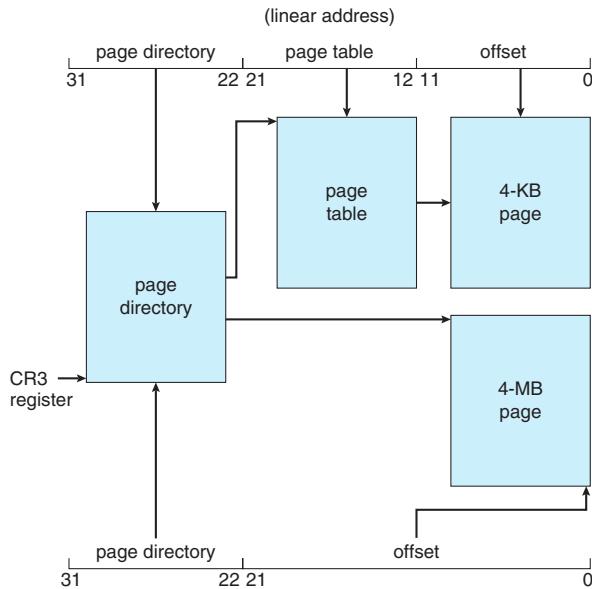
The machine has six segment registers, allowing six segments to be addressed at any one time by a process. It also has six 8-byte microprogram registers to hold the corresponding descriptors from either the LDT or the GDT. This cache lets the Pentium avoid having to read the descriptor from memory for every memory reference.

The linear address on the IA-32 is 32 bits long and is formed as follows. The segment register points to the appropriate entry in the LDT or GDT. The base and limit information about the segment in question is used to generate a **linear address**. First, the limit is used to check for address validity. If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address. This is shown in Figure 9.22. In the following section, we discuss how the paging unit turns this linear address into a physical address.

### 9.6.1.2 IA-32 Paging

The IA-32 architecture allows a page size of either 4 KB or 4 MB. For 4-KB pages, IA-32 uses a two-level paging scheme in which the division of the 32-bit linear address is as follows:

page number		page offset
<i>p</i> <sub>1</sub>	<i>p</i> <sub>2</sub>	<i>d</i>
10	10	12



**Figure 9.23** Paging in the IA-32 architecture.

The address-translation scheme for this architecture is similar to the scheme shown in Figure 9.16. The IA-32 address translation is shown in more detail in Figure 9.23. The 10 high-order bits reference an entry in the outermost page table, which IA-32 terms the **page directory**. (The CR3 register points to the page directory for the current process.) The page directory entry points to an inner page table that is indexed by the contents of the innermost 10 bits in the linear address. Finally, the low-order bits 0–11 refer to the offset in the 4-KB page pointed to in the page table.

One entry in the page directory is the **Page\_Size** flag, which—if set—indicates that the size of the page frame is 4 MB and not the standard 4 KB. If this flag is set, the page directory points directly to the 4-MB page frame, bypassing the inner page table; and the 22 low-order bits in the linear address refer to the offset in the 4-MB page frame.

To improve the efficiency of physical memory use, IA-32 page tables can be swapped to disk. In this case, an invalid bit is used in the page directory entry to indicate whether the table to which the entry is pointing is in memory or on disk. If the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table. The table can then be brought into memory on demand.

As software developers began to discover the 4-GB memory limitations of 32-bit architectures, Intel adopted a **page address extension (PAE)**, which allows 32-bit processors to access a physical address space larger than 4 GB. The fundamental difference introduced by PAE support was that paging went from a two-level scheme (as shown in Figure 9.23) to a three-level scheme, where the top two bits refer to a **page directory pointer table**. Figure 9.24 illustrates a PAE system with 4-KB pages. (PAE also supports 2-MB pages.)

PAE also increased the page-directory and page-table entries from 32 to 64 bits in size, which allowed the base address of page tables and page frames to

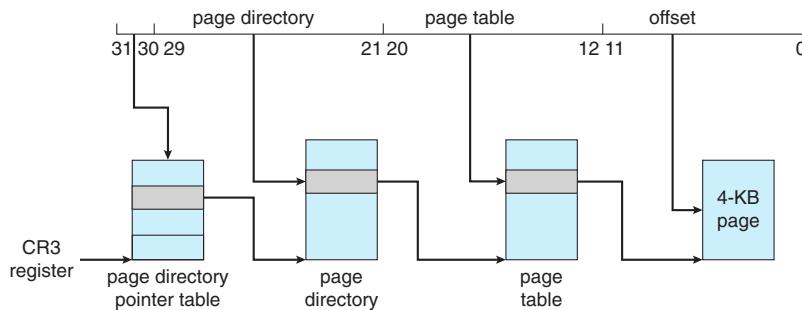


Figure 9.24 Page address extensions.

extend from 20 to 24 bits. Combined with the 12-bit offset, adding PAE support to IA-32 increased the address space to 36 bits, which supports up to 64 GB of physical memory. It is important to note that operating system support is required to use PAE. Both Linux and macOS support PAE. However, 32-bit versions of Windows desktop operating systems still provide support for only 4 GB of physical memory, even if PAE is enabled.

### 9.6.2 x86-64

Intel has had an interesting history of developing 64-bit architectures. Its initial entry was the IA-64 (later named **Itanium**) architecture, but that architecture was not widely adopted. Meanwhile, another chip manufacturer—AMD—began developing a 64-bit architecture known as x86-64 that was based on extending the existing IA-32 instruction set. The x86-64 supported much larger logical and physical address spaces, as well as several other architectural advances. Historically, AMD had often developed chips based on Intel's architecture, but now the roles were reversed as Intel adopted AMD's x86-64 architecture. In discussing this architecture, rather than using the commercial names **AMD64** and **Intel 64**, we will use the more general term **x86-64**.

Support for a 64-bit address space yields an astonishing  $2^{64}$  bytes of addressable memory—a number greater than 16 quintillion (or 16 exabytes). However, even though 64-bit systems can potentially address this much memory, in practice far fewer than 64 bits are used for address representation in current designs. The x86-64 architecture currently provides a 48-bit virtual address with support for page sizes of 4 KB, 2 MB, or 1 GB using four levels of paging hierarchy. The representation of the linear address appears in Figure 9.25. Because this addressing scheme can use PAE, virtual addresses are 48 bits in size but support 52-bit physical addresses (4,096 terabytes).

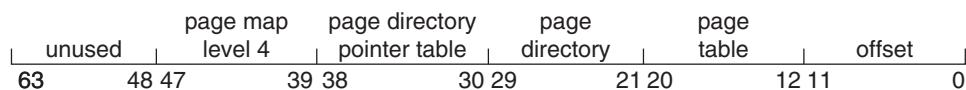


Figure 9.25 x86-64 linear address.

## 9.7 Example: ARMv8 Architecture

Although Intel chips have dominated the personal computer market for more than 30 years, chips for mobile devices such as smartphones and tablet computers often instead run on ARM processors. Interestingly, whereas Intel both designs and manufactures chips, ARM only designs them. It then licenses its architectural designs to chip manufacturers. Apple has licensed the ARM design for its iPhone and iPad mobile devices, and most Android-based smartphones use ARM processors as well. In addition to mobile devices, ARM also provides architecture designs for real-time embedded systems. Because of the abundance of devices that run on the ARM architecture, over 100 billion ARM processors have been produced, making it the most widely used architecture when measured in the quantity of chips produced. In this section, we describe the 64-bit ARMv8 architecture.

The ARMv8 has three different **translation granules**: 4 KB, 16 KB, and 64 KB. Each translation granule provides different page sizes, as well as larger sections of contiguous memory, known as **regions**. The page and region sizes for the different translation granules are shown below:

Translation Granule Size	Page Size	Region Size
4 KB	4 KB	2 MB, 1 GB
16 KB	16 KB	32 MB
64 KB	64 KB	512 MB

For 4-KB and 16-KB granules, up to four levels of paging may be used, with up to three levels of paging for 64-KB granules. Figure 9.26 illustrates the ARMv8 address structure for the 4-KB translation granule with up to four levels of paging. (Notice that although ARMv8 is a 64-bit architecture, only 48 bits are currently used.) The four-level hierarchical paging structure for the 4-KB translation granule is illustrated in Figure 9.27. (The TTBR register is the **translation table base register** and points to the level 0 table for the current thread.)

If all four levels are used, the offset (bits 0–11 in Figure 9.26) refers to the offset within a 4-KB page. However, notice that the table entries for level 1 and

### 64-BIT COMPUTING

History has taught us that even though memory capacities, CPU speeds, and similar computer capabilities seem large enough to satisfy demand for the foreseeable future, the growth of technology ultimately absorbs available capacities, and we find ourselves in need of additional memory or processing power, often sooner than we think. What might the future of technology bring that would make a 64-bit address space seem too small?

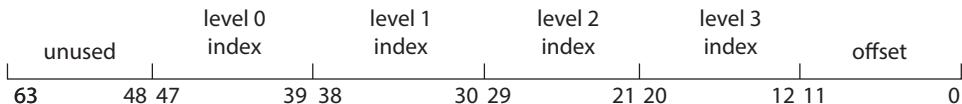


Figure 9.26 ARM 4-KB translation granule.

level 2 may refer either to another table or to a 1-GB region (level-1 table) or 2-MB region (level-2 table). As an example, if the level-1 table refers to a 1-GB region rather than a level-2 table, the low-order 30 bits (bits 0–29 in Figure 9.26) are used as an offset into this 1-GB region. Similarly, if the level-2 table refers to a 2-MB region rather than a level-3 table, the low-order 21 bits (bits 0–20 in Figure 9.26) refer to the offset within this 2-MB region.

The ARM architecture also supports two levels of TLBs. At the inner level are two **micro TLBs**—a TLB for data and another for instructions. The micro TLB supports ASIDs as well. At the outer level is a single **main TLB**. Address translation begins at the micro-TLB level. In the case of a miss, the main TLB is then checked. If both TLBs yield misses, a page table walk must be performed in hardware.

## 9.8 Summary

- Memory is central to the operation of a modern computer system and consists of a large array of bytes, each with its own address.
- One way to allocate an address space to each process is through the use of base and limit registers. The base register holds the smallest legal physical memory address, and the limit specifies the size of the range.

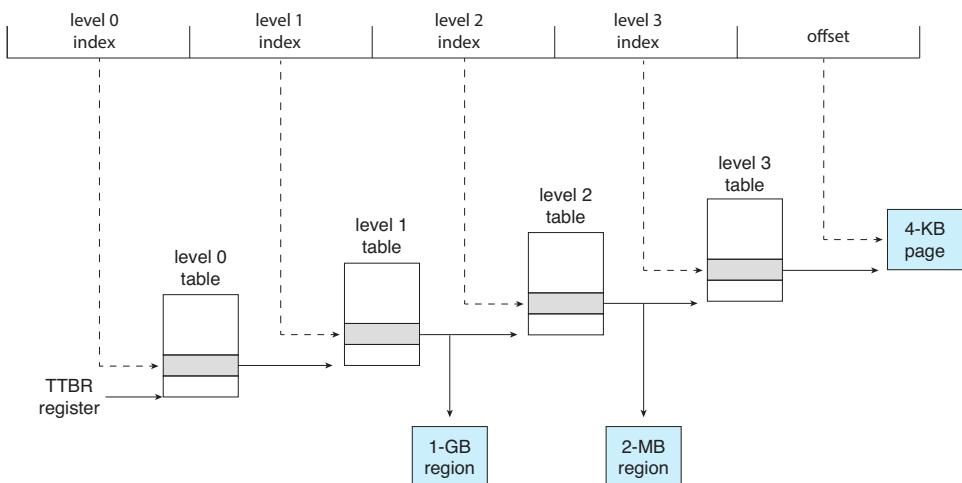


Figure 9.27 ARM four-level hierarchical paging.

- Binding symbolic address references to actual physical addresses may occur during (1) compile, (2) load, or (3) execution time.
- An address generated by the CPU is known as a logical address, which the memory management unit (MMU) translates to a physical address in memory.
- One approach to allocating memory is to allocate partitions of contiguous memory of varying sizes. These partitions may be allocated based on three possible strategies: (1) first fit, (2) best fit, and (3) worst fit.
- Modern operating systems use paging to manage memory. In this process, physical memory is divided into fixed-sized blocks called frames and logical memory into blocks of the same size called pages.
- When paging is used, a logical address is divided into two parts: a page number and a page offset. The page number serves as an index into a per-process page table that contains the frame in physical memory that holds the page. The offset is the specific location in the frame being referenced.
- A translation look-aside buffer (TLB) is a hardware cache of the page table. Each TLB entry contains a page number and its corresponding frame.
- Using a TLB in address translation for paging systems involves obtaining the page number from the logical address and checking if the frame for the page is in the TLB. If it is, the frame is obtained from the TLB. If the frame is not present in the TLB, it must be retrieved from the page table.
- Hierarchical paging involves dividing a logical address into multiple parts, each referring to different levels of page tables. As addresses expand beyond 32 bits, the number of hierarchical levels may become large. Two strategies that address this problem are hashed page tables and inverted page tables.
- Swapping allows the system to move pages belonging to a process to disk to increase the degree of multiprogramming.
- The Intel 32-bit architecture has two levels of page tables and supports either 4-KB or 4-MB page sizes. This architecture also supports page-address extension, which allows 32-bit processors to access a physical address space larger than 4 GB. The x86-64 and ARMv9 architectures are 64-bit architectures that use hierarchical paging.

## Practice Exercises

- 9.1 Name two differences between logical and physical addresses.
- 9.2 Why are page sizes always powers of 2?
- 9.3 Consider a system in which a program can be separated into two parts: code and data. The CPU knows whether it wants an instruction (instruction fetch) or data (data fetch or store). Therefore, two base-limit register pairs are provided: one for instructions and one for data. The instruction

base-limit register pair is automatically read-only, so programs can be shared among different users. Discuss the advantages and disadvantages of this scheme.

- 9.4 Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.
  - a. How many bits are there in the logical address?
  - b. How many bits are there in the physical address?
- 9.5 What is the effect of allowing two entries in a page table to point to the same page frame in memory? Explain how this effect could be used to decrease the amount of time needed to copy a large amount of memory from one place to another. What effect would updating some byte on one page have on the other page?
- 9.6 Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)?
- 9.7 Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):
  - a. 3085
  - b. 42095
  - c. 215201
  - d. 650000
  - e. 2000001
- 9.8 The BTV operating system has a 21-bit virtual address, yet on certain embedded devices, it has only a 16-bit physical address. It also has a 2-KB page size. How many entries are there in each of the following?
  - a. A conventional, single-level page table
  - b. An inverted page tableWhat is the maximum amount of physical memory in the BTV operating system?
- 9.9 Consider a logical address space of 256 pages with a 4-KB page size, mapped onto a physical memory of 64 frames.
  - a. How many bits are required in the logical address?
  - b. How many bits are required in the physical address?
- 9.10 Consider a computer system with a 32-bit logical address and 4-KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?
  - a. A conventional, single-level page table
  - b. An inverted page table

## Further Reading

The concept of paging can be credited to the designers of the Atlas system, which has been described by [Kilburn et al. (1961)] and by [Howarth et al. (1961)].

[Hennessy and Patterson (2012)] explain the hardware aspects of TLBs, caches, and MMUs. [Jacob and Mudge (2001)] describe techniques for managing the TLB. [Fang et al. (2001)] evaluate support for large pages.

PAE support for Windows systems is discussed in <http://msdn.microsoft.com/en-us/library/windows/hardware/gg487512.aspx>. An overview of the ARM architecture is provided in <http://www.arm.com/products/processors/cortex-a/cortex-a9.php>

## Bibliography

**[Fang et al. (2001)]** Z. Fang, L. Zhang, J. B. Carter, W. C. Hsieh, and S. A. McKee, “Reevaluating Online Superpage Promotion with Hardware Support”, *Proceedings of the International Symposium on High-Performance Computer Architecture*, Volume 50, Number 5 (2001).

**[Hennessy and Patterson (2012)]** J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Fifth Edition, Morgan Kaufmann (2012).

**[Howarth et al. (1961)]** D. J. Howarth, R. B. Payne, and F. H. Sumner, “The Manchester University Atlas Operating System, Part II: User’s Description”, *Computer Journal*, Volume 4, Number 3 (1961), pages 226–229.

**[Jacob and Mudge (2001)]** B. Jacob and T. Mudge, “Uniprocessor Virtual Memory Without TLBs”, *IEEE Transactions on Computers*, Volume 50, Number 5 (2001).

**[Kilburn et al. (1961)]** T. Kilburn, D. J. Howarth, R. B. Payne, and F. H. Sumner, “The Manchester University Atlas Operating System, Part I: Internal Organization”, *Computer Journal*, Volume 4, Number 3 (1961), pages 222–225.

## Chapter 9 Exercises

- 9.11 Explain the difference between internal and external fragmentation.
- 9.12 Consider the following process for generating binaries. A compiler is used to generate the object code for individual modules, and a linker is used to combine multiple object modules into a single program binary. How does the linker change the binding of instructions and data to memory addresses? What information needs to be passed from the compiler to the linker to facilitate the memory-binding tasks of the linker?
- 9.13 Given six memory partitions of 100 MB, 170 MB, 40 MB, 205 MB, 300 MB, and 185 MB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 200 MB, 15 MB, 185 MB, 75 MB, 175 MB, and 80 MB (in order)? Indicate which—if any—requests cannot be satisfied. Comment on how efficiently each of the algorithms manages memory.
- 9.14 Most systems allow a program to allocate more memory to its address space during execution. Allocation of data in the heap segments of programs is an example of such allocated memory. What is required to support dynamic memory allocation in the following schemes?
  - a. Contiguous memory allocation
  - b. Paging
- 9.15 Compare the memory organization schemes of contiguous memory allocation and paging with respect to the following issues:
  - a. External fragmentation
  - b. Internal fragmentation
  - c. Ability to share code across processes
- 9.16 On a system with paging, a process cannot access memory that it does not own. Why? How could the operating system allow access to additional memory? Why should it or should it not?
- 9.17 Explain why mobile operating systems such as iOS and Android do not support swapping.
- 9.18 Although Android does not support swapping on its boot disk, it is possible to set up a swap space using a separate SD nonvolatile memory card. Why would Android disallow swapping on its boot disk yet allow it on a secondary disk?
- 9.19 Explain why address-space identifiers (ASIDs) are used in TLBs.
- 9.20 Program binaries in many systems are typically structured as follows. Code is stored starting with a small, fixed virtual address, such as 0. The code segment is followed by the data segment, which is used for storing the program variables. When the program starts executing, the stack is allocated at the other end of the virtual address space and is allowed to grow toward lower virtual addresses. What is the significance of this structure for the following schemes?

- a. Contiguous memory allocation
  - b. Paging
- 9.21** Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers)?
- a. 21205
  - b. 164250
  - c. 121357
  - d. 16479315
  - e. 27253187
- 9.22** The MPV operating system is designed for embedded systems and has a 24-bit virtual address, a 20-bit physical address, and a 4-KB page size. How many entries are there in each of the following?
- a. A conventional, single-level page table
  - b. An inverted page table
- What is the maximum amount of physical memory in the MPV operating system?
- 9.23** Consider a logical address space of 2,048 pages with a 4-KB page size, mapped onto a physical memory of 512 frames.
- a. How many bits are required in the logical address?
  - b. How many bits are required in the physical address?
- 9.24** Consider a computer system with a 32-bit logical address and 8-KB page size. The system supports up to 1 GB of physical memory. How many entries are there in each of the following?
- a. A conventional, single-level page table
  - b. An inverted page table
- 9.25** Consider a paging system with the page table stored in memory.
- a. If a memory reference takes 50 nanoseconds, how long does a paged memory reference take?
  - b. If we add TLBs, and if 75 percent of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes 2 nanoseconds, if the entry is present.)
- 9.26** What is the purpose of paging the page tables?
- 9.27** Consider the IA-32 address-translation scheme shown in Figure 9.22.
- a. Describe all the steps taken by the IA-32 in translating a logical address into a physical address.
  - b. What are the advantages to the operating system of hardware that provides such complicated memory translation?

- c. Are there any disadvantages to this address-translation system? If so, what are they? If not, why is this scheme not used by every manufacturer?

## Programming Problems

- 9.28** Assume that a system has a 32-bit virtual address with a 4-KB page size. Write a C program that is passed a virtual address (in decimal) on the command line and have it output the page number and offset for the given address. As an example, your program would run as follows:

```
./addresses 19986
```

Your program would output:

```
The address 19986 contains:  
page number = 4  
offset = 3602
```

Writing this program will require using the appropriate data type to store 32 bits. We encourage you to use unsigned data types as well.

## Programming Projects

### Contiguous Memory Allocation

In Section 9.2, we presented different algorithms for contiguous memory allocation. This project will involve managing a contiguous region of memory of size *MAX* where addresses may range from 0 ... *MAX* – 1. Your program must respond to four different requests:

1. Request for a contiguous block of memory
2. Release of a contiguous block of memory
3. Compact unused holes of memory into one single block
4. Report the regions of free and allocated memory

Your program will be passed the initial amount of memory at startup. For example, the following initializes the program with 1 MB (1,048,576 bytes) of memory:

```
./allocator 1048576
```

Once your program has started, it will present the user with the following prompt:

```
allocator>
```

It will then respond to the following commands: RQ (request), RL (release), C (compact), STAT (status report), and X (exit).

A request for 40,000 bytes will appear as follows:

```
allocator>RQ P0 40000 W
```

The first parameter to the RQ command is the new process that requires the memory, followed by the amount of memory being requested, and finally the strategy. (In this situation, “W” refers to worst fit.)

Similarly, a release will appear as:

```
allocator>RL P0
```

This command will release the memory that has been allocated to process P0.

The command for compaction is entered as:

```
allocator>C
```

This command will compact unused holes of memory into one region.

Finally, the STAT command for reporting the status of memory is entered as:

```
allocator>STAT
```

Given this command, your program will report the regions of memory that are allocated and the regions that are unused. For example, one possible arrangement of memory allocation would be as follows:

```
Addresses [0:315000] Process P1
Addresses [315001: 512500] Process P3
Addresses [512501:625575] Unused
Addresses [625575:725100] Process P6
Addresses [725001] . . .
```

## Allocating Memory

Your program will allocate memory using one of the three approaches highlighted in Section 9.2.2, depending on the flag that is passed to the RQ command. The flags are:

- F—first fit
- B—best fit
- W—worst fit

This will require that your program keep track of the different holes representing available memory. When a request for memory arrives, it will allocate the memory from one of the available holes based on the allocation strategy. If there is insufficient memory to allocate to a request, it will output an error message and reject the request.

Your program will also need to keep track of which region of memory has been allocated to which process. This is necessary to support the STAT command and is also needed when memory is released via the RL command, as the process releasing memory is passed to this command. If a partition being released is adjacent to an existing hole, be sure to combine the two holes into a single hole.

## **Compaction**

If the user enters the C command, your program will compact the set of holes into one larger hole. For example, if you have four separate holes of size 550 KB, 375 KB, 1,900 KB, and 4,500 KB, your program will combine these four holes into one large hole of size 7,325 KB.

There are several strategies for implementing compaction, one of which is suggested in Section 9.2.3. Be sure to update the beginning address of any processes that have been affected by compaction.



# File-System Interface



For most users, the file system is the most visible aspect of a general-purpose operating system. It provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system. The file system consists of two distinct parts: a collection of files, each storing related data, and a directory structure, which organizes and provides information about all the files in the system. Most file systems live on storage devices, which we described in Chapter 11 and will continue to discuss in the next chapter. In this chapter, we consider the various aspects of files and the major directory structures. We also discuss the semantics of sharing files among multiple processes, users, and computers. Finally, we discuss ways to handle file protection, necessary when we have multiple users and want to control who may access files and how files may be accessed.

## CHAPTER OBJECTIVES

- Explain the function of file systems.
- Describe the interfaces to file systems.
- Discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures.
- Explore file-system protection.

### 13.1 File Concept

Computers can store information on various storage media, such as NVM devices, HDDs, magnetic tapes, and optical disks. So that the computer system will be convenient to use, the operating system provides a uniform logical view of stored information. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the [file](#). Files are mapped by the operating system onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent between system reboots.

A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general.

Because files are **the** method users and applications use to store and retrieve data, and because they are so general purpose, their use has stretched beyond its original confines. For example, UNIX, Linux, and some other operating systems provide a `proc` file system that uses file-system interfaces to provide access to system information (such as process details).

The information in a file is defined by its creator. Many different types of information may be stored in a file—source or executable programs, numeric or text data, photos, music, video, and so on. A file has a certain defined structure, which depends on its type. A **text fil** is a sequence of characters organized into lines (and possibly pages). A **source fil** is a sequence of functions, each of which is further organized as declarations followed by executable statements. An **executable fil** is a series of code sections that the loader can bring into memory and execute.

### 13.1.1 File Attributes

A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as `example.c`. Some systems differentiate between uppercase and lowercase characters in names, whereas other systems do not. When a file is named, it becomes independent of the process, the user, and even the system that created it. For instance, one user might create the file `example.c`, and another user might edit that file by specifying its name. The file's owner might write the file to a USB drive, send it as an e-mail attachment, or copy it across a network, and it could still be called `example.c` on the destination system. Unless there is a sharing and synchronization method, that second copy is now independent of the first and can be changed separately.

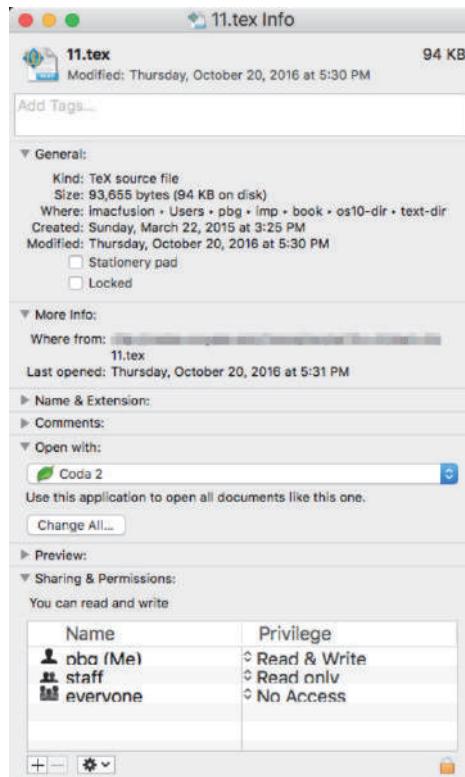
A file's attributes vary from one operating system to another but typically consist of these:

- **Name.** The symbolic file name is the only information kept in human-readable form.
- **Identifie .** This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type.** This information is needed for systems that support different types of files.
- **Location.** This information is a pointer to a device and to the location of the file on that device.

- **Size.** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection.** Access-control information determines who can do reading, writing, executing, and so on.
- **Timestamps and user identification.** This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

Some newer file systems also support **extended file attributes**, including character encoding of the file and security features such as a file checksum. Figure 13.1 illustrates a **file info window** on macOS that displays a file's attributes.

The information about all files is kept in the directory structure, which resides on the same device as the files themselves. Typically, a directory entry consists of the file's name and its unique identifier. The identifier in turn locates the other file attributes. It may take more than a kilobyte to record this information for each file. In a system with many files, the size of the directory itself may be megabytes or gigabytes. Because directories must match the volatility of the files, like files, they must be stored on the device and are usually brought into memory piecemeal, as needed.



**Figure 13.1** A file info window on macOS.

### 13.1.2 File Operations

A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. Let's examine what the operating system must do to perform each of these seven basic file operations. It should then be easy to see how other similar operations, such as renaming a file, can be implemented.

- **Creating a fil** . Two steps are necessary to create a file. First, space in the file system must be found for the file. We discuss how to allocate space for the file in Chapter 14. Second, an entry for the new file must be made in a directory.
- **Opening a fil** . Rather than have all file operations specify a file name, causing the operating system to evaluate the name, check access permissions, and so on, all operations except create and delete require a file `open()` first. If successful, the `open` call returns a file handle that is used as an argument in the other calls.
- **Writing a fil** . To write a file, we make a system call specifying both the open file handle and the information to be written to the file. The system must keep a **write pointer** to the location in the file where the next write is to take place if it is sequential. The write pointer must be updated whenever a write occurs.
- **Reading a fil** . To read from a file, we use a system call that specifies the file handle and where (in memory) the next block of the file should be put. Again, the system needs to keep a **read pointer** to the location in the file where the next read is to take place, if sequential. Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process **current-file-position pointer**. Both the read and write operations use this same pointer, saving space and reducing system complexity.
- **Repositioning within a file** . The current-file-position pointer of the open file is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a file **seek**.
- **Deleting a fil** . To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase or mark as free the directory entry. Note that some systems allow **hard links**—multiple names (directory entries) for the same file. In this case the actual file contents is not deleted until the last link is deleted.
- **Truncating a fil** . The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length. The file can then be reset to length zero, and its file space can be released.

These seven basic operations comprise the minimal set of required file operations. Other common operations include appending new information to the end of an existing file and renaming an existing file. These primitive operations can then be combined to perform other file operations. For instance, we can create a copy of a file by creating a new file and then reading from the old and writing to the new. We also want to have operations that allow a user to get and set the various attributes of a file. For example, we may want to have operations that allow a user to determine the status of a file, such as the file's length, and to set file attributes, such as the file's owner.

As mentioned, most of the file operations mentioned involve searching the directory for the entry associated with the named file. To avoid this constant searching, many systems require that an `open()` system call be made before a file is first used. The operating system keeps a table, called the **open-file table**, containing information about all open files. When a file operation is requested, the file is specified via an index into this table, so no searching is required. When the file is no longer being actively used, it is closed by the process, and the operating system removes its entry from the open-file table, potentially releasing locks. `create()` and `delete()` are system calls that work with closed rather than open files.

Some systems implicitly open a file when the first reference to it is made. The file is automatically closed when the job or program that opened the file terminates. Most systems, however, require that the programmer open a file explicitly with the `open()` system call before that file can be used. The `open()` operation takes a file name and searches the directory, copying the directory entry into the open-file table. The `open()` call can also accept access-mode information—create, read-only, read-write, append-only, and so on. This mode is checked against the file's permissions. If the request mode is allowed, the file is opened for the process. The `open()` system call typically returns a pointer to the entry in the open-file table. This pointer, not the actual file name, is used in all I/O operations, avoiding any further searching and simplifying the system-call interface.

The implementation of the `open()` and `close()` operations is more complicated in an environment where several processes may open the file simultaneously. This may occur in a system where several different applications open the same file at the same time. Typically, the operating system uses two levels of internal tables: a per-process table and a system-wide table. The per-process table tracks all files that a process has open. Stored in this table is information regarding the process's use of the file. For instance, the current file pointer for each file is found here. Access rights to the file and accounting information can also be included.

Each entry in the per-process table in turn points to a system-wide open-file table. The system-wide table contains process-independent information, such as the location of the file on disk, access dates, and file size. Once a file has been opened by one process, the system-wide table includes an entry for the file. When another process executes an `open()` call, a new entry is simply added to the process's open-file table pointing to the appropriate entry in the system-wide table. Typically, the open-file table also has an **open count** associated with each file to indicate how many processes have the file open. Each `close()` decreases this open count, and when the open count reaches zero, the file is no longer in use, and the file's entry is removed from the open-file table.

**FILE LOCKING IN JAVA**

In the Java API, acquiring a lock requires first obtaining the `FileChannel` for the file to be locked. The `lock()` method of the `FileChannel` is used to acquire the lock. The API of the `lock()` method is

```
FileLock lock(long begin, long end, boolean shared)
```

where `begin` and `end` are the beginning and ending positions of the region being locked. Setting `shared` to `true` is for shared locks; setting `shared` to `false` acquires the lock exclusively. The lock is released by invoking the `release()` of the `FileLock` returned by the `lock()` operation.

The program in Figure 13.2 illustrates file locking in Java. This program acquires two locks on the file `file.txt`. The lock for the first half of the file is an exclusive lock; the lock for the second half is a shared lock.

In summary, several pieces of information are associated with an open file.

- **File pointer.** On systems that do not include a file offset as part of the `read()` and `write()` system calls, the system must track the last read–write location as a current-file-position pointer. This pointer is unique to each process operating on the file and therefore must be kept separate from the on-disk file attributes.
- **File-open count.** As files are closed, the operating system must reuse its open-file table entries, or it could run out of space in the table. Multiple processes may have opened a file, and the system must wait for the last file to close before removing the open-file table entry. The file-open count tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.
- **Location of the file.** Most file operations require the system to read or write data within the file. The information needed to locate the file (wherever it is located, be it on mass storage, on a file server across the network, or on a RAM drive) is kept in memory so that the system does not have to read it from the directory structure for each operation.
- **Access rights.** Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/O requests.

Some operating systems provide facilities for locking an open file (or sections of a file). File locks allow one process to lock a file and prevent other processes from gaining access to it. File locks are useful for files that are shared by several processes—for example, a system log file that can be accessed and modified by a number of processes in the system.

File locks provide functionality similar to reader–writer locks, covered in Section 7.1.2. A **shared lock** is akin to a reader lock in that several processes can acquire the lock concurrently. An **exclusive lock** behaves like a writer lock; only one process at a time can acquire such a lock. It is important to note that not

---

```

import java.io.*;
import java.nio.channels.*;

public class LockingExample {
    public static final boolean EXCLUSIVE = false;
    public static final boolean SHARED = true;

    public static void main(String args[]) throws IOException {
        FileLock sharedLock = null;
        FileLock exclusiveLock = null;

        try {
            RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");

            // get the channel for the file
            FileChannel ch = raf.getChannel();

            // this locks the first half of the file - exclusive
            exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);

            /** Now modify the data . . . */

            // release the lock
            exclusiveLock.release();

            // this locks the second half of the file - shared
            sharedLock = ch.lock(raf.length()/2+1, raf.length(), SHARED);

            /** Now read the data . . . */

            // release the lock
            sharedLock.release();
        } catch (java.io.IOException ioe) {
            System.err.println(ioe);
        }
        finally {
            if (exclusiveLock != null)
                exclusiveLock.release();
            if (sharedLock != null)
                sharedLock.release();
        }
    }
}

```

---

**Figure 13.2** File-locking example in Java.

all operating systems provide both types of locks: some systems provide only exclusive file locking.

Furthermore, operating systems may provide either **mandatory** or **advisory** file-locking mechanisms. With mandatory locking, once a process acquires an exclusive lock, the operating system will prevent any other process from

accessing the locked file. For example, assume a process acquires an exclusive lock on the file `system.log`. If we attempt to open `system.log` from another process—for example, a text editor—the operating system will prevent access until the exclusive lock is released. Alternatively, if the lock is advisory, then the operating system will not prevent the text editor from acquiring access to `system.log`. Rather, the text editor must be written so that it manually acquires the lock before accessing the file. In other words, if the locking scheme is mandatory, the operating system ensures locking integrity. For advisory locking, it is up to software developers to ensure that locks are appropriately acquired and released. As a general rule, Windows operating systems adopt mandatory locking, and UNIX systems employ advisory locks.

The use of file locks requires the same precautions as ordinary process synchronization. For example, programmers developing on systems with mandatory locking must be careful to hold exclusive file locks only while they are accessing the file. Otherwise, they will prevent other processes from accessing the file as well. Furthermore, some measures must be taken to ensure that two or more processes do not become involved in a deadlock while trying to acquire file locks.

### 13.1.3 File Types

When we design a file system—indeed, an entire operating system—we always consider whether the operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways. For example, a common mistake occurs when a user tries to output the binary-object form of a program. This attempt normally produces garbage; however, the attempt can succeed if the operating system has been told that the file is a binary-object program.

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts—a name and an extension, usually separated by a period (Figure 13.3). In this way, the user and the operating system can tell from the name alone what the type of a file is. Most operating systems allow users to specify a file name as a sequence of characters followed by a period and terminated by an extension made up of additional characters. Examples include `resume.docx`, `server.c`, and `ReaderThread.cpp`.

The system uses the extension to indicate the type of the file and the type of operations that can be done on that file. Only a file with a `.com`, `.exe`, or `.sh` extension can be executed, for instance. The `.com` and `.exe` files are two forms of binary executable files, whereas the `.sh` file is a **shell script** containing, in ASCII format, commands to the operating system. Application programs also use extensions to indicate file types in which they are interested. For example, Java compilers expect source files to have a `.java` extension, and the Microsoft Word word processor expects its files to end with a `.doc` or `.docx` extension. These extensions are not always required, so a user may specify a file without the extension (to save typing), and the application will look for a file with the given name and the extension it expects. Because these extensions are not supported by the operating system, they can be considered “hints” to the applications that operate on them.

Consider, too, the macOS operating system. In this system, each file has a type, such as `.app` (for application). Each file also has a creator attribute

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

**Figure 13.3** Common file types.

containing the name of the program that created it. This attribute is set by the operating system during the `create()` call, so its use is enforced and supported by the system. For instance, a file produced by a word processor has the word processor's name as its creator. When the user opens that file, by double-clicking the mouse on the icon representing the file, the word processor is invoked automatically, and the file is loaded, ready to be edited.

The UNIX system uses a **magic number** stored at the beginning of some binary files to indicate the type of data in the file (for example, the format of an image file). Likewise, it uses a text magic number at the start of text files to indicate the type of file (which shell language a script is written in) and so on. (For more details on magic numbers and other computer jargon, see <http://www.catb.org/esr/jargon/>.) Not all files have magic numbers, so system features cannot be based solely on this information. UNIX does not record the name of the creating program, either. UNIX does allow file-name-extension hints, but these extensions are neither enforced nor depended on by the operating system; they are meant mostly to aid users in determining what type of contents the file contains. Extensions can be used or ignored by a given application, but that is up to the application's programmer.

#### 13.1.4 File Structure

File types also can be used to indicate the internal structure of the file. Source and object files have structures that match the expectations of the programs that read them. Further, certain files must conform to a required structure that

is understood by the operating system. For example, the operating system requires that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is. Some operating systems extend this idea into a set of system-supported file structures, with sets of special operations for manipulating files with those structures.

This point brings us to one of the disadvantages of having the operating system support multiple file structures: it makes the operating system large and cumbersome. If the operating system defines five different file structures, it needs to contain the code to support these file structures. In addition, it may be necessary to define every file as one of the file types supported by the operating system. When new applications require information structured in ways not supported by the operating system, severe problems may result.

For example, assume that a system supports two types of files: text files (composed of ASCII characters separated by a carriage return and line feed) and executable binary files. Now, if we (as users) want to define an encrypted file to protect the contents from being read by unauthorized people, we may find neither file type to be appropriate. The encrypted file is not ASCII text lines but rather is (apparently) random bits. Although it may appear to be a binary file, it is not executable. As a result, we may have to circumvent or misuse the operating system's file-type mechanism or abandon our encryption scheme.

Some operating systems impose (and support) a minimal number of file structures. This approach has been adopted in UNIX, Windows, and others. UNIX considers each file to be a sequence of 8-bit bytes; no interpretation of these bits is made by the operating system. This scheme provides maximum flexibility but little support. Each application program must include its own code to interpret an input file as to the appropriate structure. However, all operating systems must support at least one structure—that of an executable file—so that the system is able to load and run programs.

### 13.1.5 Internal File Structure

Internally, locating an offset within a file can be complicated for the operating system. Disk systems typically have a well-defined block size determined by the size of a sector. All disk I/O is performed in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.

For example, the UNIX operating system defines all files to be simply streams of bytes. Each byte is individually addressable by its offset from the beginning (or end) of the file. In this case, the logical record size is 1 byte. The file system automatically packs and unpacks bytes into physical disk blocks—say, 512 bytes per block—as necessary.

The logical record size, physical block size, and packing technique determine how many logical records are in each physical block. The packing can be done either by the user's application program or by the operating system. In either case, the file may be considered a sequence of blocks. All the basic I/O functions operate in terms of blocks. The conversion from logical records to physical blocks is a relatively simple software problem.

Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted. If each block were 512 bytes, for example, then a file of 1,949 bytes would be allocated four blocks (2,048 bytes); the last 99 bytes would be wasted. The waste incurred to keep everything in units of blocks (instead of bytes) is internal fragmentation. All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

## 13.2 Access Methods

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files. Others (such as mainframe operating systems) support many access methods, and choosing the right one for a particular application is a major design problem.

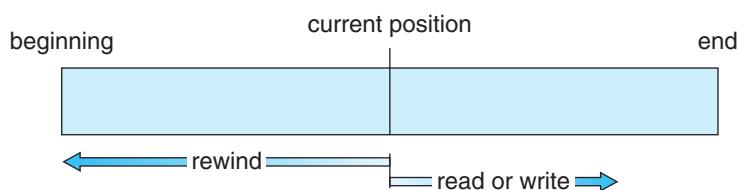
### 13.2.1 Sequential Access

The simplest access method is **sequential access**. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.

Reads and writes make up the bulk of the operations on a file. A read operation—`read_next()`—reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, the write operation—`write_next()`—appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning, and on some systems, a program may be able to skip forward or backward  $n$  records for some integer  $n$ —perhaps only for  $n = 1$ . Sequential access, which is depicted in Figure 13.4, is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.

### 13.2.2 Direct Access

Another method is **direct access** (or **relative access**). Here, a file is made up of fixed-length **logical records** that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records. Thus,



**Figure 13.4** Sequential-access file.

we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type. When a query concerning a particular subject arrives, we compute which block contains the answer and then read that block directly to provide the desired information.

As a simple example, on an airline-reservation system, we might store all the information about a particular flight (for example, flight 713) in the block identified by the flight number. Thus, the number of available seats for flight 713 is stored in block 713 of the reservation file. To store information about a larger set, such as people, we might compute a hash function on the people's names or search a small in-memory index to determine a block to read and search.

For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have `read(n)`, where  $n$  is the block number, rather than `read_next()`, and `write(n)` rather than `write_next()`. An alternative approach is to retain `read_next()` and `write_next()` and to add an operation `position_file(n)` where  $n$  is the block number. Then, to effect a `read(n)`, we would `position_file(n)` and then `read_next()`.

The block number provided by the user to the operating system is normally a **relative block number**. A relative block number is an index relative to the beginning of the file. Thus, the first relative block of the file is 0, the next is 1, and so on, even though the absolute disk address may be 14703 for the first block and 3192 for the second. The use of relative block numbers allows the operating system to decide where the file should be placed (called the **allocation problem**, as we discuss in Chapter 14) and helps to prevent the user from accessing portions of the file system that may not be part of her file. Some systems start their relative block numbers at 0; others start at 1.

How, then, does the system satisfy a request for record  $N$  in a file? Assuming we have a logical record length  $L$ , the request for record  $N$  is turned into an I/O request for  $L$  bytes starting at location  $L * (N)$  within the file (assuming the first record is  $N = 0$ ). Since logical records are of a fixed size, it is also easy to read, write, or delete a record.

Not all operating systems support both sequential and direct access for files. Some systems allow only sequential file access; others allow only direct access. Some systems require that a file be defined as sequential or direct when it is created. Such a file can be accessed only in a manner consistent with its declaration. We can easily simulate sequential access on a direct-access file by simply keeping a variable  $cp$  that defines our current position, as shown in Figure 13.5. Simulating a direct-access file on a sequential-access file, however, is extremely inefficient and clumsy.

### 13.2.3 Other Access Methods

Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file. The **index**, like an index in the back of a book, contains pointers to the various blocks. To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.

sequential access	implementation for direct access
reset	$cp = 0;$
read_next	$read cp;$ $cp = cp + 1;$
write_next	$write cp;$ $cp = cp + 1;$

**Figure 13.5** Simulation of sequential access on a direct-access file.

For example, a retail-price file might list the universal product codes (UPCs) for items, with the associated prices. Each record consists of a 10-digit UPC and a 6-digit price, for a 16-byte record. If our disk has 1,024 bytes per block, we can store 64 records per block. A file of 120,000 records would occupy about 2,000 blocks (2 million bytes). By keeping the file sorted by UPC, we can define an index consisting of the first UPC in each block. This index would have 2,000 entries of 10 digits each, or 20,000 bytes, and thus could be kept in memory. To find the price of a particular item, we can make a binary search of the index. From this search, we learn exactly which block contains the desired record and access that block. This structure allows us to search a large file doing little I/O.

With large files, the index file itself may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file contains pointers to secondary index files, which point to the actual data items.

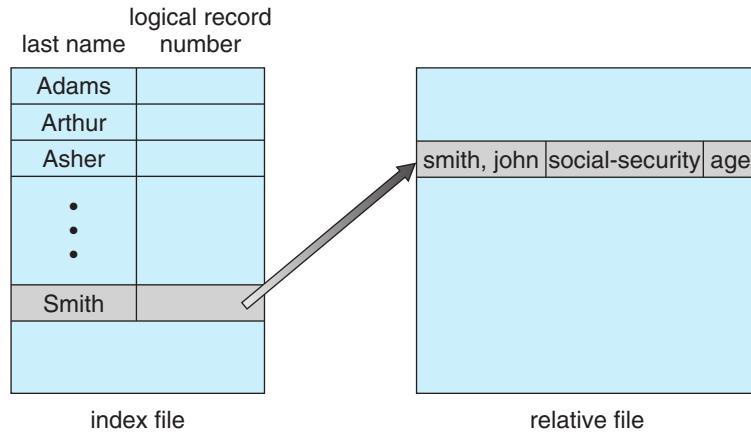
For example, IBM's indexed sequential-access method (ISAM) uses a small master index that points to disk blocks of a secondary index. The secondary index blocks point to the actual file blocks. The file is kept sorted on a defined key. To find a particular item, we first make a binary search of the master index, which provides the block number of the secondary index. This block is read in, and again a binary search is used to find the block containing the desired record. Finally, this block is searched sequentially. In this way, any record can be located from its key by at most two direct-access reads. Figure 13.6 shows a similar situation as implemented by OpenVMS index and relative files.

### 13.3 Directory Structure

The directory can be viewed as a symbol table that translates file names into their file control blocks. If we take such a view, we see that the directory itself can be organized in many ways. The organization must allow us to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory. In this section, we examine several schemes for defining the logical structure of the directory system.

When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

- **Search for a file**. We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names, and similar

**Figure 13.6** Example of index and relative files.

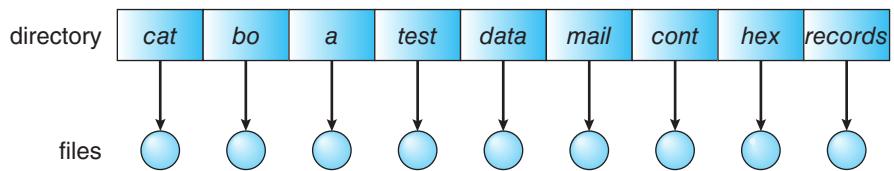
names may indicate a relationship among files, we may want to be able to find all files whose names match a particular pattern.

- **Create a file.** New files need to be created and added to the directory.
- **Delete a file.** When a file is no longer needed, we want to be able to remove it from the directory. Note a delete leaves a hole in the directory structure and the file system may have a method to defragment the directory structure.
- **List a directory.** We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
- **Rename a file.** Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
- **Traverse the file system.** We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files to magnetic tape, other secondary storage, or across a network to another system or the cloud. This technique provides a backup copy in case of system failure. In addition, if a file is no longer in use, the file can be copied to the backup target and the disk space of that file released for reuse by another file.

In the following sections, we describe the most common schemes for defining the logical structure of a directory.

### 13.3.1 Single-Level Directory

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand (Figure 13.7).



**Figure 13.7** Single-level directory.

A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names. If two users call their data file `test.txt`, then the unique-name rule is violated. For example, in one programming class, 23 students called the program for their second assignment `prog2.c`; another 11 called it `assign2.c`. Fortunately, most file systems support file names of up to 255 characters, so it is relatively easy to select unique file names.

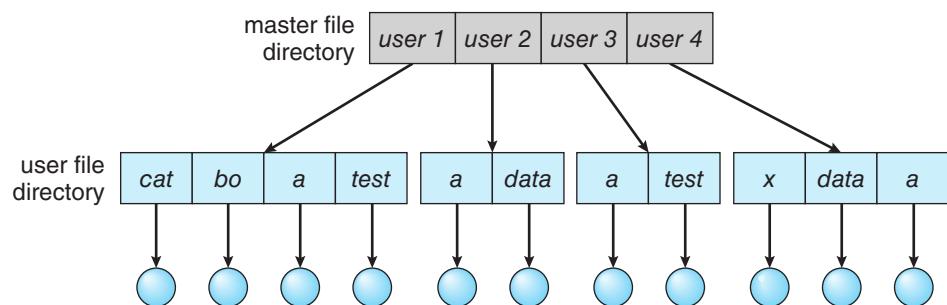
Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases. It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. Keeping track of so many files is a daunting task.

### 13.3.2 Two-Level Directory

As we have seen, a single-level directory often leads to confusion of file names among different users. The standard solution is to create a separate directory for each user.

In the two-level directory structure, each user has his own **user file directory (UFD)**. The UFDs have similar structures, but each lists only the files of a single user. When a user job starts or a user logs in, the system's **master file directory (MFD)** is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user (Figure 13.8).

When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique. To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name



**Figure 13.8** Two-level directory structure.

exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

The user directories themselves must be created and deleted as necessary. A special system program is run with the appropriate user name and account information. The program creates a new UFD and adds an entry for it to the MFD. The execution of this program might be restricted to system administrators. The allocation of disk space for user directories can be handled with the techniques discussed in Chapter 14 for files themselves.

Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. Isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files. Some systems simply do not allow local user files to be accessed by other users.

If access is to be permitted, one user must have the ability to name a file in another user's directory. To name a particular file uniquely in a two-level directory, we must give both the user name and the file name. A two-level directory can be thought of as a tree, or an inverted tree, of height 2. The root of the tree is the MFD. Its direct descendants are the UFDs. The descendants of the UFDs are the files themselves. The files are the leaves of the tree. Specifying a user name and a file name defines a path in the tree from the root (the MFD) to a leaf (the specified file). Thus, a user name and a file name define a **path name**. Every file in the system has a path name. To name a file uniquely, a user must know the path name of the file desired.

For example, if user A wishes to access her own test file named `test.txt`, she can simply refer to `test.txt`. To access the file named `test.txt` of user B (with directory-entry name `userb`), however, she might have to refer to `/userb/test.txt`. Every system has its own syntax for naming files in directories other than the user's own.

Additional syntax is needed to specify the volume of a file. For instance, in Windows a volume is specified by a letter followed by a colon. Thus, a file specification might be `C:\userb\test`. Some systems go even further and separate the volume, directory name, and file name parts of the specification. In OpenVMS, for instance, the file `login.com` might be specified as: `u:[sst.crissmeyer]login.com;1`, where `u` is the name of the volume, `sst` is the name of the directory, `crissmeyer` is the name of the subdirectory, and `1` is the version number. Other systems—such as UNIX and Linux—simply treat the volume name as part of the directory name. The first name given is that of the volume, and the rest is the directory and file. For instance, `/u/pgalvin/test` might specify volume `u`, directory `pgalvin`, and file `test`.

A special instance of this situation occurs with the system files. Programs provided as part of the system—loaders, assemblers, compilers, utility routines, libraries, and so on—are generally defined as files. When the appropriate commands are given to the operating system, these files are read by the loader and executed. Many command interpreters simply treat such a command as the name of a file to load and execute. In the directory system as we defined it above, this file name would be searched for in the current UFD. One solution would be to copy the system files into each UFD. However, copying all the system files would waste an enormous amount of space. (If the system files

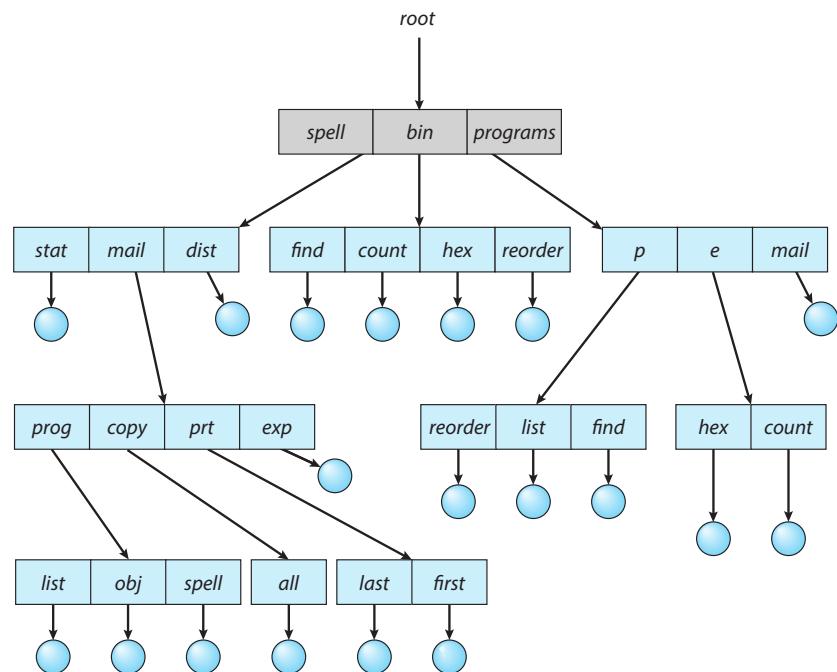
require 5 MB, then supporting 12 users would require  $5 \times 12 = 60$  MB just for copies of the system files.)

The standard solution is to complicate the search procedure slightly. A special user directory is defined to contain the system files (for example, user 0). Whenever a file name is given to be loaded, the operating system first searches the local UFD. If the file is found, it is used. If it is not found, the system automatically searches the special user directory that contains the system files. The sequence of directories searched when a file is named is called the **search path**. The search path can be extended to contain an unlimited list of directories to search when a command name is given. This method is the one most used in UNIX and Windows. Systems can also be designed so that each user has his own search path.

### 13.3.3 Tree-Structured Directories

Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height (Figure 13.9). This generalization allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.

A directory (or subdirectory) contains a set of files or subdirectories. In many implementations, a directory is simply another file, but it is treated in a special way. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special



**Figure 13.9** Tree-structured directory structure.

system calls are used to create and delete directories. In this case the operating system (or the file system code) implements another file format, that of a directory.

In normal use, each process has a current directory. The **current directory** should contain most of the files that are of current interest to the process. When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file. To change directories, a system call could be provided that takes a directory name as a parameter and uses it to redefine the current directory. Thus, the user can change her current directory whenever she wants. Other systems leave it to the application (say, a shell) to track and operate on a current directory, as each process could have different current directories.

The initial current directory of a user's login shell is designated when the user job starts or the user logs in. The operating system searches the accounting file (or some other predefined location) to find an entry for this user (for accounting purposes). In the accounting file is a pointer to (or the name of) the user's initial directory. This pointer is copied to a local variable for this user that specifies the user's initial current directory. From that shell, other processes can be spawned. The current directory of any subprocess is usually the current directory of the parent when it was spawned.

Path names can be of two types: absolute and relative. In UNIX and Linux, an **absolute path name** begins at the root (which is designated by an initial "/") and follows a path down to the specified file, giving the directory names on the path. A **relative path name** defines a path from the current directory. For example, in the tree-structured file system of Figure 13.9, if the current directory is /spell/mail, then the relative path name prt/first refers to the same file as does the absolute path name /spell/mail/prt/first.

Allowing a user to define her own subdirectories permits her to impose a structure on her files. This structure might result in separate directories for files associated with different topics (for example, a subdirectory was created to hold the text of this book) or different forms of information. For example, the directory **programs** may contain source programs; the directory **bin** may store all the binaries. (As a side note, executable files were known in many systems as "binaries" which led to them being stored in the **bin** directory.)

An interesting policy decision in a tree-structured directory concerns how to handle the deletion of a directory. If a directory is empty, its entry in the directory that contains it can simply be deleted. However, suppose the directory to be deleted is not empty but contains several files or subdirectories. One of two approaches can be taken. Some systems will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory. If any subdirectories exist, this procedure must be applied recursively to them, so that they can be deleted also. This approach can result in a substantial amount of work. An alternative approach, such as that taken by the UNIX **rm** command, is to provide an option: when a request is made to delete a directory, all that directory's files and subdirectories are also to be deleted. Either approach is fairly easy to implement; the choice is one of policy. The latter policy is more convenient, but it is also more dangerous, because an entire directory structure can be removed with one command. If that command

is issued in error, a large number of files and directories will need to be restored (assuming a backup exists).

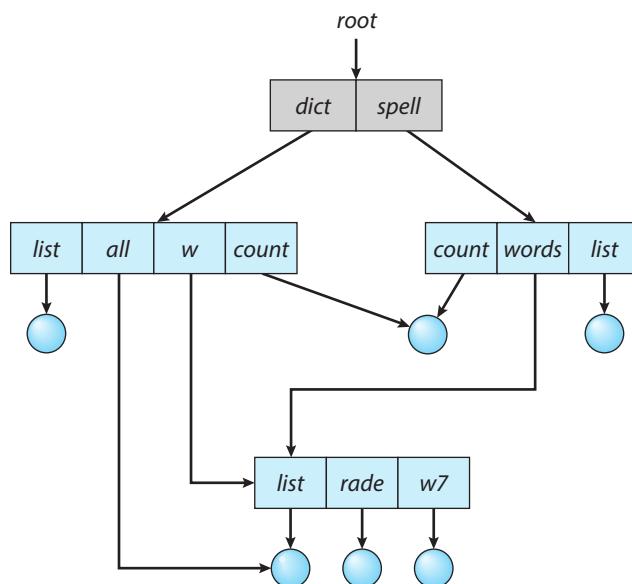
With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users. For example, user B can access a file of user A by specifying its path name. User B can specify either an absolute or a relative path name. Alternatively, user B can change her current directory to be user A's directory and access the file by its file name.

#### 13.3.4 Acyclic-Graph Directories

Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers. But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. In this situation, the common subdirectory should be *shared*. A shared directory or file exists in the file system in two (or more) places at once.

A tree structure prohibits the sharing of files or directories. An **acyclic graph**—that is, a graph with no cycles—allows directories to share subdirectories and files (Figure 13.10). The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.

It is important to note that a shared file (or directory) is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy. With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other. Sharing is



**Figure 13.10** Acyclic-graph directory structure.

particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.

When people are working as a team, all the files they want to share can be put into one directory. The home directory of each team member could contain this directory of shared files as a subdirectory. Even in the case of a single user, the user's file organization may require that some file be placed in different subdirectories. For example, a program written for a particular project should be both in the directory of all programs and in the directory for that project.

Shared files and subdirectories can be implemented in several ways. A common way, exemplified by UNIX systems, is to create a new directory entry called a link. A [link](#) is effectively a pointer to another file or subdirectory. For example, a link may be implemented as an absolute or a relative path name. When a reference to a file is made, we search the directory. If the directory entry is marked as a link, then the name of the real file is included in the link information. We [resolve](#) the link by using that path name to locate the real file. Links are easily identified by their format in the directory entry (or by having a special type on systems that support types) and are effectively indirect pointers. The operating system ignores these links when traversing directory trees to preserve the acyclic structure of the system.

Another common approach to implementing shared files is simply to duplicate all information about them in both sharing directories. Thus, both entries are identical and equal. Consider the difference between this approach and the creation of a link. The link is clearly different from the original directory entry; thus, the two are not equal. Duplicate directory entries, however, make the original and the copy indistinguishable. A major problem with duplicate directory entries is maintaining consistency when a file is modified.

An acyclic-graph directory structure is more flexible than a simple tree structure, but it is also more complex. Several problems must be considered carefully. A file may now have multiple absolute path names. Consequently, distinct file names may refer to the same file. This situation is similar to the aliasing problem for programming languages. If we are trying to traverse the entire file system—to find a file, to accumulate statistics on all files, or to copy all files to backup storage—this problem becomes significant, since we do not want to traverse shared structures more than once.

Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused? One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now-nonexistent file. Worse, if the remaining file pointers contain actual disk addresses, and the space is subsequently reused for other files, these dangling pointers may point into the middle of other files.

In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle. The deletion of a link need not affect the original file; only the link is removed. If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling. We can search for these links and remove them as well, but unless a list of the associated links is kept with each file, this search can be expensive. Alternatively, we can leave the links until an attempt is made to use them. At that time, we can determine that the file of the name given by the link does not exist and can fail to resolve the link name; the access is treated just as with any other illegal file name. (In this case, the system designer should consider carefully what to do when a file is

deleted and another file of the same name is created, before a symbolic link to the original file is used.) In the case of UNIX, symbolic links are left when a file is deleted, and it is up to the user to realize that the original file is gone or has been replaced. Microsoft Windows uses the same approach.

Another approach to deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted. We could keep a list of all references to a file (directory entries or symbolic links). When a link or a copy of the directory entry is established, a new entry is added to the file-reference list. When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty.

The trouble with this approach is the variable and potentially large size of the file-reference list. However, we really do not need to keep the entire list—we need to keep only a count of the number of references. Adding a new link or directory entry increments the reference count. Deleting a link or entry decrements the count. When the count is 0, the file can be deleted; there are no remaining references to it. The UNIX operating system uses this approach for nonsymbolic links (or **hard links**), keeping a reference count in the file information block (or inode; see Section C.7.2). By effectively prohibiting multiple references to directories, we maintain an acyclic-graph structure.

To avoid problems such as the ones just discussed, some systems simply do not allow shared directories or links.

### 13.3.5 General Graph Directory

A serious problem with using an acyclic-graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature. However, when we add links, the tree structure is destroyed, resulting in a simple graph structure (Figure 13.11).

The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse the graph and to determine when there are no more references to a file. We want to avoid traversing shared sections of an acyclic graph twice, mainly for performance reasons. If we have just searched a major shared subdirectory for a particular file without finding it, we want to avoid searching that subdirectory again; the second search would be a waste of time.

If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance. A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating. One solution is to limit arbitrarily the number of directories that will be accessed during a search.

A similar problem exists when we are trying to determine when a file can be deleted. With acyclic-graph directory structures, a value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted. However, when cycles exist, the reference count may not be 0 even when it is no longer possible to refer to a directory or file. This anomaly results from the possibility of self-referencing (or a cycle) in the directory structure. In this case, we generally need to use a **garbage collection**

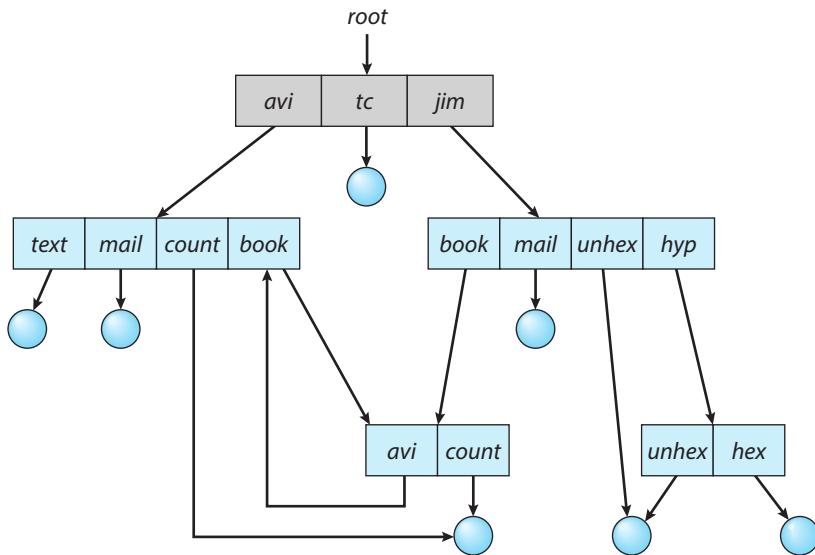


Figure 13.11 General graph directory.

scheme to determine when the last reference has been deleted and the disk space can be reallocated. Garbage collection involves traversing the entire file system, marking everything that can be accessed. Then, a second pass collects everything that is not marked onto a list of free space. (A similar marking procedure can be used to ensure that a traversal or search will cover everything in the file system once and only once.) Garbage collection for a disk-based file system, however, is extremely time consuming and is thus seldom attempted.

Garbage collection is necessary only because of possible cycles in the graph. Thus, an acyclic-graph structure is much easier to work with. The difficulty is to avoid cycles as new links are added to the structure. How do we know when a new link will complete a cycle? There are algorithms to detect cycles in graphs; however, they are computationally expensive, especially when the graph is on disk storage. A simpler algorithm in the special case of directories and links is to bypass links during directory traversal. Cycles are avoided, and no extra overhead is incurred.

## 13.4 Protection

When information is stored in a computer system, we want to keep it safe from physical damage (the issue of reliability) and improper access (the issue of protection).

Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed. File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes,

and vandalism. Files may be deleted accidentally. Bugs in the file-system software can also cause file contents to be lost. Reliability was covered in more detail in Chapter 11.

Protection can be provided in many ways. For a laptop system running a modern operating system, we might provide protection by requiring a user name and password authentication to access it, encrypting the secondary storage so even someone opening the laptop and removing the drive would have a difficult time accessing its data, and firewalling network access so that when it is in use it is difficult to break in via its network connection. In multiuser system, even valid access of the system needs more advanced mechanisms to allow only valid access of the data.

#### 13.4.1 Types of Access

The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection. Thus, we could provide complete protection by prohibiting access. Alternatively, we could provide free access with no protection. Both approaches are too extreme for general use. What is needed is controlled access.

Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:

- **Read.** Read from the file.
- **Write.** Write or rewrite the file.
- **Execute.** Load the file into memory and execute it.
- **Append.** Write new information at the end of the file.
- **Delete.** Delete the file and free its space for possible reuse.
- **List.** List the name and attributes of the file.
- **Attribute change.** Changing the attributes of the file.

Other operations, such as renaming, copying, and editing the file, may also be controlled. For many systems, however, these higher-level functions may be implemented by a system program that makes lower-level system calls. Protection is provided at only the lower level. For instance, copying a file may be implemented simply by a sequence of read requests. In this case, a user with read access can also cause the file to be copied, printed, and so on.

Many protection mechanisms have been proposed. Each has advantages and disadvantages and must be appropriate for its intended application. A small computer system that is used by only a few members of a research group, for example, may not need the same types of protection as a large corporate computer that is used for research, finance, and personnel operations. We discuss some approaches to protection in the following sections and present a more complete treatment in Chapter 17.

### 13.4.2 Access Control

The most common approach to the protection problem is to make access dependent on the identity of the user. Different users may need different types of access to a file or directory. The most general scheme to implement identity-dependent access is to associate with each file and directory an **access-control list (ACL)** specifying user names and the types of access allowed for each user. When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

This approach has the advantage of enabling complex access methodologies. The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access. This technique has two undesirable consequences:

- Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
- The directory entry, previously of fixed size, now must be of variable size, resulting in more complicated space management.

These problems can be resolved by use of a condensed version of the access list.

To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:

- **Owner.** The user who created the file is the owner.
- **Group.** A set of users who are sharing the file and need similar access is a group, or work group.
- **Other.** All other users in the system.

The most common recent approach is to combine access-control lists with the more general (and easier to implement) owner, group, and universe access-control scheme just described. For example, Solaris uses the three categories of access by default but allows access-control lists to be added to specific files and directories when more fine-grained access control is desired.

To illustrate, consider a person, Sara, who is writing a new book. She has hired three graduate students (Jim, Dawn, and Jill) to help with the project. The text of the book is kept in a file named `book.tex`. The protection associated with this file is as follows:

- Sara should be able to invoke all operations on the file.
- Jim, Dawn, and Jill should be able only to read and write the file; they should not be allowed to delete the file.
- All other users should be able to read, but not write, the file. (Sara is interested in letting as many people as possible read the text so that she can obtain feedback.)

### PERMISSIONS IN A UNIX SYSTEM

In the UNIX system, directory protection and file protection are handled similarly. Associated with each file and directory are three fields—owner, group, and universe—each consisting of the three bits **rwx**, where **r** controls read access, **w** controls write access, and **x** controls execution. Thus, a user can list the content of a subdirectory only if the **r** bit is set in the appropriate field. Similarly, a user can change his current directory to another current directory (say, **foo**) only if the **x** bit associated with the **foo** subdirectory is set in the appropriate field.

A sample directory listing from a UNIX environment is shown in below:

-rw-rw-r--	1	pbg	staff	31200	Sep 3 08:30	intro.ps
drwx-----	5	pbg	staff	512	Jul 8 09:33	private/
drwxrwxr-x	2	pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2	jwg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1	pbg	staff	9423	Feb 24 2017	program.c
-rwxr-xr-x	1	pbg	staff	20471	Feb 24 2017	program
drwx--x--x	4	tag	faculty	512	Jul 31 10:31	lib/
drwx-----	3	pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3	pbg	staff	512	Jul 8 09:35	test/

The first field describes the protection of the file or directory. A **d** as the first character indicates a subdirectory. Also shown are the number of links to the file, the owner's name, the group's name, the size of the file in bytes, the date of last modification, and finally the file's name (with optional extension).

To achieve such protection, we must create a new group—say, **text**—with members Jim, Dawn, and Jill. The name of the group, **text**, must then be associated with the file **book.tex**, and the access rights must be set in accordance with the policy we have outlined.

Now consider a visitor to whom Sara would like to grant temporary access to Chapter 1. The visitor cannot be added to the **text** group because that would give him access to all chapters. Because a file can be in only one group, Sara cannot add another group to Chapter 1. With the addition of access-control-list functionality, though, the visitor can be added to the access control list of Chapter 1.

For this scheme to work properly, permissions and access lists must be controlled tightly. This control can be accomplished in several ways. For example, in the UNIX system, groups can be created and modified only by the manager of the facility (or by any superuser). Thus, control is achieved through human interaction. Access lists are discussed further in Section 17.6.2.

With the more limited protection classification, only three fields are needed to define protection. Often, each field is a collection of bits, and each bit either allows or prevents the access associated with it. For example, the UNIX system defines three fields of three bits each—**rwx**, where **r** controls read access, **w** controls write access, and **x** controls execution. A separate field is kept for the

file owner, for the file's group, and for all other users. In this scheme, nine bits per file are needed to record protection information. Thus, for our example, the protection fields for the file `book.tex` are as follows: for the owner Sara, all bits are set; for the group `text`, the `r` and `w` bits are set; and for the universe, only the `r` bit is set.

One difficulty in combining approaches comes in the user interface. Users must be able to tell when the optional ACL permissions are set on a file. In the Solaris example, a “+” is appended to the regular permissions, as in:

```
19 -rw-r--r--+ 1 jim staff 130 May 25 22:13 file1
```

A separate set of commands, `setfacl` and `getfacl`, is used to manage the ACLs.

Windows users typically manage access-control lists via the GUI. Figure 13.12 shows a file-permission window on Windows 7 NTFS file system. In this example, user “guest” is specifically denied access to the file `ListPanel.java`.

Another difficulty is assigning precedence when permission and ACLs conflict. For example, if Walter is in a file's group, which has read permission, but the file has an ACL granting Walter read and write permission, should a write by Walter be granted or denied? Solaris and other operating systems give ACLs precedence (as they are more fine-grained and are not assigned by default). This follows the general rule that specificity should have priority.

#### 13.4.3 Other Protection Approaches

Another approach to the protection problem is to associate a password with each file. Just as access to the computer system is often controlled by a password, access to each file can be controlled in the same way. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file. The use of passwords has a few disadvantages, however. First, the number of passwords that a user needs to remember may become large, making the scheme impractical. Second, if only one password is used for all the files, then once it is discovered, all files are accessible; protection is on an all-or-none basis. Some systems allow a user to associate a password with a subdirectory, rather than with an individual file, to address this problem. More commonly encryption of a partition or individual files provides strong protection, but password management is key.

In a multilevel directory structure, we need to protect not only individual files but also collections of files in subdirectories; that is, we need to provide a mechanism for directory protection. The directory operations that must be protected are somewhat different from the file operations. We want to control the creation and deletion of files in a directory. In addition, we probably want to control whether a user can determine the existence of a file in a directory. Sometimes, knowledge of the existence and name of a file is significant in itself. Thus, listing the contents of a directory must be a protected operation. Similarly, if a path name refers to a file in a directory, the user must be allowed access to both the directory and the file. In systems where files may have numerous path names (such as acyclic and general graphs), a given user may have different access rights to a particular file, depending on the path name used.

# Virtual Machines



The term *virtualization* has many meanings, and aspects of virtualization permeate all aspects of computing. Virtual machines are one instance of this trend. Generally, with a virtual machine, guest operating systems and applications run in an environment that appears to them to be native hardware and that behaves toward them as native hardware would but that also protects, manages, and limits them.

This chapter delves into the uses, features, and implementation of virtual machines. Virtual machines can be implemented in several ways, and this chapter describes these options. One option is to add virtual machine support to the kernel. Because that implementation method is the most pertinent to this book, we explore it most fully. Additionally, hardware features provided by the CPU and even by I/O devices can support virtual machine implementation, so we discuss how those features are used by the appropriate kernel modules.

## CHAPTER OBJECTIVES

- Explore the history and benefits of virtual machines.
- Discuss the various virtual machine technologies.
- Describe the methods used to implement virtualization.
- Identify the most common hardware features that support virtualization and explain how they are used by operating-system modules.
- Discuss current virtualization research areas.

### 18.1 Overview

The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate environment is running on its own private computer. This concept may seem similar to the layered approach of operating system implementation (see Section 2.8.2), and in some ways it is. In the case of

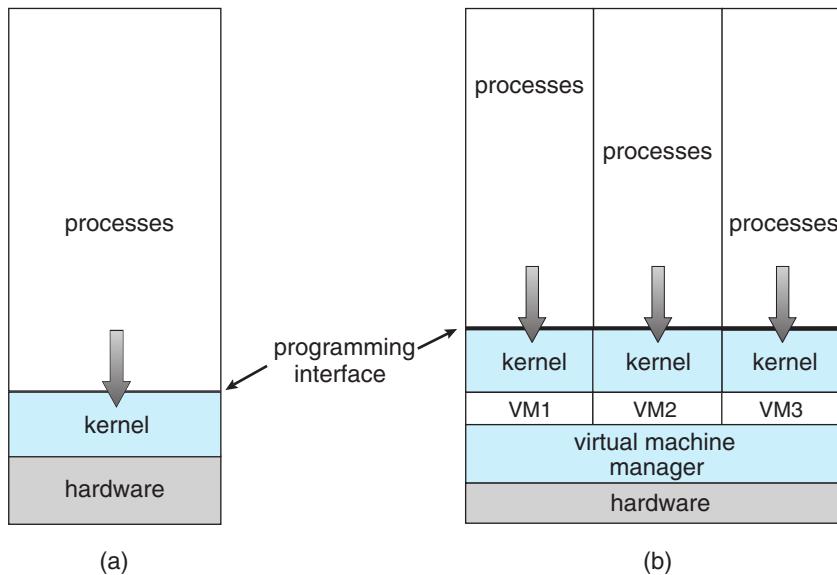
virtualization, there is a layer that creates a virtual system on which operating systems or applications can run.

Virtual machine implementations involve several components. At the base is the **host**, the underlying hardware system that runs the virtual machines. The **virtual machine manager (VMM)** (also known as a **hypervisor**) creates and runs virtual machines by providing an interface that is *identical* to the host (except in the case of paravirtualization, discussed later). Each **guest** process is provided with a virtual copy of the host (Figure 18.1). Usually, the guest process is in fact an operating system. A single physical machine can thus run multiple operating systems concurrently, each in its own virtual machine.

Take a moment to note that with virtualization, the definition of “operating system” once again blurs. For example, consider VMM software such as VMware ESX. This virtualization software is installed on the hardware, runs when the hardware boots, and provides services to applications. The services include traditional ones, such as scheduling and memory management, along with new types, such as migration of applications between systems. Furthermore, the applications are, in fact, guest operating systems. Is the VMware ESX VMM an operating system that, in turn, runs other operating systems? Certainly it acts like an operating system. For clarity, however, we call the component that provides virtual environments a VMM.

The implementation of VMMs varies greatly. Options include the following:

- Hardware-based solutions that provide support for virtual machine creation and management via firmware. These VMMs, which are commonly found in mainframe and large to midsized servers, are generally known as **type 0 hypervisors**. IBM LPARs and Oracle LDOMs are examples.



**Figure 18.1** System models. (a) Nonvirtual machine. (b) Virtual machine.

***INDIRECTION***

“All problems in computer science can be solved by another level of indirection”—David Wheeler

“. . . except for the problem of too many layers of indirection.”—Kevlin Henney

- Operating-system-like software built to provide virtualization, including VMware ESX (mentioned above), Joyent SmartOS, and Citrix XenServer. These VMMs are known as **type 1 hypervisors**.
- General-purpose operating systems that provide standard functions as well as VMM functions, including Microsoft Windows Server with HyperV and Red Hat Linux with the KVM feature. Because such systems have a feature set similar to type 1 hypervisors, they are also known as type 1.
- Applications that run on standard operating systems but provide VMM features to guest operating systems. These applications, which include VMware Workstation and Fusion, Parallels Desktop, and Oracle VirtualBox, are **type 2 hypervisors**.
- **Paravirtualization**, a technique in which the guest operating system is modified to work in cooperation with the VMM to optimize performance.
- **Programming-environment virtualization**, in which VMMs do not virtualize real hardware but instead create an optimized virtual system. This technique is used by Oracle Java and Microsoft.NET.
- **Emulators** that allow applications written for one hardware environment to run on a very different hardware environment, such as a different type of CPU.
- **Application containment**, which is not virtualization at all but rather provides virtualization-like features by segregating applications from the operating system. Oracle Solaris Zones, BSD Jails, and IBM AIX WPARs “contain” applications, making them more secure and manageable.

The variety of virtualization techniques in use today is a testament to the breadth, depth, and importance of virtualization in modern computing. Virtualization is invaluable for data-center operations, efficient application development, and software testing, among many other uses.

## 18.2 History

Virtual machines first appeared commercially on IBM mainframes in 1972. Virtualization was provided by the IBM VM operating system. This system has evolved and is still available. In addition, many of its original concepts are found in other systems, making it worth exploring.

IBM VM/370 divided a mainframe into multiple virtual machines, each running its own operating system. A major difficulty with the VM approach involved disk systems. Suppose that the physical machine had three disk drives but wanted to support seven virtual machines. Clearly, it could not allocate a disk drive to each virtual machine. The solution was to provide virtual disks—termed **minidisks** in IBM’s VM operating system. The minidisks were identical to the system’s hard disks in all respects except size. The system implemented each minidisk by allocating as many tracks on the physical disks as the minidisk needed.

Once the virtual machines were created, users could run any of the operating systems or software packages that were available on the underlying machine. For the IBM VM system, a user normally ran CMS—a single-user interactive operating system.

For many years after IBM introduced this technology, virtualization remained in its domain. Most systems could not support virtualization. However, a formal definition of virtualization helped to establish system requirements and a target for functionality. The virtualization requirements called for:

- **Fidelity.** A VMM provides an environment for programs that is essentially identical to the original machine.
- **Performance.** Programs running within that environment show only minor performance decreases.
- **Safety.** The VMM is in complete control of system resources.

These requirements still guide virtualization efforts today.

By the late 1990s, Intel 80x86 CPUs had become common, fast, and rich in features. Accordingly, developers launched multiple efforts to implement virtualization on that platform. Both **Xen** and **VMware** created technologies, still used today, to allow guest operating systems to run on the 80x86. Since that time, virtualization has expanded to include all common CPUs, many commercial and open-source tools, and many operating systems. For example, the open-source **VirtualBox** project (<http://www.virtualbox.org>) provides a program that runs on Intel x86 and AMD 64 CPUs and on Windows, Linux, macOS, and Solaris host operating systems. Possible guest operating systems include many versions of Windows, Linux, Solaris, and BSD, including even MS-DOS and IBM OS/2.

### 18.3 Benefits and Features

Several advantages make virtualization attractive. Most of them are fundamentally related to the ability to share the same hardware yet run several different execution environments (that is, different operating systems) concurrently.

One important advantage of virtualization is that the host system is protected from the virtual machines, just as the virtual machines are protected from each other. A virus inside a guest operating system might damage that operating system but is unlikely to affect the host or the other guests. Because

each virtual machine is almost completely isolated from all other virtual machines, there are almost no protection problems.

A potential disadvantage of isolation is that it can prevent sharing of resources. Two approaches to providing sharing have been implemented. First, it is possible to share a file-system volume and thus to share files. Second, it is possible to define a network of virtual machines, each of which can send information over the virtual communications network. The network is modeled after physical communication networks but is implemented in software. Of course, the VMM is free to allow any number of its guests to use physical resources, such as a physical network connection (with sharing provided by the VMM), in which case the allowed guests could communicate with each other via the physical network.

One feature common to most virtualization implementations is the ability to freeze, or **suspend**, a running virtual machine. Many operating systems provide that basic feature for processes, but VMMs go one step further and allow copies and **snapshots** to be made of the guest. The copy can be used to create a new VM or to move a VM from one machine to another with its current state intact. The guest can then **resume** where it was, as if on its original machine, creating a **clone**. The snapshot records a point in time, and the guest can be reset to that point if necessary (for example, if a change was made but is no longer wanted). Often, VMMs allow many snapshots to be taken. For example, snapshots might record a guest's state every day for a month, making restoration to any of those snapshot states possible. These abilities are used to good advantage in virtual environments.

A virtual machine system is a perfect vehicle for operating-system research and development. Normally, changing an operating system is a difficult task. Operating systems are large and complex programs, and a change in one part may cause obscure bugs to appear in some other part. The power of the operating system makes changing it particularly dangerous. Because the operating system executes in kernel mode, a wrong change in a pointer could cause an error that would destroy the entire file system. Thus, it is necessary to test all changes to the operating system carefully.

Of course, the operating system runs on and controls the entire machine, so the system must be stopped and taken out of use while changes are made and tested. This period is commonly called **system-development time**. Since it makes the system unavailable to users, system-development time on shared systems is often scheduled late at night or on weekends, when system load is low.

A virtual-machine system can eliminate much of this latter problem. System programmers are given their own virtual machine, and system development is done on the virtual machine instead of on a physical machine. Normal system operation is disrupted only when a completed and tested change is ready to be put into production.

Another advantage of virtual machines for developers is that multiple operating systems can run concurrently on the developer's workstation. This virtualized workstation allows for rapid porting and testing of programs in varying environments. In addition, multiple versions of a program can run, each in its own isolated operating system, within one system. Similarly, quality-assurance engineers can test their applications in multiple environments without buying, powering, and maintaining a computer for each environment.

A major advantage of virtual machines in production data-center use is system **consolidation**, which involves taking two or more separate systems and running them in virtual machines on one system. Such physical-to-virtual conversions result in resource optimization, since many lightly used systems can be combined to create one more heavily used system.

Consider, too, that management tools that are part of the VMM allow system administrators to manage many more systems than they otherwise could. A virtual environment might include 100 physical servers, each running 20 virtual servers. Without virtualization, 2,000 servers would require several system administrators. With virtualization and its tools, the same work can be managed by one or two administrators. One of the tools that make this possible is **templating**, in which one standard virtual machine image, including an installed and configured guest operating system and applications, is saved and used as a source for multiple running VMs. Other features include managing the patching of all guests, backing up and restoring the guests, and monitoring their resource use.

Virtualization can improve not only resource utilization but also resource management. Some VMMs include a **live migration** feature that moves a running guest from one physical server to another without interrupting its operation or active network connections. If a server is overloaded, live migration can thus free resources on the source host while not disrupting the guest. Similarly, when host hardware must be repaired or upgraded, guests can be migrated to other servers, the evacuated host can be maintained, and then the guests can be migrated back. This operation occurs without downtime and without interruption to users.

Think about the possible effects of virtualization on how applications are deployed. If a system can easily add, remove, and move a virtual machine, then why install applications on that system directly? Instead, the application could be preinstalled on a tuned and customized operating system in a virtual machine. This method would offer several benefits for application developers. Application management would become easier, less tuning would be required, and technical support of the application would be more straightforward. System administrators would find the environment easier to manage as well. Installation would be simple, and redeploying the application to another system would be much easier than the usual steps of uninstalling and reinstalling. For widespread adoption of this methodology to occur, though, the format of virtual machines must be standardized so that any virtual machine will run on any virtualization platform. The “Open Virtual Machine Format” is an attempt to provide such standardization, and it could succeed in unifying virtual machine formats.

Virtualization has laid the foundation for many other advances in computer facility implementation, management, and monitoring. **Cloud computing**, for example, is made possible by virtualization in which resources such as CPU, memory, and I/O are provided as services to customers using Internet technologies. By using APIs, a program can tell a cloud computing facility to create thousands of VMs, all running a specific guest operating system and application, that others can access via the Internet. Many multiuser games, photo-sharing sites, and other web services use this functionality.

In the area of desktop computing, virtualization is enabling desktop and laptop computer users to connect remotely to virtual machines located in

remote data centers and access their applications as if they were local. This practice can increase security, because no data are stored on local disks at the user's site. The cost of the user's computing resource may also decrease. The user must have networking, CPU, and some memory, but all that these system components need to do is display an image of the guest as it runs remotely (via a protocol such as [RDP](#)). Thus, they need not be expensive, high-performance components. Other uses of virtualization are sure to follow as it becomes more prevalent and hardware support continues to improve.

## 18.4 Building Blocks

Although the virtual machine concept is useful, it is difficult to implement. Much work is required to provide an *exact* duplicate of the underlying machine. This is especially a challenge on dual-mode systems, where the underlying machine has only user mode and kernel mode. In this section, we examine the building blocks that are needed for efficient virtualization. Note that these building blocks are not required by type 0 hypervisors, as discussed in Section 18.5.2.

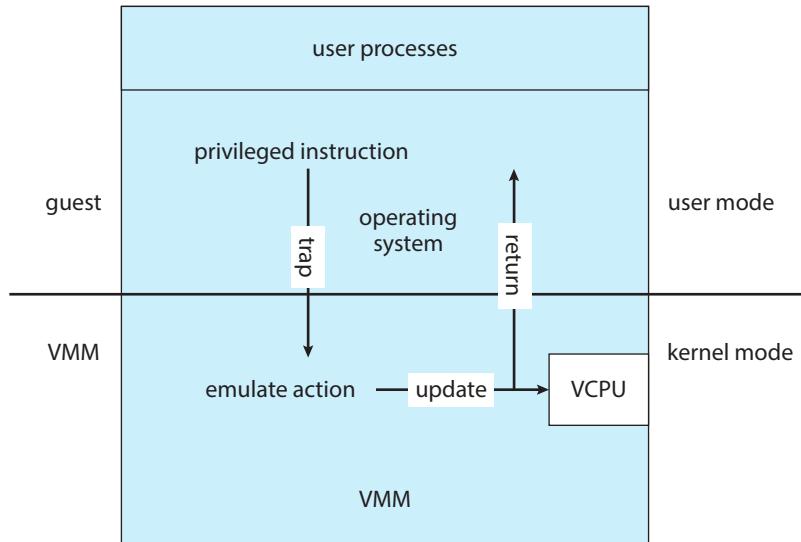
The ability to virtualize depends on the features provided by the CPU. If the features are sufficient, then it is possible to write a VMM that provides a guest environment. Otherwise, virtualization is impossible. VMMs use several techniques to implement virtualization, including trap-and-emulate and binary translation. We discuss each of these techniques in this section, along with the hardware support needed to support virtualization.

As you read the section, keep in mind that an important concept found in most virtualization options is the implementation of a [virtual CPU \(VCPU\)](#). The VCPU does not execute code. Rather, it represents the state of the CPU as the guest machine believes it to be. For each guest, the VMM maintains a VCPU representing that guest's current CPU state. When the guest is context-switched onto a CPU by the VMM, information from the VCPU is used to load the right context, much as a general-purpose operating system would use the PCB.

### 18.4.1 Trap-and-Emulate

On a typical dual-mode system, the virtual machine guest can execute only in user mode (unless extra hardware support is provided). The kernel, of course, runs in kernel mode, and it is not safe to allow user-level code to run in kernel mode. Just as the physical machine has two modes, so must the virtual machine. Consequently, we must have a virtual user mode and a virtual kernel mode, both of which run in physical user mode. Those actions that cause a transfer from user mode to kernel mode on a real machine (such as a system call, an interrupt, or an attempt to execute a privileged instruction) must also cause a transfer from virtual user mode to virtual kernel mode in the virtual machine.

How can such a transfer be accomplished? The procedure is as follows: When the kernel in the guest attempts to execute a privileged instruction, that is an error (because the system is in user mode) and causes a trap to the VMM in the real machine. The VMM gains control and executes (or "emulates") the action that was attempted by the guest kernel on the part of the guest. It



**Figure 18.2** Trap-and-emulate virtualization implementation.

then returns control to the virtual machine. This is called the **trap-and-emulate** method and is shown in Figure 18.2.

With privileged instructions, time becomes an issue. All nonprivileged instructions run natively on the hardware, providing the same performance for guests as native applications. Privileged instructions create extra overhead, however, causing the guest to run more slowly than it would natively. In addition, the CPU is being multiprogrammed among many virtual machines, which can further slow down the virtual machines in unpredictable ways.

This problem has been approached in various ways. IBM VM, for example, allows normal instructions for the virtual machines to execute directly on the hardware. Only the privileged instructions (needed mainly for I/O) must be emulated and hence execute more slowly. In general, with the evolution of hardware, the performance of trap-and-emulate functionality has been improved, and cases in which it is needed have been reduced. For example, many CPUs now have extra modes added to their standard dual-mode operation. The VCPU need not keep track of what mode the guest operating system is in, because the physical CPU performs that function. In fact, some CPUs provide guest CPU state management in hardware, so the VMM need not supply that functionality, removing the extra overhead.

#### 18.4.2 Binary Translation

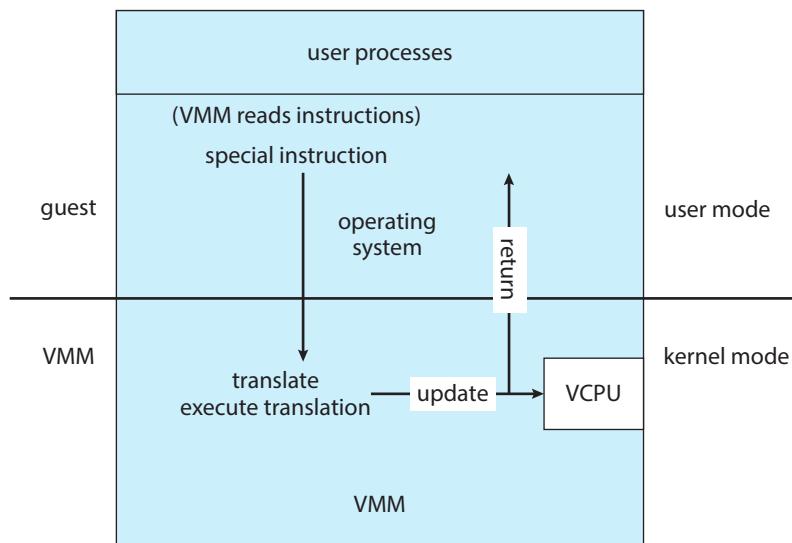
Some CPUs do not have a clean separation of privileged and nonprivileged instructions. Unfortunately for virtualization implementers, the Intel x86 CPU line is one of them. No thought was given to running virtualization on the x86 when it was designed. (In fact, the first CPU in the family—the Intel 4004, released in 1971—was designed to be the core of a calculator.) The chip has maintained backward compatibility throughout its lifetime, preventing changes that would have made virtualization easier through many generations.

Let's consider an example of the problem. The command `popf` loads the flag register from the contents of the stack. If the CPU is in privileged mode, all of the flags are replaced from the stack. If the CPU is in user mode, then only some flags are replaced, and others are ignored. Because no trap is generated if `popf` is executed in user mode, the trap-and-emulate procedure is rendered useless. Other x86 instructions cause similar problems. For the purposes of this discussion, we will call this set of instructions *special instructions*. As recently as 1998, using the trap-and-emulate method to implement virtualization on the x86 was considered impossible because of these special instructions.

This previously insurmountable problem was solved with the implementation of the **binary translation** technique. Binary translation is fairly simple in concept but complex in implementation. The basic steps are as follows:

1. If the guest VCPU is in user mode, the guest can run its instructions natively on a physical CPU.
2. If the guest VCPU is in kernel mode, then the guest believes that it is running in kernel mode. The VMM examines every instruction the guest executes in virtual kernel mode by reading the next few instructions that the guest is going to execute, based on the guest's program counter. Instructions other than special instructions are run natively. Special instructions are translated into a new set of instructions that perform the equivalent task—for example, changing the flags in the VCPU.

Binary translation is shown in Figure 18.3. It is implemented by translation code within the VMM. The code reads native binary instructions dynamically from the guest, on demand, and generates native binary code that executes in place of the original code.



**Figure 18.3** Binary translation virtualization implementation.

The basic method of binary translation just described would execute correctly but perform poorly. Fortunately, the vast majority of instructions would execute natively. But how could performance be improved for the other instructions? We can turn to a specific implementation of binary translation, the VMware method, to see one way of improving performance. Here, caching provides the solution. The replacement code for each instruction that needs to be translated is cached. All later executions of that instruction run from the translation cache and need not be translated again. If the cache is large enough, this method can greatly improve performance.

Let's consider another issue in virtualization: memory management, specifically the page tables. How can the VMM keep page-table state both for guests that believe they are managing the page tables and for the VMM itself? A common method, used with both trap-and-emulate and binary translation, is to use [nested page tables \(NPTs\)](#). Each guest operating system maintains one or more page tables to translate from virtual to physical memory. The VMM maintains NPTs to represent the guest's page-table state, just as it creates a VCPU to represent the guest's CPU state. The VMM knows when the guest tries to change its page table, and it makes the equivalent change in the NPT. When the guest is on the CPU, the VMM puts the pointer to the appropriate NPT into the appropriate CPU register to make that table the active page table. If the guest needs to modify the page table (for example, fulfilling a page fault), then that operation must be intercepted by the VMM and appropriate changes made to the nested and system page tables. Unfortunately, the use of NPTs can cause TLB misses to increase, and many other complexities need to be addressed to achieve reasonable performance.

Although it might seem that the binary translation method creates large amounts of overhead, it performed well enough to launch a new industry aimed at virtualizing Intel x86-based systems. VMware tested the performance impact of binary translation by booting one such system, Windows XP, and immediately shutting it down while monitoring the elapsed time and the number of translations produced by the binary translation method. The result was 950,000 translations, taking 3 microseconds each, for a total increase of 3 seconds (about 5 percent) over native execution of Windows XP. To achieve that result, developers used many performance improvements that we do not discuss here. For more information, consult the bibliographical notes at the end of this chapter.

#### 18.4.3 Hardware Assistance

Without some level of hardware support, virtualization would be impossible. The more hardware support available within a system, the more feature-rich and stable the virtual machines can be and the better they can perform. In the Intel x86 CPU family, Intel added new virtualization support (the [VT-x](#) instructions) in successive generations beginning in 2005. Now, binary translation is no longer needed.

In fact, all major general-purpose CPUs now provide extended hardware support for virtualization. For example, AMD virtualization technology ([AMD-V](#)) has appeared in several AMD processors starting in 2006. It defines two new modes of operation—host and guest—thus moving from a dual-mode to a

multimode processor. The VMM can enable host mode, define the characteristics of each guest virtual machine, and then switch the system to guest mode, passing control of the system to a guest operating system that is running in the virtual machine. In guest mode, the virtualized operating system thinks it is running on native hardware and sees whatever devices are included in the host's definition of the guest. If the guest tries to access a virtualized resource, then control is passed to the VMM to manage that interaction. The functionality in Intel VT-x is similar, providing root and nonroot modes, equivalent to host and guest modes. Both provide guest VCPU state data structures to load and save guest CPU state automatically during guest context switches. In addition, **virtual machine control structures (VMCSs)** are provided to manage guest and host state, as well as various guest execution controls, exit controls, and information about why guests exit back to the host. In the latter case, for example, a nested page-table violation caused by an attempt to access unavailable memory can result in the guest's exit.

AMD and Intel have also addressed memory management in the virtual environment. With AMD's RVI and Intel's EPT memory-management enhancements, VMMs no longer need to implement software NPTs. In essence, these CPUs implement nested page tables in hardware to allow the VMM to fully control paging while the CPUs accelerate the translation from virtual to physical addresses. The NPTs add a new layer, one representing the guest's view of logical-to-physical address translation. The CPU page-table walking function (traversing the data structure to find the desired data) includes this new layer as necessary, walking through the guest table to the VMM table to find the physical address desired. A TLB miss results in a performance penalty, because more tables (the guest and host page tables) must be traversed to complete the lookup. Figure 18.4 shows the extra translation work performed by the hardware to translate from a guest virtual address to a final physical address.

I/O is another area improved by hardware assistance. Consider that the standard direct-memory-access (DMA) controller accepts a target memory address and a source I/O device and transfers data between the two without operating-system action. Without hardware assistance, a guest might try to set up a DMA transfer that affects the memory of the VMM or other guests. In CPUs that provide hardware-assisted DMA (such as Intel CPUs with VT-d), even DMA has a level of indirection. First, the VMM sets up **protection domains** to tell the CPU which physical memory belongs to each guest. Next, it assigns the I/O devices to the protection domains, allowing them direct access to those memory regions and only those regions. The hardware then transforms the address in a DMA request issued by an I/O device to the host physical memory address associated with the I/O. In this manner, DMA transfers are passed through between a guest and a device without VMM interference.

Similarly, interrupts must be delivered to the appropriate guest and must not be visible to other guests. By providing an interrupt remapping feature, CPUs with virtualization hardware assistance automatically deliver an interrupt destined for a guest to a core that is currently running a thread of that guest. That way, the guest receives interrupts without any need for the VMM to intercede in their delivery. Without interrupt remapping, malicious guests could generate interrupts that could be used to gain control of the host system. (See the bibliographical notes at the end of this chapter for more details.)

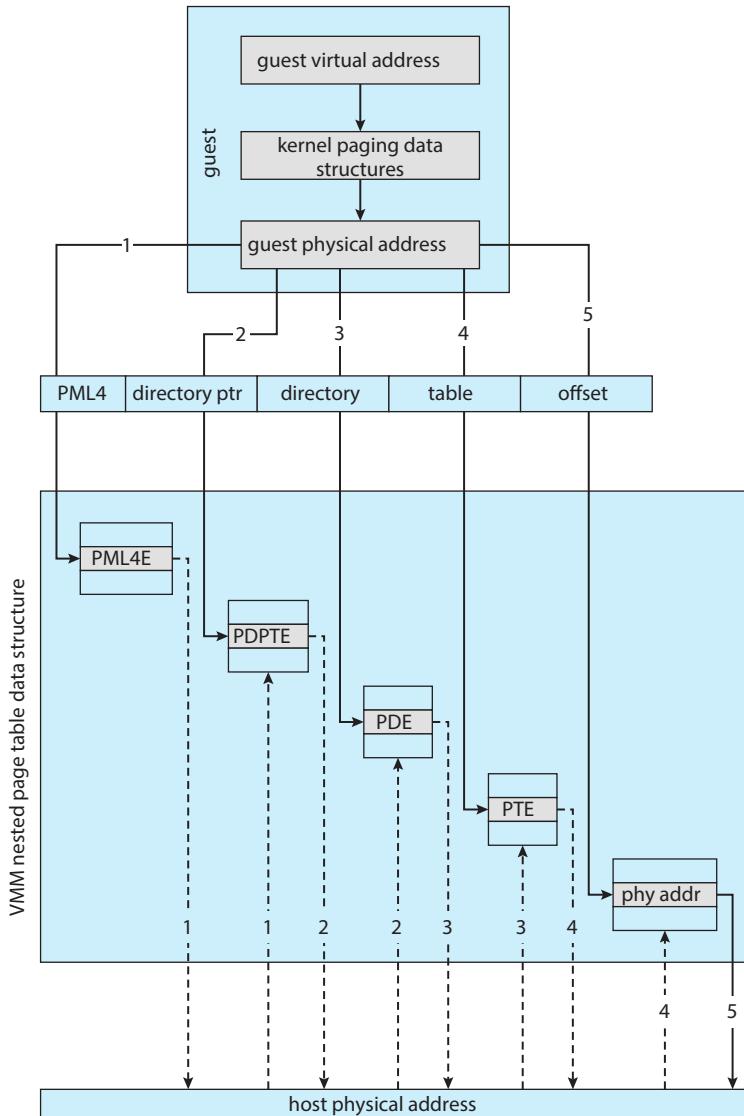


Figure 18.4 Nested page tables.

ARM architectures, specifically ARM v8 (64-bit) take a slightly different approach to hardware support of virtualization. They provide an entire exception level—EL2—which is even more privileged than that of the kernel (EL1). This allows the running of a secluded hypervisor, with its own MMU access and interrupt trapping. To allow for paravirtualization, a special instruction (HVC) is added. It allows the hypervisor to be called from guest kernels. This instruction can only be called from within kernel mode (EL1).

An interesting side effect of hardware-assisted virtualization is that it allows for the creation of thin hypervisors. A good example is macOS's hypervisor framework (“HyperVisor.framework”), which is an operating-system-supplied library that allows the creation of virtual machines in a few lines of

code. The actual work is done via system calls, which have the kernel call the privileged virtualization CPU instructions on behalf of the hypervisor process, allowing management of virtual machines without the hypervisor needing to load a kernel module of its own to execute those calls.

## 18.5 Types of VMs and Their Implementations

We've now looked at some of the techniques used to implement virtualization. Next, we consider the major types of virtual machines, their implementation, their functionality, and how they use the building blocks just described to create a virtual environment. Of course, the hardware on which the virtual machines are running can cause great variation in implementation methods. Here, we discuss the implementations in general, with the understanding that VMMs take advantage of hardware assistance where it is available.

### 18.5.1 The Virtual Machine Life Cycle

Let's begin with the virtual machine life cycle. Whatever the hypervisor type, at the time a virtual machine is created, its creator gives the VMM certain parameters. These parameters usually include the number of CPUs, amount of memory, networking details, and storage details that the VMM will take into account when creating the guest. For example, a user might want to create a new guest with two virtual CPUs, 4 GB of memory, 10 GB of disk space, one network interface that gets its IP address via DHCP, and access to the DVD drive.

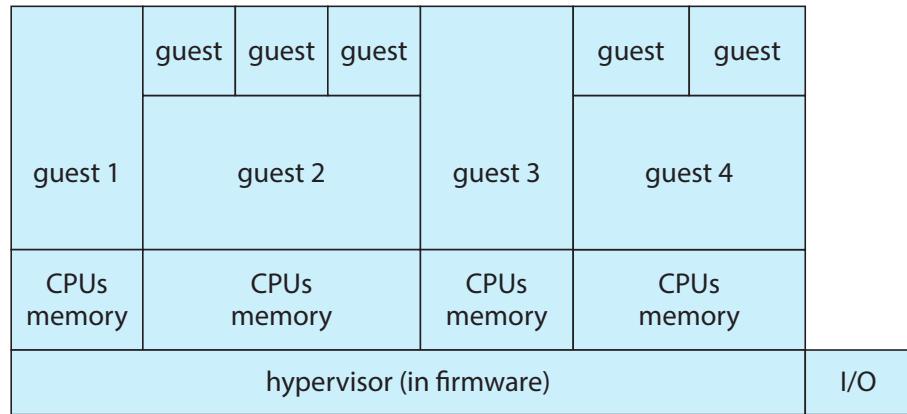
The VMM then creates the virtual machine with those parameters. In the case of a type 0 hypervisor, the resources are usually dedicated. In this situation, if there are not two virtual CPUs available and unallocated, the creation request in our example will fail. For other hypervisor types, the resources are dedicated or virtualized, depending on the type. Certainly, an IP address cannot be shared, but the virtual CPUs are usually multiplexed on the physical CPUs as discussed in Section 18.6.1. Similarly, memory management usually involves allocating more memory to guests than actually exists in physical memory. This is more complicated and is described in Section 18.6.2.

Finally, when the virtual machine is no longer needed, it can be deleted. When this happens, the VMM first frees up any used disk space and then removes the configuration associated with the virtual machine, essentially forgetting the virtual machine.

These steps are quite simple compared with building, configuring, running, and removing physical machines. Creating a virtual machine from an existing one can be as easy as clicking the “clone” button and providing a new name and IP address. This ease of creation can lead to **virtual machine sprawl**, which occurs when there are so many virtual machines on a system that their use, history, and state become confusing and difficult to track.

### 18.5.2 Type 0 Hypervisor

Type 0 hypervisors have existed for many years under many names, including “partitions” and “domains.” They are a hardware feature, and that brings its own positives and negatives. Operating systems need do nothing special to take advantage of their features. The VMM itself is encoded in the firmware and

**Figure 18.5** Type 0 hypervisor.

loaded at boot time. In turn, it loads the guest images to run in each partition. The feature set of a type 0 hypervisor tends to be smaller than those of the other types because it is implemented in hardware. For example, a system might be split into four virtual systems, each with dedicated CPUs, memory, and I/O devices. Each guest believes that it has dedicated hardware because it does, simplifying many implementation details.

I/O presents some difficulty, because it is not easy to dedicate I/O devices to guests if there are not enough. What if a system has two Ethernet ports and more than two guests, for example? Either all guests must get their own I/O devices, or the system must provide I/O device sharing. In these cases, the hypervisor manages shared access or grants all devices to a **control partition**. In the control partition, a guest operating system provides services (such as networking) via daemons to other guests, and the hypervisor routes I/O requests appropriately. Some type 0 hypervisors are even more sophisticated and can move physical CPUs and memory between running guests. In these cases, the guests are paravirtualized, aware of the virtualization and assisting in its execution. For example, a guest must watch for signals from the hardware or VMM that a hardware change has occurred, probe its hardware devices to detect the change, and add or subtract CPUs or memory from its available resources.

Because type 0 virtualization is very close to raw hardware execution, it should be considered separately from the other methods discussed here. A type 0 hypervisor can run multiple guest operating systems (one in each hardware partition). All of those guests, because they are running on raw hardware, can in turn be VMMs. Essentially, each guest operating system in a type 0 hypervisor is a native operating system with a subset of hardware made available to it. Because of that, each can have its own guest operating systems (Figure 18.5). Other types of hypervisors usually cannot provide this virtualization-within-virtualization functionality.

### 18.5.3 Type 1 Hypervisor

Type 1 hypervisors are commonly found in company data centers and are, in a sense, becoming “the data-center operating system.” They are special-purpose operating systems that run natively on the hardware, but rather than providing

system calls and other interfaces for running programs, they create, run, and manage guest operating systems. In addition to running on standard hardware, they can run on type 0 hypervisors, but not on other type 1 hypervisors. Whatever the platform, guests generally do not know they are running on anything but the native hardware.

Type 1 hypervisors run in kernel mode, taking advantage of hardware protection. Where the host CPU allows, they use multiple modes to give guest operating systems their own control and improved performance. They implement device drivers for the hardware they run on, since no other component could do so. Because they are operating systems, they must also provide CPU scheduling, memory management, I/O management, protection, and even security. Frequently, they provide APIs, but those APIs support applications in guests or external applications that supply features like backups, monitoring, and security. Many type 1 hypervisors are closed-source commercial offerings, such as VMware ESX, while some are open source or hybrids of open and closed source, such as Citrix XenServer and its open Xen counterpart.

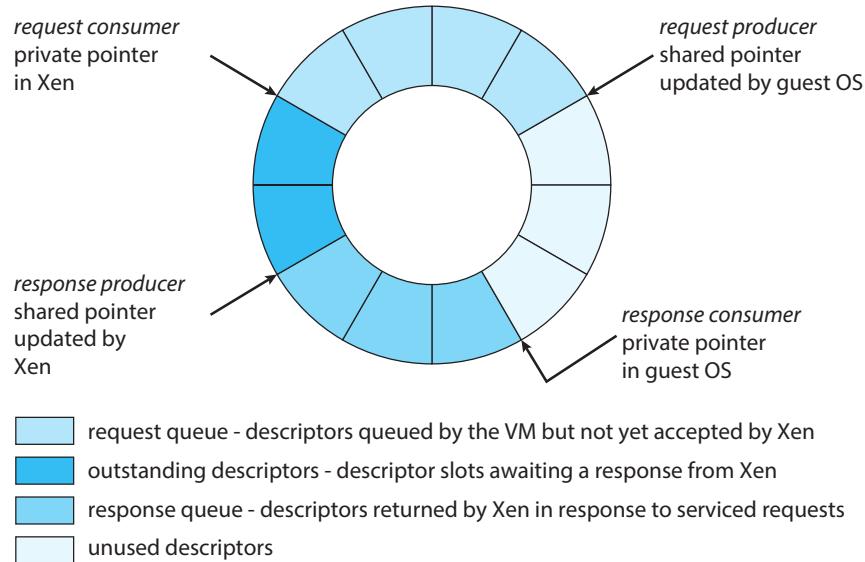
By using type 1 hypervisors, data-center managers can control and manage the operating systems and applications in new and sophisticated ways. An important benefit is the ability to consolidate more operating systems and applications onto fewer systems. For example, rather than having ten systems running at 10 percent utilization each, a data center might have one server manage the entire load. If utilization increases, guests and their applications can be moved to less-loaded systems live, without interruption of service. Using snapshots and cloning, the system can save the states of guests and duplicate those states—a much easier task than restoring from backups or installing manually or via scripts and tools. The price of this increased manageability is the cost of the VMM (if it is a commercial product), the need to learn new management tools and methods, and the increased complexity.

Another type of type 1 hypervisor includes various general-purpose operating systems with VMM functionality. Here, an operating system such as Red-Hat Enterprise Linux, Windows, or Oracle Solaris performs its normal duties as well as providing a VMM allowing other operating systems to run as guests. Because of their extra duties, these hypervisors typically provide fewer virtualization features than other type 1 hypervisors. In many ways, they treat a guest operating system as just another process, but they provide special handling when the guest tries to execute special instructions.

#### 18.5.4 Type 2 Hypervisor

Type 2 hypervisors are less interesting to us as operating-system explorers, because there is very little operating-system involvement in these application-level virtual machine managers. This type of VMM is simply another process run and managed by the host, and even the host does not know that virtualization is happening within the VMM.

Type 2 hypervisors have limits not associated with some of the other types. For example, a user needs administrative privileges to access many of the hardware assistance features of modern CPUs. If the VMM is being run by a standard user without additional privileges, the VMM cannot take advantage of these features. Due to this limitation, as well as the extra overhead of running



**Figure 18.6** Xen I/O via shared circular buffer.<sup>1</sup>

a general-purpose operating system as well as guest operating systems, type 2 hypervisors tend to have poorer overall performance than type 0 or type 1.

As is often the case, the limitations of type 2 hypervisors also provide some benefits. They run on a variety of general-purpose operating systems, and running them requires no changes to the host operating system. A student can use a type 2 hypervisor, for example, to test a non-native operating system without replacing the native operating system. In fact, on an Apple laptop, a student could have versions of Windows, Linux, Unix, and less common operating systems all available for learning and experimentation.

### 18.5.5 Paravirtualization

As we've seen, paravirtualization works differently than the other types of virtualization. Rather than try to trick a guest operating system into believing it has a system to itself, paravirtualization presents the guest with a system that is similar but not identical to the guest's preferred system. The guest must be modified to run on the paravirtualized virtual hardware. The gain for this extra work is more efficient use of resources and a smaller virtualization layer.

The Xen VMM became the leader in paravirtualization by implementing several techniques to optimize the performance of guests as well as of the host system. For example, as mentioned earlier, some VMMs present virtual devices to guests that appear to be real devices. Instead of taking that approach, the Xen VMM presented clean and simple device abstractions that allow efficient I/O as well as good I/O-related communication between the guest and the VMM. For

---

<sup>1</sup>Barham, Paul. "Xen and the Art of Virtualization". SOSP '03 Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, p 164-177. ©2003 Association for Computing Machinery, Inc

each device used by each guest, there was a circular buffer shared by the guest and the VMM via shared memory. Read and write data are placed in this buffer, as shown in Figure 18.6.

For memory management, Xen did not implement nested page tables. Rather, each guest had its own set of page tables, set to read-only. Xen required the guest to use a specific mechanism, a **hypcall** from the guest to the hypervisor VMM, when a page-table change was needed. This meant that the guest operating system's kernel code must have been changed from the default code to these Xen-specific methods. To optimize performance, Xen allowed the guest to queue up multiple page-table changes asynchronously via hypercalls and then checked to ensure that the changes were complete before continuing operation.

Xen allowed virtualization of x86 CPUs without the use of binary translation, instead requiring modifications in the guest operating systems like the one described above. Over time, Xen has taken advantage of hardware features supporting virtualization. As a result, it no longer requires modified guests and essentially does not need the paravirtualization method. Paravirtualization is still used in other solutions, however, such as type 0 hypervisors.

### 18.5.6 Programming-Environment Virtualization

Another kind of virtualization, based on a different execution model, is the virtualization of programming *environments*. Here, a programming language is designed to run within a custom-built virtualized environment. For example, Oracle's Java has many features that depend on its running in the **Java virtual machine (JVM)**, including specific methods for security and memory management.

If we define virtualization as including only duplication of hardware, this is not really virtualization at all. But we need not limit ourselves to that definition. Instead, we can define a virtual environment, based on APIs, that provides a set of features we want to have available for a particular language and programs written in that language. Java programs run within the JVM environment, and the JVM is compiled to be a native program on systems on which it runs. This arrangement means that Java programs are written once and then can run on any system (including all of the major operating systems) on which a JVM is available. The same can be said of **interpreted languages**, which run inside programs that read each instruction and interpret it into native operations.

### 18.5.7 Emulation

Virtualization is probably the most common method for running applications designed for one operating system on a different operating system, but on the same CPU. This method works relatively efficiently because the applications were compiled for the instruction set that the target system uses.

But what if an application or operating system needs to run on a different CPU? Here, it is necessary to translate all of the source CPU's instructions so that they are turned into the equivalent instructions of the target CPU. Such an environment is no longer virtualized but rather is fully emulated.

**Emulation** is useful when the host system has one system architecture and the guest system was compiled for a different architecture. For example,

suppose a company has replaced its outdated computer system with a new system but would like to continue to run certain important programs that were compiled for the old system. The programs could be run in an emulator that translates each of the outdated system's instructions into the native instruction set of the new system. Emulation can increase the life of programs and allow us to explore old architectures without having an actual old machine.

As may be expected, the major challenge of emulation is performance. Instruction-set emulation may run an order of magnitude slower than native instructions, because it may take ten instructions on the new system to read, parse, and simulate an instruction from the old system. Thus, unless the new machine is ten times faster than the old, the program running on the new machine will run more slowly than it did on its native hardware. Another challenge for emulator writers is that it is difficult to create a correct emulator because, in essence, this task involves writing an entire CPU in software.

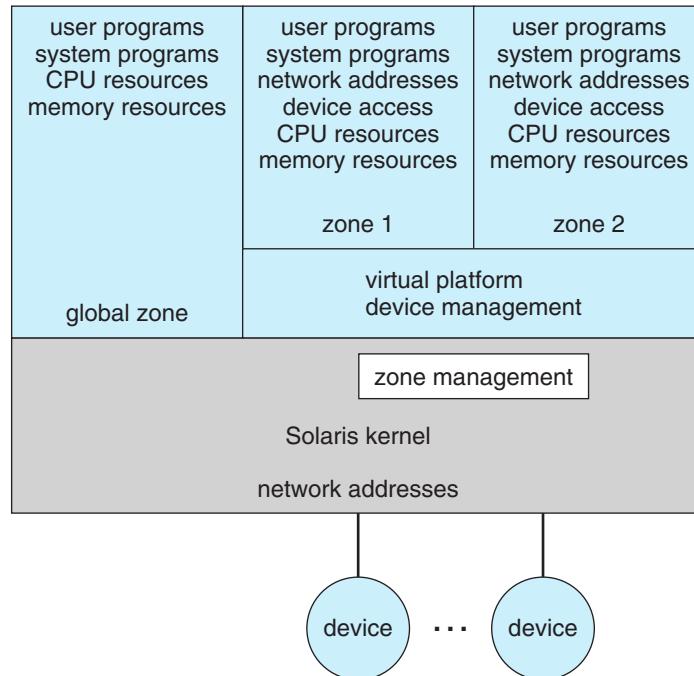
In spite of these challenges, emulation is very popular, particularly in gaming circles. Many popular video games were written for platforms that are no longer in production. Users who want to run those games frequently can find an emulator of such a platform and then run the game unmodified within the emulator. Modern systems are so much faster than old game consoles that even the Apple iPhone has game emulators and games available to run within them.

#### 18.5.8 Application Containment

The goal of virtualization in some instances is to provide a method to segregate applications, manage their performance and resource use, and create an easy way to start, stop, move, and manage them. In such cases, perhaps full-fledged virtualization is not needed. If the applications are all compiled for the same operating system, then we do not need complete virtualization to provide these features. We can instead use application containment.

Consider one example of application containment. Starting with version 10, Oracle Solaris has included [containers](#), or [zones](#), that create a virtual layer between the operating system and the applications. In this system, only one kernel is installed, and the hardware is not virtualized. Rather, the operating system and its devices are virtualized, providing processes within a zone with the impression that they are the only processes on the system. One or more containers can be created, and each can have its own applications, network stacks, network address and ports, user accounts, and so on. CPU and memory resources can be divided among the zones and the system-wide processes. Each zone, in fact, can run its own scheduler to optimize the performance of its applications on the allotted resources. Figure 18.7 shows a Solaris 10 system with two containers and the standard “global” user space.

Containers are much lighter weight than other virtualization methods. That is, they use fewer system resources and are faster to instantiate and destroy, more similar to processes than virtual machines. For this reason, they are becoming more commonly used, especially in cloud computing. FreeBSD was perhaps the first operating system to include a container-like feature (called “jails”), and AIX has a similar feature. Linux added the [LXC](#) container feature in 2014. It is now included in the common Linux distributions via



**Figure 18.7** Solaris 10 with two zones.

a flag in the `clone()` system call. (The source code for LXC is available at <https://linuxcontainers.org/lxc/downloads>.)

Containers are also easy to automate and manage, leading to orchestration tools like **Docker** and **Kubernetes**. Orchestration tools are means of automating and coordinating systems and services. Their aim is to make it simple to run entire suites of distributed applications, just as operating systems make it simple to run a single program. These tools offer rapid deployment of full applications, consisting of many processes within containers, and also offer monitoring and other administration features. For more on Docker, see <https://www.docker.com/what-docker>. Information about Kubernetes can be found at <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes>.

## 18.6 Virtualization and Operating-System Components

Thus far, we have explored the building blocks of virtualization and the various types of virtualization. In this section, we take a deeper dive into the operating-system aspects of virtualization, including how the VMM provides core operating-system functions like scheduling, I/O, and memory management. Here, we answer questions such as these: How do VMMs schedule CPU use when guest operating systems believe they have dedicated CPUs? How can memory management work when many guests require large amounts of memory?

### 18.6.1 CPU Scheduling

A system with virtualization, even a single-CPU system, frequently acts like a multiprocessor system. The virtualization software presents one or more virtual CPUs to each of the virtual machines running on the system and then schedules the use of the physical CPUs among the virtual machines.

The significant variations among virtualization technologies make it difficult to summarize the effect of virtualization on scheduling. First, let's consider the general case of VMM scheduling. The VMM has a number of physical CPUs available and a number of threads to run on those CPUs. The threads can be VMM threads or guest threads. Guests are configured with a certain number of virtual CPUs at creation time, and that number can be adjusted throughout the life of the VM. When there are enough CPUs to allocate the requested number to each guest, the VMM can treat the CPUs as dedicated and schedule only a given guest's threads on that guest's CPUs. In this situation, the guests act much like native operating systems running on native CPUs.

Of course, in other situations, there may not be enough CPUs to go around. The VMM itself needs some CPU cycles for guest management and I/O management and can steal cycles from the guests by scheduling its threads across all of the system CPUs, but the impact of this action is relatively minor. More difficult is the case of **overcommitment**, in which the guests are configured for more CPUs than exist in the system. Here, a VMM can use standard scheduling algorithms to make progress on each thread but can also add a fairness aspect to those algorithms. For example, if there are six hardware CPUs and twelve guest-allocated CPUs, the VMM can allocate CPU resources proportionally, giving each guest half of the CPU resources it believes it has. The VMM can still present all twelve virtual CPUs to the guests, but in mapping them onto physical CPUs, the VMM can use its scheduler to distribute them appropriately.

Even given a scheduler that provides fairness, any guest operating-system scheduling algorithm that assumes a certain amount of progress in a given amount of time will most likely be negatively affected by virtualization. Consider a time-sharing operating system that tries to allot 100 milliseconds to each time slice to give users a reasonable response time. Within a virtual machine, this operating system receives only what CPU resources the virtualization system gives it. A 100-millisecond time slice may take much more than 100 milliseconds of virtual CPU time. Depending on how busy the system is, the time slice may take a second or more, resulting in very poor response times for users logged into that virtual machine. The effect on a real-time operating system can be even more serious.

The net outcome of such scheduling is that individual virtualized operating systems receive only a portion of the available CPU cycles, even though they believe they are receiving all of the cycles and indeed are scheduling all of the cycles. Commonly, the time-of-day clocks in virtual machines are incorrect because timers take longer to trigger than they would on dedicated CPUs. Virtualization can thus undo the scheduling-algorithm efforts of the operating systems within virtual machines.

To correct for this, the VMM makes an application available for each type of operating system that the system administrator can install into the guests. This application corrects clock drift and can have other functions, such as virtual device management.

### 18.6.2 Memory Management

Efficient memory use in general-purpose operating systems is a major key to performance. In virtualized environments, there are more users of memory (the guests and their applications, as well as the VMM), leading to more pressure on memory use. Further adding to this pressure is the fact that VMMs typically overcommit memory, so that the total memory allocated to guests exceeds the amount that physically exists in the system. The extra need for efficient memory use is not lost on the implementers of VMMs, who take extensive measures to ensure the optimal use of memory.

For example, VMware ESX uses several methods of memory management. Before memory optimization can occur, the VMM must establish how much real memory each guest should use. To do that, the VMM first evaluates each guest's maximum memory size. General-purpose operating systems do not expect the amount of memory in the system to change, so VMMs must maintain the illusion that the guest has that amount of memory. Next, the VMM computes a target real-memory allocation for each guest based on the configured memory for that guest and other factors, such as overcommitment and system load. It then uses the three low-level mechanisms listed below to reclaim memory from the guests

1. Recall that a guest believes it controls memory allocation via its page-table management, whereas in reality the VMM maintains a nested page table that translates the guest page table to the real page table. The VMM can use this extra level of indirection to optimize the guest's use of memory without the guest's knowledge or help. One approach is to provide double paging. Here, the VMM has its own page-replacement algorithms and loads pages into a backing store that the guest believes is physical memory. Of course, the VMM knows less about the guest's memory access patterns than the guest does, so its paging is less efficient, creating performance problems. VMMs do use this method when other methods are not available or are not providing enough free memory. However, it is not the preferred approach.
2. A common solution is for the VMM to install in each guest a pseudo-device driver or kernel module that the VMM controls. (A **pseudo-device driver** uses device-driver interfaces, appearing to the kernel to be a device driver, but does not actually control a device. Rather, it is an easy way to add kernel-mode code without directly modifying the kernel.) This **balloon memory manager** communicates with the VMM and is told to allocate or deallocate memory. If told to allocate, it allocates memory and tells the operating system to pin the allocated pages into physical memory. Recall that pinning locks a page into physical memory so that it cannot be moved or paged out. To the guest, these pinned pages appear to decrease the amount of physical memory it has available, creating memory pressure. The guest then may free up other physical memory to be sure it has enough free memory. Meanwhile, the VMM, knowing that the pages pinned by the balloon process will never be used, removes those physical pages from the guest and allocates them to another guest. At the same time, the guest is using its own memory-management and paging algorithms to manage the available memory, which is the most

efficient option. If memory pressure within the entire system decreases, the VMM will tell the balloon process within the guest to unpin and free some or all of the memory, allowing the guest more pages for its use.

3. Another common method for reducing memory pressure is for the VMM to determine if the same page has been loaded more than once. If this is the case, the VMM reduces the number of copies of the page to one and maps the other users of the page to that one copy. VMware, for example, randomly samples guest memory and creates a hash for each page sampled. That hash value is a “thumbprint” of the page. The hash of every page examined is compared with other hashes stored in a hash table. If there is a match, the pages are compared byte by byte to see if they really are identical. If they are, one page is freed, and its logical address is mapped to the other’s physical address. This technique might seem at first to be ineffective, but consider that guests run operating systems. If multiple guests run the same operating system, then only one copy of the active operating-system pages need be in memory. Similarly, multiple guests could be running the same set of applications, again a likely source of memory sharing.

The overall effect of these mechanisms is to enable guests to behave and perform as if they had the full amount of memory requested, although in reality they have less.

### 18.6.3 I/O

In the area of I/O, hypervisors have some leeway and can be less concerned with how they represent the underlying hardware to their guests. Because of the wide variation in I/O devices, operating systems are used to dealing with varying and flexible I/O mechanisms. For example, an operating system’s device-driver mechanism provides a uniform interface to the operating system whatever the I/O device. Device-driver interfaces are designed to allow third-party hardware manufacturers to provide device drivers connecting their devices to the operating system. Usually, device drivers can be dynamically loaded and unloaded. Virtualization takes advantage of this built-in flexibility by providing specific virtualized devices to guest operating systems.

As described in Section 18.5, VMMs vary greatly in how they provide I/O to their guests. I/O devices may be dedicated to guests, for example, or the VMM may have device drivers onto which it maps guest I/O. The VMM may also provide idealized device drivers to guests. In this case, the guest sees an easy-to-control device, but in reality that simple device driver communicates to the VMM, which sends the requests to a more complicated real device through a more complex real device driver. I/O in virtual environments is complicated and requires careful VMM design and implementation.

Consider the case of a hypervisor and hardware combination that allows devices to be dedicated to a guest and allows the guest to access those devices directly. Of course, a device dedicated to one guest is not available to any other guests, but this direct access can still be useful in some circumstances. The reason to allow direct access is to improve I/O performance. The less the hypervisor has to do to enable I/O for its guests, the faster the I/O can occur. With type 0 hypervisors that provide direct device access, guests can often

run at the same speed as native operating systems. When type 0 hypervisors instead provide shared devices, performance may suffer.

With direct device access in type 1 and 2 hypervisors, performance can be similar to that of native operating systems if certain hardware support is present. The hardware needs to provide DMA pass-through with facilities like VT-d, as well as direct interrupt delivery (interrupts going directly to the guests). Given how frequently interrupts occur, it should be no surprise that the guests on hardware without these features have worse performance than if they were running natively.

In addition to direct access, VMMs provide shared access to devices. Consider a disk drive to which multiple guests have access. The VMM must provide protection while the device is being shared, assuring that a guest can access only the blocks specified in the guest's configuration. In such instances, the VMM must be part of every I/O, checking it for correctness as well as routing the data to and from the appropriate devices and guests.

In the area of networking, VMMs also have work to do. General-purpose operating systems typically have one Internet protocol (IP) address, although they sometimes have more than one—for example, to connect to a management network, backup network, and production network. With virtualization, each guest needs at least one IP address, because that is the guest's main mode of communication. Therefore, a server running a VMM may have dozens of addresses, and the VMM acts as a virtual switch to route the network packets to the addressed guests.

The guests can be “directly” connected to the network by an IP address that is seen by the broader network (this is known as **bridging**). Alternatively, the VMM can provide a **network address translation (NAT)** address. The NAT address is local to the server on which the guest is running, and the VMM provides routing between the broader network and the guest. The VMM also provides firewalling to guard connections between guests within the system and between guests and external systems.

#### 18.6.4 Storage Management

An important question in determining how virtualization works is this: If multiple operating systems have been installed, what and where is the boot disk? Clearly, virtualized environments need to approach storage management differently than do native operating systems. Even the standard multiboot method of slicing the boot disk into partitions, installing a boot manager in one partition, and installing each other operating system in another partition is not sufficient, because partitioning has limits that would prevent it from working for tens or hundreds of virtual machines.

Once again, the solution to this problem depends on the type of hypervisor. Type 0 hypervisors often allow root disk partitioning, partly because these systems tend to run fewer guests than other systems. Alternatively, a disk manager may be part of the control partition, and that disk manager may provide disk space (including boot disks) to the other partitions.

Type 1 hypervisors store the guest root disk (and configuration information) in one or more files in the file systems provided by the VMM. Type 2 hypervisors store the same information in the host operating system's file systems. In essence, a **disk image**, containing all of the contents of the root disk

of the guest, is contained in one file in the VMM. Aside from the potential performance problems that causes, this is a clever solution, because it simplifies copying and moving guests. If the administrator wants a duplicate of the guest (for testing, for example), she simply copies the associated disk image of the guest and tells the VMM about the new copy. Booting the new virtual machine brings up an identical guest. Moving a virtual machine from one system to another that runs the same VMM is as simple as halting the guest, copying the image to the other system, and starting the guest there.

Guests sometimes need more disk space than is available in their root disk image. For example, a nonvirtualized database server might use several file systems spread across many disks to store various parts of the database. Virtualizing such a database usually involves creating several files and having the VMM present those to the guest as disks. The guest then executes as usual, with the VMM translating the disk I/O requests coming from the guest into file I/O commands to the correct files.

Frequently, VMMs provide a mechanism to capture a physical system as it is currently configured and convert it to a guest that the VMM can manage and run. This **physical-to-virtual (P-to-V)** conversion reads the disk blocks of the physical system's disks and stores them in files on the VMM's system or on shared storage that the VMM can access. VMMs also provide a **virtual-to-physical (V-to-P)** procedure for converting a guest to a physical system. This procedure is sometimes needed for debugging: a problem could be caused by the VMM or associated components, and the administrator could attempt to solve the problem by removing virtualization from the problem variables. V-to-P conversion can take the files containing all of the guest data and generate disk blocks on a physical disk, recreating the guest as a native operating system and applications. Once the testing is concluded, the original system can be reused for other purposes when the virtual machine returns to service, or the virtual machine can be deleted and the original system can continue to run.

### 18.6.5 Live Migration

One feature not found in general-purpose operating systems but found in type 0 and type 1 hypervisors is the live migration of a running guest from one system to another. We mentioned this capability earlier. Here, we explore the details of how live migration works and why VMMs can implement it relatively easily while general-purpose operating systems, in spite of some research attempts, cannot.

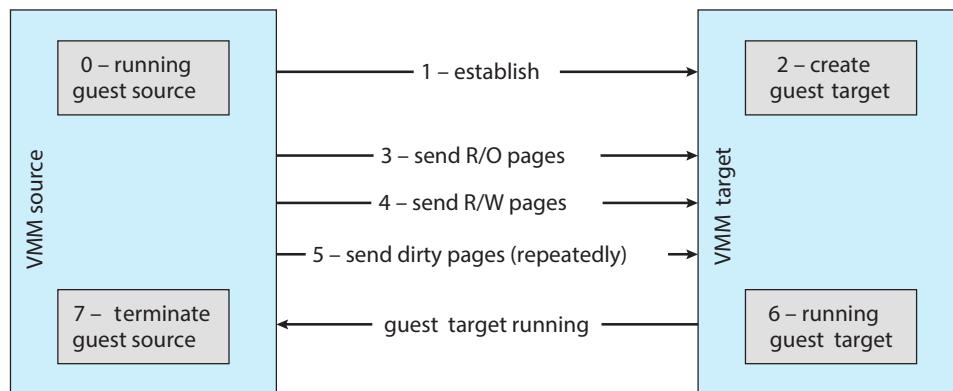
First, let's consider how live migration works. A running guest on one system is copied to another system running the same VMM. The copy occurs with so little interruption of service that users logged in to the guest, as well as network connections to the guest, continue without noticeable impact. This rather astonishing ability is very powerful in resource management and hardware administration. After all, compare it with the steps necessary without virtualization: we must warn users, shut down the processes, possibly move the binaries, and restart the processes on the new system. Only then can users access the services again. With live migration, we can decrease the load on an overloaded system or make hardware or system changes with no discernable disruption for users.

Live migration is made possible by the well-defined interface between each guest and the VMM and the limited state the VMM maintains for the guest. The VMM migrates a guest via the following steps:

1. The source VMM establishes a connection with the target VMM and confirms that it is allowed to send a guest.
2. The target creates a new guest by creating a new VCPU, new nested page table, and other state storage.
3. The source sends all read-only memory pages to the target.
4. The source sends all read-write pages to the target, marking them as clean.
5. The source repeats step 4, because during that step some pages were probably modified by the guest and are now dirty. These pages need to be sent again and marked again as clean.
6. When the cycle of steps 4 and 5 becomes very short, the source VMM freezes the guest, sends the VCPU's final state, other state details, and the final dirty pages, and tells the target to start running the guest. Once the target acknowledges that the guest is running, the source terminates the guest.

This sequence is shown in Figure 18.8.

We conclude this discussion with a few interesting details and limitations concerning live migration. First, for network connections to continue uninterrupted, the network infrastructure needs to understand that a MAC address—the hardware networking address—can move between systems. Before virtualization, this did not happen, as the MAC address was tied to physical hardware. With virtualization, the MAC must be movable for existing networking connections to continue without resetting. Modern network switches understand this and route traffic wherever the MAC address is, even accommodating a move.



**Figure 18.8** Live migration of a guest between two servers.

**Started on** Wednesday, 19 April 2023, 8:46 PM

**State** Finished

**Completed on** Wednesday, 19 April 2023, 9:26 PM

**Time taken** 39 mins 31 secs

**Grade** 15.60 out of 30.00 (51.99%)

Question 1

Incorrect

Mark 0.00 out of 2.00

Select all the correct statements about synchronization primitives.

Select one or more:

- a. Blocking means moving the process to a wait queue and calling scheduler
- b. Blocking means moving the process to a wait queue and spinning ✗
- c. Semaphores are always a good substitute for spinlocks ✗
- d. Spinlocks are good for multiprocessor scenarios, for small critical sections
- e. All synchronization primitives are implemented essentially with some hardware assistance.
- f. Blocking means one process passing over control to another process ✗
- g. Semaphores can be used for synchronization scenarios like ordered execution ✓
- h. Mutexes can be implemented without any hardware assistance
- i. Spinlocks consume CPU time
- j. Mutexes can be implemented using spinlock
- k. Thread that is going to block should not be holding any spinlock
- l. Mutexes can be implemented using blocking and wakeup

Your answer is incorrect.

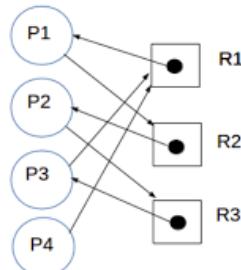
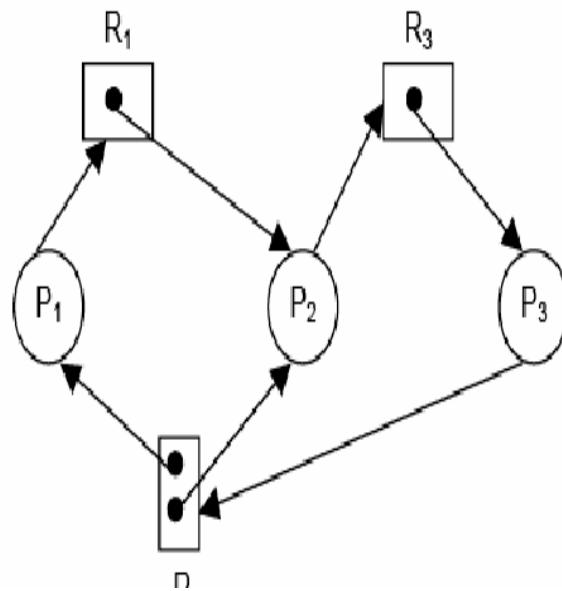
The correct answers are: Spinlocks are good for multiprocessor scenarios, for small critical sections, Spinlocks consume CPU time, Semaphores can be used for synchronization scenarios like ordered execution, Mutexes can be implemented using spinlock, Mutexes can be implemented using blocking and wakeup, Thread that is going to block should not be holding any spinlock, Blocking means moving the process to a wait queue and calling scheduler, All synchronization primitives are implemented essentially with some hardware assistance.

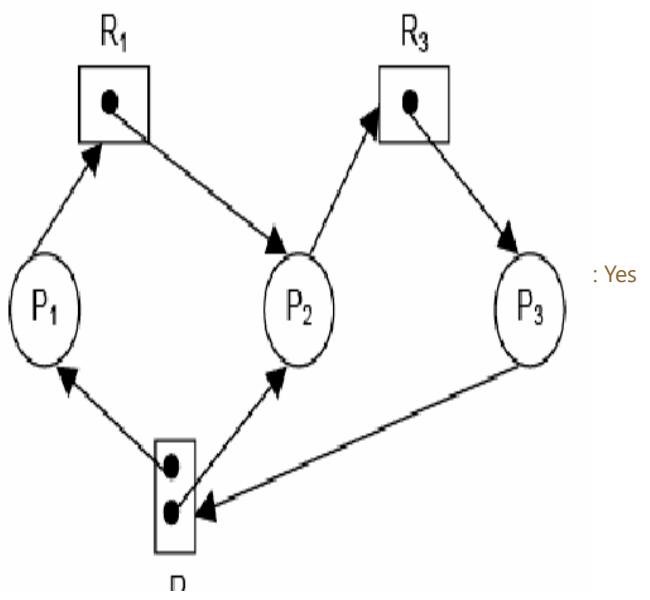
**Question 2**

Partially correct

Mark 0.50 out of 1.00

For each of the resource allocation diagram shown,  
infer whether the graph contains at least one deadlock or not.

**Yes**      **No**



<b>Started on</b>	Wednesday, 19 April 2023, 6:16 PM
<b>State</b>	Finished
<b>Completed on</b>	Wednesday, 19 April 2023, 8:31 PM
<b>Time taken</b>	2 hours 14 mins
<b>Overdue</b>	14 mins 47 secs
<b>Grade</b>	<b>14.08</b> out of 30.00 ( <b>46.92%</b> )

Question **1**

Incorrect

Mark 0.00 out of 1.00

Select all correct statements about file system recovery (without journaling) programs e.g. fsck

Select one or more:

- a. Recovery programs are needed only if the file system has a delayed-write policy. ✓
- b. Recovery is possible due to redundancy in file system data structures
- c. It is possible to lose data as part of recovery ✓
- d. A recovery program, most typically, builds the file system data structure and checks for inconsistencies ✓
- e. Recovery programs recalculate most of the metadata summaries (e.g. free inode count) ✓
- f. They are used to recover deleted files ✗
- g. They may take very long time to execute ✓
- h. Even with a write-through policy, it is possible to need a recovery program.
- i. They can make changes to the on-disk file system

Your answer is incorrect.

The correct answers are: Recovery is possible due to redundancy in file system data structures, A recovery program, most typically, builds the file system data structure and checks for inconsistencies, It is possible to lose data as part of recovery, They may take very long time to execute, They can make changes to the on-disk file system, Recovery programs recalculate most of the metadata summaries (e.g. free inode count), Recovery programs are needed only if the file system has a delayed-write policy., Even with a write-through policy, it is possible to need a recovery program.

**Question 2**

Partially correct

Mark 0.70 out of 1.00

Mark the statements as True or False, w.r.t. thrashing

True	False	
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Thrashing is particular to demand paging systems, and does not apply to pure paging systems.
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	During thrashing the CPU is under-utilised as most time is spent in I/O
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Thrashing occurs when the total size of all process's locality exceeds total memory size.
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Thrashing can be limited if local replacement is used.
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Thrashing occurs because some process is doing lot of disk I/O.
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The working set model is an attempt at approximating the locality of a process.
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Thrashing can occur even if entire memory is not in use.
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	mmap() solves the problem of thrashing.
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.

Thrashing is particular to demand paging systems, and does not apply to pure paging systems.: True

During thrashing the CPU is under-utilised as most time is spent in I/O: True

Thrashing occurs when the total size of all process's locality exceeds total memory size.: True

Thrashing can be limited if local replacement is used.: True

Thrashing occurs because some process is doing lot of disk I/O.: False

The working set model is an attempt at approximating the locality of a process.: True

Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.: True

Thrashing can occur even if entire memory is not in use.: False

mmap() solves the problem of thrashing.: False

Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.: False

**Question 3**

Partially correct

Mark 0.25 out of 1.00

Match each suggested semaphore implementation (discussed in class)

with the problems that it faces

```
struct semaphore {  
    int val;  
    spinlock lk;  
};  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0) {  
        spinunlock(&(s->sl));  
        spinlock(&(s->sl));  
    }  
    (s->val)--;  
    spinunlock(&(s->sl));  
}
```

too much spinning, bounded wait not guaranteed ✓

```
struct semaphore {  
    int val;  
    spinlock lk;  
};  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0)  
    ;  
    (s->val)--;  
    spinunlock(&(s->sl));  
}
```

blocks holding a spinlock ✗

```

struct semaphore {
    int val;
    spinlock lk;
    list l;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->s1 = 0;
}

block(semaphore *s) {
    listappend(s->l, current);
    spinunlock(&(s->s1));
    schedule();
}

wait(semaphore *s) {
    spinlock(&(s->s1));
    while(s->val <=0) {
        block(s);
    }
    (s->val)--;
    spinunlock(&(s->s1));
}

signal(semaphore *s) {
    spinlock(*(s->s1));
    (s->val)++;
    x = dequeue(s->s1) and enqueue(readyq, x);
    spinunlock(*(s->s1));
}

```

too much spinning, bounded wait not guaranteed



```

struct semaphore {
    int val;
    spinlock lk;
    list l;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->s1 = 0;
}

block(semaphore *s) {
    listappend(s->l, current);
    schedule();
}

wait(semaphore *s) {
    spinlock(&(s->s1));
    while(s->val <=0) {
        block(s);
    }
    (s->val)--;
    spinunlock(&(s->s1));
}

```

deadlock



Your answer is partially correct.

You have correctly selected 1.

The correct answer is:

```

struct semaphore {
    int val;
    spinlock lk;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        spinunlock(&(s->sl));
        spinlock(&(s->sl));
    }
    (s->val)--;
    spinunlock(&(s->sl));
}

```

→ too much spinning, bounded wait not guaranteed,

```

struct semaphore {
    int val;
    spinlock lk;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0)
    ;
    (s->val)--;
    spinunlock(&(s->sl));
}

```

→ deadlock,

```

struct semaphore {
    int val;
    spinlock lk;
    list l;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

block(semaphore *s) {
    listappend(s->l, current);
    spinunlock(&(s->sl));
    schedule();
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        block(s);
    }
    (s->val)--;
    spinunlock(&(s->sl));
}

signal(semaphore *s) {
    spinlock(*(&(s->sl)));
    (s->val)++;
    x = dequeue(s->sl) and enqueue(readyq, x);
    spinunlock(*(&(s->sl)));
}

```

→ not holding lock after unblock,

```
struct semaphore {
    int val;
    spinlock lk;
    list l;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

block(semaphore *s) {
    listappend(s->l, current);
    schedule();
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        block(s);
    }
    (s->val)--;
    spinunlock(&(s->sl));
}
```

→ blocks holding a spinlock

**Question 4**

Partially correct

Mark 0.88 out of 1.00

Mark the statements as True or False, w.r.t. passing of arguments to system calls in xv6 code.

True	False	
<input checked="" type="radio"/>	<input type="radio"/> 	The arguments to system call originally reside on process stack.
<input checked="" type="radio"/>	<input type="radio"/> 	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer
<input checked="" type="radio"/>	<input type="radio"/> 	The arguments are accessed in the kernel code using esp on the trapframe.
<input type="radio"/> 	<input checked="" type="radio"/>	Integer arguments are stored in eax, ebx, ecx, etc. registers
<input checked="" type="radio"/>	<input type="radio"/> 	Integer arguments are copied from user memory to kernel memory using argint()
<input type="radio"/> 	<input checked="" type="radio"/>	String arguments are first copied to trapframe and then from trapframe to kernel's other variables.
<input checked="" type="radio"/>	<input type="radio"/> 	The functions like argint(), argstr() make the system call arguments available in the kernel.
<input type="radio"/> 	<input checked="" type="radio"/>	The arguments to system call are copied to kernel stack in trapasm.S

The arguments to system call originally reside on process stack.: True

String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer: True

The arguments are accessed in the kernel code using esp on the trapframe.: True

Integer arguments are stored in eax, ebx, ecx, etc. registers: False

Integer arguments are copied from user memory to kernel memory using argint(): True

String arguments are first copied to trapframe and then from trapframe to kernel's other variables.: False

The functions like argint(), argstr() make the system call arguments available in the kernel.: True

The arguments to system call are copied to kernel stack in trapasm.S: False

**Question 5**

Correct

Mark 1.00 out of 1.00

Map the technique with its feature/problem

static loading	wastage of physical memory	✓
static linking	large executable file	✓
dynamic linking	small executable file	✓
dynamic loading	allocate memory only if needed	✓

The correct answer is: static loading → wastage of physical memory, static linking → large executable file, dynamic linking → small executable file, dynamic loading → allocate memory only if needed

**Question 6**

Partially correct

Mark 0.10 out of 2.00

Compare paging with demand paging and select the correct statements.

Select one or more:

- a. Demand paging requires additional hardware support, compared to paging.
- b. Demand paging always increases effective memory access time. ✓
- c. Both demand paging and paging support shared memory pages. ✓
- d. TLB hit ratio has zero impact in effective memory access time in demand paging. ✗
- e. Paging requires NO hardware support in CPU
- f. Paging requires some hardware support in CPU ✓
- g. The meaning of valid-invalid bit in page table is different in paging and demand-paging. ✓
- h. Calculations of number of bits for page number and offset are same in paging and demand paging. ✓
- i. With paging, it's possible to have user programs bigger than physical memory. ✗
- j. With demand paging, it's possible to have user programs bigger than physical memory.

Your answer is partially correct.

You have correctly selected 5.

The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

Question **7**

Correct

Mark 1.00 out of 1.00

Given that the memory access time is 150 ns, probability of a page fault is 0.9 and page fault handling time is 6 ms,  
The effective memory access time in nanoseconds is:

Answer: 5400015 ✓

The correct answer is: 5400015.00

Question **8**

Correct

Mark 1.00 out of 1.00

Assuming a 8- KB page size, what is the page numbers for the address 803160 reference in decimal :  
(give answer also in decimal)

Answer: 98 ✓

The correct answer is: 98

Question 9

Partially correct

Mark 1.50 out of 2.00

For Virtual File System to work, which of the following changes are required to be done to an existing OS code (e.g. xv6)?

- a. Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount() ✓
- b. A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/" ✓
- c. The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems.
- d. The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode. ✓
- e. The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2\_read, ext2\_write, ntfs\_read, ntfs\_write) using function pointers. ✓
- f. The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup.
- g. Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open()) ✓
- h. The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories ✓

The correct answers are: A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/", The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2\_read, ext2\_write, ntfs\_read, ntfs\_write) using function pointers., The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup., The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems., The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode., The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories, Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount(), Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open())

Question 10

Partially correct

Mark 0.33 out of 1.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.

"..." means some code.

```
void  
yield(void)  
{  
...  
release(&ptable.lock);  
}
```

Release the lock held by some another process



```
void  
panic(char *s)  
{  
...  
panicked = 1;
```

If you don't do this, a process may be running on two processors parallelly



```
void  
acquire(struct spinlock *lk)  
{  
...  
getcallerpcs(&lk, lk->pcs);  
}
```

Disable interrupts to avoid another process's pointer being returned



Your answer is partially correct.

You have correctly selected 1.

The correct answer is:

```
void  
yield(void)  
{  
...  
release(&ptable.lock);  
} → Release the lock held by some another process, void  
panic(char *s)  
{  
...  
panicked = 1; → Ensure that no printing happens on other processors, void  
acquire(struct spinlock *lk)  
{  
...  
getcallerpcs(&lk, lk->pcs); → Traverse ebp chain to get sequence of instructions followed in functions calls
```

Question 11

Partially correct

Mark 0.75 out of 1.00

Select all correct statements about journalling (logging) in file systems like ext3

Select one or more:

- a. A different device driver is always needed to access the journal
- b. The purpose of journal is to speed up file system recovery ✓
- c. Most typically a transaction in journal is recorded atomically (full or none) ✓
- d. Journals are often stored circularly
- e. Journals must be maintained on the same device that hosts the file system
- f. the journal contains a summary of all changes made as part of a single transaction ✓
- g. Journal is hosted in the same device that hosts the swap space

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: The purpose of journal is to speed up file system recovery, the journal contains a summary of all changes made as part of a single transaction, Most typically a transaction in journal is recorded atomically (full or none), Journals are often stored circularly

Question **12**

Partially correct

Mark 1.00 out of 2.00

Select all the correct statements about synchronization primitives.

Select one or more:

- a. Thread that is going to block should not be holding any spinlock
- b. Semaphores can be used for synchronization scenarios like ordered execution ✓
- c. Mutexes can be implemented using spinlock ✓
- d. Blocking means one process passing over control to another process
- e. Semaphores are always a good substitute for spinlocks
- f. Blocking means moving the process to a wait queue and spinning
- g. Blocking means moving the process to a wait queue and calling scheduler
- h. All synchronization primitives are implemented essentially with some hardware assistance.
- i. Spinlocks are good for multiprocessor scenarios, for small critical sections
- j. Mutexes can be implemented without any hardware assistance
- k. Spinlocks consume CPU time ✓
- l. Mutexes can be implemented using blocking and wakeup ✓

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: Spinlocks are good for multiprocessor scenarios, for small critical sections, Spinlocks consume CPU time, Semaphores can be used for synchronization scenarios like ordered execution, Mutexes can be implemented using spinlock, Mutexes can be implemented using blocking and wakeup, Thread that is going to block should not be holding any spinlock, Blocking means moving the process to a wait queue and calling scheduler, All synchronization primitives are implemented essentially with some hardware assistance.

**Question 13**

Partially correct

Mark 0.50 out of 2.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.

"..." means some code.

```
void  
yield(void)  
{  
...  
release(&ptable.lock);  
}
```

Ensure that no printing happens on other processors



```
struct proc*  
myproc(void) {  
...  
pushcli();  
c = mycpu();  
p = c->proc;  
popcli();  
...  
}
```

Disable interrupts to avoid another process's pointer being returned



```
static inline uint  
xchg(volatile uint *addr, uint newval)  
{  
    uint result;  
  
    // The + in "+m" denotes a read-modify-write  
    // operand.  
    asm volatile("lock; xchgl %0, %1" :  
        "+m" (*addr), "=a" (result) :  
        "1" (newval) :  
        "cc");  
    return result;  
}
```

Atomic compare and swap instruction (to be expanded inline into code)



```
void  
acquire(struct spinlock *lk)  
{  
...  
__sync_synchronize();
```

Avoid a self-deadlock



```
void
acquire(struct spinlock *lk)
{
...
getcallerpcs(&lk, lk->pcs);
```

Disable interrupts to avoid deadlocks



```
void
acquire(struct spinlock *lk)
{
pushcli();
```

Traverse ebp chain to get sequence of instructions followed in functions calls



```
void
sleep(void *chan, struct spinlock *lk)
{
...
if(lk != &phtable.lock){
    acquire(&phtable.lock);
    release(lk);
}
```

Release the lock held by some another process



```
void
panic(char *s)
{
...
panicked = 1;
```

If you don't do this, a process may be running on two processors parallely



Your answer is partially correct.

You have correctly selected 2.

The correct answer is: void

```
yield(void)
{
...
release(&phtable.lock);
}
```

→ Release the lock held by some another process, **struct proc\***

```
myproc(void) {
...
pushcli();
c = mycpu();
p = c->proc;
popcli();
...
}
```

→ Disable interrupts to avoid another process's pointer being returned, **static inline uint**

```
xchg(volatile uint *addr, uint newval)
{
    uint result;
```

```
// The + in "+m" denotes a read-modify-write operand.  
asm volatile("lock; xchgl %0, %1" :  
    "+m" (*addr), "=a" (result) :  
    "1" (newval) :  
    "cc");  
return result;  
} → Atomic compare and swap instruction (to be expanded inline into code), void  
acquire(struct spinlock *lk)  
{  
...  
_sync_synchronize();
```

```
→ Tell compiler not to reorder memory access beyond this line, void  
acquire(struct spinlock *lk)  
{  
...  
getcallerpcs(&lk, lk->pcs);
```

```
→ Traverse ebp chain to get sequence of instructions followed in functions calls, void  
acquire(struct spinlock *lk)  
{  
pushcli();  
→ Disable interrupts to avoid deadlocks, void  
sleep(void *chan, struct spinlock *lk)  
{  
...  
if(lk != &ptable.lock){  
    acquire(&ptable.lock);  
    release(lk);  
} → Avoid a self-deadlock, void  
panic(char *s)  
{  
...  
panicked = 1; → Ensure that no printing happens on other processors
```

**Question 14**

Partially correct

Mark 0.86 out of 1.00

Select T/F for statements about Volume Managers.

Do pay attention to the use of the words physical partition and physical volume.

True	False	
<input checked="" type="radio"/>	<input type="radio"/> X	A logical volume may span across multiple physical partitions
<input type="radio"/> ✓	<input type="radio"/> X	A logical volume may span across multiple physical volumes
<input checked="" type="radio"/>	<input type="radio"/> X	A physical partition should be initialized as a physical volume, before it can be used by volume manager.
<input checked="" type="radio"/>	<input type="radio"/> X	A volume group consists of multiple physical volumes
<input checked="" type="radio"/>	<input type="radio"/> X	The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.
<input checked="" type="radio"/>	<input type="radio"/> X	A logical volume can be extended in size but upto the size of volume group
<input checked="" type="radio"/>	<input type="radio"/> X	The volume manager stores additional metadata on the physical disk partitions

A logical volume may span across multiple physical partitions: True

A logical volume may span across multiple physical volumes: True

A physical partition should be initialized as a physical volume, before it can be used by volume manager.: True

A volume group consists of multiple physical volumes: True

The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.: True

A logical volume can be extended in size but upto the size of volume group: True

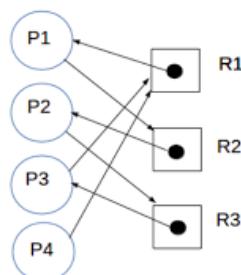
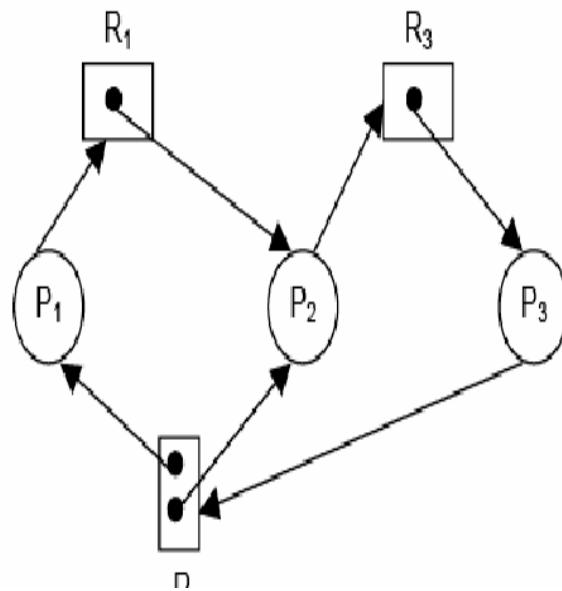
The volume manager stores additional metadata on the physical disk partitions: True

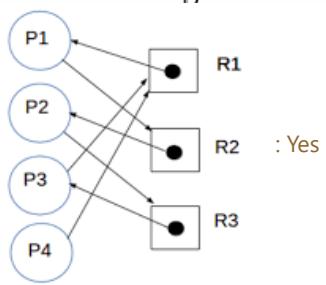
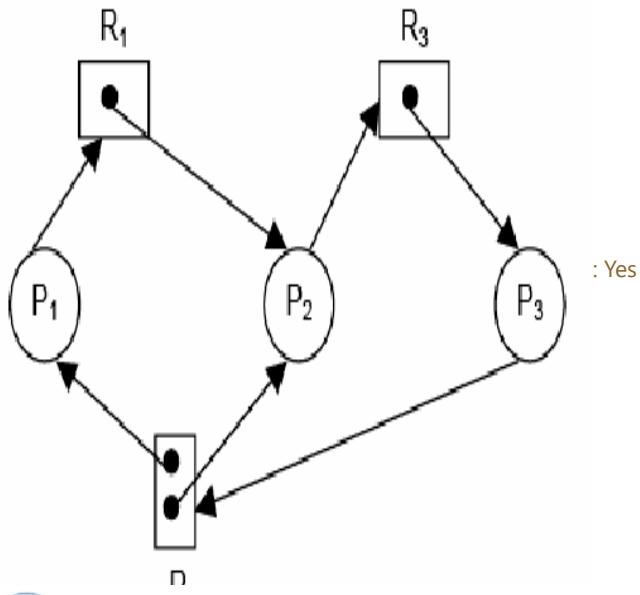
**Question 15**

Partially correct

Mark 0.50 out of 1.00

For each of the resource allocation diagram shown,  
infer whether the graph contains at least one deadlock or not.

**Yes**      **No**



Question **16**

Partially correct

Mark 0.80 out of 1.00

Select all the correct statements w.r.t user and kernel threads

Select one or more:

- a. all three models, that is many-one, one-one, many-many , require a user level thread library ✓
- b. A process may not block in many-one model, if a thread makes a blocking system call
- c. one-one model can be implemented even if there are no kernel threads
- d. one-one model increases kernel's scheduling load ✓
- e. A process blocks in many-one model even if a single thread makes a blocking system call
- f. many-one model can be implemented even if there are no kernel threads ✓
- g. many-one model gives no speedup on multicore processors ✓

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: many-one model can be implemented even if there are no kernel threads, all three models, that is many-one, one-one, many-many , require a user level thread library, one-one model increases kernel's scheduling load, many-one model gives no speedup on multicore processors, A process blocks in many-one model even if a single thread makes a blocking system call

Question 17

Incorrect

Mark 0.00 out of 1.00

Match the code with its functionality

S1 = 0; S2 = 0;

P2:

Statement1;

Signal(S2);

P1:

Wait(S2);

Statementn2;

Signal(S1);

Execution order P1, P2, P3



P3:

Wait(S1);

Statement S3;

S = 0

P1:

Statement1;

Signal(S)

Execution order P3, P2, P1



P2:

Wait(S)

Statment2;

S = 5

Wait(S)

Critical Section

Execution order P2, then P1



Signal(S)

S = 1

Wait(S)

Critical Section

Counting semaphore



Signal(S);

Your answer is incorrect.

The correct answer is: S1 = 0; S2 = 0;

P2:

Statement1;

Signal(S2);

P1:

Wait(S2);

Statemetn2;

Signal(S1);

P3:

Wait(S1);

Statement S3; → Execution order P2, P1, P3, S = 0

P1:

Statement1;

Signal(S)

P2:

Wait(S)

Statement2; → Execution order P1, then P2, S = 5

Wait(S)

Critical Section

Signal(S) → Counting semaphore, S = 1

Wait(S)

Critical Section

Signal(S); → Binary Semaphore for mutual exclusion

Question **18**

Not answered

Marked out of 2.00

Write all changes required to xv6 to add a buddy allocator.

Every change should be mentioned in terms of either of the following:

- (a) pseudo-code of new function to be added
- (b) prototype of any new function or new system call to be added
- (c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added
- (d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure
- (e) Name and a one-line description of new userland functionality to be added
- (f) Changes to Makefile
- (g) Any other change in a maximum of 20 words per change.

**Question 19**

Correct

Mark 1.00 out of 1.00

Note: for this question you get full marks if you select all and only correct options, you get ZERO if at least one option is wrong or not selected.

Select all the correct statements about log structured file systems.

- a. a transaction is said to be committed when all operations are written to file system
- b. even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery ✓
- c. file system recovery may end up losing data ✓
- d. file system recovery recovers all the lost data
- e. log may be kept on same block device or another block device ✓

Your answer is correct.

The correct answers are: file system recovery may end up losing data, log may be kept on same block device or another block device, even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery

**Question 20**

Incorrect

Mark 0.00 out of 1.00

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now:

11010010

Now, there is a request for a chunk of 45 bytes.

After this allocation, the bitmap, indicating the status of the buddy allocator will be

Answer: 11001110

✗

The correct answer is: 11011110

Question **21**

Complete

Mark 0.25 out of 3.00

List down all changes required to xv6 code, in order to add the system call chown().

Every change should be mentioned in terms of either of the following:

- (a) pseudo-code of new function to be added
  - (b) prototype of any new function or new system call to be added
  - (c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added
  - (d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure
  - (e) Name and a one-line description of new userland functionality to be added
  - (f) Changes to Makefile
  - (g) Any other change in a maximum of 20 words per change.
- 
- (a) int chown(char\* path, char \*owner\_name);
  - d) there is no need to change the data structure and no need to add new data structure
  - (e) allow user to change the owner of the file
  - (f) \_chown\

Comment:

**Question 22**

Correct

Mark 1.00 out of 1.00

Calculate the average waiting time using  
FCFS scheduling  
for the following workload  
assuming that they arrive in this order during the first time unit:

Process Burst Time

P1	2
P2	6
P3	2
P4	3

Write only a number in the answer upto two decimal points.

Answer: 5



P2 waits for 2 units

P3 waits for 2+6 units

P4 waits for 2 + 6 +2 units of time

Total waiting = 2 + 2 + 6 + 2 + 6 + 2 = 20 units

Average waiting time = 20/4 = 5

The correct answer is: 5

**Question 23**

Partially correct

Mark 0.67 out of 1.00

Given that a kernel has 1000 KB of total memory, and holes of sizes (in that order) 300 KB, 200 KB, 100 KB, 250 KB. For each of the requests on the left side, match it with the chunk chosen using the specified algorithm.

Consider each request as first request.

150 KB, first fit	200 KB	✗
150 KB, best fit	300 KB	✗
220 KB, best fit	250 KB	✓
200 KB, first fit	300 KB	✓
100 KB, worst fit	300 KB	✓
50 KB, worst fit	300 KB	✓

The correct answer is: 150 KB, first fit → 300 KB, 150 KB, best fit → 200 KB, 220 KB, best fit → 250 KB, 200 KB, first fit → 300 KB, 100 KB, worst fit → 300 KB, 50 KB, worst fit → 300 KB

[◀ Random Quiz - 6 \(xv6 file system\)](#)

Jump to...

[Homework questions: Basics of MM, xv6 booting ►](#)

**Started on** Wednesday, 19 April 2023, 8:46 PM

**State** Finished

**Completed on** Wednesday, 19 April 2023, 9:26 PM

**Time taken** 39 mins 31 secs

**Grade** 15.60 out of 30.00 (51.99%)

Question 1

Incorrect

Mark 0.00 out of 2.00

Select all the correct statements about synchronization primitives.

Select one or more:

- a. Blocking means moving the process to a wait queue and calling scheduler
- b. Blocking means moving the process to a wait queue and spinning ✗
- c. Semaphores are always a good substitute for spinlocks ✗
- d. Spinlocks are good for multiprocessor scenarios, for small critical sections
- e. All synchronization primitives are implemented essentially with some hardware assistance.
- f. Blocking means one process passing over control to another process ✗
- g. Semaphores can be used for synchronization scenarios like ordered execution ✓
- h. Mutexes can be implemented without any hardware assistance
- i. Spinlocks consume CPU time
- j. Mutexes can be implemented using spinlock
- k. Thread that is going to block should not be holding any spinlock
- l. Mutexes can be implemented using blocking and wakeup

Your answer is incorrect.

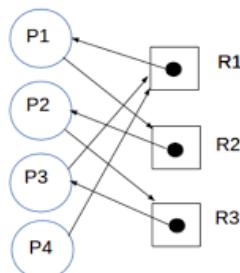
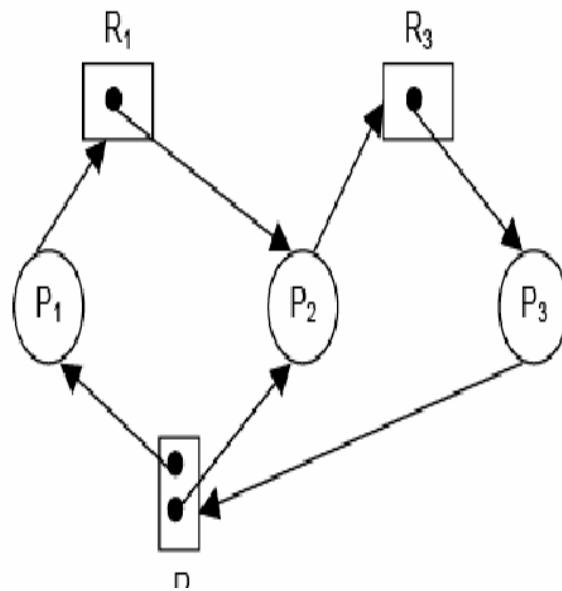
The correct answers are: Spinlocks are good for multiprocessor scenarios, for small critical sections, Spinlocks consume CPU time, Semaphores can be used for synchronization scenarios like ordered execution, Mutexes can be implemented using spinlock, Mutexes can be implemented using blocking and wakeup, Thread that is going to block should not be holding any spinlock, Blocking means moving the process to a wait queue and calling scheduler, All synchronization primitives are implemented essentially with some hardware assistance.

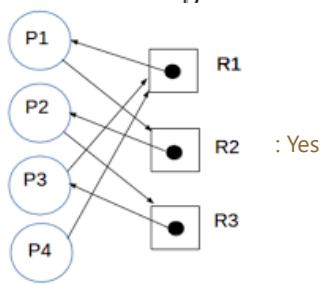
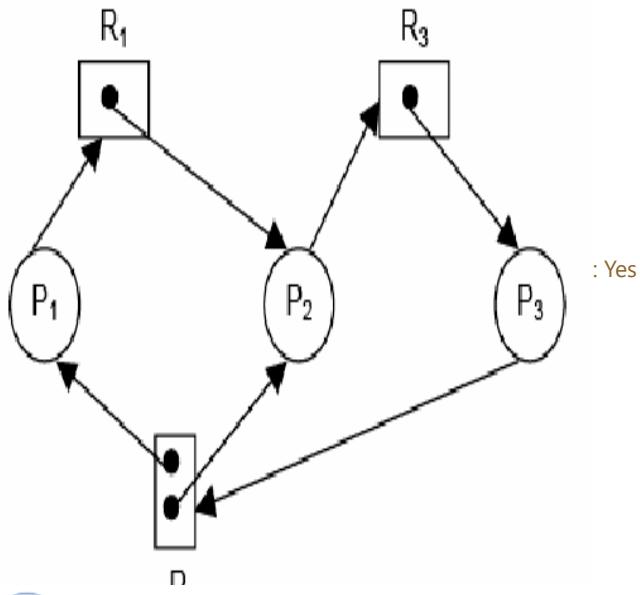
**Question 2**

Partially correct

Mark 0.50 out of 1.00

For each of the resource allocation diagram shown,  
infer whether the graph contains at least one deadlock or not.

**Yes**      **No**



**Question 3**

Partially correct

Mark 0.25 out of 1.00

Match each suggested semaphore implementation (discussed in class)

with the problems that it faces

```
struct semaphore {
    int val;
    spinlock lk;
    list l;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

block(semaphore *s) {
    listappend(s->l, current);
    spinunlock(&(s->sl));
    schedule();
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        block(s);
    }
    (s->val)--;
    spinunlock(&(s->sl));
}

signal(semaphore *s) {
    spinlock(*(s->sl));
    (s->val)++;
    x = dequeue(s->sl) and enqueue(readyq, x);
    spinunlock(*(s->sl));
}
```

deadlock



```
struct semaphore {
    int val;
    spinlock lk;
    list l;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

block(semaphore *s) {
    listappend(s->l, current);
    schedule();
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        block(s);
    }
    (s->val)--;
    spinunlock(&(s->sl));
}
```

not holding lock after unblock



```
struct semaphore {
    int val;
    spinlock lk;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        spinunlock(&(s->sl));
        spinlock(&(s->sl));
    }
    (s->val)--;
    spinunlock(&(s->sl));
}
```

too much spinning, bounded wait not guaranteed



```
struct semaphore {
    int val;
    spinlock lk;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0)
    ;
    (s->val)--;
    spinunlock(&(s->sl));
}
```

blocks holding a spinlock



Your answer is partially correct.

You have correctly selected 1.

The correct answer is:

```

struct semaphore {
    int val;
    spinlock lk;
    list l;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

block(semaphore *s) {
    listappend(s->l, current);
    spinunlock(&(s->sl));
    schedule();
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        block(s);
    }
    (s->val)--;
    spinunlock(&(s->sl));
}

signal(semaphore *s) {
    spinlock(*(s->sl));
    (s->val)++;
    x = dequeue(s->sl) and enqueue(readyq, x);
    spinunlock(*(s->sl));
}

```

→ not holding lock after unblock,

```

struct semaphore {
    int val;
    spinlock lk;
    list l;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

block(semaphore *s) {
    listappend(s->l, current);
    schedule();
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        block(s);
    }
    (s->val)--;
    spinunlock(&(s->sl));
}

```

→ blocks holding a spinlock,

```

struct semaphore {
    int val;
    spinlock lk;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        spinunlock(&(s->sl));
        spinlock(&(s->sl));
    }
    (s->val)--;
    spinunlock(&(s->sl));
}

```

→ too much spinning, bounded wait not guaranteed,

```

struct semaphore {
    int val;
    spinlock lk;
};

sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}

wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0)
    ;
    (s->val)--;
    spinunlock(&(s->sl));
}

```

→ deadlock

Question 4

Partially correct

Mark 0.75 out of 1.00

Match the code with its functionality

S = 1  
Wait(S)  
Critical Section      Binary Semaphore for mutual exclusion ✓  
Signal(S);

S = 0  
P1:  
Statement1;  
Signal(S)      Execution order P1, then P2 ✓  
P2:

Wait(S)  
Statement2;  
S = 5  
Wait(S)  
Critical Section      Execution order P1, P2, P3 ✗  
Signal(S)

S1 = 0; S2 = 0;  
P2:  
Statement1;  
Signal(S2);  
  
P1:  
Wait(S2);      Execution order P2, P1, P3 ✓  
Statement2;  
Signal(S1);

P3:  
Wait(S1);  
Statement S3;

Your answer is partially correct.

You have correctly selected 3.

The correct answer is: S = 1

Wait(S)  
Critical Section  
Signal(S); → Binary Semaphore for mutual exclusion, S = 0  
P1:  
Statement1;  
Signal(S)

P2:  
Wait(S)  
Statement2; → Execution order P1, then P2, S = 5  
Wait(S)  
Critical Section  
Signal(S) → Counting semaphore, S1 = 0; S2 = 0;

P2:

Statement1;

Signal(S2);

P1:

Wait(S2);

Statemetn2;

Signal(S1);

P3:

Wait(S1);

Statement S3; → Execution order P2, P1, P3

**Question 5**

Partially correct

Mark 1.71 out of 2.00

Compare paging with demand paging and select the correct statements.

Select one or more:

- a. With paging, it's possible to have user programs bigger than physical memory.
- b. With demand paging, it's possible to have user programs bigger than physical memory. ✓
- c. Paging requires some hardware support in CPU ✓
- d. Paging requires NO hardware support in CPU
- e. Calculations of number of bits for page number and offset are same in paging and demand paging. ✓
- f. TLB hit ration has zero impact in effective memory access time in demand paging.
- g. Demand paging always increases effective memory access time. ✓
- h. The meaning of valid-invalid bit in page table is different in paging and demand-paging.
- i. Demand paging requires additional hardware support, compared to paging. ✓
- j. Both demand paging and paging support shared memory pages. ✓

Your answer is partially correct.

You have correctly selected 6.

The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

Question 6

Partially correct

Mark 0.33 out of 1.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.

"..." means some code.

```
void  
acquire(struct spinlock *lk)  
{  
...  
    getcallerpcs(&lk, lk->pcs);
```

Disable interrupts to avoid another process's pointer being returned



```
void  
panic(char *s)  
{  
...  
    panicked = 1;
```

If you don't do this, a process may be running on two processors parallelly



```
void  
yield(void)  
{  
...  
    release(&ptable.lock);
```

Release the lock held by some another process



Your answer is partially correct.

You have correctly selected 1.

The correct answer is:

```
void  
acquire(struct spinlock *lk)  
{  
...  
    getcallerpcs(&lk, lk->pcs); → Traverse ebp chain to get sequence of instructions followed in functions calls,  
    void  
    panic(char *s)  
{  
...  
    panicked = 1; → Ensure that no printing happens on other processors, void  
    yield(void)  
{  
...  
    release(&ptable.lock);  
} → Release the lock held by some another process
```

Question 7

Correct

Mark 1.00 out of 1.00

Given that a kernel has 1000 KB of total memory, and holes of sizes (in that order) 300 KB, 200 KB, 100 KB, 250 KB. For each of the requests on the left side, match it with the chunk chosen using the specified algorithm.

Consider each request as first request.

150 KB, first fit	300 KB	✓
220 KB, best fit	250 KB	✓
100 KB, worst fit	300 KB	✓
200 KB, first fit	300 KB	✓
50 KB, worst fit	300 KB	✓
150 KB, best fit	200 KB	✓

The correct answer is: 150 KB, first fit → 300 KB, 220 KB, best fit → 250 KB, 100 KB, worst fit → 300 KB, 200 KB, first fit → 300 KB, 50 KB, worst fit → 300 KB, 150 KB, best fit → 200 KB

**Question 8**

Correct

Mark 1.00 out of 1.00

Select T/F for statements about Volume Managers.

Do pay attention to the use of the words physical partition and physical volume.

True	False	
<input checked="" type="radio"/>	<input type="radio"/> X	A logical volume may span across multiple physical partitions
<input checked="" type="radio"/>	<input type="radio"/> X	A volume group consists of multiple physical volumes
<input checked="" type="radio"/>	<input type="radio"/> X	A logical volume may span across multiple physical volumes
<input checked="" type="radio"/>	<input type="radio"/> X	A logical volume can be extended in size but upto the size of volume group
<input checked="" type="radio"/>	<input type="radio"/> X	The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.
<input checked="" type="radio"/>	<input type="radio"/> X	A physical partition should be initialized as a physical volume, before it can be used by volume manager.
<input checked="" type="radio"/>	<input type="radio"/> X	The volume manager stores additional metadata on the physical disk partitions

A logical volume may span across multiple physical partitions: True

A volume group consists of multiple physical volumes: True

A logical volume may span across multiple physical volumes: True

A logical volume can be extended in size but upto the size of volume group: True

The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.: True

A physical partition should be initialized as a physical volume, before it can be used by volume manager.: True

The volume manager stores additional metadata on the physical disk partitions: True

Question **9**

Incorrect

Mark 0.00 out of 1.00

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now:

11010010

Now, there is a request for a chunk of 45 bytes.

After this allocation, the bitmap, indicating the status of the buddy allocator will be

Answer: 11001101



The correct answer is: 11011110

Question **10**

Not answered

Marked out of 2.00

Write all changes required to xv6 to add a buddy allocator.

Every change should be mentioned in terms of either of the following:

- (a) pseudo-code of new function to be added
- (b) prototype of any new function or new system call to be added
- (c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added
- (d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure
- (e) Name and a one-line description of new userland functionality to be added
- (f) Changes to Makefile
- (g) Any other change in a maximum of 20 words per change.

Question 11

Partially correct

Mark 0.80 out of 1.00

Select all the correct statements w.r.t user and kernel threads

Select one or more:

- a. many-one model can be implemented even if there are no kernel threads ✓
- b. one-one model can be implemented even if there are no kernel threads
- c. all three models, that is many-one, one-one, many-many , require a user level thread library ✓
- d. many-one model gives no speedup on multicore processors ✓
- e. one-one model increases kernel's scheduling load ✓
- f. A process blocks in many-one model even if a single thread makes a blocking system call
- g. A process may not block in many-one model, if a thread makes a blocking system call

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: many-one model can be implemented even if there are no kernel threads, all three models, that is many-one, one-one, many-many , require a user level thread library, one-one model increases kernel's scheduling load, many-one model gives no speedup on multicore processors, A process blocks in many-one model even if a single thread makes a blocking system call

**Question 12**

Partially correct

Mark 0.50 out of 1.00

Select all correct statements about journalling (logging) in file systems like ext3

Select one or more:

- a. the journal contains a summary of all changes made as part of a single transaction ✓
- b. Journals are often stored circularly
- c. Journals must be maintained on the same device that hosts the file system
- d. The purpose of journal is to speed up file system recovery ✓
- e. Most typically a transaction in journal is recorded atomically (full or none)
- f. Journal is hosted in the same device that hosts the swap space
- g. A different device driver is always needed to access the journal

Your answer is partially correct.

You have correctly selected 2.

The correct answers are: The purpose of journal is to speed up file system recovery, the journal contains a summary of all changes made as part of a single transaction, Most typically a transaction in journal is recorded atomically (full or none), Journals are often stored circularly

**Question 13**

Correct

Mark 1.00 out of 1.00

Note: for this question you get full marks if you select all and only correct options, you get ZERO if at least one option is wrong or not selected.

Select all the correct statements about log structured file systems.

- a. a transaction is said to be committed when all operations are written to file system
- b. file system recovery may end up losing data ✓
- c. log may be kept on same block device or another block device ✓
- d. even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery ✓
- e. file system recovery recovers all the lost data

Your answer is correct.

The correct answers are: file system recovery may end up losing data, log may be kept on same block device or another block device, even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery

Question **14**

Complete

Mark 0.25 out of 3.00

List down all changes required to xv6 code, in order to add the system call chown().

Every change should be mentioned in terms of either of the following:

- (a) pseudo-code of new function to be added
- (b) prototype of any new function or new system call to be added
- (c) pseudo-code of changes to an existing function, describing lines to be removed, and lines to be added
- (d) **precise** declaration of new data structures to be added in C, or changes to the existing data structure
- (e) Name and a one-line description of new userland functionality to be added
- (f) Changes to Makefile
- (g) Any other change in a maximum of 20 words per change.
  - (a) int chown (char \* path , char \*ownername);
  - (d)there is no need of new data structures or changes to the existing data structure
  - (e)by using this system call ,user to change the owner if the file
  - (f)\_trychown\ at the end of UPROGS

Comment:

**Question 15**

Partially correct

Mark 1.00 out of 2.00

For Virtual File System to work, which of the following changes are required to be done to an existing OS code (e.g. xv6)?

- a. The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems.
- b. The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode. ✓
- c. Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount() ✓
- d. A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/" ✓
- e. Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open()) ✓
- f. The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup.
- g. The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories
- h. The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2\_read, ext2\_write, ntfs\_read, ntfs\_write) using function pointers.

The correct answers are: A mount() system call should be provided to mount a partition onto some directory in existing namespace rooted at "/", The filesystem related system calls (e.g. read, write) need to invoke the file system specific functions (e.g. ext2\_read, ext2\_write, ntfs\_read, ntfs\_write) using function pointers., The file system specific function pointers, for file system system-calls, need to be setup in the generic inode during lookup., The operating system in-memory inode needs to be a generic-inode representing "inode" like data structure across multiple file systems., The generic inode needs to have a field representing if this inode is a mount point and also to refer/point to the root of the mounted file system's inode., The lookup() operation needs to check if it's crossing a mount point and call FS specific operations to read inodes/directories, Each file-system writer needs to provide the set of function pointers for VFS, and these function pointers need to be setup in generic inode of "/" of that file system during mount(), Each open() needs to copy the function pointers from the inode of the parent directory into the inode of the child (if not already done), unless it's traversing a mount point. (This may be done as part of lookup() which is called by open())

**Question 16**

Partially correct

Mark 0.75 out of 1.00

Map the technique with its feature/problem

dynamic linking	small executable file	✓
static loading	wastage of physical memory	✓
static linking	large executable file	✓
dynamic loading	small executable file	✗

The correct answer is: dynamic linking → small executable file, static loading → wastage of physical memory, static linking → large executable file, dynamic loading → allocate memory only if needed

**Question 17**

Correct

Mark 1.00 out of 1.00

Mark the statements as True or False, w.r.t. thrashing

True	False	
<input checked="" type="radio"/>	<input type="radio"/> X	Thrashing can be limited if local replacement is used.
<input type="radio"/> X	<input checked="" type="radio"/>	Thrashing occurs because some process is doing lot of disk I/O.
<input checked="" type="radio"/>	<input type="radio"/> X	Thrashing occurs when the total size of all process's locality exceeds total memory size.
<input checked="" type="radio"/>	<input type="radio"/> X	During thrashing the CPU is under-utilised as most time is spent in I/O
<input checked="" type="radio"/>	<input type="radio"/> X	Thrashing is particular to demand paging systems, and does not apply to pure paging systems.
<input type="radio"/> X	<input checked="" type="radio"/>	Thrashing can occur even if entire memory is not in use.
<input checked="" type="radio"/>	<input type="radio"/> X	Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.
<input type="radio"/> X	<input checked="" type="radio"/>	Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.
<input checked="" type="radio"/>	<input type="radio"/> X	The working set model is an attempt at approximating the locality of a process.
<input type="radio"/> X	<input checked="" type="radio"/>	mmap() solves the problem of thrashing.

Thrashing can be limited if local replacement is used.: True

Thrashing occurs because some process is doing lot of disk I/O.: False

Thrashing occurs when the total size of all process's locality exceeds total memory size.: True

During thrashing the CPU is under-utilised as most time is spent in I/O: True

Thrashing is particular to demand paging systems, and does not apply to pure paging systems.: True

Thrashing can occur even if entire memory is not in use.: False

Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.: True

Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.: False

The working set model is an attempt at approximating the locality of a process.: True

mmap() solves the problem of thrashing.: False

Question **18**

Partially correct

Mark 0.75 out of 2.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.

"..." means some code.

```
struct proc*
myproc(void) {
...
pushcli();
c = mycpu();
p = c->proc;
popcli();
...
}
```

Traverse ebp chain to get sequence of instructions followed in functions calls



```
static inline uint
xchg(volatile uint *addr, uint newval)
{
    uint result;

// The + in "+m" denotes a read-modify-write
// operand.
asm volatile("lock; xchgl %0, %1" :
    "+m" (*addr), "=a" (result) :
    "1" (newval) :
    "cc");
    return result;
}

void
sleep(void *chan, struct spinlock *lk)
{
...
if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
}
}

void
panic(char *s)
{
...
panicked = 1;
```

Atomic compare and swap instruction (to be expanded inline into code)



Release the lock held by some another process



```
void
acquire(struct spinlock *lk)
{
    pushcli();
```

If you don't do this, a process may be running on two processors parallelly



Release the lock held by some another process



```
void
acquire(struct spinlock *lk)
{
...
__sync_synchronize();
```

Tell compiler not to reorder memory access beyond this line



```
void
acquire(struct spinlock *lk)
{
...
getcallerpcs(&lk, lk->pcs);
```

If you don't do this, a process may be running on two processors parallelly



```
void
yield(void)
{
...
release(&ptable.lock);
}
```

Release the lock held by some another process



Your answer is partially correct.

You have correctly selected 3.

The correct answer is: **struct proc\***

```
myproc(void) {
...
pushcli();
c = mycpu();
p = c->proc;
popcli();
...
}
```

→ Disable interrupts to avoid another process's pointer being returned, **static inline uint xchg(volatile uint \*addr, uint newval)**

```
{
    uint result;

// The + in "+m" denotes a read-modify-write operand.
asm volatile("lock; xchgl %0, %1" :
    "+m" (*addr), "=a" (result) :
    "1" (newval) :
    "cc");
    return result;
} → Atomic compare and swap instruction (to be expanded inline into code), void sleep(void *chan, struct spinlock *lk)
{
```

```
...
if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
} → Avoid a self-deadlock, void
panic(char *s)
{
...
panicked = 1; → Ensure that no printing happens on other processors, void
acquire(struct spinlock *lk)
{
    pushcli();
    → Disable interrupts to avoid deadlocks, void
acquire(struct spinlock *lk)
{
...
__sync_synchronize();
```

```
→ Tell compiler not to reorder memory access beyond this line, void
acquire(struct spinlock *lk)
{
...
getcallerpcs(&lk, lk->pcs);
```

```
→ Traverse ebp chain to get sequence of instructions followed in functions calls, void
yield(void)
{
...
release(&ptable.lock);
}
```

→ Release the lock held by some another process

Question **19**

Correct

Mark 1.00 out of 1.00

Calculate the average waiting time using  
FCFS scheduling  
for the following workload  
assuming that they arrive in this order during the first time unit:

Process Burst Time

P1	2
P2	6
P3	2
P4	3

Write only a number in the answer upto two decimal points.

Answer:  ✓

P2 waits for 2 units

P3 waits for 2+6 units

P4 waits for 2 + 6 +2 units of time

Total waiting =  $2 + 2 + 6 + 2 + 6 + 2 = 20$  units

Average waiting time =  $20/4 = 5$

The correct answer is: 5

**Question 20**

Incorrect

Mark 0.00 out of 1.00

Select all correct statements about file system recovery (without journaling) programs e.g. fsck

Select one or more:

- a. Recovery programs recalculate most of the metadata summaries (e.g. free inode count) ✓
- b. It is possible to lose data as part of recovery ✓
- c. Even with a write-through policy, it is possible to need a recovery program. ✓
- d. They can make changes to the on-disk file system ✓
- e. A recovery program, most typically, builds the file system data structure and checks for inconsistencies ✓
- f. They may take very long time to execute ✓
- g. They are used to recover deleted files ✗
- h. Recovery programs are needed only if the file system has a delayed-write policy.
- i. Recovery is possible due to redundancy in file system data structures

Your answer is incorrect.

The correct answers are: Recovery is possible due to redundancy in file system data structures, A recovery program, most typically, builds the file system data structure and checks for inconsistencies, It is possible to lose data as part of recovery, They may take very long time to execute, They can make changes to the on-disk file system, Recovery programs recalculate most of the metadata summaries (e.g. free inode count), Recovery programs are needed only if the file system has a delayed-write policy., Even with a write-through policy, it is possible to need a recovery program.

**Question 21**

Correct

Mark 1.00 out of 1.00

Given that the memory access time is 150 ns, probability of a page fault is 0.8 and page fault handling time is 6 ms,  
The effective memory access time in nanoseconds is:

Answer: 4800030 ✓

The correct answer is: 4800030.00

Question **22**

Correct

Mark 1.00 out of 1.00

Assuming a 8- KB page size, what is the page numbers for the address 1005699 reference in decimal :

(give answer also in decimal)

Answer: 123



The correct answer is: 123

**Question 23**

Correct

Mark 1.00 out of 1.00

Mark the statements as True or False, w.r.t. passing of arguments to system calls in xv6 code.

True	False	
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	String arguments are first copied to trapframe and then from trapframe to kernel's other variables.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The arguments to system call are copied to kernel stack in trapasm.S
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The arguments are accessed in the kernel code using esp on the trapframe.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	Integer arguments are copied from user memory to kernel memory using argint()
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The arguments to system call originally reside on process stack.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	Integer arguments are stored in eax, ebx, ecx, etc. registers
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The functions like argint(), argstr() make the system call arguments available in the kernel.

String arguments are first copied to trapframe and then from trapframe to kernel's other variables.: False

The arguments to system call are copied to kernel stack in trapasm.S: False

The arguments are accessed in the kernel code using esp on the trapframe.: True

String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer: True

Integer arguments are copied from user memory to kernel memory using argint(): True

The arguments to system call originally reside on process stack.: True

Integer arguments are stored in eax, ebx, ecx, etc. registers: False

The functions like argint(), argstr() make the system call arguments available in the kernel.: True

[◀ Random Quiz - 6 \(xv6 file system\)](#)

Jump to...

[Homework questions: Basics of MM, xv6 booting ►](#)

**Started on** Friday, 17 March 2023, 2:33 PM

**State** Finished

**Completed on** Friday, 17 March 2023, 4:54 PM

**Time taken** 2 hours 21 mins

**Grade** 6.37 out of 10.00 (63.73%)

Question 1

Incorrect

Mark 0.00 out of 0.50

The first instruction that runs when you do "make qemu" is

cli

from bootasm.S

Why?

- a. "cli" stands for clear screen and the screen should be cleared before OS boots.
- b. "cli" that is Command Line Interface needs to be enabled first
- c. "cli" clears all registers and makes them zero, so that processor is as good as "new"
- d. "cli" clears the pipeline of the CPU so that it is as good as "fresh" CPU
- e. "cli" enables interrupts, it is required because the kernel supports interrupts.
- f. "cli" disables interrupts. It is required because as of now there are no interrupt handlers available X
- g. It disables interrupts. It is required because the interrupt handlers of kernel are not yet installed.
- h. "cli" enables interrupts, it is required because the kernel must handle interrupts.

Your answer is incorrect.

The correct answer is: It disables interrupts. It is required because the interrupt handlers of kernel are not yet installed.

**Question 2**

Incorrect

Mark 0.00 out of 0.50

The struct buf has a sleeplock, and not a spinlock, because

- a. struct buf is used as a general purpose cache by kernel and cache operations take lot of time, so better to use sleeplock rather than spinlock
- b. sleeplock is preferable because it is used in interrupt context and spinlock can not be used in interrupt context
- c. It could be a spinlock, but xv6 has chosen sleeplock for purpose of demonstrating how to use a sleeplock.
- d. struct buf is used for disk I/O which takes lot of time, so sleeping/blocking is the only option available. X
- e. struct buf is used for disk I/O which takes lot of time, so sleeping/blocking is preferred to spinning/busy-wait for the desired buf.

Your answer is incorrect.

The correct answer is: struct buf is used for disk I/O which takes lot of time, so sleeping/blocking is preferred to spinning/busy-wait for the desired buf.

**Question 3**

Partially correct

Mark 0.25 out of 0.50

when is each of the following stacks allocated?

kernel stack of process

during fork() in allocproc() ✓

kernel stack for scheduler, on first processor

on the arrival of a hardware interrupt X

user stack of process

during exec() X

kernel stack for the scheduler, on other processors

in main()->startothers() ✓

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: kernel stack of process → during fork() in allocproc(), kernel stack for scheduler, on first processor → in entry.S, user stack of process → during fork() in copyuvml(), kernel stack for the scheduler, on other processors → in main()->startothers()

## Question 4

Partially correct

Mark 0.56 out of 0.75

Mark statements as True/False w.r.t. ptable.lock

True	False	
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	A process can sleep on ptable.lock if it can't acquire it.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	It is taken by one process but released by another process, running on same processor
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	ptable.lock protects the proc[] array and all struct proc in the array
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The swtch() in scheduler() is called without holding the ptable.lock when control jumps to it from sched()
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	the rule of "never block holding a spinlock" does not apply to ptable.lock in xv6
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	One sequence of function calls which takes and releases the ptable.lock is this: iderw->sleep, acquire(ptable.lock)->sched->swtch()->scheduler()->swtch()->yield(), release(ptable.lock)
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	ptable.lock can be held by different processes on different processors at the same time
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	ptable.lock is acquired but never released

A process can sleep on ptable.lock if it can't acquire it.: False

It is taken by one process but released by another process, running on same processor: True

ptable.lock protects the proc[] array and all struct proc in the array: True

The swtch() in scheduler() is called without holding the ptable.lock when control jumps to it from sched(): False

the rule of "never block holding a spinlock" does not apply to ptable.lock in xv6: True

One sequence of function calls which takes and releases the ptable.lock is this:

iderw-&gt;sleep, acquire(ptable.lock)-&gt;sched-&gt;swtch()-&gt;scheduler()-&gt;swtch()-&gt;yield(), release(ptable.lock): True

ptable.lock can be held by different processes on different processors at the same time: False

ptable.lock is acquired but never released: False

Question 5

Correct

Mark 0.50 out of 0.50

Consider the following command and its output:

```
$ ls -lht xv6.img kernel
-rw-rw-r-- 1 abhijit abhijit 4.9M Feb 15 11:09 xv6.img
-rwxrwxr-x 1 abhijit abhijit 209K Feb 15 11:09 kernel*
```

Following code in bootmain()

```
readseg((uchar*)elf, 4096, 0);
```

and following selected lines from Makefile

```
xv6.img: bootblock kernel
    dd if=/dev/zero of=xv6.img count=10000
    dd if=bootblock of=xv6.img conv=notrunc
    dd if=kernel of=xv6.img seek=1 conv=notrunc

kernel: $(OBJS) entry.o entryother initcode kernel.ld
    $(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother
    $(OBJDUMP) -S kernel > kernel.asm
    $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

Also read the code of bootmain() in xv6 kernel.

Select the options that describe the meaning of these lines and their correlation.

- a. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(). ✓
- b. Althought the size of the kernel file is 209 Kb, only 4Kb out of it is the actual kernel code and remaining part is all zeroes.
- c. The bootmain() code does not read the kernel completely in memory
- d. Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes. ✓
- e. readseg() reads first 4k bytes of kernel in memory ✓
- f. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is not read as it is user programs.
- g. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✓
- h. The kernel.ld file contains instructions to the linker to link the kernel properly ✓
- i. The kernel.asm file is the final kernel file

Your answer is correct.

The correct answers are: The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(), readseg() reads first 4k bytes of kernel in memory, The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files, The kernel.ld file contains instructions to the linker to link the kernel properly, Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.

**Question 6**

Partially correct

Mark 0.43 out of 0.75

code line, MMU setting: Match the line of xv6 code with the MMU setup employed

movw %ax, %gs	real mode	✗
jmp *%eax	protected mode with segmentation and 4 MB pages	✓
inb \$0x64,%al	real mode	✓
ljmp \$(SEG_KCODE<<3), \$start32	protected mode with only segmentation	✗
readseg((uchar*)elf, 4096, 0);	protected mode with only segmentation	✓
orl \$CRO_PE, %eax	real mode	✓
movl \$(V2P_WO(entrypgdir)), %eax	protected mode with segmentation and 4 MB pages	✗

The correct answer is: movw %ax, %gs → protected mode with only segmentation, jmp \*%eax → protected mode with segmentation and 4 MB pages, inb \$0x64,%al → real mode, ljmp \$(SEG\_KCODE<<3), \$start32 → real mode, readseg((uchar\*)elf, 4096, 0); → protected mode with only segmentation, orl \$CRO\_PE, %eax → real mode, movl \$(V2P\_WO(entrypgdir)), %eax → protected mode with only segmentation

**Question 7**

Correct

Mark 0.50 out of 0.50

Which of the following is DONE by allocproc() ?

- a. setup kernel memory mappings for the process
- b. Select an UNUSED struct proc for use ✓
- c. allocate kernel stack for the process ✓
- d. ensure that the process starts in trapret()
- e. setup the contents of the trapframe of the process properly
- f. allocate PID to the process ✓
- g. ensure that the process starts in forkret() ✓
- h. setup the trapframe and context pointers appropriately ✓

The correct answers are: Select an UNUSED struct proc for use, allocate PID to the process, allocate kernel stack for the process, setup the trapframe and context pointers appropriately, ensure that the process starts in forkret()

## Question 8

Partially correct

Mark 0.25 out of 0.50

Mark statements as True/False w.r.t. the creation of free page list in xv6.

True	False	
<input checked="" type="radio"/>	<input checked="" type="radio"/>	kmem.use_lock is set to 1 after free page list is created, so that kmem.lock is taken before accessing kmem.freelist.
<input checked="" type="radio"/>	<input checked="" type="radio"/>	The pointers that link the pages together are in the first 4 bytes of the pages themselves
<input checked="" type="radio"/>	<input checked="" type="radio"/>	if(kmem.use_lock) acquire(&kmem.lock); is not done when called from kinit1() because there is no need to take the lock when kinit1() is running because interrupts are disabled and only one processor is running
<input checked="" type="radio"/>	<input checked="" type="radio"/>	free page list is a singly circular linked list.
<input checked="" type="radio"/>	<input checked="" type="radio"/>	if(kmem.use_lock) acquire(&kmem.lock); this "if" condition is true, when kinit2() runs because multi-processor support has been enabled by now.
<input checked="" type="radio"/>	<input checked="" type="radio"/>	the kmem.lock is used by kfree() and kalloc() only.

kmem.use\_lock is set to 1 after free page list is created, so that kmem.lock is taken before accessing kmem.freelist.: True

The pointers that link the pages together are in the first 4 bytes of the pages themselves: True

if(kmem.use\_lock)

acquire(&amp;kmem.lock);

is not done when called from kinit1() because there is no need to take the lock when kinit1() is running because interrupts are disabled and only one processor is running: True

free page list is a singly circular linked list.: False

if(kmem.use\_lock)

acquire(&amp;kmem.lock);

this "if" condition is true, when kinit2() runs because multi-processor support has been enabled by now.: False

the kmem.lock is used by kfree() and kalloc() only.: True

## Question 9

Partially correct

Mark 0.55 out of 1.00

Given below is code of sleeplock in xv6.

```
// Long-term locks for processes
struct sleeplock {
    uint locked;          // Is the lock held?
    struct spinlock lk;  // spinlock protecting this sleep lock

    // For debugging:
    char *name;           // Name of lock.
    int pid;              // Process holding lock
};

void
acquiresleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    while (lk->locked) {
        sleep(lk, &lk->lk);
    }
    lk->locked = 1;
    lk->pid = myproc()->pid;
    release(&lk->lk);
}

void
releasesleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    lk->locked = 0;
    lk->pid = 0;
    wakeup(lk);
    release(&lk->lk);
}
```

Mark the statements as True/False w.r.t. this code.

True	False
<input checked="" type="radio"/> <input type="radio"/>	Wakeup() will wakeup the first process waiting for the lock
<input checked="" type="radio"/> <input type="radio"/>	<pre>acquire(&amp;lk-&gt;lk); while (lk-&gt;locked) {     sleep(lk, &amp;lk-&gt;lk); } could also be written as acquire(&amp;lk-&gt;lk); if (lk-&gt;locked) {     sleep(lk, &amp;lk-&gt;lk); }</pre>

True	False	
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	A process has acquired the sleeplock when it comes out of sleep(): <span style="color: red;">✗</span>
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	sleep() is called holding a spinlock. This could be avoided by releasing the lock before calling sleep() and acquiring it again after call to sleep(): <span style="color: red;">✗</span>
<input checked="" type="radio"/> ✗	<input checked="" type="radio"/> ✗	sleep() is the function which blocks a process. <span style="color: green;">✓</span>
<input checked="" type="radio"/> ✗	<input checked="" type="radio"/> ✗	The process which called acquiresleep() and then got blocked, is woken up by the timer interrupt <span style="color: green;">✓</span> it's woken up by another process which called releasesleep() and then wakeup()
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	the 'spinlock lk' protects 'locked' variable, but not the 'name' nor the 'pid' <span style="color: red;">✗</span>
<input checked="" type="radio"/> ✗	<input checked="" type="radio"/> ✗	the 'spinlock lk' is needed in a sleeplock, because access to the sleeplock for locking/unlocking itself creates a critical section <span style="color: green;">✓</span>
<input checked="" type="radio"/> ✗	<input checked="" type="radio"/> ✗	The spinlock lk->lk is held when the process comes out of sleep(): <span style="color: green;">✓</span>
<input checked="" type="radio"/> ✗	<input checked="" type="radio"/> ✗	Sleeplock() will ensure that either the process gets the lock or the process gets blocked. <span style="color: green;">✓</span>
<input checked="" type="radio"/> ✗	<input checked="" type="radio"/> ✗	All processes waiting for the sleeplock will have a race for aquiring lk->lk spinlock, because all are woken up <span style="color: green;">✓</span> wakeup() wakes up all processes, and they "thunder" to take the spinlock.

Wakeup() will wakeup the first process waiting for the lock: False

```
acquire(&lk->lk);
while (!lk->locked) {
    sleep(lk, &lk->lk);
}
```

could also be written as

```
acquire(&lk->lk);
if (!lk->locked) {
    sleep(lk, &lk->lk);
}: False
```

A process has acquired the sleeplock when it comes out of sleep(): False

sleep() is called holding a spinlock. This could be avoided by releasing the lock before calling sleep() and acquiring it again after call to sleep(): False

The process which called acquiresleep() and then got blocked, is woken up by the timer interrupt: True

The 'spinlock lk' protects 'locked' variable, but not the 'name' nor the 'pid': False

The 'spinlock lk' is needed in a sleeplock, because access to the sleeplock for locking/unlocking itself creates a critical section: True

The spinlock lk->lk is held when the process comes out of sleep(): True

Sleeplock() will ensure that either the process gets the lock or the process gets blocked.: True

All processes waiting for the sleeplock will have a race for aquiring lk->lk spinlock, because all are woken up: True

## Question 10

Partially correct

Mark 0.50 out of 1.00

Mark the statements as True/False w.r.t. swtch()

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	p->context used in scheduler()->swtch() was <b>Generally</b> set when the process was interrupted earlier, and came via sched()->swtch()
<input checked="" type="radio"/>	<input type="radio"/>	swtch() is written in assembly language, because it violates calling convention, by changing the stack itself.
<input type="radio"/>	<input checked="" type="radio"/>	swtch() is written in assembly language because it violates the calling convention by pushing parameters on the stack on its own.
<input type="radio"/>	<input checked="" type="radio"/>	swtch() called from scheduler() changes the stack from the process's kernel stack to the scheduler's kernel stack.
<input checked="" type="radio"/>	<input type="radio"/>	swtch() is called only from sched() or scheduler()
<input type="radio"/>	<input checked="" type="radio"/>	switch stores the old context on new stack, and restores new context from old stack.
<input type="radio"/>	<input checked="" type="radio"/>	sched() is the only place when p->context is set
<input type="radio"/>	<input checked="" type="radio"/>	movl %esp, (%eax) means, *(c->scheduler) = contents of esp When swtch() is called from scheduler()
<input checked="" type="radio"/>	<input type="radio"/>	push in swtch() happens on old stack, while pop happens from new stack
<input checked="" type="radio"/>	<input type="radio"/>	swtch() changes the context from "old" to "new"

p->context used in scheduler()->swtch() was **Generally** set when the process was interrupted earlier, and came via sched()->swtch(): True  
 swtch() is written in assembly language, because it violates calling convention, by changing the stack itself.: True

swtch() is written in assembly language because it violates the calling convention by pushing parameters on the stack on its own.: False  
 swtch() called from scheduler() changes the stack from the process's kernel stack to the scheduler's kernel stack.: False

swtch() is called only from sched() or scheduler(): True

switch stores the old context on new stack, and restores new context from old stack.: False

sched() is the only place when p->context is set: False

movl %esp, (%eax)

means, \*(c->scheduler) = contents of esp

When swtch() is called from scheduler(): False

push in swtch() happens on old stack, while pop happens from new stack: True

swtch() changes the context from "old" to "new": True

Question **11**

Correct

Mark 0.25 out of 0.25

Select the odd one out

- a. Kernel stack of new process to kernel stack of scheduler ✓
- b. Kernel stack of new process to Process stack of new process
- c. Kernel stack of running process to kernel stack of scheduler
- d. Process stack of running process to kernel stack of running process
- e. Kernel stack of scheduler to kernel stack of new process

The correct answer is: Kernel stack of new process to kernel stack of scheduler

Question **12**

Correct

Mark 0.25 out of 0.25

The variable 'end' used as argument to kinit1 has the value

- a. 80102da0
- b. 81000000
- c. 80000000
- d. 80110000
- e. 801154a8 ✓
- f. 8010a48c

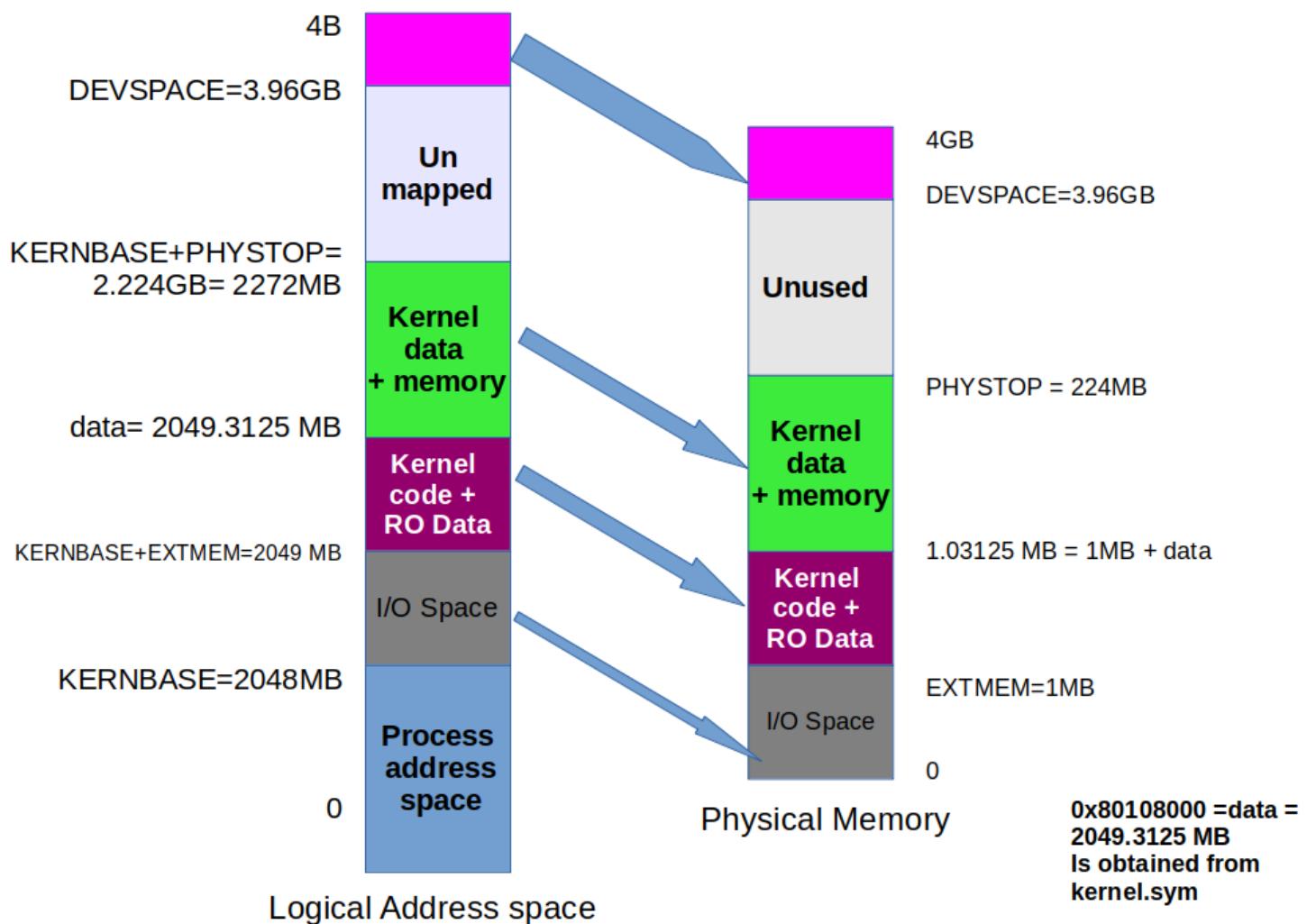
The correct answer is: 801154a8

## Question 13

Partially correct

Mark 0.36 out of 0.50

With respect to this diagram, mark statements as True/False.



True      False

- |                                  |                       |  |                                     |
|----------------------------------|-----------------------|--|-------------------------------------|
| <input checked="" type="radio"/> | <input type="radio"/> | This diagram only shows the absolutely defined virtual->physical mappings, not the mappings defined at run time by kernel.             | <input checked="" type="checkbox"/> |
| <input checked="" type="radio"/> | <input type="radio"/> | "Kernel data + memory" on right side, here refers to the region from which pages are allocated to the kernel and process both.         | <input checked="" type="checkbox"/> |
| <input checked="" type="radio"/> | <input type="radio"/> | The kernel virtual addresses start from KERNLINK = KERNBASE + EXTMEM   | <input checked="" type="checkbox"/> |
| <input checked="" type="radio"/> | <input type="radio"/> | The process's pages are mapped into physical memory from 1.03125 MB to PHYSTOP.  | <input checked="" type="checkbox"/> |
| <input checked="" type="radio"/> | <input type="radio"/> | The kernel file, after compilation, has maximum virtual address up to "data" as shown in the diagram, which is equal to "end" variable | <input checked="" type="checkbox"/> |

**True**    **False**

<input checked="" type="radio"/>	<input checked="" type="radio"/>	When bootloader loads the kernel, then physical memory from EXTMEM upto EXTMEM + data is occupied.	✓
<input checked="" type="radio"/>	<input checked="" type="radio"/>	PHYSTOP can be changed , but that needs kernel recompilation and re-execution.	✗

This diagram only shows the absolutely defined virtual->physical mappings, not the mappings defined at run time by kernel.: True  
"Kernel data + memory" on right side, here refers to the region from which pages are allocated to the kernel and process both.: True

The kernel virtual addresses start from KERNLINK = KERNBASE + EXTMEM: True

The process's pages are mapped into physical memory from 1.03125 MB to PHYSTOP.: True

The kernel file, after compilation, has maximum virtual address up to "data" as shown in the diagram, which is equal to "end" variable: True

When bootloader loads the kernel, then physical memory from EXTMEM upto EXTMEM + data is occupied.: True

PHYSTOP can be changed , but that needs kernel recompilation and re-execution.: True

**Question 14**

Correct

Mark 0.25 out of 0.25

Which of the following is not a task of the code of swtch() function

- a. Save the return value of the old context code ✓
- b. Load the new context
- c. Change the kernel stack location ✓
- d. Save the old context
- e. Switch stacks
- f. Jump to next context EIP

The correct answers are: Save the return value of the old context code, Change the kernel stack location

Question **15**

Correct

Mark 0.50 out of 0.50

We often use terms like "swtch() changes stack from process's kernel stack to scheduler's stack", or "the values are pushed on stack", or "the stack is initialized to the new page", etc. while discussing xv6 on x86.

Which of the following most accurately describes the meaning of "stack" in such sentences?

- a. The stack variable used in the program being discussed
- b. The region of memory where the kernel remembers all the function calls made
- c. The region of memory allocated by kernel for storing the parameters of functions
- d. The ss:esp pair ✓
- e. The "stack" variable declared in "stack.S" in xv6
- f. the region of memory which is currently used as stack by processor
- g. The stack segment

Your answer is correct.

The correct answer is: The ss:esp pair

**Question 16**

Partially correct

Mark 0.31 out of 0.50

Mark the statements as True/False, with respect to the use of the variable "chan" in struct proc.

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	The value of 'chan' is changed only in sleep() ✓
<input type="radio"/>	<input checked="" type="radio"/>	when chan is NULL, the 'state' in proc must be RUNNABLE. ✗
<input checked="" type="radio"/>	<input type="radio"/>	chan stores the address of the variable, representing a condition, for which the process is waiting. ✓
<input checked="" type="radio"/>	<input type="radio"/>	'chan' is used only by the sleep() and wakeup1() functions. ✓
<input type="radio"/>	<input checked="" type="radio"/>	chan is the head pointer to a linked list of processes, waiting for a particular event to occur ✗
<input checked="" type="radio"/>	<input type="radio"/>	When chan is not NULL, the 'state' in struct proc must be SLEEPING ✓
<input checked="" type="radio"/>	<input type="radio"/>	in xv6, the address of an appropriate variable is used as a "condition" for a waiting process. ✓
<input type="radio"/>	<input checked="" type="radio"/>	Changing the state of a process automatically changes the value of 'chan' ✗

The value of 'chan' is changed only in sleep(): True

when chan is NULL, the 'state' in proc must be RUNNABLE.: False

chan stores the address of the variable, representing a condition, for which the process is waiting.: True

'chan' is used only by the sleep() and wakeup1() functions.: True

chan is the head pointer to a linked list of processes, waiting for a particular event to occur: False

When chan is not NULL, the 'state' in struct proc must be SLEEPING: True

in xv6, the address of an appropriate variable is used as a "condition" for a waiting process.: True

Changing the state of a process automatically changes the value of 'chan': False

**Question 17**

Correct

Mark 0.25 out of 0.25

Match function with its meaning

iderw	Issue a disk read/write for a buffer, block the issuing process	✓
idewait	Wait for disc controller to be ready	✓
idestart	tell disc controller to start I/O for the first buffer on idequeue	✓
ideinit	Initialize the disc controller	✓
ideintr	disk interrupt handler, transfer data from controller to buffer, wake up processes waiting for this buffer, start I/O for next buffer	✓

Your answer is correct.

The correct answer is: iderw → Issue a disk read/write for a buffer, block the issuing process, idewait → Wait for disc controller to be ready, idestart → tell disc controller to start I/O for the first buffer on idequeue, ideinit → Initialize the disc controller, ideintr → disk interrupt handler, transfer data from controller to buffer, wake up processes waiting for this buffer, start I/O for next buffer

**Question 18**

Correct

Mark 0.25 out of 0.25

Why is there a call to kinit2? Why is it not merged with knit1?

- a. Because there is a limit on the values that the arguments to knit1() can take.
- b. call to seginit() makes it possible to actually use PHYSTOP in argument to kinit2()
- c. When kinit1() is called there is a need for few page frames, but later kinit2() is called to serve need of more page frames
- d. knit2 refers to virtual addresses beyond 4MB, which are not mapped before kalloc() is called✓

The correct answer is: knit2 refers to virtual addresses beyond 4MB, which are not mapped before kalloc() is called

Question **19**

Correct

Mark 0.25 out of 0.25

Which of the following call sequence is impossible in xv6?

- a. Process 1: write() -> sys\_write()-> file\_write() -- timer interrupt -> trap() -> yield() -> sched() -> switch() (jumps to)-> scheduler() -> swtch() (jumps to)-> Process 2 (return call sequence) sched() -> yield() -> trap-> user-code
- b. Process 1: write() -> sys\_write()-> file\_write() -> writei() -> bread() -> bget() -> iderw() -> sleep() -> sched() -> switch() (jumps to)-> scheduler() ->switch()(jumps to)-> Process 2 (return call sequence) sched() -> yield() -> trap-> user-code
- c. Process 1: timer interrupt -> trap() -> yield() -> sched() -> switch() -> scheduler()-> Process 2 runs -> write -> sys\_write() -> trap()-> ... ✓

Your answer is correct.

The correct answer is: Process 1: timer interrupt -> trap() -> yield() -> sched() -> switch() -> scheduler()-> Process 2 runs -> write -> sys\_write() -> trap()-> ...

Question **20**

Partially correct

Mark 0.17 out of 0.50

Mark statements as True/False, w.r.t. the given diagram

**Started on** Friday, 24 February 2023, 2:44 PM

**State** Finished

**Completed on** Friday, 24 February 2023, 4:39 PM

**Time taken** 1 hour 55 mins

**Grade** **8.78** out of 10.00 (**87.77%**)

Question **1**

Correct

Mark 0.50 out of 0.50

What is meant by formatting a disk/partition?

- a. erasing all data on the disk/partition
- b. storing all the necessary programs on the disk/partition
- c. creating layout of empty directory tree/graph data structure ✓
- d. writing zeroes on all sectors

The correct answer is: creating layout of empty directory tree/graph data structure

**Question 2**

Partially correct

Mark 0.44 out of 0.50

How does the compiler calculate addresses for the different parts of a C program, when paging is used?

Text	starting with 0	✓
Static variables	Immediately after the text, along with globals	✓
Local variables	An offset with respect to stack pointer (esp)	✓
#include files	No memory allocated, they are handled by linker	✗
#define	No memory allocated, they are handled by pre-processor	✓
typedef	No memory allocated, as they are not variables, but only conceptual definition of a type	✓
Global variables	Immediately after the text	✓
malloced memory	Heap (handled by the malloc-free library, using OS's system calls)	✓

Your answer is partially correct.

You have correctly selected 7.

The correct answer is: Text → starting with 0, Static variables → Immediately after the text, along with globals, Local variables → An offset with respect to stack pointer (esp), #include files → No memory allocated for the file, but if it contains variables, then variables may be allocated memory, #define → No memory allocated, they are handled by pre-processor, typedef → No memory allocated, as they are not variables, but only conceptual definition of a type, Global variables → Immediately after the text, malloced memory → Heap (handled by the malloc-free library, using OS's system calls)

**Question 3**

Correct

Mark 0.50 out of 0.50

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

```
$ ls ./tmp/asdfksdf >/tmp/ddd 2>&1
```

**Program 1**

```
int main(int argc, char *argv[]) {  
    int fd, n, i;  
    char buf[128];  
  
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);  
    close(1);  
    dup(fd);  
    close(2);  
    dup(fd);  
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);  
}
```

**Program 2**

```
int main(int argc, char *argv[]) {  
    int fd, n, i;  
    char buf[128];  
  
    close(1);  
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);  
    close(2);  
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);  
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);  
}
```

Select all the correct statements about the programs

Select one or more:

- a. Program 1 makes sure that there is one file offset used for '2' and '1' ✓
- b. Program 1 does 1>&2
- c. Both programs are correct
- d. Both program 1 and 2 are incorrect
- e. Program 2 does 1>&2
- f. Program 2 makes sure that there is one file offset used for '2' and '1'
- g. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd
- h. Program 1 is correct for > /tmp/ddd but not for 2>&1
- i. Program 2 is correct for > /tmp/ddd but not for 2>&1
- j. Program 1 ensures 2>&1 and does not ensure > /tmp/ddd
- k. Only Program 1 is correct ✓
- l. Only Program 2 is correct

Your answer is correct.

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

Question 4

Partially correct

Mark 0.43 out of 0.50

You must have seen the error message "Segmentation fault, core dumped" very often.

With respect to this error message, mark the statements as True/False.

True	False	
<input type="radio"/> ✗	<input checked="" type="radio"/>	The term "core" refers to the core code of the kernel.
<input type="radio"/> ✗	<input checked="" type="radio"/>	On Linux, the message is printed only because the memory management scheme is segmentation
<input checked="" type="radio"/>	<input type="radio"/> ✗	The process has definitely performed illegal memory access.
<input checked="" type="radio"/>	<input type="radio"/> ✗	On Linux, the process was sent a SIGSEGV signal and the default handler for the signal is "Term", so the process is terminated.
<input type="radio"/> ✗	<input checked="" type="radio"/>	The illegal memory access was detected by the kernel and the process was punished by kernel.
<input checked="" type="radio"/>	<input type="radio"/> ✗	The core file can be analysed later using a debugger, to determine what went wrong.
<input checked="" type="radio"/>	<input type="radio"/> ✗	The image of the process is stored in a file called "core", if the ulimit allows so.

The term "core" refers to the core code of the kernel.: False

On Linux, the message is printed only because the memory management scheme is segmentation: False

The process has definitely performed illegal memory access.: True

On Linux, the process was sent a SIGSEGV signal and the default handler for the signal is "Term", so the process is terminated.: True

The illegal memory access was detected by the kernel and the process was punished by kernel.: False

The core file can be analysed later using a debugger, to determine what went wrong.: True

The image of the process is stored in a file called "core", if the ulimit allows so.: True

**Question 5**

Correct

Mark 0.50 out of 0.50

Which of the following instructions should be privileged?

Select one or more:

- a. Access memory management unit of the processor ✓
- b. Turn off interrupts. ✓
- c. Read the clock.
- d. Set value of timer. ✓
- e. Switch from user to kernel mode. ✓ This instruction (like INT) is itself privileged - and that is why it not only changes the mode, but also ensures a jump to an ISR (kernel code)
- f. Access a general purpose register
- g. Set value of a memory location
- h. Modify entries in device-status table ✓
- i. Access I/O device. ✓

Your answer is correct.

The correct answers are: Set value of timer., Access memory management unit of the processor, Turn off interrupts., Modify entries in device-status table, Access I/O device., Switch from user to kernel mode.

**Question 6**

Correct

Mark 0.50 out of 0.50

Map the block allocation scheme with the problem it suffers from

(Match pairs 1-1, match a scheme with the problem that it suffers from relatively the most, compared to others)

Indexed Allocation	Overhead of reading metadata blocks	✓
Continuous allocation	need for compaction	✓
Linked allocation	Too many seeks	✓

Your answer is correct.

The correct answer is: Indexed Allocation → Overhead of reading metadata blocks, Continuous allocation → need for compaction, Linked allocation → Too many seeks

## Question 7

Partially correct

Mark 0.43 out of 0.50

Following code claims to implement the command

/bin/ls -l | /usr/bin/head -3 | /usr/bin/tail -1

Fill in the blanks to make the code work.

Note: Do not include space in writing any option. x[1][2] should be written without any space, and so is the case with [1] or [2]. Pay attention to exact syntax and do not write any extra character like ';' or = etc.

```
int main(int argc, char *argv[]) {
    int pid1, pid2;
    int pfd[ 2 ] ✓ ][2];
    pipe( pfd[0] ✓ );
    pid1 = fork() ✓ ;
    if(pid1 != 0) {
        close(pfd[0][0] ✓ );
        close( 1 ✓ );
        dup( pfd[0][1] ✓ );
        execl("/bin/ls", "/bin/ls", " -1 ", NULL);
    }
    pipe( pfd[1] ✓ );
    pid2 ✓ = fork();
    if(pid2 == 0) {
        close( pfd[0][1] ✗ );
        close(0);
        dup( pfd[1][0] ✗ );
        close(pfd[1][0] ✓ );
        close( 1 ✓ );
        dup( pfd[1][1] ✓ );
        execl("/usr/bin/head", "/usr/bin/head", " -3 ", NULL);
    } else {
        close(pfd[1][1] ✓ );
        close( 0 ✓ );
        dup( pfd[1][0] ✓ );
        close(pfd[0][0] ✓ );
    }
}
```

```
execl("/usr/bin/tail", "/usr/bin/tail", " -1 ", NULL);  
}  
}
```

**Question 8**

Partially correct

Mark 0.45 out of 0.50

Match the elements of C program to their place in memory

Arguments	Stack	✓
Function code	Code	✓
#define MACROS	No memory needed	✗
#include files	No memory needed	✓
Global Static variables	Data	✓
Mallocoed Memory	Heap	✓
Global variables	Data	✓
Local Static variables	Data	✓
Local Variables	Stack	✓
Code of main()	Code	✓

The correct answer is: Arguments → Stack, Function code → Code, #define MACROS → No Memory needed, #include files → No memory needed, Global Static variables → Data, Mallocoed Memory → Heap, Global variables → Data, Local Static variables → Data, Local Variables → Stack, Code of main() → Code

**Question 9**

Correct

Mark 0.50 out of 0.50

Doing a lookup on the pathname /a/b/b/c/d for opening the file "d" requires reading  ✓ no. of inodes. Assume that there are no hard/soft links on the path.

Write the answer as a number.

The correct answer is: 6

**Question 10**

Partially correct

Mark 0.42 out of 0.50

Mark the statements about device drivers by marking as True or False.

True	False	
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	Different devices of the same type (e.g. 2 IDE hard disks) must need different device drivers.
<input checked="" type="radio"/> ✗	<input checked="" type="radio"/> ✗	It's possible that a particular hardware has multiple device drivers available for it.
<input checked="" type="radio"/> ✗	<input checked="" type="radio"/> ✗	Device driver is part of OS code
<input checked="" type="radio"/> ✗	<input checked="" type="radio"/> ✗	Device driver is an intermediary between the hardware controller and OS
<input checked="" type="radio"/> ✗	<input checked="" type="radio"/> ✗	Device driver is part of hardware
<input checked="" type="radio"/> ✗	<input checked="" type="radio"/> ✗	Writing a device driver mandatorily demands reading the technical documentation about the hardware.

Different devices of the same type (e.g. 2 IDE hard disks) must need different device drivers.: False

It's possible that a particular hardware has multiple device drivers available for it.: True

Device driver is part of OS code: True

Device driver is an intermediary between the hardware controller and OS: True

Device driver is part of hardware: False

Writing a device driver mandatorily demands reading the technical documentation about the hardware.: True

**Question 11**

Correct

Mark 0.50 out of 0.50

Map each signal with its meaning

SIGSEGV	Invalid Memory Reference	✓
SIGALRM	Timer Signal from alarm()	✓
SIGPIPE	Broken Pipe	✓
SIGUSR1	User Defined Signal	✓
SIGCHLD	Child Stopped or Terminated	✓

The correct answer is: SIGSEGV → Invalid Memory Reference, SIGALRM → Timer Signal from alarm(), SIGPIPE → Broken Pipe, SIGUSR1 → User Defined Signal, SIGCHLD → Child Stopped or Terminated

**Question 12**

Partially correct

Mark 0.25 out of 0.50

Select all the blocks that may need to be written back to disk (if updated, of-course), as "Yes", when an operation of deleting a file is carried out on ext2 file system.

An option has to be correct entirely to be marked "Yes"

Block bitmap(s) for all the blocks of the file

 Yes ✓

Data blocks of the file

 Yes ✗

One or multiple data blocks of the parent directory

 Yes ✗

Superblock

 Yes ✓

Possibly one block bitmap corresponding to the parent directory

 Yes ✓

One or more data bitmap blocks for the parent directory

 Yes ✗

Your answer is partially correct.

only one data block of parent directory. multiple blocks not possible. an entry is always contained within one single block

You have correctly selected 3.

The correct answer is: Block bitmap(s) for all the blocks of the file → Yes, Data blocks of the file → No, One or multiple data blocks of the parent directory → No, Superblock → Yes, Possibly one block bitmap corresponding to the parent directory → Yes, One or more data bitmap blocks for the parent directory → No

**Question 13**

Correct

Mark 0.50 out of 0.50

Select the compiler's view of the process's address space, for each of the following MMU schemes:  
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Segmentation	many continuous chunks of variable size	✓
Segmentation, then paging	many continuous chunks of variable size	✓
Paging	one continuous chunk	✓
Relocation + Limit	one continuous chunk	✓

Your answer is correct.

The correct answer is: Segmentation → many continuous chunks of variable size, Segmentation, then paging → many continuous chunks of variable size, Paging → one continuous chunk, Relocation + Limit → one continuous chunk

**Question 14**

Partially correct

Mark 0.30 out of 0.50

Mark the statements about named and un-named pipes as True or False

True	False	
<input checked="" type="radio"/>	<input type="radio"/> X	Named pipe exists as a file
<input type="radio"/>	<input checked="" type="radio"/> X	Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.
<input checked="" type="radio"/>	<input type="radio"/> X	Both types of pipes are an extension of the idea of "message passing".
<input type="radio"/> X	<input checked="" type="radio"/>	A named pipe has a name decided by the kernel.
<input checked="" type="radio"/>	<input type="radio"/> X	Un-named pipes are inherited by a child process from parent.
<input checked="" type="radio"/>	<input type="radio"/> X	Named pipes can exist beyond the life-time of processes using them.
<input checked="" type="radio"/>	<input type="radio"/> X	Both types of pipes provide FIFO communication.
<input type="radio"/> X	<input checked="" type="radio"/>	The buffers for named-pipe are in process-memory while the buffers for the un-named pipe are in kernel memory.
<input type="radio"/> X	<input checked="" type="radio"/>	Named pipes can be used for communication between only "related" processes.
<input type="radio"/> X	<input checked="" type="radio"/>	The pipe() system call can be used to create either a named or un-named pipe.

Named pipe exists as a file.: True

Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.: True

Both types of pipes are an extension of the idea of "message passing": True

A named pipe has a name decided by the kernel.: False

Un-named pipes are inherited by a child process from parent.: True

Named pipes can exist beyond the life-time of processes using them.: True

Both types of pipes provide FIFO communication.: True

The buffers for named-pipe are in process-memory while the buffers for the un-named pipe are in kernel memory.: False

Named pipes can be used for communication between only "related" processes.: False

The pipe() system call can be used to create either a named or un-named pipe.: False

**Question 15**

Partially correct

Mark 0.29 out of 0.50

Mark the statements as True/False w.r.t. the basic concepts of memory management.

True	False	
<input type="radio"/> ✗	<input checked="" type="checkbox"/> ✗	The kernel refers to the page table for converting each virtual address to physical address.
<input checked="" type="checkbox"/> ✗	<input type="radio"/> ✗	When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.
<input type="radio"/> ✗	<input checked="" type="checkbox"/> ✗	The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.
<input type="radio"/> ✗	<input checked="" type="checkbox"/> ✗	The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.
<input checked="" type="checkbox"/> ✗	<input type="radio"/> ✗	The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.
<input type="radio"/> ✗	<input checked="" type="checkbox"/> ✗	When a process is executing, each virtual address is converted into physical address by the kernel directly.
<input checked="" type="checkbox"/> ✗	<input type="radio"/> ✗	The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.

The kernel refers to the page table for converting each virtual address to physical address.: False

When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.: True

The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.: False

The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.: False

The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.: True

When a process is executing, each virtual address is converted into physical address by the kernel directly.: False

The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.: True

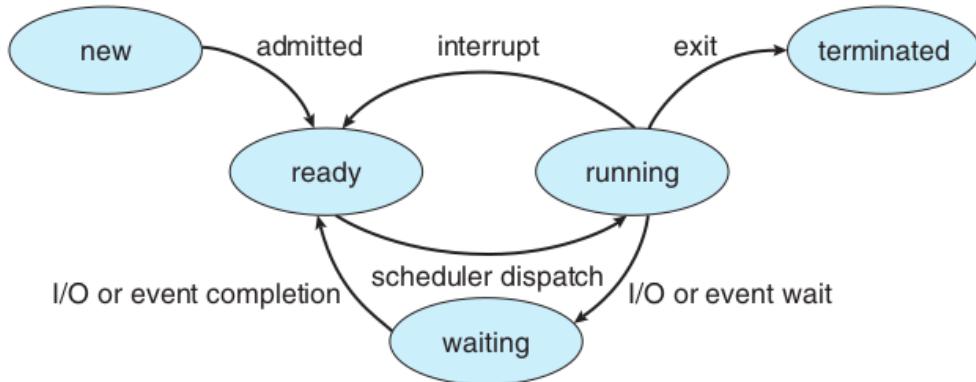
Question 16

Correct

Mark 0.50 out of 0.50

Mark statements True/False w.r.t. change of states of a process. Note that a statement is true only if the claim and argument both are true.

Reference: The process state diagram (and your understanding of how kernel code works). Note - the diagram does not show zombie state!



**Figure 3.2** Diagram of process state.

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	Every forked process has to go through ZOMBIE state, at least for a small duration.
<input type="radio"/>	<input checked="" type="radio"/>	A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first
<input checked="" type="radio"/>	<input type="radio"/>	Only a process in READY state is considered by scheduler
<input checked="" type="radio"/>	<input type="radio"/>	A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet
<input type="radio"/>	<input checked="" type="radio"/>	A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.

Every forked process has to go through ZOMBIE state, at least for a small duration.: True

A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first: False

Only a process in READY state is considered by scheduler: True

A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet: True

A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False

**Question 17**

Partially correct

Mark 0.45 out of 0.50

Select Yes if the mentioned element should be a part of PCB

Select No otherwise.

Yes	No	
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Memory management information about that process
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/>	Pointer to IDT
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/>	PID of Init
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	PID
<input type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/>	Function pointers to all system calls
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	EIP at the time of context switch
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Process state
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	List of opened files
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Process context
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Pointer to the parent process

Memory management information about that process: Yes

Pointer to IDT: No

PID of Init: No

PID: Yes

Function pointers to all system calls: No

EIP at the time of context switch: Yes

Process state: Yes

List of opened files: Yes

Process context: Yes

Pointer to the parent process: Yes

Question **18**

Correct

Mark 0.50 out of 0.50

Predict the output of the program given here.

Assume that all the path names for the programs are correct. For example "/usr/bin/echo" will actually run echo command.

Assume that there is no mixing of printf output on screen if two of them run concurrently.

In the answer replace a new line by a single space.

For example::

good

output

should be written as good output

--

```
main() {  
    int i;  
    i = fork();  
    if(i == 0)  
        execl("/usr/bin/echo", "/usr/bin/echo", "hi", 0);  
    else  
        wait(0);  
    fork();  
    execl("/usr/bin/echo", "/usr/bin/echo", "one", 0);  
}
```

Answer: hi one one



The correct answer is: hi one one

Question **19**

Correct

Mark 0.50 out of 0.50

How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security) ?

Select one:

- a. It prohibits one process from accessing other process's memory
- b. It prohibits invocation of kernel code completely, if a user program is running
- c. It prohibits a user mode process from running privileged instructions✓
- d. It disallows hardware interrupts when a process is running

Your answer is correct.

The correct answer is: It prohibits a user mode process from running privileged instructions

Question **20**

Partially correct

Mark 0.33 out of 0.50

Which of the following parts of a C program do not have any corresponding machine code ?

- a. pointer dereference
- b. #directives✓
- c. typedefs✓
- d. global variables
- e. function calls
- f. expressions
- g. local variable declaration

Your answer is partially correct.

You have correctly selected 2.

The correct answers are: #directives, typedefs, global variables

[◀ Quiz-1 Preparation questions](#)

Jump to...

[Quiz - 2 \(17 March 2023\) ►](#)

**Started on** Monday, 16 January 2023, 9:00 PM

**State** Finished

**Completed on** Monday, 16 January 2023, 10:07 PM

**Time taken** 1 hour 6 mins

**Grade** 13.25 out of 15.00 (88.33%)

Question **1**

Correct

Mark 1.00 out of 1.00

When you turn your computer ON, you are often shown an option like "Press F9 for boot options". What does this mean?

- a. The choice of booting slowly or fast
- b. The choice of which OS to boot from
- c. The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded ✓
- d. The choice of the boot loader (e.g. GRUB or Windows-Loader)

The correct answer is: The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded

Question **2**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about bootloader.

Every wrong selection will deduct marks proportional to  $1/n$  where  $n$  is total wrong choices in the question.

You will get minimum a zero.

- a. Modern Bootloaders often allow configuring the way an OS boots ✓
- b. Bootloader must be one sector in length
- c. The bootloader loads the BIOS
- d. Bootloaders allow selection of OS to boot from ✓
- e. LILO is a bootloader ✓

Your answer is correct.

The correct answers are: LILO is a bootloader, Modern Bootloaders often allow configuring the way an OS boots, Bootloaders allow selection of OS to boot from

**Question 3**

Correct

Mark 2.00 out of 2.00

What will this program do?

```
int main() {  
    fork();  
    execl("/bin/ls", "/bin/ls", NULL);  
    printf("hello");  
}
```

- a. one process will run ls, another will print hello
- b. run ls once
- c. run ls twice and print hello twice
- d. run ls twice and print hello twice, but output will appear in some random order
- e. run ls twice✓

Your answer is correct.

The correct answer is: run ls twice

**Question 4**

Correct

Mark 1.00 out of 1.00

Order the following events in boot process (from 1 onwards)

Login interface	5	✓
Init	4	✓
OS	3	✓
BIOS	1	✓
Shell	6	✓
Boot loader	2	✓

Your answer is correct.

The correct answer is: Login interface → 5, Init → 4, OS → 3, BIOS → 1, Shell → 6, Boot loader → 2

Question 5

Partially correct

Mark 0.50 out of 1.00

Select the correct statements about hard and soft links

Select one or more:

- a. Soft link shares the inode of actual file
- b. Deleting a hard link deletes the file, only if link count was 1 ✓
- c. Soft links increase the link count of the actual file inode
- d. Soft links can span across partitions while hard links can't ✓
- e. Deleting a soft link deletes only the actual file
- f. Deleting a hard link always deletes the file ✗
- g. Hard links can span across partitions while soft links can't
- h. Deleting a soft link deletes the link, not the actual file ✓
- i. Hard links increase the link count of the actual file inode
- j. Deleting a soft link deletes both the link and the actual file
- k. Hard links share the inode ✓
- l. Hard links enforce separation of filename from its metadata in on-disk data structures.

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: Soft links can span across partitions while hard links can't, Hard links increase the link count of the actual file inode, Deleting a soft link deletes the link, not the actual file, Deleting a hard link deletes the file, only if link count was 1, Hard links share the inode, Hard links enforce separation of filename from its metadata in on-disk data structures.

Question **6**

Partially correct

Mark 1.75 out of 3.00

Select correct statements about mounting

Select one or more:

- a. Mounting makes all disk partitions available as one name space
- b. Mounting deletes all data at the mount-point
- c. Even in operating systems with a pluggable kernel module for file systems, the code for mounting any particular file system must X
- d. It's possible to mount a partition on one computer, into namespace of another computer. ✓
- e. Mounting is attaching a disk-partition with a filesystem on it, into another file system name-space ✓
- f. On Linuxes mounting can be done only while booting the OS
- g. The existing name-space at the mount-point is no longer visible after mounting ✓
- h. The mount point can be a file as well
- i. The mount point must be a directory ✓
- j. In operating systems with a pluggable kernel module for file systems, the code for mounting a particular file system is provided by the module of that file system. ✓

Your answer is partially correct.

You have correctly selected 5.

The correct answers are: Mounting is attaching a disk-partition with a filesystem on it, into another file system name-space, The mount point must be a directory, The existing name-space at the mount-point is no longer visible after mounting, Mounting makes all disk partitions available as one name space, In operating systems with a pluggable kernel module for file systems, the code for mounting a particular file system is provided by the module of that file system., It's possible to mount a partition on one computer, into namespace of another computer.

Question **7**

Correct

Mark 1.00 out of 1.00

Compare multiprogramming with multitasking

- a. A multitasking system is not necessarily multiprogramming
- b. A multiprogramming system is not necessarily multitasking ✓

The correct answer is: A multiprogramming system is not necessarily multitasking

Question **8**

Correct

Mark 1.00 out of 1.00

Is the terminal a part of the kernel on GNU/Linux systems?

- a. yes
- b. no ✓ wrong

The correct answer is: no

Question 9

Correct

Mark 1.00 out of 1.00

Consider the following programs

[exec1.c](#)

```
#include <unistd.h>
#include <stdio.h>
int main() {
    execl("./exec2", "./exec2", NULL);
}
```

[exec2.c](#)

```
#include <unistd.h>
#include <stdio.h>
int main() {
    execl("/bin/ls", "/bin/ls", NULL);
    printf("hello\n");
}
```

Compiled as

```
cc  exec1.c  -o exec1
cc  exec2.c  -o exec2
```

And run as

[\\$ ./exec1](#)

Explain the output of the above command ([./exec1](#))

Assume that /bin/ls , i.e. the 'ls' program exists.

Select one:

- a. Execution fails as the call to execl() in exec2 fails
- b. Execution fails as the call to execl() in exec1 fails
- c. "ls" runs on current directory ✓
- d. Execution fails as one exec can't invoke another exec
- e. Program prints hello

Your answer is correct.

The correct answer is: "ls" runs on current directory

Question 10

Correct

Mark 1.00 out of 1.00

A process blocks itself means

- a. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
- b. The kernel code of system call calls scheduler
- c. The application code calls the scheduler
- d. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler ✓

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

Question 11

Correct

Mark 1.00 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

- a. Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode ✓
- b. The two modes are essential for a multitasking system ✓
- c. There is an instruction like 'iret' to return from kernel mode to user mode ✓
- d. The two modes are essential for a multiprogramming system ✓
- e. The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously ✓

Your answer is correct.

The correct answers are: The two modes are essential for a multiprogramming system, The two modes are essential for a multitasking system, There is an instruction like 'iret' to return from kernel mode to user mode, The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously, Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode

Question **12**

Correct

Mark 1.00 out of 1.00

which of the following is not a difference between real mode and protected mode

- a. in real mode the addressable memory is less than in protected mode
- b. in real mode the addressable memory is more than in protected mode ✓
- c. processor starts in real mode
- d. in real mode general purpose registers are 16 bit, in protected mode they are 32 bit
- e. in real mode the segment is multiplied by 16, in protected mode segment is used as index in GDT

The correct answer is: in real mode the addressable memory is more than in protected mode

[◀ Random Quiz - 1 \(Pre-Requisite Quiz\)](#)

Jump to...

[Random Quiz - 3 \(processes, memory management, event driven kernel\), compilation-linking-loading, ipc-pipes ►](#)

**Started on** Thursday, 2 February 2023, 9:01 PM

**State** Finished

**Completed on** Thursday, 2 February 2023, 10:55 PM

**Time taken** 1 hour 53 mins

**Grade** 16.46 out of 20.00 (82.28%)

Question 1

Complete

Mark 1.00 out of 1.00

Which of the following are NOT a part of job of a typical compiler?

- a. Check the program for logical errors
- b. Convert high level language code to machine code
- c. Invoke the linker to link the function calls with their code, extern globals with their declaration
- d. Suggest alternative pieces of code that can be written
- e. Check the program for syntactical errors
- f. Process the # directives in a C program

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

**Question 2**

Complete

Mark 1.00 out of 1.00

Consider the following code and MAP the file to which each fd points at the end of the code.

```
int main(int argc, char *argv[]) {  
    int fd1, fd2 = 1, fd3 = 1, fd4 = 1;  
  
    fd1 = open("/tmp/1", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);  
    fd2 = open("/tmp/2", O_RDONLY);  
    fd3 = open("/tmp/3", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);  
    close(0);  
    close(1);  
    dup(fd2);  
    dup(fd3);  
    close(fd3);  
    dup2(fd2, fd4);  
    printf("%d %d %d %d\n", fd1, fd2, fd3, fd4);  
    return 0;  
}
```

fd4	/tmp/2
fd2	/tmp/2
1	/tmp/3
fd1	/tmp/1
fd3	closed
0	/tmp/2
2	stderr

The correct answer is: fd4 → /tmp/2, fd2 → /tmp/2, 1 → /tmp/3, fd1 → /tmp/1, fd3 → closed, 0 → /tmp/2, 2 → stderr

Question 3

Complete

Mark 0.75 out of 1.00

Select the sequence of events that are NOT possible, assuming an interruptible kernel code

Select one or more:

- a. P1 running  
keyboard hardware interrupt  
keyboard interrupt handler running  
interrupt handler returns  
P1 running  
P1 makes system call  
system call returns  
P1 running  
timer interrupt  
scheduler  
P2 running
- b. P1 running  
P1 makes system call and blocks  
Scheduler  
P2 running  
P2 makes system call and blocks  
Scheduler  
P3 running  
Hardware interrupt  
Interrupt unblocks P1  
Interrupt returns  
P3 running  
Timer interrupt  
Scheduler  
P1 running
- c. P1 running  
P1 makes system call  
system call returns  
P1 running  
timer interrupt  
Scheduler running  
P2 running
- d. P1 running  
P1 makes system call and blocks  
Scheduler  
P2 running  
P2 makes system call and blocks  
Scheduler  
P1 running again
- e. P1 running  
P1 makes system call  
Scheduler  
P2 running  
P2 makes system call and blocks  
Scheduler  
P1 running again

- f. P1 running  
P1 makes system call  
timer interrupt  
Scheduler  
P2 running  
timer interrupt  
Scheuler  
P1 running  
P1's system call return

The correct answers are: P1 running

P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again,

P1 running

P1 makes system call

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again

Question 4

Complete

Mark 0.00 out of 1.00

Select all the correct statements about named pipes and ordinary(unnamed) pipe

Select one or more:

- a. named pipes can be used between multiple processes but ordinary pipes can not be used
- b. named pipe can be used between any processes
- c. both named and unnamed pipes require some kind of agreed protocol to be effectively used among multiple processes
- d. named pipes are more efficient than ordinary pipes
- e. ordinary pipe can only be used between related processes
- f. a named pipe exists as a file on the file system
- g. named pipe exists even if the processes using it do exit()

The correct answers are: ordinary pipe can only be used between related processes, named pipe can be used between any processes, a named pipe exists as a file on the file system, named pipe exists even if the processes using it do exit(), both named and unnamed pipes require some kind of agreed protocol to be effectively used among multiple processes

Question 5

Complete

Mark 4.50 out of 5.00

Following code claims to implement the command

/bin/ls -l | /usr/bin/head -3 | /usr/bin/tail -1

Fill in the blanks to make the code work.

Note: Do not include space in writing any option. x[1][2] should be written without any space, and so is the case with [1] or [2]. Pay attention to exact syntax and do not write any extra character like ';' or = etc.

```
int main(int argc, char *argv[]) {
    int pid1, pid2;
    int pfd[ 2 ][2];
    pipe( pfd[0] );
    pid1 = fork();
    if(pid1 != 0) {
        close(pfd[0][0]);
        close( 1 );
        dup( pfd[0][1] );
        execl("/bin/ls", "/bin/ls", " -l ", NULL);
    }
    pipe( pfd[1] );
    pid2 = fork();
    if(pid2 == 0) {
        close( pfd[0][1] );
        close(0);
        dup( pfd[1][0] );
        close(pfd[1][0]);
        close( 1 );
        dup( pfd[1][1] );
        execl("/usr/bin/head", "/usr/bin/head", " -3 ", NULL);
    } else {
        close(pfd[1][1]);
        close( 0 );
        dup( pfd[1][0] );
        close(pfd[0][0]);
    }
}
```

```

execl("/usr/bin/tail", "/usr/bin/tail", " -1 ", NULL);
}
}

```

**Question 6**

Complete

Mark 1.00 out of 1.00

Select the compiler's view of the process's address space, for each of the following MMU schemes:

(Assume that each scheme, e.g. paging/segmentation/etc is effectively utilised)

- |                           |   |
|---------------------------|---|
| Paging                    | one continuous chunk                    |
| Segmentation, then paging | many continuous chunks of variable size |
| Segmentation              | many continuous chunks of variable size |
| Relocation + Limit        | one continuous chunk                    |

The correct answer is: Paging → one continuous chunk, Segmentation, then paging → many continuous chunks of variable size, Segmentation → many continuous chunks of variable size, Relocation + Limit → one continuous chunk

**Question 7**

Complete

Mark 1.40 out of 2.00

Match the elements of C program to their place in memory

- |                         |                  |
|-------------------------|------------------|
| Local Variables         | Stack            |
| Allocated Memory        | Heap             |
| Global Static variables | Data             |
| Global variables        | Stack            |
| Code of main()          | Code             |
| Local Static variables  | Data             |
| Function code           | Data             |
| Arguments               | Stack            |
| #define MACROS          | No memory needed |
| #include files          | No memory needed |

The correct answer is: Local Variables → Stack, Allocated Memory → Heap, Global Static variables → Data, Global variables → Data, Code of main() → Code, Local Static variables → Data, Function code → Code, Arguments → Stack, #define MACROS → No Memory needed, #include files → No memory needed

Question 8

Complete

Mark 0.63 out of 1.00

Consider the image given below, which explains how paging works.

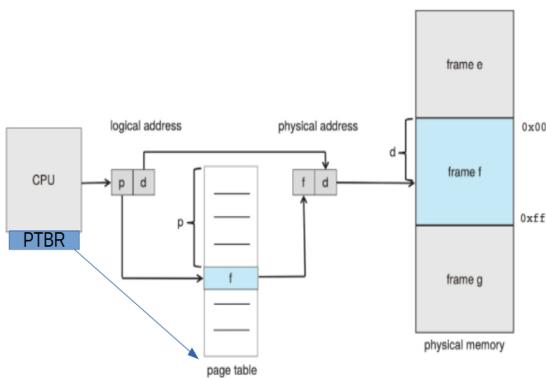


Figure 9.8 Paging hardware.

Mention whether each statement is True or False, with respect to this image.

**True      False**

- The physical address may not be of the same size (in bits) as the logical address
- The locating of the page table using PTBR also involves paging translation
- Size of page table is always determined by the size of RAM
- The page table is indexed using frame number
- The PTBR is present in the CPU as a register
- The page table is indexed using page number
- Maximum Size of page table is determined by number of bits used for page number
- The page table is itself present in Physical memory

The physical address may not be of the same size (in bits) as the logical address: True

The locating of the page table using PTBR also involves paging translation: False

Size of page table is always determined by the size of RAM: False

The page table is indexed using frame number: False

The PTBR is present in the CPU as a register: True

The page table is indexed using page number: True

Maximum Size of page table is determined by number of bits used for page number: True

The page table is itself present in Physical memory: True

Question 9

Complete

Mark 0.67 out of 1.00

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

(Note: non-interruptible kernel code means, if the kernel code is executing, then interrupts will be disabled).

Note: A possible sequence may have some missing steps in between. An impossible sequence will have n and n+1th steps such that n+1th step can not follow n'th step.

Select one or more:

- a. P1 running

P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P3 running

Hardware interrupt

Interrupt unblocks P1

Interrupt returns

P3 running

Timer interrupt

Scheduler

P1 running

- b. P1 running

P1 makes system call

system call returns

P1 running

timer interrupt

Scheduler running

P2 running

- c. P1 running

P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again

- d. P1 running

keyboard hardware interrupt

keyboard interrupt handler running

interrupt handler returns

P1 running

P1 makes system call

system call returns

P1 running

timer interrupt

scheduler

P2 running

- e. P1 running

P1 makes system call

timer interrupt

Scheduler

P2 running  
timer interrupt  
Scheuler  
P1 running  
P1's system call return

f.

P1 running  
P1 makes system call  
Scheduler  
P2 running  
P2 makes system call and blocks  
Scheduler  
P1 running again

The correct answers are: P1 running

P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again, P1 running

P1 makes system call

timer interrupt

Scheduler

P2 running

timer interrupt

Scheuler

P1 running

P1's system call return,

P1 running

P1 makes system call

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again

**Question 10**

Complete

Mark 1.00 out of 1.00

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

\$ ls ./tmp/asdfksdf >/tmp/ddd 2>&1

**Program 1**

```
int main(int argc, char *argv[]) {  
    int fd, n, i;  
    char buf[128];  
  
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);  
    close(1);  
    dup(fd);  
    close(2);  
    dup(fd);  
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);  
}
```

**Program 2**

```
int main(int argc, char *argv[]) {  
    int fd, n, i;  
    char buf[128];  
  
    close(1);  
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);  
    close(2);  
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);  
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);  
}
```

Select all the correct statements about the programs

Select one or more:

- a. Both programs are correct
- b. Program 2 is correct for > /tmp/ddd but not for 2>&1
- c. Program 1 is correct for > /tmp/ddd but not for 2>&1
- d. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd
- e. Program 2 makes sure that there is one file offset used for '2' and '1'
- f. Program 1 ensures 2>&1 and does not ensure > /tmp/ddd
- g. Only Program 2 is correct
- h. Program 1 does 1>&2
- i. Program 1 makes sure that there is one file offset used for '2' and '1'
- j. Program 2 does 1>&2
- k. Both program 1 and 2 are incorrect
- l. Only Program 1 is correct

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

Question **11**

Complete

Mark 0.71 out of 1.00

Order the events that occur on a timer interrupt:

Jump to a code pointed by IDT

2

Save the context of the currently running process

3

Execute the code of the new process

6

Jump to scheduler code

4

Change to kernel stack of currently running process

1

Select another process for execution

5

Set the context of the new process

7

The correct answer is: Jump to a code pointed by IDT → 2, Save the context of the currently running process → 3, Execute the code of the new process → 7, Jump to scheduler code → 4, Change to kernel stack of currently running process → 1, Select another process for execution → 5, Set the context of the new process → 6

Question **12**

Complete

Mark 0.80 out of 1.00

Select all the correct statements about zombie processes

Select one or more:

- a. A process becomes zombie when its parent finishes
- b. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it
- c. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent
- d. Zombie processes are harmless even if OS is up for long time
- e. A process can become zombie if it finishes, but the parent has finished before it
- f. A zombie process occupies space in OS data structures
- g. init() typically keeps calling wait() for zombie processes to get cleaned up
- h. A zombie process remains zombie forever, as there is no way to clean it up

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

**Question 13**

Complete

Mark 1.00 out of 1.00

Select the state that is not possible after the given state, for a process:

New:  RunningReady :  WaitingRunning: :  None of theseWaiting:  Running**Question 14**

Complete

Mark 1.00 out of 1.00

Select the order in which the various stages of a compiler execute.

Pre-processing  1Intermediate code generation  3Linking  4Syntactical Analysis  2Loading  does not exist

The correct answer is: Pre-processing → 1, Intermediate code generation → 3, Linking → 4, Syntactical Analysis → 2, Loading → does not exist

**Question 15**

Complete

Mark 1.00 out of 1.00

A process blocks itself means

- a. The application code calls the scheduler
- b. The kernel code of system call calls scheduler
- c. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
- d. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

Jump to...

[Random Quiz 4 : Scheduling, signals, segmentation, paging, compilation, process-state ►](#)

---

**Started on** Thursday, 9 March 2023, 6:20 PM

**State** Finished

---

**Completed on** Thursday, 9 March 2023, 7:30 PM

**Time taken** 1 hour 10 mins

**Overdue** 14 mins

---

**Grade** **5.55** out of 10.00 (**55.53%**)

**Question 1**

Partially correct

Mark 0.18 out of 1.00

W.r.t. Memory management in xv6,

xv6 uses physical memory upto 224 MB only  
Mark statements True or False

True	False	
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	xv6 uses physical memory upto 224 MB only
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The stack allocated in entry.S is used as stack for scheduler's context for first processor
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The free page-frame are created out of nearly 222 MB
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	PHYSTOP can be increased to some extent, simply by editing memlayout.h
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() changes CR3 to use page directory of new process
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The process's address space gets mapped on frames, obtained from ~2MB:224MB range
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The kernel's page table given by kpgdir variable is used as stack for scheduler's context
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The kernel code and data take up less than 2 MB space
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir

xv6 uses physical memory upto 224 MB only: True

The stack allocated in entry.S is used as stack for scheduler's context for first processor: True

The free page-frame are created out of nearly 222 MB: True

The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True

PHYSTOP can be increased to some extent, simply by editing memlayout.h: True

The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False

The process's address space gets mapped on frames, obtained from ~2MB:224MB range: True

The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False

The kernel code and data take up less than 2 MB space: True

The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False

The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True

Question **2**

Correct

Mark 1.00 out of 1.00

What's the trapframe in xv6?

- a. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S ✓
- b. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by code in trapasm.S only
- c. The IDT table
- d. A frame of memory that contains all the trap handler's addresses
- e. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware only
- f. A frame of memory that contains all the trap handler code
- g. A frame of memory that contains all the trap handler code's function pointers

Your answer is correct.

The correct answer is: The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S

Question 3

Partially correct

Mark 0.75 out of 1.00

Select the correct statements about interrupt handling in xv6 code

- a. The trapframe pointer in struct proc, points to a location on process's kernel stack ✓
- b. The CS and EIP are changed only after pushing user code's SS,ESP on stack ✓
- c. On any interrupt/syscall/exception the control first jumps in trapasm.S
- d. All the 256 entries in the IDT are filled in xv6 code ✓
- e. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt
- f. Before going to alltraps, the kernel stack contains upto 5 entries.
- g. The trapframe pointer in struct proc, points to a location on user stack
- h. The function trap() is the called only in case of hardware interrupt
- i. The CS and EIP are changed immediately (as the first thing) on a hardware interrupt
- j. xv6 uses the 0x64th entry in IDT for system calls
- k. xv6 uses the 64th entry in IDT for system calls ✓
- l. On any interrupt/syscall/exception the control first jumps in vectors.S ✓
- m. The function trap() is the called even if any of the hardware interrupt/system-call/exception occurs ✓

Your answer is partially correct.

You have correctly selected 6.

The correct answers are: All the 256 entries in the IDT are filled in xv6 code, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in vectors.S, Before going to alltraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on process's kernel stack, The function trap() is the called even if any of the hardware interrupt/system-call/exception occurs, The CS and EIP are changed only after pushing user code's SS,ESP on stack

Question 4

Partially correct

Mark 0.55 out of 1.00

Consider the following command and its output:

```
$ ls -lht xv6.img kernel
-rw-rw-r-- 1 abhijit abhijit 4.9M Feb 15 11:09 xv6.img
-rwxrwxr-x 1 abhijit abhijit 209K Feb 15 11:09 kernel*
```

Following code in bootmain()

```
readseg((uchar*)elf, 4096, 0);
```

and following selected lines from Makefile

```
xv6.img: bootblock kernel
    dd if=/dev/zero of=xv6.img count=10000
    dd if=bootblock of=xv6.img conv=notrunc
    dd if=kernel of=xv6.img seek=1 conv=notrunc

kernel: $(OBJS) entry.o entryother initcode kernel.ld
    $(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother
    $(OBJDUMP) -S kernel > kernel.asm
    $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

Also read the code of bootmain() in xv6 kernel.

Select the options that describe the meaning of these lines and their correlation.

- a. Althought the size of the kernel file is 209 Kb, only 4Kb out of it is the actual kernel code and remaining part is all zeroes.
- b. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is not X read as it is user programs.
- c. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(). ✓
- d. The kernel.asm file is the final kernel file
- e. Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.
- f. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✓
- g. The kernel.ld file contains instructions to the linker to link the kernel properly ✓
- h. readseg() reads first 4k bytes of kernel in memory ✓
- i. The bootmain() code does not read the kernel completely in memory

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(), readseg() reads first 4k bytes of kernel in memory, The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files, The kernel.ld file contains instructions to the linker to link the kernel properly, Althought the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.

**Question 5**

Correct

Mark 1.00 out of 1.00

For each function/code-point, select the status of segmentation setup in xv6

bootmain()	gdt setup with 3 entries, at start32 symbol of bootasm.S	✓
kvmalloc() in main()	gdt setup with 3 entries, at start32 symbol of bootasm.S	✓
entry.S	gdt setup with 3 entries, at start32 symbol of bootasm.S	✓
bootasm.S	gdt setup with 3 entries, at start32 symbol of bootasm.S	✓
after startothers() in main()	gdt setup with 5 entries (0 to 4) on all processors	✓
after seginit() in main()	gdt setup with 5 entries (0 to 4) on one processor	✓

Your answer is correct.

The correct answer is: bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S, kvmalloc() in main() → gdt setup with 3 entries, at start32 symbol of bootasm.S, entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, bootasm.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, after startothers() in main() → gdt setup with 5 entries (0 to 4) on all processors, after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor

**Question 6**

Correct

Mark 1.00 out of 1.00

Some part of the bootloader of xv6 is written in assembly while some part is written in C. Why is that so?

Select all the appropriate choices

- a. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time
- b. The setting up of the most essential memory management infrastructure needs assembly code ✓
- c. The code for reading ELF file can not be written in assembly
- d. The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C ✓

Your answer is correct.

The correct answers are: The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C, The setting up of the most essential memory management infrastructure needs assembly code

Question 7

Incorrect

Mark 0.00 out of 1.00

xv6.img: bootblock kernel

```
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

Consider above lines from the Makefile. Which of the following is INCORRECT?

- a. The kernel is located at block-1 of the xv6.img ✗
- b. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk. ✓
- c. The size of the kernel file is nearly 5 MB ✓
- d. The bootblock is located on block-0 of the xv6.img ✗
- e. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies 10,000 blocks on the disk. ✗
- f. Blocks in xv6.img after kernel may be all zeroes. ✗
- g. xv6.img is the virtual processor used by the qemu emulator
- h. The size of the xv6.img is nearly 5 MB ✗
- i. The xv6.img is the virtual disk that is created by combining the bootblock and the kernel file. ✗
- j. The bootblock may be 512 bytes or less (looking at the Makefile instruction) ✗
- k. The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

Your answer is incorrect.

The correct answers are: xv6.img is the virtual processor used by the qemu emulator, The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk., The size of the kernel file is nearly 5 MB, The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

Question 8

Partially correct

Mark 0.07 out of 1.00

Select all the correct statements about code of bootmain() in xv6

```
void
bootmain(void)
{
    struct elfhdr *elf;
    struct proghdr *ph, *eph;
    void (*entry) (void);
    uchar* pa;

    elf = (struct elfhdr*)0x10000; // scratch space

    // Read 1st page off disk
    readseg((uchar*)elf, 4096, 0);

    // Is this an ELF executable?
    if(elf->magic != ELF_MAGIC)
        return; // let bootasm.S handle error

    // Load each program segment (ignores ph flags).
    ph = (struct proghdr*)((uchar*)elf + elf->phoff);
    eph = ph + elf->phnum;
    for(; ph < eph; ph++) {
        pa = (uchar*)ph->paddr;
        readseg(pa, ph->filesz, ph->off);
        if(ph->memsz > ph->filesz)
            stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
    }

    // Call the entry point from the ELF header.
    // Does not return!
    entry = (void(*)(void)) (elf->entry);
    entry();
}
```

Also, inspect the relevant parts of the xv6 code. binary files, etc and run commands as you deem fit to answer this question.

- a. The kernel file has only two program headers ✓
- b. The kernel file gets loaded at the Physical address 0x10000 in memory. ✓
- c. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✗
- d. The readseg finally invokes the disk I/O code using assembly instructions
- e. The elf->entry is set by the linker in the kernel file and it's 8010000c
- f. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded
- g. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it. ✓
- h. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✗
- i. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory.

- j. The condition if(ph->memsz > ph->filesz) is never true.
- k. The stosb() is used here, to fill in some space in memory with zeroes ✓

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: The kernel file gets loaded at the Physical address 0x10000 in memory., The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it., The elf->entry is set by the linker in the kernel file and it's 8010000c, The readseg finally invokes the disk I/O code using assembly instructions, The stosb() is used here, to fill in some space in memory with zeroes, The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded, The kernel file has only two program headers

Question 9

Correct

Mark 1.00 out of 1.00

In bootasm.S, on the line

```
1jmp    $(SEG_KCODE<<3), $start32
```

The SEG\_KCODE << 3, that is shifting of 1 by 3 bits is done because

- a. While indexing the GDT using CS, the value in CS is always divided by 8
- b. The code segment is 16 bit and only upper 13 bits are used for segment number ✓
- c. The value 8 is stored in code segment
- d. The code segment is 16 bit and only lower 13 bits are used for segment number
- e. The ljmp instruction does a divide by 8 on the first argument

Your answer is correct.

The correct answer is: The code segment is 16 bit and only upper 13 bits are used for segment number

**Question 10**

Incorrect

Mark 0.00 out of 1.00

For each line of code mentioned on the left side, select the location of sp/esp that is in use

`ljmp $(SEG_KCODE<<3), $start32  
in bootasm.S`

0x7c00 to 0



`readseg((uchar*)elf, 4096, 0);  
in bootmain.c`

Immaterial as the stack is not used here



`jmp *%eax  
in entry.S`

0x7c00 to 0



`cli  
in bootasm.S`

Choose...

`call bootmain  
in bootasm.S`

Immaterial as the stack is not used here



Your answer is incorrect.

The correct answer is: `ljmp $(SEG_KCODE<<3), $start32`

`in bootasm.S → Immaterial as the stack is not used here, readseg((uchar*)elf, 4096, 0);`

`in bootmain.c → 0x7c00 to 0, jmp *%eax`

`in entry.S → The 4KB area in kernel image, loaded in memory, named as 'stack', cli`

`in bootasm.S → Immaterial as the stack is not used here, call bootmain`

`in bootasm.S → 0x7c00 to 0`

[◀ Random Quiz 4 : Scheduling, signals, segmentation, paging, compilation, process-state](#)

Jump to...

[Random Quiz - 6 \(xv6 file system\) ►](#)

**Started on** Friday, 31 March 2023, 6:18 PM

**State** Finished

**Completed on** Friday, 31 March 2023, 7:00 PM

**Time taken** 41 mins 52 secs

**Grade** 4.92 out of 15.00 (32.83%)

Question 1

Partially correct

Mark 0.67 out of 1.00

Select all the actions taken by iget()

- a. Returns an inode with given dev+inode-number from cache, if it exists in cache ✓
- b. Returns the inode with reference count incremented ✓
- c. Returns a free-inode , with dev+inode-number set, if not found in cache
- d. Panics if inode does not exist in cache
- e. Returns a valid inode if not found in cache
- f. Returns the inode with inode-cache lock held
- g. Returns the inode locked

Your answer is partially correct.

You have correctly selected 2.

The correct answers are: Returns an inode with given dev+inode-number from cache, if it exists in cache, Returns the inode with reference count incremented, Returns a free-inode , with dev+inode-number set, if not found in cache

Question **2**

Incorrect

Mark 0.00 out of 1.00

Note: for this question you get full marks if you select all and only correct options, you get ZERO if at least one option is wrong or not selected.

Select all the correct statements about log structured file systems.

- a. ext4 is a log structured file system ✗ it's a journaled file system, not log structured
- b. file system recovery recovers all the lost data ✗
- c. log structured file systems considerably improve the recovery time ✓
- d. xv6 has a log structured file system
- e. ext2 is by default a log structured file system ✗

Your answer is incorrect.

The correct answers are: xv6 has a log structured file system, log structured file systems considerably improve the recovery time

**Question 3**

Partially correct

Mark 0.86 out of 2.00

Select T/F w.r.t physical disk handling in xv6 code

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	device files are not supported <span style="color: red;">✗</span>
<input type="radio"/>	<input checked="" type="radio"/>	The code supports IDE, and not SATA/SCSI <span style="color: red;">✗</span>
<input checked="" type="radio"/>	<input checked="" type="radio"/>	log is kept on the same device as the file system <span style="color: green;">✓</span>
<input checked="" type="radio"/>	<input checked="" type="radio"/>	disk driver handles only one buffer at a time <span style="color: green;">✓</span>
<input checked="" type="radio"/>	<input checked="" type="radio"/>	the superblock does not contain number of free blocks <span style="color: green;">✓</span>
<input checked="" type="radio"/>	<input type="radio"/>	only direct blocks are supported <span style="color: red;">✗</span>
<input checked="" type="radio"/>	<input checked="" type="radio"/>	only 2 disks are handled by default <span style="color: red;">✗</span>

device files are not supported: False

The code supports IDE, and not SATA/SCSI: True

log is kept on the same device as the file system: True

disk driver handles only one buffer at a time: True

the superblock does not contain number of free blocks: True

only direct blocks are supported: False

only 2 disks are handled by default: True

**Question 4**

Partially correct

Mark 1.00 out of 2.00

Marks the statements as True/False w.r.t. "struct buf"

True	False	
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	B_VALID means the buffer is empty and can be reused
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	Lock on a buffer is acquired in bget, and released in brelse
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The reference count (refcnt) in struct buf is = number of processes accessing the buffer
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	A buffer can have both B_VALID and B_DIRTY flags set
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The "next" pointer chain gives the buffers in LRU order
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	A buffer can be both on the MRU/LRU list and also on ideoqueue list.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The buffers are maintained in LRU order, in the function brelse
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	B_DIRTY flag means the buffer contains modified data

B\_VALID means the buffer is empty and can be reused: False

Lock on a buffer is acquired in bget, and released in brelse: True

The reference count (refcnt) in struct buf is = number of processes accessing the buffer: True

A buffer can have both B\_VALID and B\_DIRTY flags set: False

The "next" pointer chain gives the buffers in LRU order: False

A buffer can be both on the MRU/LRU list and also on ideoqueue list.: True

The buffers are maintained in LRU order, in the function brelse: True

B\_DIRTY flag means the buffer contains modified data: True

Question **5**

Partially correct

Mark 0.40 out of 1.00

Arrange the following in their typical order of use in xv6.

1.  iget
2.  ilock
3.  iunlock
4.  iput
5.  use inode

Your answer is partially correct.

Grading type: Relative to the next item (including last)

Grade details: 2 / 5 = 40%

Here are the scores for each item in this response:

1. 1 / 1 = 100%
2. 0 / 1 = 0%
3. 1 / 1 = 100%
4. 0 / 1 = 0%
5. 0 / 1 = 0%

The correct order for these items is as follows:

1. iget
2. ilock
3. use inode
4. iunlock
5. iput

**Question 6**

Partially correct

Mark 0.50 out of 1.00

Compare XV6 and EXT2 file systems.

Select True/False for each point.

True	False	
<input checked="" type="radio"/> ✗	<input type="radio"/> ✓	In both ext2 and xv6, the superblock gives location of first inode block <span style="color: red;">✗</span>
<input type="checkbox"/> ✓	<input checked="" type="radio"/> ✗	xv6 contains journal, ext2 does not <span style="color: red;">✗</span>
<input checked="" type="radio"/> ✗	<input checked="" type="radio"/> ✓	Ext2 contains superblock but xv6 does not. <span style="color: green;">✓</span>
<input checked="" type="radio"/> ✗	<input checked="" type="radio"/> ✓	xv6 contains inode bitmap, but ext2 does not <span style="color: green;">✓</span>
<input checked="" type="radio"/> ✗	<input type="radio"/> ✓	Both xv6 and ext2 contain magic number <span style="color: red;">✗</span>
<input type="checkbox"/> ✓	<input checked="" type="radio"/> ✗	Ext2 contains group descriptors but xv6 does not <span style="color: green;">✓</span>

In both ext2 and xv6, the superblock gives location of first inode block: False

xv6 contains journal, ext2 does not: True

Ext2 contains superblock but xv6 does not.: False

xv6 contains inode bitmap, but ext2 does not: False

Both xv6 and ext2 contain magic number: False

Ext2 contains group descriptors but xv6 does not: True

**Question 7**

Incorrect

Mark 0.00 out of 1.00

Maximum size of a file on xv6 in **bytes** is

(just write a numeric answer)

Answer: 512

✗

The correct answer is: 71680

**Question 8**

Partially correct

Mark 0.17 out of 1.00

Select all the actions taken by ilock()

- a. Mark the in-memory inode as valid, if needed
- b. Read the inode from disk, if needed ✓
- c. Take the sleeplock on the inode, always
- d. Get the inode from the inode-cache
- e. Copy the on-disk inode into in-memory inode, if neeed ✓
- f. Take the sleeplock on the inode, optionally ✗
- g. Lock all the buffers of the file in memory

Your answer is partially correct.

You have correctly selected 2.

The correct answers are: Read the inode from disk, if needed, Copy the on-disk inode into in-memory inode, if neeed, Take the sleeplock on the inode, always, Mark the in-memory inode as valid, if needed

**Question 9**

Incorrect

Mark 0.00 out of 1.00

The lines

```
if(ip->type != T_DIR){  
    iunlockput(ip);  
    return 0;  
}
```

in namex() function

mean

- a. The last path component (which is a file, and not a directory) has been resolved, so release the lock (using iunlockput) and return
- b. One of the sub-components on the given path name, was a directory, but it was not supposed to be a directory, hence an error ✗
- c. There was a syntax error in the pathname specified
- d. One of the sub-components on the given path name, was not a directory, hence it's an error
- e. One of the sub-components on the given path name, did not exist, hence it's an error ✗
- f. ilock is held on the inode, and hence it's an error if it is a directory ✗
- g. No directory entry was found for the file to be opened, hence an error

Your answer is incorrect.

The correct answer is: One of the sub-components on the given path name, was not a directory, hence it's an error

**Question 10**

Partially correct

Mark 0.50 out of 1.00

Suppose an application on xv6 does the following:

```
int main() {
    char arr[128];
    int fd = open("README", O_RDONLY);
    read(fd, arr, 100);
}
```

Assume that the code works.

Which of the following things are true about xv6 kernel code, w.r.t. the above C program.

True	False	
<input checked="" type="radio"/> ✘	<input type="radio"/> ✓	The "memmove(dst, bp->data + off%BSIZE, m);" in readi() will copy the data from the disk to the kernel buffers
<input type="radio"/> ✓	<input checked="" type="radio"/> ✘	The process will be made to sleep only once
<input checked="" type="radio"/> ✘	<input type="radio"/> ✓	The data is transferred from disk to kernel buffers first, and then address of arr is mapped to the kernel buffers
<input type="radio"/> ✓	<input checked="" type="radio"/> ✘	The ONLY function that gets called on return devsw[ip->major].read(ip, dst, n); is consoleread
<input checked="" type="radio"/> ✘	<input type="radio"/> ✓	The loop in readi() will always read a different block using bread()
<input type="radio"/> ✓	<input checked="" type="radio"/> ✘	value of fd will be 3

The "memmove(dst, bp->data + off%BSIZE, m);" in readi() will copy the data from the disk to the kernel buffers: False

The process will be made to sleep only once: True

The data is transferred from disk to kernel buffers first, and then address of arr is mapped to the kernel buffers: False

The ONLY function that gets called on return devsw[ip->major].read(ip, dst, n); is consoleread: True

The loop in readi() will always read a different block using bread(): False

value of fd will be 3: True

Question 11

Partially correct

Mark 0.83 out of 1.00

Map the function in xv6's file system code, to its perceived logical layer.

ideintr	disk driver	✓
filestat()	file descriptor	✓
ialloc	inode	✓
bread	buffer cache	✓
balloc	system call	✗
skipelem	pathname lookup	✓
bmap	pathname lookup	✗
sys_chdir()	system call	✓
stati	inode	✓
commit	logging	✓
namei	pathname lookup	✓
dirlookup	directory	✓

Your answer is partially correct.

You have correctly selected 10.

The correct answer is: ideintr → disk driver, filestat() → file descriptor, ialloc → inode, bread → buffer cache, balloc → block allocation on disk, skipelem → pathname lookup, bmap → inode, sys\_chdir() → system call, stati → inode, commit → logging, namei → pathname lookup, dirlookup → directory

**Question 12**

Incorrect

Mark 0.00 out of 1.00

Match function with its functionality

nameiparent	Write a new entry in a given directory	✗
dirlink	Link a directory with another directory	✗
namex	Search a given name in a given directory	✗
dirlookup	Lookup (search) for a given directory	✗

Your answer is incorrect.

The correct answer is: nameiparent → return in-memory inode for parent directory of a given pathname, dirlink → Write a new entry in a given directory, namex → return in-memory inode for a given pathname, dirlookup → Search a given name in a given directory

**Question 13**

Incorrect

Mark 0.00 out of 1.00

An inode is read from disk as a part of this function

- a. ilock
- b. sys\_read ✗
- c. iget
- d. readi
- e. iread

Your answer is incorrect.

The correct answer is: ilock

[◀ Random Quiz - 5: xv6 make, bootloader, interrupt handling, memory management](#)

Jump to...

[\(Random Quiz - 7 \) Pre-Endsem Quiz ►](#)

**Started on** Thursday, 16 February 2023, 9:00 PM

**State** Finished

**Completed on** Thursday, 16 February 2023, 10:43 PM

**Time taken** 1 hour 43 mins

**Grade** 13.00 out of 15.00 (86.68%)

Question 1

Correct

Mark 1.00 out of 1.00

Which of the following statements is false ?

Select one:

- a. A process scheduling algorithm is preemptive if the CPU can be forcibly removed from a process.
- b. Time sharing systems generally use preemptive CPU scheduling.
- c. Real time systems generally use non preemptive CPU scheduling. ✓
- d. Response time is more predictable in preemptive systems than in non preemptive systems.

Your answer is correct.

The correct answer is: Real time systems generally use non preemptive CPU scheduling.

**Question 2**

Correct

Mark 1.00 out of 1.00

Order the sequence of events, in scheduling process P1 after process P0

timer interrupt occurs

 2 ✓

Process P1 is running

 6 ✓

context of P0 is saved in P0's PCB

 3 ✓

Control is passed to P1

 5 ✓

context of P1 is loaded from P1's PCB

 4 ✓

Process P0 is running

 1 ✓

Your answer is correct.

The correct answer is: timer interrupt occurs → 2, Process P1 is running → 6, context of P0 is saved in P0's PCB → 3, Control is passed to P1 → 5, context of P1 is loaded from P1's PCB → 4, Process P0 is running → 1

**Question 3**

Correct

Mark 1.00 out of 1.00

Select the compiler's view of the process's address space, for each of the following MMU schemes:

(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Paging

 one continuous chunk ✓

Relocation + Limit

 one continuous chunk ✓

Segmentation

 many continuous chunks of variable size ✓

Segmentation, then paging

 many continuous chunks of variable size ✓

Your answer is correct.

The correct answer is: Paging → one continuous chunk, Relocation + Limit → one continuous chunk, Segmentation → many continuous chunks of variable size, Segmentation, then paging → many continuous chunks of variable size

Question 4

Correct

Mark 1.00 out of 1.00

Match the names of PCB structures with kernel

- |       |                    |   |
|-------|--------------------|---|
| linux | struct task_struct | ✓ |
| xv6   | struct proc        | ✓ |

The correct answer is: linux → struct task\_struct, xv6 → struct proc

Question 5

Correct

Mark 1.00 out of 1.00

Select the state that is not possible after the given state, for a process:

- |          |               |   |
|----------|---------------|---|
| New:     | Running       | ✓ |
| Ready :  | Waiting       | ✓ |
| Running: | None of these | ✓ |
| Waiting: | Running       | ✓ |

Question 6

Partially correct

Mark 0.47 out of 1.00

Select all the correct statements about zombie processes

Select one or more:

- a. A zombie process occupies space in OS data structures
- b. init() typically keeps calling wait() for zombie processes to get cleaned up ✓
- c. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent ✓
- d. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it ✓
- e. Zombie processes are harmless even if OS is up for long time ✗
- f. A process can become zombie if it finishes, but the parent has finished before it ✓
- g. A process becomes zombie when its parent finishes
- h. A zombie process remains zombie forever, as there is no way to clean it up

Your answer is partially correct.

You have correctly selected 4.

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

## Question 7

Partially correct

Mark 0.86 out of 1.00

Mark True/False

Statements about scheduling and scheduling algorithms

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	xv6 code does not care about Processor Affinity
<input type="radio"/>	<input checked="" type="radio"/>	Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.
<input checked="" type="radio"/>	<input type="radio"/>	Response time will be quite poor on non-interruptible kernels
<input checked="" type="radio"/>	<input type="radio"/>	Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.
<input checked="" type="radio"/>	<input type="radio"/>	Processor Affinity refers to memory accesses of a process being stored on cache of that processor
<input checked="" type="radio"/>	<input type="radio"/>	A scheduling algorithm is non-preemptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.
<input type="radio"/>	<input checked="" type="radio"/>	On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread  It's the negation of this. Time NOT spent in idle thread.

xv6 code does not care about Processor Affinity: True

Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.: True

Response time will be quite poor on non-interruptible kernels: True

Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.: True

Processor Affinity refers to memory accesses of a process being stored on cache of that processor: True

A scheduling algorithm is non-preemptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.: True

On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread: False

**Question 8**

Correct

Mark 1.00 out of 1.00

Mark whether the concept is related to scheduling or not.

Yes	No	
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	file-table
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	context-switch
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	timer interrupt
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	ready-queue
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	runnable process

file-table: No

context-switch: Yes

timer interrupt: Yes

ready-queue: Yes

runnable process: Yes

**Question 9**

Partially correct

Mark 0.50 out of 1.00

Select all the correct statements about signals

Select one or more:

- a. Signals are delivered to a process by another process
- b. The signal handler code runs in user mode of CPU
- c. A signal handler can be invoked asynchronously or synchronously depending on signal type ✓
- d. SIGKILL definitely kills a process because it can't be caught or ignored, and its default action terminates the process ✓
- e. Signal handlers once replaced can't be restored
- f. The signal handler code runs in kernel mode of CPU ✗
- g. Signals are delivered to a process by kernel ✓
- h. SIGKILL definitely kills a process because its code runs in kernel mode of CPU

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: Signals are delivered to a process by kernel, A signal handler can be invoked asynchronously or synchronously depending on signal type, The signal handler code runs in user mode of CPU, SIGKILL definitely kills a process because it can't be caught or ignored, and its default action terminates the process

**Question 10**

Correct

Mark 1.00 out of 1.00

Map each signal with its meaning

SIGPIPE	Broken Pipe	✓
SIGCHLD	Child Stopped or Terminated	✓
SIGUSR1	User Defined Signal	✓
SIGALRM	Timer Signal from alarm()	✓
SIGSEGV	Invalid Memory Reference	✓

The correct answer is: SIGPIPE → Broken Pipe, SIGCHLD → Child Stopped or Terminated, SIGUSR1 → User Defined Signal, SIGALRM → Timer Signal from alarm(), SIGSEGV → Invalid Memory Reference

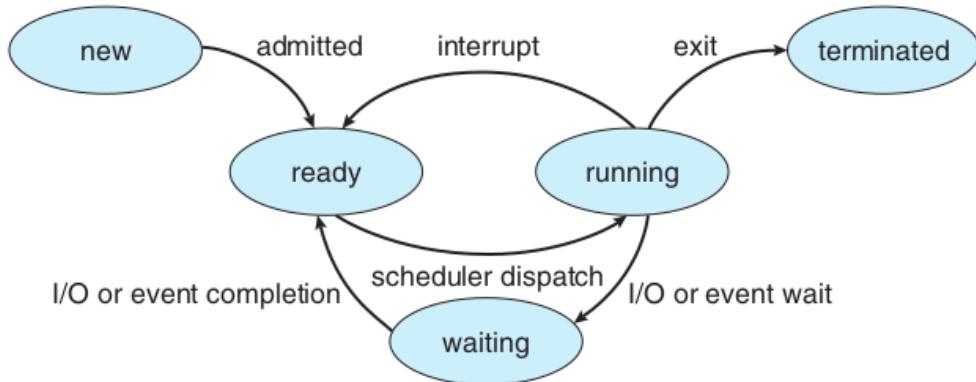
**Question 11**

Partially correct

Mark 0.80 out of 1.00

Mark statements True/False w.r.t. change of states of a process. Note that a statement is true only if the claim and argument both are true.

Reference: The process state diagram (and your understanding of how kernel code works). Note - the diagram does not show zombie state!



**Figure 3.2** Diagram of process state.

True	False	
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Every forked process has to go through ZOMBIE state, at least for a small duration.
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	Only a process in READY state is considered by scheduler
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/>	A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/>	A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.

Every forked process has to go through ZOMBIE state, at least for a small duration.: True

Only a process in READY state is considered by scheduler: True

A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet: True

A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first: False

A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False

Question **12**

Correct

Mark 1.00 out of 1.00

Which of the following parts of a C program do not have any corresponding machine code ?

- a. pointer dereference
- b. function calls
- c. global variables ✓
- d. local variable declaration
- e. #directives ✓
- f. typedefs ✓
- g. expressions

Your answer is correct.

The correct answers are: #directives, typedefs, global variables

Question **13**

Partially correct

Mark 1.38 out of 2.00

Select all the correct statements about the state of a process.

- a. A process in ready state is ready to be scheduled ✓
- b. A waiting process starts running after the wait is over
- c. A process changes from running to ready state on a timer interrupt or any I/O wait
- d. A process waiting for I/O completion is typically woken up by the particular interrupt handler code ✓
- e. Processes in the ready queue are in the ready state ✓
- f. A process changes from running to ready state on a timer interrupt
- g. It is not maintained in the data structures by kernel, it is only for conceptual understanding of programmers
- h. A process waiting for any condition is woken up by another process only ✗
- i. A running process may terminate, or go to wait or become ready again ✓
- j. A process can self-terminate only when it's running ✓
- k. Typically, it's represented as a number in the PCB ✓
- l. A process that is running is not on the ready queue ✓
- m. Changing from running state to waiting state results in "giving up the CPU" ✓
- n. A process in ready state is ready to receive interrupts

Your answer is partially correct.

You have correctly selected 8.

The correct answers are: Typically, it's represented as a number in the PCB, A process in ready state is ready to be scheduled, Processes in the ready queue are in the ready state, A process that is running is not on the ready queue, A running process may terminate, or go to wait or become ready again, A process changes from running to ready state on a timer interrupt, Changing from running state to waiting state results in "giving up the CPU", A process can self-terminate only when it's running, A process waiting for I/O completion is typically woken up by the particular interrupt handler code

Question **14**

Correct

Mark 1.00 out of 1.00

Which of the following are NOT a part of job of a typical compiler?

- a. Invoke the linker to link the function calls with their code, extern globals with their declaration
- b. Check the program for logical errors ✓
- c. Convert high level language code to machine code
- d. Check the program for syntactical errors
- e. Suggest alternative pieces of code that can be written ✓
- f. Process the # directives in a C program

Your answer is correct.

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

◀ Random Quiz - 3 (processes, memory management, event driven kernel), compilation-linking-loading, ipc-pipes

Jump to...

Random Quiz - 5: xv6 make, bootloader, interrupt handling, memory management ►

[Dashboard](#) / [My courses](#) / [Computer Engineering & IT](#) / [CEIT-Even-sem-21-22](#) / [OS-even-sem-21-22](#) / [7 February - 13 February](#)

/ [Quiz-1: 10 AM](#)

**Started on** Saturday, 12 February 2022, 10:00:21 AM

**State** Finished

**Completed on** Saturday, 12 February 2022, 11:25:53 AM

**Time taken** 1 hour 25 mins

**Grade** 4.94 out of 10.00 (49%)

**Question 1**

Complete

Mark 0.00 out of 0.50

Select all the correct statements about code of bootmain() in xv6

```

void
bootmain(void)
{
    struct elfhdr *elf;
    struct proghdr *ph, *eph;
    void (*entry)(void);
    uchar* pa;

    elf = (struct elfhdr*)0x10000; // scratch space

    // Read 1st page off disk
    readseg((uchar*)elf, 4096, 0);

    // Is this an ELF executable?
    if(elf->magic != ELF_MAGIC)
        return; // let bootasm.S handle error

    // Load each program segment (ignores ph flags).
    ph = (struct proghdr*)((uchar*)elf + elf->phoff);
    eph = ph + elf->phnum;
    for(; ph < eph; ph++){
        pa = (uchar*)ph->paddr;
        readseg(pa, ph->filesz, ph->off);
        if(ph->memsz > ph->filesz)
            stobs(pa + ph->filesz, 0, ph->memsz - ph->filesz);
    }

    // Call the entry point from the ELF header.
    // Does not return!
    entry = (void(*)(void))(elf->entry);
    entry();
}

```

Also, inspect the relevant parts of the xv6 code. binary files, etc and run commands as you deem fit to answer this question.

- a. The elf->entry is set by the linker in the kernel file and it's 0x80000000
- b. The condition if(ph->memsz > ph->filesz) is never true.
- c. The readseg finally invokes the disk I/O code using assembly instructions
- d. The elf->entry is set by the linker in the kernel file and it's 0x80000000
- e. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded
- f. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory.
- g. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it.
- h. The elf->entry is set by the linker in the kernel file and it's 8010000c
- i. The kernel file gets loaded at the Physical address 0x10000 in memory.
- j. The stobs() is used here, to fill in some space in memory with zeroes
- k. The kernel file has only two program headers

The correct answers are: The kernel file gets loaded at the Physical address 0x10000 in memory., The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it., The elf->entry is set by the linker in the kernel file and it's 8010000c, The readseg finally invokes the disk I/O code using assembly instructions, The stosb() is used here, to fill in some space in memory with zeroes, The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded, The kernel file has only two program headers

**Question 2**

Complete

Mark 0.50 out of 0.50

What's the trapframe in xv6?

- a. A frame of memory that contains all the trap handler's addresses
- b. A frame of memory that contains all the trap handler code's function pointers
- c. The IDT table
- d. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S
- e. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by code in trapasm.S only
- f. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware only
- g. A frame of memory that contains all the trap handler code

The correct answer is: The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S

**Question 3**

Complete

Mark 0.21 out of 0.50

Order the events that occur on a timer interrupt:

- |   |   |
|---|---|
| Jump to a code pointed by IDT                       | 3 |
| Jump to scheduler code                              | 2 |
| Select another process for execution                | 5 |
| Change to kernel stack of currently running process | 4 |
| Set the context of the new process                  | 6 |
| Save the context of the currently running process   | 1 |
| Execute the code of the new process                 | 7 |

The correct answer is: Jump to a code pointed by IDT → 2, Jump to scheduler code → 4, Select another process for execution → 5, Change to kernel stack of currently running process → 1, Set the context of the new process → 6, Save the context of the currently running process → 3, Execute the code of the new process → 7

**Question 4**

Complete

Mark 0.50 out of 0.50

Suppose a program does a scanf() call.

Essentially the scanf does a read() system call.

This call will obviously "block" waiting for the user input.

In terms of OS data structures and execution of code, what does it mean?

Select one:

- a. OS code for read() will move PCB of current process to a wait queue and call scheduler
- b. OS code for read() will call scheduler
- c. OS code for read() will move the PCB of this process to a wait queue and return from the system call
- d. read() will return and process will be taken to a wait queue
- e. read() returns and process calls scheduler()

The correct answer is: OS code for read() will move PCB of current process to a wait queue and call scheduler

**Question 5**

Complete

Mark 0.50 out of 0.50

In bootasm.S, on the line

```
ljmp    $(SEG_KCODE<<3), $start32
```

The SEG\_KCODE << 3, that is shifting of 1 by 3 bits is done because

- a. The value 8 is stored in code segment
- b. The code segment is 16 bit and only upper 13 bits are used for segment number
- c. While indexing the GDT using CS, the value in CS is always divided by 8
- d. The ljmp instruction does a divide by 8 on the first argument
- e. The code segment is 16 bit and only lower 13 bits are used for segment number

The correct answer is: The code segment is 16 bit and only upper 13 bits are used for segment number

**Question 6**

Complete

Mark 0.40 out of 0.50

Select Yes if the mentioned element should be a part of PCB

Select No otherwise.

**Yes      No**

Memory management information about that process

Process context

Function pointers to all system calls

PID

EIP at the time of context switch

List of opened files

PID of Init

Process state

Pointer to the parent process

Pointer to IDT

Memory management information about that process: Yes

Process context: Yes

Function pointers to all system calls: No

PID: Yes

EIP at the time of context switch: Yes

List of opened files: Yes

PID of Init: No

Process state: Yes

Pointer to the parent process: Yes

Pointer to IDT: No

**Question 7**

Complete

Mark 0.40 out of 1.00

Mark the statements, w.r.t. the scheduler of xv6 as True or False

**True      False**

The work of selecting and scheduling a process is done only in `scheduler()` and not in `sched()`

`swtch` is a function that saves old context, loads new context, and returns to last EIP in the new context

The variable `c->scheduler` on first processor uses the stack allocated entry.S

`sched()` and `scheduler()` are co-routines

The function `scheduler()` executes using the kernel-only stack

When a process is scheduled for execution, it resumes execution in `sched()` after the call to `swtch()`

`swtch` is a function that does not return to the caller

the control returns to `mycpu()->intena = intena; ()`; after `swtch(&p->context, mycpu()->scheduler);` in `sched()`

the control returns to `switchkvm();` after `swtch(&(c->scheduler), p->context);` in `scheduler()`

`sched()` calls `scheduler()` and `scheduler()` calls `sched()`

The work of selecting and scheduling a process is done only in `scheduler()` and not in `sched()`: True

`swtch` is a function that saves old context, loads new context, and returns to last EIP in the new context: True

The variable `c->scheduler` on first processor uses the stack allocated entry.S: True

`sched()` and `scheduler()` are co-routines: True

The function `scheduler()` executes using the kernel-only stack: True

When a process is scheduled for execution, it resumes execution in `sched()` after the call to `swtch()`

: True

`swtch` is a function that does not return to the caller: True

the control returns to `mycpu()->intena = intena; ()`; after `swtch(&p->context, mycpu()->scheduler);` in `sched()`:

False

the control returns to `switchkvm();` after `swtch(&(c->scheduler), p->context);` in `scheduler()`: False

`sched()` calls `scheduler()` and `scheduler()` calls `sched()`: False

**Question 8**

Complete

Mark 0.00 out of 0.50

Consider the following programs

**exec1.c**

```
#include <unistd.h>
#include <stdio.h>
int main() {
    exec("./exec2", "./exec2", NULL);
}
```

**exec2.c**

```
#include <unistd.h>
#include <stdio.h>
int main() {
    exec("/bin/ls", "/bin/ls", NULL);
    printf("hello\n");
}
```

Compiled as

```
cc  exec1.c -o exec1
cc  exec2.c -o exec2
```

And run as

```
$ ./exec1
```

Explain the output of the above command (./exec1)

Assume that /bin/ls , i.e. the 'ls' program exists.

Select one:

- a. "ls" runs on current directory
- b. Execution fails as the call to execl() in exec1 fails
- c. Execution fails as one exec can't invoke another exec
- d. Program prints hello
- e. Execution fails as the call to execl() in exec2 fails

The correct answer is: "ls" runs on current directory

**Question 9**

Complete

Mark 0.00 out of 0.50

For each line of code mentioned on the left side, select the location of sp/esp that is in use

**cli****in bootasm.S**

0x7c00 to 0

**readseg((uchar\*)elf, 4096, 0);****in bootmain.c**

Immaterial as the stack is not used here

**ljmp \$(SEG\_KCODE<<3), \$start32****in bootasm.S**

0x10000 to 0x7c00

**call bootmain****in bootasm.S**

The 4KB area in kernel image, loaded in memory, named as 'stack'

**jmp \*%eax****in entry.S**

0x7c00 to 0x10000

The correct answer is: **cli**

**in bootasm.S** → Immateral as the stack is not used here, **readseg((uchar\*)elf, 4096, 0);**

**in bootmain.c** → 0x7c00 to 0, **ljmp \$(SEG\_KCODE<<3), \$start32**

**in bootasm.S** → Immateral as the stack is not used here, **call bootmain**

**in bootasm.S** → 0x7c00 to 0, **jmp \*%eax**

**in entry.S** → The 4KB area in kernel image, loaded in memory, named as 'stack'

**Question 10**

Complete

Mark 0.63 out of 1.00

Select the correct statements about interrupt handling in xv6 code

- a. xv6 uses the 64th entry in IDT for system calls
- b. xv6 uses the 0x64th entry in IDT for system calls
- c. On any interrupt/syscall/exception the control first jumps in vectors.S
- d. The function trap() is called irrespective of hardware interrupt/system-call/exception
- e. Before going to altraps, the kernel stack contains upto 5 entries.
- f. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt
- g. The trapframe pointer in struct proc, points to a location on kernel stack
- h. The function trap() is called only in case of hardware interrupt
- i. The trapframe pointer in struct proc, points to a location on user stack
- j. On any interrupt/syscall/exception the control first jumps in trapasm.S
- k. The CS and EIP are changed only after pushing user code's SS,ESP on stack
- l. The CS and EIP are changed only immediately on a hardware interrupt
- m. All the 256 entries in the IDT are filled

The correct answers are: All the 256 entries in the IDT are filled, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in vectors.S, Before going to altraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on kernel stack, The function trap() is called irrespective of hardware interrupt/system-call/exception, The CS and EIP are changed only after pushing user code's SS,ESP on stack

**Question 11**

Complete

Mark 0.50 out of 0.50

Order the sequence of events, in scheduling process P1 after process P0

timer interrupt occurs

2

Process P0 is running

1

context of P1 is loaded from P1's PCB

4

Process P1 is running

6

Control is passed to P1

5

context of P0 is saved in P0's PCB

3

The correct answer is: timer interrupt occurs → 2, Process P0 is running → 1, context of P1 is loaded from P1's PCB → 4, Process P1 is running → 6, Control is passed to P1 → 5, context of P0 is saved in P0's PCB → 3

**Question 12**

Complete

Mark 0.00 out of 1.00

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

(Note: non-interruptible kernel code means, if the kernel code is executing, then interrupts will be disabled).

Note: A possible sequence may have some missing steps in between. An impossible sequence will have n and n+1th steps such that n+1th step can not follow n'th step.

Select one or more:

a. P1 running

P1 makes system call  
timer interrupt  
Scheduler  
P2 running  
timer interrupt  
Scheduler  
P1 running  
P1's system call return

b. P1 running

P1 makes system call  
system call returns  
P1 running  
timer interrupt  
Scheduler running  
P2 running

c. P1 running

P1 makes system call and blocks  
Scheduler  
P2 running  
P2 makes system call and blocks  
Scheduler  
P3 running  
Hardware interrupt  
Interrupt unblocks P1  
Interrupt returns  
P3 running  
Timer interrupt  
Scheduler  
P1 running

d. P1 running

keyboard hardware interrupt  
keyboard interrupt handler running  
interrupt handler returns  
P1 running  
P1 makes system call  
system call returns  
P1 running  
timer interrupt  
scheduler  
P2 running

e.

P1 running  
P1 makes system call  
Scheduler  
P2 running  
P2 makes system call and blocks  
Scheduler  
P1 running again

f. P1 running

P1 makes system call and blocks  
 Scheduler  
 P2 running  
 P2 makes system call and blocks  
 Scheduler  
 P1 running again

The correct answers are: P1 running

P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again, P1 running

P1 makes system call

timer interrupt

Scheduler

P2 running

timer interrupt

Scheuler

P1 running

P1's system call return,

P1 running

P1 makes system call

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again

### Question 13

Complete

Mark 0.50 out of 0.50

Some part of the bootloader of xv6 is written in assembly while some part is written in C. Why is that so?

Select all the appropriate choices

- a. The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C
- b. The setting up of the most essential memory management infrastructure needs assembly code
- c. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time
- d. The code for reading ELF file can not be written in assembly

The correct answers are: The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C, The setting up of the most essential memory management infrastructure needs assembly code

**Question 14**

Complete

Mark 0.17 out of 0.50

The bootmain() function has this code

```
elf = (struct elfhdr*)0x10000; // scratch space  
readseg((uchar*)elf, 4096, 0);
```

Mark the statements as True or False with respect to this code.

In these statements 0x1000 is referred to as ADDRESS

**True      False**

- The value ADDRESS is changed to a 0 the program could still work
- If the value of ADDRESS is changed to a higher number (upto a limit), the program could still work
- This line loads the kernel code at ADDRESS
- If the value of ADDRESS is changed to a lower number (upto a limit), the program could still work
- This line effectively loads the ELF header and the program headers at ADDRESS
- If the value of ADDRESS is changed, then the program will not work

The value ADDRESS is changed to a 0 the program could still work: False

If the value of ADDRESS is changed to a higher number (upto a limit), the program could still work: True

This line loads the kernel code at ADDRESS: False

If the value of ADDRESS is changed to a lower number (upto a limit), the program could still work: True

This line effectively loads the ELF header and the program headers at ADDRESS: False

If the value of ADDRESS is changed, then the program will not work: False

**Question 15**

Complete

Mark 0.50 out of 1.00

Which parts of the xv6 code in bootasm.S bootmain.c , entry.S and in the codepath related to scheduler() and trap handling() can also be written in some other way, and still ensure that xv6 works properly?

Writing code is not necessary. You only need to comment on which part of the code could be changed to something else or written in another fashion.

Maximum two points to be written.

We can use a scheduling algorithm. We can use the kernel stack in scheduler function in entry.S and bootmain.c .

**Question 16**

Complete

Mark 0.13 out of 0.50

Select all the correct statements about zombie processes

Select one or more:

- a. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent
- b. Zombie processes are harmless even if OS is up for long time
- c. A zombie process occupies space in OS data structures
- d. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it
- e. A process can become zombie if it finishes, but the parent has finished before it
- f. init() typically keeps calling wait() for zombie processes to get cleaned up
- g. A process becomes zombie when its parent finishes
- h. A zombie process remains zombie forever, as there is no way to clean it up

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

◀ Extra Reading on Linkers: A writeup by Ian Taylor (keep changing url string from 38 to 39, and so on)

Jump to...



**Started on** Saturday, 20 February 2021, 2:51 PM

**State** Finished

**Completed on** Saturday, 20 February 2021, 3:55 PM

**Time taken** 1 hour 3 mins

**Grade** 7.30 out of 20.00 (37%)

#### Question 1

Partially correct

Mark 0.80 out of 1.00

Select all the correct statements about the state of a process.

- a. A process can self-terminate only when it's running ✓
- b. Typically, it's represented as a number in the PCB ✓
- c. A process that is running is not on the ready queue ✓
- d. Processes in the ready queue are in the ready state ✓
- e. It is not maintained in the data structures by kernel, it is only for conceptual understanding of programmers
- f. Changing from running state to waiting state results in "giving up the CPU" ✓
- g. A process in ready state is ready to receive interrupts
- h. A waiting process starts running after the wait is over ✗
- i. A process changes from running to ready state on a timer interrupt ✓
- j. A process in ready state is ready to be scheduled ✓
- k. A running process may terminate, or go to wait or become ready again ✓
- l. A process waiting for I/O completion is typically woken up by the particular interrupt handler code ✓
- m. A process waiting for any condition is woken up by another process only
- n. A process changes from running to ready state on a timer interrupt or any I/O wait

Your answer is partially correct.

You have selected too many options.

The correct answers are: Typically, it's represented as a number in the PCB, A process in ready state is ready to be scheduled, Processes in the ready queue are in the ready state, A process that is running is not on the ready queue, A running process may terminate, or go to wait or become ready again, A process changes from running to ready state on a timer interrupt, Changing from running state to waiting state results in "giving up the CPU", A process can self-terminate only when it's running, A process waiting for I/O completion is typically woken up by the particular interrupt handler code

**Question 2**

Incorrect

Mark 0.00 out of 1.00

For each line of code mentioned on the left side, select the location of sp/esp that is in use

`jmp *%eax`  
in entry.S

0x7c00 to 0x10000



`ljmp $(SEG_KCODE<<3), $start32`  
in bootasm.S

0x10000 to 0x7c00



`call bootmain`  
in bootasm.S

0x7c00 to 0x10000



`cli`  
in bootasm.S

0x7c00 to 0



`readseg((uchar*)elf, 4096, 0);`  
in bootmain.c

The 4KB area in kernel image, loaded in memory, named as 'stack'



Your answer is incorrect.

The correct answer is: `jmp *%eax`

`in entry.S` → The 4KB area in kernel image, loaded in memory, named as 'stack', `ljmp $(SEG_KCODE<<3), $start32`

`in bootasm.S` → Immateriel as the stack is not used here, `call bootmain`

`in bootasm.S` → 0x7c00 to 0, `cli`

`in bootasm.S` → Immateriel as the stack is not used here, `readseg((uchar*)elf, 4096, 0);`

`in bootmain.c` → 0x7c00 to 0

**Question 3**

Correct

Mark 0.25 out of 0.25

Order the following events in boot process (from 1 onwards)

Boot loader	2	✓
Shell	6	✓
BIOS	1	✓
OS	3	✓
Init	4	✓
Login interface	5	✓

Your answer is correct.

The correct answer is: Boot loader → 2, Shell → 6, BIOS → 1, OS → 3, Init → 4, Login interface → 5

**Question 4**

Partially correct

Mark 0.30 out of 0.50

Consider the following command and its output:

```
$ ls -lht xv6.img kernel
-rw-rw-r-- 1 abhijit abhijit 4.9M Feb 15 11:09 xv6.img
-rwxrwxr-x 1 abhijit abhijit 209K Feb 15 11:09 kernel*
```

Following code in bootmain()

```
readseg((uchar*)elf, 4096, 0);
```

and following selected lines from Makefile

```
xv6.img: bootblock kernel
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

```
kernel: $(OBJS) entry.o entryother initcode kernel.ld
$(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother
$(OBJDUMP) -S kernel > kernel.asm
$(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

Also read the code of bootmain() in xv6 kernel.

Select the options that describe the meaning of these lines and their correlation.

- a. Although the size of the kernel file is 209 Kb, only 4Kb out of it is the actual kernel code and remaining part is all zeroes.
- b. The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files ✓
- c. The kernel.ld file contains instructions to the linker to link the kernel properly ✓
- d. The bootmain() code does not read the kernel completely in memory
- e. readseg() reads first 4k bytes of kernel in memory
- f. Although the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.
- g. The kernel.asm file is the final kernel file
- h. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is not read as it is user programs.
- i. The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(). ✓

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: The kernel disk image is ~5MB, the kernel within it is 209 kb, but bootmain() initially reads only first 4kb, and the later part is read using program headers in bootmain(), readseg() reads first 4k bytes of kernel in memory, The kernel is compiled by linking multiple .o files created from .c files; and the entry.o, initcode, entryother files, The kernel.ld file contains instructions to the linker to link the kernel properly, Although the size of the xv6.img file is ~5MB, only some part out of it is the bootloader+kernel code and remaining part is all zeroes.

**Question 5**

Partially correct

Mark 0.50 out of 1.00

```
int f() {  
    int count;  
    for (count = 0; count < 2; count++) {  
        if (fork() == 0)  
            printf("Operating-System\\n");  
    }  
    printf("TYCOMP\\n");  
}
```

The number of times "Operating-System" is printed, is:

Answer:

The correct answer is: 7.00

**Question 6**

Partially correct

Mark 0.40 out of 0.50

Select Yes/True if the mentioned element must be a part of PCB

Select No/False otherwise.

Yes	No	
<input checked="" type="radio"/>	<input type="radio"/> X	PID
<input checked="" type="radio"/>	<input type="radio"/> X	Process context
<input checked="" type="radio"/>	<input type="radio"/> X	List of opened files
<input checked="" type="radio"/>	<input type="radio"/> X	Process state
<input type="radio"/> X	<input checked="" type="radio"/>	Parent's PID
<input type="radio"/> X	<input checked="" type="radio"/>	Pointer to IDT
<input type="radio"/> X	<input checked="" type="radio"/>	Function pointers to all system calls
<input checked="" type="radio"/>	<input type="radio"/> X	Memory management information about that process
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Pointer to the parent process
<input checked="" type="radio"/>	<input type="radio"/> X	EIP at the time of context switch

PID: Yes

Process context: Yes

List of opened files: Yes

Process state: Yes

Parent's PID: No

Pointer to IDT: No

Function pointers to all system calls: No

Memory management information about that process: Yes

Pointer to the parent process: Yes

EIP at the time of context switch: Yes

**Question 7**

Incorrect

Mark 0.00 out of 1.00

Select all the correct statements about code of bootmain() in xv6

```

void
bootmain(void)
{
    struct elfhdr *elf;
    struct proghdr *ph, *eph;
    void (*entry)(void);
    uchar* pa;

    elf = (struct elfhdr*)0x10000; // scratch space

    // Read 1st page off disk
    readseg((uchar*)elf, 4096, 0);

    // Is this an ELF executable?
    if(elf->magic != ELF_MAGIC)
        return; // let bootasm.S handle error

    // Load each program segment (ignores ph flags).
    ph = (struct proghdr*)((uchar*)elf + elf->phoff);
    eph = ph + elf->phnum;
    for(; ph < eph; ph++){
        pa = (uchar*)ph->paddr;
        readseg(pa, ph->filesz, ph->off);
        if(ph->memsz > ph->filesz)
            stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
    }

    // Call the entry point from the ELF header.
    // Does not return!
    entry = (void(*)(void))(elf->entry);
    entry();
}

```

Also, inspect the relevant parts of the xv6 code. binary files, etc and run commands as you deem fit to answer this question.

- a. The kernel file gets loaded at the Physical address 0x10000 +0x80000000 in memory. ✗
- b. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✗
- c. The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded ✓
- d. The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it. ✓
- e. The kernel file has only two program headers ✓
- f. The elf->entry is set by the linker in the kernel file and it's 0x80000000 ✗
- g. The readseg finally invokes the disk I/O code using assembly instructions ✓
- h. The elf->entry is set by the linker in the kernel file and it's 8010000c ✓
- i. The kernel file gets loaded at the Physical address 0x10000 in memory. ✓
- j. The condition if(ph->memsz > ph->filesz) is never true. ✗
- k. The stosb() is used here, to fill in some space in memory with zeroes ✓

Your answer is incorrect.

The correct answers are: The kernel file gets loaded at the Physical address 0x10000 in memory., The kernel file in memory is not necessarily a continuously filled in chunk, it may have holes in it., The elf->entry is set by the linker in the kernel file and it's 8010000c, The readseg finally invokes the disk I/O code using assembly instructions, The stosb() is used here, to fill in some space in memory with zeroes, The kernel ELF file contains actual physical address where particular sections of 'kernel' file should be loaded, The kernel file has only two program headers

**Question 8**

Partially correct

Mark 0.13 out of 0.25

Which of the following are NOT a part of job of a typical compiler?

- a. Check the program for logical errors ✓
- b. Convert high level language code to machine code
- c. Process the # directives in a C program
- d. Invoke the linker to link the function calls with their code, extern globals with their declaration
- e. Check the program for syntactical errors
- f. Suggest alternative pieces of code that can be written

Your answer is partially correct.

You have correctly selected 1.

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

**Question 9**

Correct

Mark 0.25 out of 0.25

Rank the following storage systems from slowest (first) to fastest(last)

Cache	6	✓
Hard Disk	3	✓
RAM	5	✓
Optical Disks	2	✓
Non volatile memory	4	✓
Registers	7	✓
Magnetic Tapes	1	✓

Your answer is correct.

The correct answer is: Cache → 6, Hard Disk → 3, RAM → 5, Optical Disks → 2, Non volatile memory → 4, Registers → 7, Magnetic Tapes → 1

**Question 10**

Partially correct

Mark 0.21 out of 0.50

Which of the following parts of a C program do not have any corresponding machine code ?

- a. local variable declaration
- b. global variables
- c. function calls ✗
- d. #directives ✓
- e. expressions
- f. pointer dereference
- g. typedefs ✓

Your answer is partially correct.

You have correctly selected 2.

The correct answers are: #directives, typedefs, global variables

**Question 11**

Correct

Mark 0.25 out of 0.25

Match a system call with it's description

pipe	create an unnamed FIFO storage with 2 ends - one for reading and another for writing	✓
dup	create a copy of the specified file descriptor into smallest available file descriptor	✓
dup2	create a copy of the specified file descriptor into another specified file descriptor	✓
exec	execute a binary file overlaying the image of current process	✓
fork	create an identical child process	✓

Your answer is correct.

The correct answer is: pipe → create an unnamed FIFO storage with 2 ends - one for reading and another for writing, dup → create a copy of the specified file descriptor into smallest available file descriptor, dup2 → create a copy of the specified file descriptor into another specified file descriptor, exec → execute a binary file overlaying the image of current process, fork → create an identical child process

**Question 12**

Correct

Mark 0.25 out of 0.25

Match the register with the segment used with it.

eip	cs	✓
edi	es	✓
esi	ds	✓
ebp	ss	✓
esp	ss	✓

Your answer is correct.

The correct answer is: eip → cs, edi → es, esi → ds, ebp → ss, esp → ss

**Question 13**

Correct

Mark 0.25 out of 0.25

What's the trapframe in xv6?

- a. A frame of memory that contains all the trap handler code
- b. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware only
- c. The IDT table
- d. A frame of memory that contains all the trap handler code's function pointers
- e. A frame of memory that contains all the trap handler's addresses
- f. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S ✓
- g. The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by code in trapasm.S only

Your answer is correct.

The correct answer is: The sequence of values, including saved registers, constructed on the stack when an interrupt occurs, built by hardware + code in trapasm.S

**Question 14**

Incorrect

Mark 0.00 out of 0.50

Select all the correct statements about linking and loading.

Select one or more:

- a. Continuous memory management schemes can support dynamic linking and dynamic loading. ✗
- b. Loader is last stage of the linker program ✗
- c. Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently) ✓
- d. Dynamic linking and loading is not possible without demand paging or demand segmentation. ✓
- e. Dynamic linking essentially results in relocatable code. ✓
- f. Continuous memory management schemes can support static linking and static loading. (may be inefficiently) ✓
- g. Loader is part of the operating system ✓
- h. Static linking leads to non-relocatable code ✗
- i. Dynamic linking is possible with continuous memory management, but variable sized partitions only. ✗

Your answer is incorrect.

The correct answers are: Continuous memory management schemes can support static linking and static loading. (may be inefficiently), Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently), Dynamic linking essentially results in relocatable code., Loader is part of the operating system, Dynamic linking and loading is not possible without demand paging or demand segmentation.

**Question 15**

Incorrect

Mark 0.00 out of 0.25

In bootasm.S, on the line

```
ljmp    $(SEG_KCODE<<3), $start32
```

The SEG\_KCODE << 3, that is shifting of 1 by 3 bits is done because

- a. The value 8 is stored in code segment
- b. The code segment is 16 bit and only upper 13 bits are used for segment number
- c. The code segment is 16 bit and only lower 13 bits are used for segment number ✗
- d. While indexing the GDT using CS, the value in CS is always divided by 8
- e. The ljmp instruction does a divide by 8 on the first argument

Your answer is incorrect.

The correct answer is: The code segment is 16 bit and only upper 13 bits are used for segment number

**Question 16**

Partially correct

Mark 0.07 out of 0.50

Order the events that occur on a timer interrupt:

Change to kernel stack

1	✗
---	---

Jump to a code pointed by IDT

2	✗
---	---

Jump to scheduler code

5	✗
---	---

Set the context of the new process

4	✗
---	---

Save the context of the currently running process

3	✓
---	---

Execute the code of the new process

6	✗
---	---

Select another process for execution

7	✗
---	---

Your answer is partially correct.

You have correctly selected 1.

The correct answer is: Change to kernel stack → 2, Jump to a code pointed by IDT → 1, Jump to scheduler code → 4, Set the context of the new process → 6, Save the context of the currently running process → 3, Execute the code of the new process → 7, Select another process for execution → 5

**Question 17**

Incorrect

Mark 0.00 out of 1.00

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

```
$ ls . /tmp/asdfksdf >/tmp/ddd 2>&1
```

**Program 1**

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(1);
    dup(fd);
    close(2);
    dup(fd);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

**Program 2**

```
int main(int argc, char *argv[]) {
    int fd, n, i;
    char buf[128];

    close(1);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    close(2);
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);
}
```

Select all the correct statements about the programs

Select one or more:

- a. Both programs are correct ✗
- b. Program 2 makes sure that there is one file offset used for '2' and '1' ✗
- c. Only Program 2 is correct ✗
- d. Program 2 does 1>&2 ✗
- e. Program 2 ensures 2>&1 and does not ensure >/tmp/ddd ✗
- f. Program 1 makes sure that there is one file offset used for '2' and '1' ✓
- g. Program 1 is correct for >/tmp/ddd but not for 2>&1 ✗
- h. Program 1 does 1>&2 ✗
- i. Both program 1 and 2 are incorrect ✗
- j. Program 2 is correct for >/tmp/ddd but not for 2>&1 ✗
- k. Only Program 1 is correct ✓
- l. Program 1 ensures 2>&1 and does not ensure >/tmp/ddd ✗

Your answer is incorrect.

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

**Question 18**

Correct

Mark 0.25 out of 0.25

Select the option which best describes what the CPU does during its powered ON lifetime

- a. Ask the user what is to be done, and execute that task
- b. Ask the OS what is to be done, and execute that task
- c. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, Ask the User or the OS what is to be done next, repeat
- d. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per ✓ the instruction itself, repeat
- e. Fetch instruction specified by OS, Decode and execute it, repeat
- f. Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, Ask OS what is to be done next, repeat

The correct answer is: Fetch instructions specified by location given by PC, Decode and Execute it, during execution increment PC or change PC as per the instruction itself, repeat

**Question 19**

Partially correct

Mark 0.86 out of 1.00

Consider the following code and MAP the file to which each fd points at the end of the code.

```
int main(int argc, char *argv[]) {
    int fd1, fd2 = 1, fd3 = 1, fd4 = 1;

    fd1 = open("/tmp/1", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
    fd2 = open("/tmp/2", O_RDONLY);
    fd3 = open("/tmp/3", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
    close(0);
    close(1);
    dup(fd2);
    dup(fd3);
    close(fd3);
    dup2(fd2, fd4);
    printf("%d %d %d %d\n", fd1, fd2, fd3, fd4);
    return 0;
}
```

1	closed	✗
fd4	/tmp/2	✓
fd2	/tmp/2	✓
fd1	/tmp/1	✓
2	stderr	✓
0	/tmp/2	✓
fd3	closed	✓

Your answer is partially correct.

You have correctly selected 6.

The correct answer is: 1 → /tmp/3, fd4 → /tmp/2, fd2 → /tmp/2, fd1 → /tmp/1, 2 → stderr, 0 → /tmp/2, fd3 → closed

**Question 20**

Incorrect

Mark 0.00 out of 2.00

Following code claims to implement the command

/bin/ls -l | /usr/bin/head -3 | /usr/bin/tail -1

Fill in the blanks to make the code work.

Note: Do not include space in writing any option. x[1][2] should be written without any space, and so is the case with [1] or [2]. Pay attention to exact syntax and do not write any extra character like ';' or = etc.

```
int main(int argc, char *argv[]) {
```

```
    int pid1, pid2;
```

```
    int pfd[
```

```
    1
```

```
    x ] [2];
```

```
    pipe(
```

```
    2
```

```
    x );
```

```
    pid1 =
```

```
    3
```

```
    x ;
```

```
    if(pid1 != 0) {
```

```
        close(pfd[0]
```

```
        0
```

```
    x );
```

```
    close(
```

```
    pid1
```

```
    x );
```

```
    dup(
```

```
    pid2
```

```
    x );
```

```
    execl("/bin/ls", "/bin/ls", "
```

```
    1
```

```
    x ", NULL);
```

```
    }
```

```
    pipe(
```

```
    
```

```
    x );
```

```
    
```

```
    x = fork();
```

```
    if(pid2 == 0) {
```

```
        close(
```

```
        
```

```
        x ;
```

```
        close(0);
```

```
        dup(
```

```
        
```

```
        x );
```

```
        close(pfd[1]
```

```
        
```

```
✗ );
close(
  
✗ );
dup(
  
✗ );
execl("/usr/bin/head", "/usr/bin/head", "  
  
✗ ", NULL);
} else {
close(pfd
  
✗ );
close(
  
✗ );
dup(
  
✗ );
close(pfd
  
✗ );
execl("/usr/bin/tail", "/usr/bin/tail", "  
  
✗ ", NULL);
}  
}
```

**Question 21**

Partially correct

Mark 0.11 out of 1.00

Select all the correct statements about calling convention on x86 32-bit.

- a. Return address is one location above the ebp ✓
- b. Parameters may be passed in registers or on stack ✓
- c. Space for local variables is allocated by subtracting the stack pointer inside the code of the called function ✓
- d. The ebp pointers saved on the stack constitute a chain of activation records ✓
- e. The two lines in the beginning of each function, "push %ebp; mov %esp, %ebp", create space for local variables ✗
- f. Parameters may be passed in registers or on stack ✓
- g. The return value is either stored on the stack or returned in the eax register ✗
- h. Parameters are pushed on the stack in left-right order
- i. during execution of a function, ebp is pointing to the old ebp
- j. Space for local variables is allocated by subtracting the stack pointer inside the code of the caller function ✗
- k. Compiler may allocate more memory on stack than needed ✓

Your answer is partially correct.

You have selected too many options.

The correct answers are: Compiler may allocate more memory on stack than needed, Parameters may be passed in registers or on stack, Return address is one location above the ebp, during execution of a function, ebp is pointing to the old ebp, Space for local variables is allocated by subtracting the stack pointer inside the code of the called function, The ebp pointers saved on the stack constitute a chain of activation records

**Question 22**

Correct

Mark 1.00 out of 1.00

Match the program with its output (ignore newlines in the output. Just focus on the count of the number of 'hi')

main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }

hi ✓

main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }

hi hi ✓

main() { int i = NULL; fork(); printf("hi\n"); }

hi hi ✓

main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }

hi ✓

Your answer is correct.

The correct answer is: main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi, main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi hi, main() { int i = NULL; fork(); printf("hi\n"); } → hi hi, main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi

**Question 23**

Incorrect

Mark 0.00 out of 0.50

Some part of the bootloader of xv6 is written in assembly while some part is written in C. Why is that so?

Select all the appropriate choices

- a. The code in assembly is required for transition to protected mode, from real mode; but calling convention was applicable all the time ✗
- b. The setting up of the most essential memory management infrastructure needs assembly code ✓
- c. The code for reading ELF file can not be written in assembly ✗
- d. The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C ✓

Your answer is incorrect.

The correct answers are: The code in assembly is required for transition to protected mode, from real mode; after that calling convention applies, hence code can be written in C, The setting up of the most essential memory management infrastructure needs assembly code

**Question 24**

Incorrect

Mark 0.00 out of 0.50

```
xv6.img: bootblock kernel
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

Consider above lines from the Makefile. Which of the following is incorrect?

- a. The size of the kernel file is nearly 5 MB ✓
- b. The kernel is located at block-1 of the xv6.img ✗
- c. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies 10,000 blocks on the disk. ✗
- d. The size of xv6.img is exactly = (size of bootblock) + (size of kernel) ✗
- e. The bootblock is located on block-0 of the xv6.img ✗
- f. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk. ✓
- g. The bootblock may be 512 bytes or less (looking at the Makefile instruction) ✗
- h. The xv6.img is the virtual disk that is created by combining the bootblock and the kernel file. ✗
- i. The size of the xv6.img is nearly 5 MB ✗
- j. xv6.img is the virtual processor used by the qemu emulator ✓
- k. Blocks in xv6.img after kernel may be all zeroes. ✗

Your answer is incorrect.

The correct answers are: xv6.img is the virtual processor used by the qemu emulator, The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk., The size of the kernel file is nearly 5 MB, The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

**Question 25**

Incorrect

Mark 0.00 out of 1.00

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

Select one or more:

a. P1 running

P1 makes system call

timer interrupt

Scheduler

P2 running

timer interrupt

Scheuler

P1 running

P1's system call return

b. P1 running

P1 makes sytem call and blocks

Scheduler

P2 running

P2 makes sytem call and blocks

Scheduler

P1 running again



c. P1 running

P1 makes system call

system call returns

P1 running

timer interrupt

Scheduler running

P2 running

d. P1 running

P1 makes sytem call and blocks

Scheduler

P2 running

P2 makes sytem call and blocks

Scheduler

P3 running

Hardware interrupt

Interrupt unblocks P1

Interrupt returns

P3 running

Timer interrupt

Scheduler

P1 running



e.

P1 running

P1 makes sytem call

Scheduler

P2 running

P2 makes sytem call and blocks

Scheduler

P1 running again

f. P1 running

keyboard hardware interrupt

keyboard interrupt handler running

interrupt handler returns

P1 running

P1 makes sytem call

system call returns



P1 running  
timer interrupt  
scheduler  
P2 running

Your answer is incorrect.

The correct answers are: P1 running

P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again, P1 running

P1 makes system call

timer interrupt

Scheduler

P2 running

timer interrupt

Scheuler

P1 running

P1's system call return,

P1 running

P1 makes system call

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again

**Question 26**

Correct

Mark 0.25 out of 0.25

Which of the following are the files related to bootloader in xv6?

- a. bootasm.s and entry.S
- b. bootasm.S and bootmain.c ✓
- c. bootasm.S, bootmain.c and bootblock.c
- d. bootmain.c and bootblock.S

Your answer is correct.

The correct answer is: bootasm.S and bootmain.c

**Question 27**

Correct

Mark 0.25 out of 0.25

Match the following parts of a C program to the layout of the process in memory

Instructions	Text section	✓
Local Variables	Stack Section	✓
Dynamically allocated memory	Heap Section	✓
Global and static data	Data section	✓

Your answer is correct.

The correct answer is:

Instructions → Text section, Local Variables → Stack Section,  
 Dynamically allocated memory → Heap Section,  
 Global and static data → Data section

**Question 28**

Incorrect

Mark 0.00 out of 0.50

What will this program do?

```
int main() {
    fork();
    execl("/bin/ls", "/bin/ls", NULL);
    printf("hello");
}
```

- a. one process will run ls, another will print hello
- b. run ls once ✗
- c. run ls twice
- d. run ls twice and print hello twice
- e. run ls twice and print hello twice, but output will appear in some random order

Your answer is incorrect.

The correct answer is: run ls twice

**Question 29**

Correct

Mark 0.25 out of 0.25

What is the OS Kernel?

- a. The code that controls hardware, abstracts access to hardware resources using system calls, creates an environment for processes to be created and run ✓ correct
- b. The set of tools like compiler, linker, loader, terminal, shell, etc.
- c. Only the system programs like compiler, linker, loader, etc.
- d. Everything that I see on my screen

The correct answer is: The code that controls hardware, abstracts access to hardware resources using system calls, creates an environment for processes to be created and run

**Question 30**

Correct

Mark 0.50 out of 0.50

Which of the following is/are not saved during context switch?

- a. Program Counter
- b. General Purpose Registers
- c. Bus ✓
- d. Stack Pointer
- e. MMU related registers/information
- f. Cache ✓
- g. TLB ✓

Your answer is correct.

The correct answers are: TLB, Cache, Bus

**Question 31**

Partially correct

Mark 0.10 out of 0.25

Select the order in which the various stages of a compiler execute.

Linking	3	
Syntactical Analysis	2	
Pre-processing	1	
Intermediate code generation	does not exist	
Loading	4	

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: Linking → 4, Syntactical Analysis → 2, Pre-processing → 1, Intermediate code generation → 3, Loading → does not exist

**Question 32**

Partially correct

Mark 0.08 out of 0.50

Order the sequence of events, in scheduling process P1 after process P0

context of P0 is saved in P0's PCB	2	
context of P1 is loaded from P1's PCB	3	
Process P1 is running	5	
timer interrupt occurs	6	
Process P0 is running	1	
Control is passed to P1	4	

Your answer is partially correct.

You have correctly selected 1.

The correct answer is: context of P0 is saved in P0's PCB → 3, context of P1 is loaded from P1's PCB → 4, Process P1 is running → 6, timer interrupt occurs → 2, Process P0 is running → 1, Control is passed to P1 → 5

**Question 33**

Not answered

Marked out of 1.00

Select the correct statements about interrupt handling in xv6 code

- a. On any interrupt/syscall/exception the control first jumps in vectors.S
- b. The trapframe pointer in struct proc, points to a location on user stack
- c. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt
- d. xv6 uses the 64th entry in IDT for system calls
- e. The CS and EIP are changed only after pushing user code's SS,ESP on stack
- f. The trapframe pointer in struct proc, points to a location on kernel stack
- g. The function trap() is called only in case of hardware interrupt
- h. The CS and EIP are changed only immediately on a hardware interrupt
- i. All the 256 entries in the IDT are filled
  
- j. On any interrupt/syscall/exception the control first jumps in trapasm.S
- k. The function trap() is called irrespective of hardware interrupt/system-call/exception
- l. xv6 uses the 0x64th entry in IDT for system calls
- m. Before going to alltraps, the kernel stack contains upto 5 entries.

Your answer is incorrect.

The correct answers are: All the 256 entries in the IDT are filled, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in trapasm.S, Before going to alltraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on kernel stack, The function trap() is called irrespective of hardware interrupt/system-call/exception, The CS and EIP are changed only after pushing user code's SS,ESP on stack

[◀ \(Assignment\) Change free list management in xv6](#)

Jump to...

**Started on** Tuesday, 22 March 2022, 1:59:34 PM

**State** Finished

**Completed on** Tuesday, 22 March 2022, 4:14:53 PM

**Time taken** 2 hours 15 mins

**Grade** 24.57 out of 40.00 (61%)

**Question 1**

Partially correct

Mark 0.10 out of 1.00

Select all the correct statements w.r.t user and kernel threads

Select one or more:

a. many-one model can be implemented even if there are no kernel threads ✓

b. many-one model gives no speedup on multicore processors

c. all three models, that is many-one, one-one, many-many , require a user level thread library ✓

d. A process blocks in many-one model even if a single thread makes a blocking system call

e. A process may not block in many-one model, if a thread makes a blocking system call ✗

f. one-one model increases kernel's scheduling load ✓

g. one-one model can be implemented even if there are no kernel threads

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: many-one model can be implemented even if there are no kernel threads, all three models, that is many-one, one-one, many-many , require a user level thread library, one-one model increases kernel's scheduling load, many-one model gives no speedup on multicore processors, A process blocks in many-one model even if a single thread makes a blocking system call

**Question 2**

Partially correct

Mark 0.50 out of 1.00

Select all correct statements w.r.t. Major and Minor page faults on Linux

- a. Thrashing is possible only due to major page faults
- b. Minor page fault may occur because the page was freed, but still tagged and available in the free page list
- c. Minor page fault may occur because of a page fault during fork(), on code of an already running process ✓
- d. Major page faults are likely to occur in more numbers at the beginning of the process ✓
- e. Minor page fault may occur because the page was a shared memory page ✓
- f. Minor page faults are an improvement of the page buffering techniques

The correct answers are: Minor page fault may occur because the page was a shared memory page, Minor page fault may occur because of a page fault during fork(), on code of an already running process, Minor page fault may occur because the page was freed, but still tagged and available in the free page list, Major page faults are likely to occur in more numbers at the beginning of the process, Thrashing is possible only due to major page faults, Minor page faults are an improvement of the page buffering techniques

**Question 3**

Correct

Mark 2.00 out of 2.00

W.r.t. Memory management in xv6,

xv6 uses physical memory upto 224 MB only  
Mark statements True or False

True	False	
<input checked="" type="radio"/>	<input type="radio"/> ✗	The stack allocated in entry.S is used as stack for scheduler's context for first processor
<input type="radio"/> ✗	<input checked="" type="radio"/>	The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context
<input checked="" type="radio"/>	<input type="radio"/> ✗	The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir
<input checked="" type="radio"/>	<input type="radio"/> ✗	The kernel code and data take up less than 2 MB space
<input checked="" type="radio"/>	<input type="radio"/> ✗	The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context
<input checked="" type="radio"/>	<input type="radio"/> ✗	xv6 uses physical memory upto 224 MB only
<input checked="" type="radio"/>	<input type="radio"/> ✗	The free page-frame are created out of nearly 222 MB
<input type="radio"/> ✗	<input checked="" type="radio"/>	The switchkvm() call in scheduler() changes CR3 to use page directory of new process
<input checked="" type="radio"/>	<input type="radio"/> ✗	PHYSTOP can be increased to some extent, simply by editing memlayout.h
<input checked="" type="radio"/>	<input type="radio"/> ✗	The process's address space gets mapped on frames, obtained from ~2MB:224MB range
<input type="radio"/> ✗	<input checked="" type="radio"/>	The kernel's page table given by kpgdir variable is used as stack for scheduler's context

The stack allocated in entry.S is used as stack for scheduler's context for first processor: True

The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False

The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True

The kernel code and data take up less than 2 MB space: True

The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True

xv6 uses physical memory upto 224 MB only: True

The free page-frame are created out of nearly 222 MB: True

The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False

PHYSTOP can be increased to some extent, simply by editing memlayout.h: True

The process's address space gets mapped on frames, obtained from ~2MB:224MB range: True

The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False

**Question 4**

Correct

Mark 1.00 out of 1.00

Given that a kernel has 1000 KB of total memory, and holes of sizes (in that order) 300 KB, 200 KB, 100 KB, 250 KB. For each of the requests on the left side, match it with the chunk chosen using the specified algorithm.

Consider each request as first request.

50 KB, worst fit	300 KB	✓
100 KB, worst fit	300 KB	✓
200 KB, first fit	300 KB	✓
150 KB, best fit	200 KB	✓
220 KB, best fit	250 KB	✓
150 KB, first fit	300 KB	✓

The correct answer is: 50 KB, worst fit → 300 KB, 100 KB, worst fit → 300 KB, 200 KB, first fit → 300 KB, 150 KB, best fit → 200 KB, 220 KB, best fit → 250 KB, 150 KB, first fit → 300 KB

**Question 5**

Partially correct

Mark 0.60 out of 1.00

Choice of the global or local replacement strategy is a subjective choice for kernel programmers. There are advantages and disadvantages on either side. Out of the following statements, that advocate either global or local replacement strategy, select those statements that have a logically **CONSISTENT** argument. (That is any statement that is logically correct about either global or local replacement)

**Consistent**    **Inconsistent**

<input checked="" type="radio"/>	<input type="radio"/> X	Local replacement can lead to under-utilisation of memory, because a process may not use all the pages allocated to it all the time.	<input checked="" type="checkbox"/>
<input checked="" type="radio"/>	<input type="radio"/> X	Global replacement may give highly variable per process completion time because number of page faults become un-predictable.	X
<input checked="" type="radio"/>	<input type="radio"/> X	Global replacement can be preferred when greater throughput (number of processes completing per unit time) is a concern, because each process tries to complete at the expense of others, thus leading to overall more processes completing (unless thrashing occurs).	<input checked="" type="checkbox"/>
<input checked="" type="radio"/>	<input type="radio"/> X	Local replacement results in more predictable per-process completion time because number of page faults can be better predicted.	<input checked="" type="checkbox"/>
<input checked="" type="radio"/>	<input type="radio"/> X	Local replacement can be preferred when avoiding thrashing is a major concern because with local replacement and minimum number of frames allocated, a process is always able to progress and cascading inter-process page faults are avoided.	X

Local replacement can lead to under-utilisation of memory, because a process may not use all the pages allocated to it all the time.: Consistent

Global replacement may give highly variable per process completion time because number of page faults become un-predictable.: Consistent

Global replacement can be preferred when greater throughput (number of processes completing per unit time) is a concern, because each process tries to complete at the expense of others, thus leading to overall more processes completing (unless thrashing occurs).: Consistent

Local replacement results in more predictable per-process completion time because number of page faults can be better predicted.: Consistent

Local replacement can be preferred when avoiding thrashing is a major concern because with local replacement and minimum number of frames allocated, a process is always able to progress and cascading inter-process page faults are avoided.: Consistent

**Question 6**

Correct

Mark 1.00 out of 1.00

Map the functionality/use with function/variable in xv6 code.

Setup kernel part of a page table, and switch to that page table

kvmalloc()

Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices

setupkvm()

return a free page, if available; 0, otherwise

kalloc()

Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed

mappages()

Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary

walkpgdir()

Array listing the kernel memory mappings, to be used by setupkvm()

kmap[]

Your answer is correct.

The correct answer is: Setup kernel part of a page table, and switch to that page table → kvmalloc(), Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices → setupkvm(), return a free page, if available; 0, otherwise → kalloc(), Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed → mappages(), Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary → walkpgdir(), Array listing the kernel memory mappings, to be used by setupkvm() → kmap[]

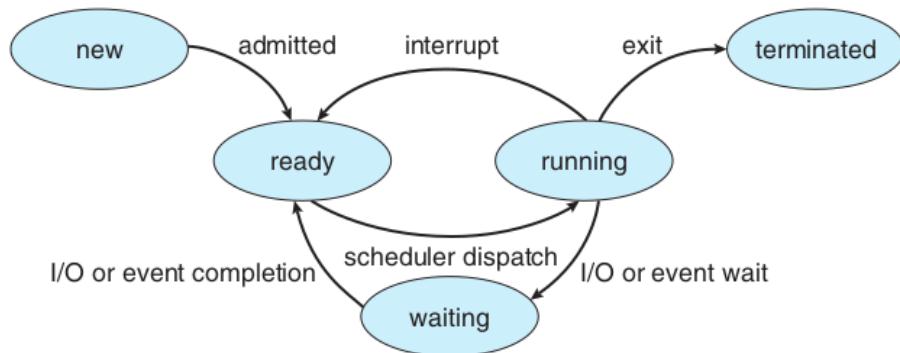
## Question 7

Partially correct

Mark 0.20 out of 1.00

Mark statements True/False w.r.t. change of states of a process. Note that a statement is true only if the claim and argument both are true.

Reference: The process state diagram (and your understanding of how kernel code works). Note - the diagram does not show zombie state!



**Figure 3.2** Diagram of process state.

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	Every forked process has to go through ZOMBIE state, at least for a small duration.
<input type="radio"/>	<input checked="" type="radio"/>	A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first
<input checked="" type="radio"/>	<input type="radio"/>	A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet
<input checked="" type="radio"/>	<input type="radio"/>	Only a process in READY state is considered by scheduler
<input type="radio"/>	<input checked="" type="radio"/>	A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.

Every forked process has to go through ZOMBIE state, at least for a small duration.: True

A process only in RUNNING state can become TERMINATED because scheduler moves it to ZOMBIE state first: False

A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred and it has not been moved to ready queue yet: True

Only a process in READY state is considered by scheduler: True

A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False

**Question 8**

Correct

Mark 2.00 out of 2.00

Consider the reference string

6 4 2 0 1 2 6 9 2 0 5

If the number of page frames is 3, then total number of page faults (including initial), using FIFO replacement is:

Answer: 

#6# 6,4# 6,4,2 #0,4,2# 0,1,2 #0,1,6 #9,1,6# 9,2,6# 9,2,0 #5,2,0

The correct answer is: 10

**Question 9**

Incorrect

Mark 0.00 out of 1.00

Select all the correct statements about linking and loading.

Select one or more:

- a. Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently)
- b. Dynamic linking essentially results in relocatable code.
- c. Continuous memory management schemes can support static linking and static loading. (may be inefficiently)
- d. Loader is last stage of the linker program
- e. Dynamic linking and loading is not possible without demand paging or demand segmentation.
- f. Static linking leads to non-relocatable code
- g. Continuous memory management schemes can support dynamic linking and dynamic loading.
- h. Dynamic linking is possible with continuous memory management, but variable sized partitions only.
- i. Loader is part of the operating system

Your answer is incorrect.

The correct answers are: Continuous memory management schemes can support static linking and static loading. (may be inefficiently), Continuous memory management schemes can support static linking and dynamic loading. (may be inefficiently), Dynamic linking essentially results in relocatable code., Loader is part of the operating system, Dynamic linking and loading is not possible without demand paging or demand segmentation.

**Question 10**

Correct

Mark 1.00 out of 1.00

Consider a computer system with a 32-bit logical address and 4- KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?

Write answer as a decimal number.

A conventional, single-level page table:

1048576



An inverted page table:

131072

**Question 11**

Incorrect

Mark 0.00 out of 1.00

W.r.t. xv6 code, match the state of a process with a code that sets the state

RUNNING	Choose...
ZOMBIE	Choose...
EMBRYO	Choose...
SLEEPING	Choose...
UNUSED	Choose...
RUNNABLE	scheduler()



The correct answer is: RUNNING → scheduler(), ZOMBIE → exit(), called by process itself, EMBRYO → fork()->allocproc() before setting up the UVM, SLEEPING → sleep(), called by any process blocking itself, UNUSED → wait(), called by parent process, RUNNABLE → wakeup(), called by an interrupt handler

**Question 12**

Not answered

Marked out of 1.00

Select the correct statements about interrupt handling in xv6 code

- a. The trapframe pointer in struct proc, points to a location on user stack
- b. The CS and EIP are changed only immediately on a hardware interrupt
- c. The CS and EIP are changed only after pushing user code's SS,ESP on stack
- d. The trapframe pointer in struct proc, points to a location on kernel stack
- e. On any interrupt/syscall/exception the control first jumps in vectors.S
- f. The function trap() is called irrespective of hardware interrupt/system-call/exception
- g. All the 256 entries in the IDT are filled
- h. The function trap() is called only in case of hardware interrupt
- i. xv6 uses the 64th entry in IDT for system calls
- j. xv6 uses the 0x64th entry in IDT for system calls
- k. Before going to alptraps, the kernel stack contains upto 5 entries.
- l. Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt
- m. On any interrupt/syscall/exception the control first jumps in trapasm.S

Your answer is incorrect.

The correct answers are: All the 256 entries in the IDT are filled, Each entry in IDT essentially gives the values of CS and EIP to be used in handling that interrupt, xv6 uses the 64th entry in IDT for system calls, On any interrupt/syscall/exception the control first jumps in vectors.S, Before going to alptraps, the kernel stack contains upto 5 entries., The trapframe pointer in struct proc, points to a location on kernel stack, The function trap() is called irrespective of hardware interrupt/system-call/exception, The CS and EIP are changed only after pushing user code's SS,ESP on stack

**Question 13**

Correct

Mark 1.00 out of 1.00

The complete range of virtual addresses (after main() in main.c is over), from which the free pages used by kalloc() and kfree() is derived, are:

- a. end, (4MB + PHYSTOP)
- b. P2V(end), PHYSTOP
- c. end, P2V(4MB + PHYSTOP)
- d. end, PHYSTOP
- e. end, 4MB
- f. end, P2V(PHYSTOP) ✓
- g. P2V(end), P2V(PHYSTOP)

Your answer is correct.

The correct answer is: end, P2V(PHYSTOP)

**Question 14**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about MMU and its functionality (on a non-demand paged system)

Select one or more:

- a. Illegal memory access is detected in hardware by MMU and a trap is raised ✓
- b. Illegal memory access is detected by operating system
- c. MMU is a separate chip outside the processor
- d. The operating system interacts with MMU for every single address translation
- e. Logical to physical address translations in MMU are done in hardware, automatically ✓
- f. Logical to physical address translations in MMU are done with specific machine instructions
- g. MMU is inside the processor ✓
- h. The Operating system sets up relevant CPU registers to enable proper MMU translations ✓

Your answer is correct.

The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

**Question 15**

Incorrect

Mark 0.00 out of 2.00

Order the following events, in the creation of init() process in xv6:

1. ✗ initcode is selected by scheduler for execution
2. ✗ kernel stack is allocated for initcode process
3. ✗ values are set in the trapframe of initcode
4. ✗ sys\_exec runs
5. ✗ initcode process is set to be runnable
6. ✗ code is set to start in forkret() when process gets scheduled
7. ✗ Arguments are setup on process stack for /init
8. ✗ trapframe and context pointers are set to proper location
9. ✗ trap() runs
10. ✗ userinit() is called
11. ✗ the header of "/init" ELF file is ready by kernel
12. ✗ empty struct proc is obtained for initcode
13. ✗ Stack is allocated for "/init" process
14. ✗ function pointer from syscalls[] array is invoked
15. ✗ memory mappings are created for "/init" process
16. ✗ page table mappings of 'initcode' are replaced by mappings of 'init'
17. ✗ initcode calls exec system call
18. ✗ initcode process runs
19. ✗ name of process "/init" is copied in struct proc

Your answer is incorrect.

Grading type: Relative to the next item (including last)

Grade details: 0 / 19 = 0%

Here are the scores for each item in this response:

1. 0 / 1 = 0%
2. 0 / 1 = 0%
3. 0 / 1 = 0%
4. 0 / 1 = 0%
5. 0 / 1 = 0%
6. 0 / 1 = 0%
7. 0 / 1 = 0%
8. 0 / 1 = 0%
9. 0 / 1 = 0%
10. 0 / 1 = 0%
11. 0 / 1 = 0%
12. 0 / 1 = 0%
13. 0 / 1 = 0%
14. 0 / 1 = 0%
15. 0 / 1 = 0%

- 16. 0 / 1 = 0%
- 17. 0 / 1 = 0%
- 18. 0 / 1 = 0%
- 19. 0 / 1 = 0%

The correct order for these items is as follows:

1. userinit() is called
2. empty struct proc is obtained for initcode
3. kernel stack is allocated for initcode process
4. trapframe and context pointers are set to proper location
5. code is set to start in forkret() when process gets scheduled
6. kernel memory mappings are created for initcode
7. values are set in the trapframe of initcode
8. initcode process is set to be runnable
9. initcode is selected by scheduler for execution
10. initcode process runs
11. initcode calls exec system call
12. trap() runs
13. function pointer from syscalls[] array is invoked
14. sys\_exec runs
15. the header of "/init" ELF file is ready by kernel
16. memory mappings are created for "/init" process
17. Stack is allocated for "/init" process
18. Arguments on setup on process stack for /init
19. name of process "/init" is copied in struct proc
20. page table mappings of 'initcode' are replaced by makpings of 'init'

**Question 16**

Partially correct

Mark 0.56 out of 1.00

Mark the statements as True or False, w.r.t. mmap()

True	False	
<input checked="" type="radio"/>	<input checked="" type="radio"/>	mmap() can be implemented on both demand paged and non-demand paged systems.
<input checked="" type="radio"/>	<input checked="" type="radio"/>	MAP_FIXED guarantees that the mapping is always done at the specified address
<input checked="" type="radio"/>	<input checked="" type="radio"/>	MAP_PRIVATE leads to a mapping that is copy-on-write
<input checked="" type="radio"/>	<input checked="" type="radio"/>	on failure mmap() returns (void *)-1
<input checked="" type="radio"/>	<input checked="" type="radio"/>	MAP_SHARED leads to a mapping that is copy-on-write
<input checked="" type="radio"/>	<input checked="" type="radio"/>	mmap() results in changes to buffer-cache of the kernel.
<input checked="" type="radio"/>	<input checked="" type="radio"/>	on failure mmap() returns NULL
<input checked="" type="radio"/>	<input checked="" type="radio"/>	mmap() results in changes to page table of a process.
<input checked="" type="radio"/>	<input checked="" type="radio"/>	mmap() is a system call

mmap() can be implemented on both demand paged and non-demand paged systems.: True

MAP\_FIXED guarantees that the mapping is always done at the specified address: False

MAP\_PRIVATE leads to a mapping that is copy-on-write: True

on failure mmap() returns (void \*)-1: True

MAP\_SHARED leads to a mapping that is copy-on-write: False

mmap() results in changes to buffer-cache of the kernel.: False

on failure mmap() returns NULL: False

mmap() results in changes to page table of a process.: True

mmap() is a system call: True

**Question 17**

Incorrect

Mark 0.00 out of 1.00

If one thread opens a file with read privileges then

Select one:

- a. other threads in the same process can also read from that file
- b. none of these
- c. any other thread cannot read from that file
- d. other threads in the another process can also read from that file

Your answer is incorrect.

The correct answer is: other threads in the same process can also read from that file

**Question 18**

Partially correct

Mark 0.60 out of 1.00

Mark the statements about named and un-named pipes as True or False

True	False	
<input checked="" type="radio"/>	<input type="radio"/> 	Named pipe exists as a file
<input checked="" type="radio"/>	<input type="radio"/> 	Un-named pipes are inherited by a child process from parent.
<input type="radio"/> 	<input checked="" type="radio"/>	The buffers for named-pipe are in process-memory while the buffers for the un-named pipe are in kernel memory.
<input checked="" type="radio"/>	<input type="radio"/> 	Both types of pipes are an extension of the idea of "message passing".
<input type="radio"/> 	<input checked="" type="radio"/>	A named pipe has a name decided by the kernel.
<input checked="" type="radio"/>	<input type="radio"/> 	Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.
<input checked="" type="radio"/>	<input type="radio"/> 	Both types of pipes provide FIFO communication.
<input type="radio"/> 	<input checked="" type="radio"/>	Named pipes can be used for communication between only "related" processes.
<input checked="" type="radio"/>	<input type="radio"/> 	Named pipes can exist beyond the life-time of processes using them.
<input type="radio"/> 	<input checked="" type="radio"/>	The pipe() system call can be used to create either a named or un-named pipe.

Named pipe exists as a file.: True

Un-named pipes are inherited by a child process from parent.: True

The buffers for named-pipe are in process-memory while the buffers for the un-named pipe are in kernel memory.: False

Both types of pipes are an extension of the idea of "message passing": True

A named pipe has a name decided by the kernel.: False

Un-named pipes can be used for communication between only "related" processes, if the common ancestor created it.: True

Both types of pipes provide FIFO communication.: True

Named pipes can be used for communication between only "related" processes.: False

Named pipes can exist beyond the life-time of processes using them.: True

The pipe() system call can be used to create either a named or un-named pipe.: False

**Question 19**

Partially correct

Mark 0.67 out of 1.00

Select the most common causes of use of IPC by processes

- a. More modular code
- b. Breaking up a large task into small tasks and speeding up computation, on multiple core machines ✓
- c. More security checks
- d. Sharing of information of common interest ✓
- e. Get the kernel performance statistics

The correct answers are: Sharing of information of common interest, Breaking up a large task into small tasks and speeding up computation, on multiple core machines, More modular code

**Question 20**

Correct

Mark 1.00 out of 1.00

For each function/code-point, select the status of segmentation setup in xv6

after seginit() in main()	gdt setup with 5 entries (0 to 4) on one processor	✓
bootmain()	gdt setup with 3 entries, at start32 symbol of bootasm.S	✓
after startothers() in main()	gdt setup with 5 entries (0 to 4) on all processors	✓
entry.S	gdt setup with 3 entries, at start32 symbol of bootasm.S	✓
kvmalloc() in main()	gdt setup with 3 entries, at start32 symbol of bootasm.S	✓
bootasm.S	gdt setup with 3 entries, at start32 symbol of bootasm.S	✓

Your answer is correct.

The correct answer is: after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor, bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S, after startothers() in main() → gdt setup with 5 entries (0 to 4) on all processors, entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, kvmalloc() in main() → gdt setup with 3 entries, at start32 symbol of bootasm.S, bootasm.S → gdt setup with 3 entries, at start32 symbol of bootasm.S

**Question 21**

Partially correct

Mark 0.50 out of 1.00

Mark whether the given sequence of events is possible or not-possible. Also, select the reason for your answer.

For each sequence it's a not-possible sequence if some important event is not mentioned in the sequence.

Assume that the kernel code is non-interruptible and uniprocessor system.

Process P1, user code executing

Timer interrupt

Context changes to kernel context

Generic interrupt handler runs

Generic interrupt handler calls Scheduler

Scheduler selects P2 for execution

After scheduler, Process P2 user code executing

This sequence of events is:  not-possible ✓

Because

Generic interrupt handler can not call scheduler ✗

**Question 22**

Partially correct

Mark 0.63 out of 1.00

Mark the statements as True or False, w.r.t. passing of arguments to system calls in xv6 code.

True	False	
<input checked="" type="radio"/> ✘	<input type="radio"/> ✓	Integer arguments are stored in eax, ebx, ecx, etc. registers
<input type="radio"/> ✓	<input checked="" type="radio"/> ✘	String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer
<input checked="" type="radio"/> ✓	<input checked="" type="radio"/> ✘	The functions like argint(), argstr() make the system call arguments available in the kernel.
<input checked="" type="radio"/> ✘	<input type="radio"/> ✓	String arguments are first copied to trapframe and then from trapframe to kernel's other variables.
<input checked="" type="radio"/> ✓	<input checked="" type="radio"/> ✘	The arguments to system call originally reside on process stack.
<input checked="" type="radio"/> ✘	<input type="radio"/> ✓	The arguments to system call are copied to kernel stack in trapasm.S
<input checked="" type="radio"/> ✓	<input checked="" type="radio"/> ✘	Integer arguments are copied from user memory to kernel memory using argint()
<input checked="" type="radio"/> ✓	<input checked="" type="radio"/> ✘	The arguments are accessed in the kernel code using esp on the trapframe.

Integer arguments are stored in eax, ebx, ecx, etc. registers: False

String arguments are NOT copied in kernel memory, but just pointed to by a kernel memory pointer: True

The functions like argint(), argstr() make the system call arguments available in the kernel.: True

String arguments are first copied to trapframe and then from trapframe to kernel's other variables.: False

The arguments to system call originally reside on process stack.: True

The arguments to system call are copied to kernel stack in trapasm.S: False

Integer arguments are copied from user memory to kernel memory using argint(): True

The arguments are accessed in the kernel code using esp on the trapframe.: True

**Question 23**

Not answered

Marked out of 1.00

Given below is a sequence of reference bits on pages before the second chance algorithm runs. Before the algorithm runs, the counter is at the page marked (x). Write the sequence of reference bits after the second chance algorithm has executed once. In the answer write PRECISELY one space BETWEEN each number and do not mention (x).

0 0 1(x) 1 0 1 1

Answer:



The correct answer is: 0 0 0 0 0 1 1

**Question 24**

Correct

Mark 2.00 out of 2.00

For the reference string

3 4 3 5 2

using FIFO replacement policy for pages,

consider the number of page faults for 2, 3 and 4 page frames.

Select the correct statement.

Select one:

- a. Exhibit Balady's anomaly between 3 and 4 frames
- b. Do not exhibit Balady's anomaly
- c. Exhibit Balady's anomaly between 2 and 3 frames



Your answer is correct.

The correct answer is: Do not exhibit Balady's anomaly

**Question 25**

Correct

Mark 1.00 out of 1.00

For the reference string

3 4 3 5 2

using LRU replacement policy for pages,

consider the number of page faults for 2, 3 and 4 page frames.

Select the most correct statement.

Select one:

- a. LRU will never exhibit Balady's anomaly ✓
- b. Exhibit Balady's anomaly between 2 and 3 frames
- c. This example does not exhibit Balady's anomaly
- d. Exhibit Balady's anomaly between 3 and 4 frames

Your answer is correct.

The correct answer is: LRU will never exhibit Balady's anomaly

**Question 26**

Partially correct

Mark 0.55 out of 1.00

Select all the correct statements about process states.

Note that in this question you lose marks for every incorrect choice that you make, proportional to actual number of incorrect choices.

- a. Process state is implemented as a string
- b. Process state is stored in the PCB ✓
- c. A process becomes ZOMBIE when another process bites into its memory
- d. Process state is stored in the processor ✗
- e. The scheduler can change state of a process from RUNNABLE to RUNNING and vice-versa
- f. The scheduler can change state of a process from RUNNABLE to RUNNING ✓
- g. A process becomes ZOMBIE when it calls exit() ✓
- h. Process state is changed only by interrupt handlers
- i. Process state can be implemented as just a number

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: Process state is stored in the PCB, Process state can be implemented as just a number, The scheduler can change state of a process from RUNNABLE to RUNNING, A process becomes ZOMBIE when it calls exit()

**Question 27**

Partially correct

Mark 0.38 out of 1.00

Consider a demand-paging system with the following time-measured utilizations:

CPU utilization : 20%

Paging disk: 97.7%

Other I/O devices: 5%

For each of the following, indicate whether it will (or is likely to) improve CPU utilization (even if by a small amount). Explain your answers.

a. Install a faster CPU : Yes ✗

b. Install a bigger paging disk. : Yes ✗

c. Increase the degree of multiprogramming. : Yes ✗

d. Decrease the degree of multiprogramming. : Yes ✓

e. Install more main memory.: Yes ✓

f. Install a faster hard disk or multiple controllers with multiple hard disks. : Yes ✓

g. Add prepaging to the page-fetch algorithms. :

May be ✗

h. Increase the page size. : May be ✗

**Question 28**

Incorrect

Mark 0.00 out of 1.00

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now:

10011010

Now, there is a request for a chunk of 50 bytes.

After this allocation, the bitmap, indicating the status of the buddy allocator will be

Answer: 10110010

✗

The correct answer is: 11111010

**Question 29**

Partially correct

Mark 0.75 out of 1.00

Select the correct points of comparison between POSIX and System V shared memory.

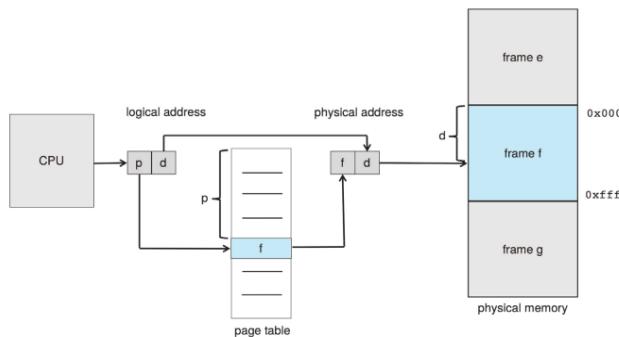
- a. POSIX shared memory is newer than System V shared memory ✓
- b. POSIX shared memory is "thread safe", System V is not ✓
- c. System V is more prevalent than POSIX even today ✓
- d. POSIX allows giving name to shared memory, System V does not

The correct answers are: POSIX shared memory is newer than System V shared memory, POSIX shared memory is "thread safe", System V is not, POSIX allows giving name to shared memory, System V does not, System V is more prevalent than POSIX even today

**Question 30**

Partially correct

Mark 0.67 out of 1.00

**Figure 9.8** Paging hardware.

Mark the statements as True or False, w.r.t. the above diagram (note that the diagram does not cover all details of what actually happens!)

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	The combining of f and d is done by MMU
<input type="radio"/>	<input checked="" type="radio"/>	There are total 3 memory references in this diagram
<input checked="" type="radio"/>	<input type="radio"/>	The split of logical address into p and d is done by MMU
<input checked="" type="radio"/>	<input type="radio"/>	The page table is in physical memory and must be continuous
<input type="radio"/>	<input checked="" type="radio"/>	Using the offset d in the physical page-frame is done by MMU
<input checked="" type="radio"/>	<input type="radio"/>	The logical address issued by CPU is the same one generated by compiler

The combining of f and d is done by MMU: True

There are total 3 memory references in this diagram: False

The split of logical address into p and d is done by MMU: True

The page table is in physical memory and must be continuous: True

Using the offset d in the physical page-frame is done by MMU: False

The logical address issued by CPU is the same one generated by compiler: True

**Question 31**

Partially correct

Mark 0.50 out of 1.00

Select all the correct statements about signals

Select one or more:

- a. SIGKILL definitely kills a process because its code runs in kernel mode of CPU
- b. Signals are delivered to a process by another process ✗
- c. The signal handler code runs in kernel mode of CPU
- d. SIGKILL definitely kills a process because it can't be caught or ignored, and its default action terminates the process ✓
- e. The signal handler code runs in user mode of CPU ✓
- f. A signal handler can be invoked asynchronously or synchronously depending on signal type ✓
- g. Signal handlers once replaced can't be restored
- h. Signals are delivered to a process by kernel

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: Signals are delivered to a process by kernel, A signal handler can be invoked asynchronously or synchronously depending on signal type, The signal handler code runs in user mode of CPU, SIGKILL definitely kills a process because it can't be caught or ignored, and its default action terminates the process

**Question 32**

Correct

Mark 1.00 out of 1.00

The data structure used in kalloc() and kfree() in xv6 is

- a. Singly linked circular list
- b. Singly linked NULL terminated list ✓
- c. Double linked NULL terminated list
- d. Doubly linked circular list

Your answer is correct.

The correct answer is: Singly linked NULL terminated list

**Question 33**

Partially correct

Mark 1.78 out of 2.00

Match the description of a memory management function with the name of the function that provides it, in xv6

Load contents from ELF into existing pages	loaduvm()	✓
Mark the page as in-accessible	clearpteu()	✓
setup the kernel part in the page table	setupkvm()	✓
Switch to kernel page table	switchkvm()	✓
Create a copy of the page table of a process	copyuvm()	✓
Copy the code pages of a process	No such function	✓
Setup and load the user page table for initcode process	inituvm()	✓
Switch to user page table	switchuvm()	✓
Load contents from ELF into pages after allocating the pages first	inituvm()	✗

The correct answer is: Load contents from ELF into existing pages → loaduvm(), Mark the page as in-accessible → clearpteu(), setup the kernel part in the page table → setupkvm(), Switch to kernel page table → switchkvm(), Create a copy of the page table of a process → copyuvm(), Copy the code pages of a process → No such function, Setup and load the user page table for initcode process → inituvm(), Switch to user page table → switchuvm(), Load contents from ELF into pages after allocating the pages first → No such function

**Question 34**

Partially correct

Mark 0.60 out of 1.00

Mark the statements as True or False, w.r.t. thrashing

True	False	
<input checked="" type="radio"/> ✘	<input type="radio"/> ✓	Thrashing occurs because some process is doing lot of disk I/O.
<input checked="" type="radio"/> ✓	<input checked="" type="radio"/> ✘	Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.
<input checked="" type="radio"/> ✘	<input checked="" type="radio"/> ✓	mmap() solves the problem of thrashing.
<input checked="" type="radio"/> ✓	<input checked="" type="radio"/> ✘	The working set model is an attempt at approximating the locality of a process.
<input checked="" type="radio"/> ✓	<input checked="" type="radio"/> ✘	Thrashing is particular to demand paging systems, and does not apply to pure paging systems.
<input checked="" type="radio"/> ✘	<input type="radio"/> ✓	Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.
<input checked="" type="radio"/> ✘	<input checked="" type="radio"/> ✓	Thrashing can occur even if entire memory is not in use.
<input checked="" type="radio"/> ✓	<input checked="" type="radio"/> ✘	During thrashing the CPU is under-utilised as most time is spent in I/O
<input checked="" type="radio"/> ✓	<input checked="" type="radio"/> ✘	Thrashing can be limited if local replacement is used.
<input checked="" type="radio"/> ✓	<input checked="" type="radio"/> ✘	Thrashing occurs when the total size of all processes's locality exceeds total memory size.

Thrashing occurs because some process is doing lot of disk I/O.: False

Processes keep changing their locality of reference, and a high rate of page faults occur when they are changing the locality.: True

mmap() solves the problem of thrashing.: False

The working set model is an attempt at approximating the locality of a process.: True

Thrashing is particular to demand paging systems, and does not apply to pure paging systems.: True

Processes keep changing their locality of reference, and least number of page faults occur when they are changing the locality.: False

Thrashing can occur even if entire memory is not in use.: False

During thrashing the CPU is under-utilised as most time is spent in I/O: True

Thrashing can be limited if local replacement is used.: True

Thrashing occurs when the total size of all processes's locality exceeds total memory size.: True

**Question 35**

Correct

Mark 1.00 out of 1.00

After virtual memory is implemented

(select T/F for each of the following) One Program's size can be larger than physical memory size

True	False	
<input checked="" type="radio"/>	<input type="radio"/> ✗	Cumulative size of all programs can be larger than physical memory size
<input checked="" type="radio"/>	<input type="radio"/> ✗	Code need not be completely in memory
<input checked="" type="radio"/>	<input type="radio"/> ✗	One Program's size can be larger than physical memory size
<input type="radio"/> ✗	<input checked="" type="radio"/>	Virtual addresses become available to executing process
<input type="radio"/> ✗	<input checked="" type="radio"/>	Virtual access to memory is granted to all processes
<input checked="" type="radio"/>	<input type="radio"/> ✗	Relatively less I/O may be possible during process execution
<input checked="" type="radio"/>	<input type="radio"/> ✗	Logical address space could be larger than physical address space

Cumulative size of all programs can be larger than physical memory size: True

Code need not be completely in memory: True

One Program's size can be larger than physical memory size: True

Virtual addresses become available to executing process: False

Virtual access to memory is granted to all processes: False

Relatively less I/O may be possible during process execution: True

Logical address space could be larger than physical address space: True

◀ (Optional Assignment) lseek system call in xv6

Jump to...

Feedback on Quiz-2 ►

**Started on** Thursday, 18 March 2021, 2:46 PM

**State** Finished

**Completed on** Thursday, 18 March 2021, 3:50 PM

**Time taken** 1 hour 4 mins

**Grade** 10.36 out of 20.00 (52%)

**Question 1**

Partially correct

Mark 0.57 out of 1.00

Mark True, the actions done as part of code of swtch() in swtch.S, in xv6

**True**

**False**

<input checked="" type="radio"/>	<input checked="" type="radio"/>	Restore new callee saved registers from kernel stack of new context	✓
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Save old callee saved registers on kernel stack of old context	✓
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Save old callee saved registers on user stack of old context	✓
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Switch from old process context to new process context	✗
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Switch from one stack (old) to another(new)	✗
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Restore new callee saved registers from user stack of new context	✓
<input checked="" type="radio"/>	<input checked="" type="radio"/>	Jump to code in new context	✗

Restore new callee saved registers from kernel stack of new context: True

Save old callee saved registers on kernel stack of old context: True

Save old callee saved registers on user stack of old context: False

Switch from old process context to new process context: False

Switch from one stack (old) to another(new): True

Restore new callee saved registers from user stack of new context: False

Jump to code in new context: False

**Question 2**

Partially correct

Mark 0.17 out of 0.50

For each function/code-point, select the status of segmentation setup in xv6

bootmain()	gdt setup with 3 entries, right from first line of code of bootloader	✗
kvmalloc() in main()	gdt setup with 5 entries (0 to 4) on one processor	✗
after startothers() in main()	gdt setup with 5 entries (0 to 4) on all processors	✓
after seginit() in main()	gdt setup with 5 entries (0 to 4) on all processors	✗
bootasm.S	gdt setup with 3 entries, right from first line of code of bootloader	✗
entry.S	gdt setup with 3 entries, at start32 symbol of bootasm.S	✓

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: bootmain() → gdt setup with 3 entries, at start32 symbol of bootasm.S, kvmalloc() in main() → gdt setup with 3 entries, at start32 symbol of bootasm.S, after startothers() in main() → gdt setup with 5 entries (0 to 4) on all processors, after seginit() in main() → gdt setup with 5 entries (0 to 4) on one processor, bootasm.S → gdt setup with 3 entries, at start32 symbol of bootasm.S, entry.S → gdt setup with 3 entries, at start32 symbol of bootasm.S

**Question 3**

Partially correct

Mark 0.38 out of 1.00

Compare paging with demand paging and select the correct statements.

Select one or more:

- a. The meaning of valid-invalid bit in page table is different in paging and demand-paging. ✓
- b. Demand paging requires additional hardware support, compared to paging. ✓
- c. Paging requires some hardware support in CPU
- d. With paging, it's possible to have user programs bigger than physical memory. ✗
- e. Both demand paging and paging support shared memory pages. ✓
- f. Demand paging always increases effective memory access time.
- g. With demand paging, it's possible to have user programs bigger than physical memory. ✓
- h. Calculations of number of bits for page number and offset are same in paging and demand paging. ✓
- i. TLB hit ration has zero impact in effective memory access time in demand paging.
- j. Paging requires NO hardware support in CPU

Your answer is partially correct.

You have correctly selected 5.

The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

**Question 4**

Partially correct

Mark 0.44 out of 0.50

Suppose a processor supports base(relocation register) + limit scheme of MMU.

Assuming this, mark the statements as True/False

True	False	
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The OS may terminate the process while handling the interrupt of memory violation
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The hardware detects any memory access beyond the limit value and raises an interrupt
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/>	The hardware may terminate the process while handling the interrupt of memory violation
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The OS sets up the relocation and limit registers when the process is scheduled
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The process sets up its own relocation and limit registers when the process is scheduled
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The OS detects any memory access beyond the limit value and raises an interrupt
<input type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The compiler generates machine code assuming appropriately sized segments for code, data and stack.

The OS may terminate the process while handling the interrupt of memory violation: True

The hardware detects any memory access beyond the limit value and raises an interrupt: True

The hardware may terminate the process while handling the interrupt of memory violation: False

The OS sets up the relocation and limit registers when the process is scheduled: True

The compiler generates machine code assuming continuous memory address space for process, and calculating appropriate sizes for code, and data;: True

The process sets up its own relocation and limit registers when the process is scheduled: False

The OS detects any memory access beyond the limit value and raises an interrupt: False

The compiler generates machine code assuming appropriately sized segments for code, data and stack.: False

**Question 5**

Correct

Mark 0.50 out of 0.50

Consider the following list of free chunks, in continuous memory management:

10k, 25k, 12k, 7k, 9k, 13k

Suppose there is a request for chunk of size 9k, then the free chunk selected under each of the following schemes will be

Best fit:

9k



First fit:

10k



Worst fit:

25k

**Question 6**

Partially correct

Mark 0.50 out of 1.00

Select all the correct statements about MMU and its functionality

Select one or more:

- a. MMU is a separate chip outside the processor
- b. MMU is inside the processor ✓
- c. Logical to physical address translations in MMU are done with specific machine instructions
- d. The operating system interacts with MMU for every single address translation ✗
- e. Illegal memory access is detected in hardware by MMU and a trap is raised ✓
- f. The Operating system sets up relevant CPU registers to enable proper MMU translations
- g. Logical to physical address translations in MMU are done in hardware, automatically ✓
- h. Illegal memory access is detected by operating system

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

**Question 7**

Incorrect

Mark 0.00 out of 0.50

Assuming a 8- KB page size, what is the page numbers for the address 874815 reference in decimal :  
(give answer also in decimal)

Answer:  ×

The correct answer is: 107

**Question 8**

Incorrect

Mark 0.00 out of 0.25

Select the compiler's view of the process's address space, for each of the following MMU schemes:  
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Segmentation, then paging	<input type="checkbox"/> Many continuous chunks each of page size	×
Relocation + Limit	<input type="checkbox"/> Many continuous chunks of same size	×
Segmentation	<input type="checkbox"/> one continuous chunk	×
Paging	<input type="checkbox"/> many continuous chunks of variable size	×

Your answer is incorrect.

The correct answer is: Segmentation, then paging → many continuous chunks of variable size, Relocation + Limit → one continuous chunk, Segmentation → many continuous chunks of variable size, Paging → one continuous chunk

**Question 9**

Incorrect

Mark 0.00 out of 0.50

Suppose the memory access time is 180ns and TLB hit ratio is 0.3, then effective memory access time is (in nanoseconds);

Answer:  ×

The correct answer is: 306.00

**Question 10**

Correct

Mark 0.50 out of 0.50

In xv6, The struct context is given as

```
struct context {
    uint edi;
    uint esi;
    uint ebx;
    uint ebp;
    uint eip;
};
```

Select all the reasons that explain why only these 5 registers are included in the struct context.

- a. The segment registers are same across all contexts, hence they need not be saved ✓
- b. esp is not saved in context, because context{} is on stack and it's address is always argument to swtch() ✓
- c. xv6 tries to minimize the size of context to save memory space
- d. esp is not saved in context, because it's not part of the context
- e. eax, ecx, edx are caller save, hence no need to save ✓

Your answer is correct.

The correct answers are: The segment registers are same across all contexts, hence they need not be saved, eax, ecx, edx are caller save, hence no need to save, esp is not saved in context, because context{} is on stack and it's address is always argument to swtch()

**Question 11**

Partially correct

Mark 0.83 out of 1.50

Arrange the following events in order, in page fault handling:

Disk interrupt wakes up the process

7	✓
---	---

The reference bit is found to be invalid by MMU

1	✓
---	---

OS makes available an empty frame

6	✗
---	---

Restart the instruction that caused the page fault

9	✓
---	---

A hardware interrupt is issued

3	✗
---	---

OS schedules a disk read for the page (from backing store)

5	✓
---	---

Process is kept in wait state

4	✗
---	---

Page tables are updated for the process

8	✓
---	---

Operating system decides that the page was not in memory

2	✗
---	---

Your answer is partially correct.

You have correctly selected 5.

The correct answer is: Disk interrupt wakes up the process → 7, The reference bit is found to be invalid by MMU → 1, OS makes available an empty frame → 4, Restart the instruction that caused the page fault → 9, A hardware interrupt is issued → 2, OS schedules a disk read for the page (from backing store) → 5, Process is kept in wait state → 6, Page tables are updated for the process → 8, Operating system decides that the page was not in memory → 3

**Question 12**

Incorrect

Mark 0.00 out of 0.50

Suppose a kernel uses a buddy allocator. The smallest chunk that can be allocated is of size 32 bytes. One bit is used to track each such chunk, where 1 means allocated and 0 means free. The chunk looks like this as of now:

00001010

Now, there is a request for a chunk of 70 bytes.

After this allocation, the bitmap, indicating the status of the buddy allocator will be

Answer: 11101010



The correct answer is: 11111010

**Question 13**

Incorrect

Mark 0.00 out of 0.25

The complete range of virtual addresses (after main() in main.c is over), from which the free pages used by kalloc() and kfree() is derived, are:

- a. end, 4MB
- b. P2V(end), P2V(PHYSTOP)
- c. end, P2V(4MB + PHYSTOP)
- d. P2V(end), PHYSTOP ✗
- e. end, (4MB + PHYSTOP)
- f. end, PHYSTOP
- g. end, P2V(PHYSTOP)

Your answer is incorrect.

The correct answer is: end, P2V(PHYSTOP)

**Question 14**

Partially correct

Mark 0.33 out of 0.50

Match the pair

Hashed page table	Linear search on collision done by OS (e.g. SPARC Solaris) typically	✓
Inverted Page table	Linear/Parallel search using frame number in page table	✗
Hierarchical Paging	More memory access time per hierarchy	✓

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: Hashed page table → Linear search on collision done by OS (e.g. SPARC Solaris) typically, Inverted Page table → Linear/Parallel search using page number in page table, Hierarchical Paging → More memory access time per hierarchy

**Question 15**

Partially correct

Mark 0.29 out of 0.50

After virtual memory is implemented

(select T/F for each of the following) One Program's size can be larger than physical memory size

True	False	
<input checked="" type="radio"/>	<input type="radio"/> ✗	Code need not be completely in memory
<input checked="" type="radio"/>	<input type="radio"/> ✗	Cumulative size of all programs can be larger than physical memory size
<input type="radio"/> ✗	<input checked="" type="radio"/>	Virtual access to memory is granted
<input checked="" type="radio"/>	<input type="radio"/> ✗	Logical address space could be larger than physical address space
<input type="radio"/> ✗	<input checked="" type="radio"/>	Virtual addresses are available
<input checked="" type="radio"/>	<input checked="" type="radio"/> ✗	Relatively less I/O may be possible during process execution
<input checked="" type="radio"/>	<input type="radio"/> ✗	One Program's size can be larger than physical memory size

Code need not be completely in memory: True

Cumulative size of all programs can be larger than physical memory size: True

Virtual access to memory is granted: False

Logical address space could be larger than physical address space: True

Virtual addresses are available: False

Relatively less I/O may be possible during process execution: True

One Program's size can be larger than physical memory size: True

**Question 16**

Partially correct

Mark 0.64 out of 1.00

W.r.t. Memory management in xv6,

xv6 uses physical memory upto 224 MB only  
Mark statements True or False**True      False**

<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The stack allocated in entry.S is used as stack for scheduler's context for first processor	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The free page-frame are created out of nearly 222 MB	✗
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The kernel code and data take up less than 2 MB space	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() changes CR3 to use page directory of new process	✗
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	PHYSTOP can be increased to some extent, simply by editing memlayout.h	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	xv6 uses physical memory upto 224 MB only	✗
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The process's address space gets mapped on frames, obtained from ~2MB:224MB range	✓
<input checked="" type="radio"/>	<input type="radio"/> <input checked="" type="checkbox"/>	The kernel's page table given by kpgdir variable is used as stack for scheduler's context	✗

The switchkvm() call in scheduler() is invoked after control comes to it from sched(), thus demanding execution in kernel's context: True

The stack allocated in entry.S is used as stack for scheduler's context for first processor: True

The switchkvm() call in scheduler() changes CR3 to use page directory kpgdir: True

The free page-frame are created out of nearly 222 MB: True

The kernel code and data take up less than 2 MB space: True

The switchkvm() call in scheduler() changes CR3 to use page directory of new process: False

The switchkvm() call in scheduler() is invoked after control comes to it from swtch() scheduler(), thus demanding execution in new process's context: False

PHYSTOP can be increased to some extent, simply by editing memlayout.h: True

xv6 uses physical memory upto 224 MB only: True

The process's address space gets mapped on frames, obtained from ~2MB:224MB range: True

The kernel's page table given by kpgdir variable is used as stack for scheduler's context: False

**Question 17**

Incorrect

Mark 0.00 out of 1.50

Consider the reference string

6 4 2 0 1 2 6 9 2 0 5

If the number of page frames is 3, then total number of page faults (including initial), using LRU replacement is:

Answer:  ✖

#6# 6,4# 6,4,2 # 0,4,2#0,1,2#6,1,2#6,9,2#0,9,2#0,5,2

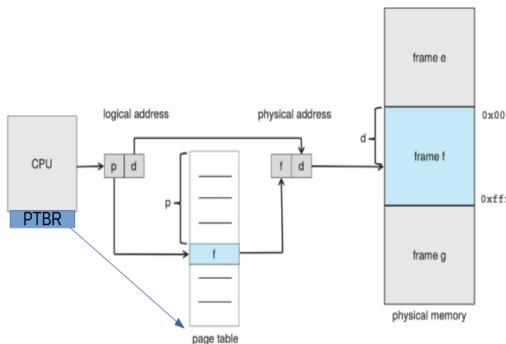
The correct answer is: 9

**Question 18**

Partially correct

Mark 0.31 out of 0.50

Consider the image given below, which explains how paging works.



**Figure 9.8** Paging hardware.

Mention whether each statement is True or False, with respect to this image.

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	The PTBR is present in the CPU as a register
<input type="radio"/>	<input checked="" type="radio"/>	The page table is indexed using frame number
<input checked="" type="radio"/>	<input type="radio"/>	The page table is indexed using page number
<input type="radio"/>	<input checked="" type="radio"/>	The locating of the page table using PTBR also involves paging translation
<input type="radio"/>	<input checked="" type="radio"/>	Size of page table is always determined by the size of RAM
<input checked="" type="radio"/>	<input type="radio"/>	The page table is itself present in Physical memory
<input checked="" type="radio"/>	<input type="radio"/>	Maximum Size of page table is determined by number of bits used for page number
<input checked="" type="radio"/>	<input type="radio"/>	The physical address may not be of the same size (in bits) as the logical address

The PTBR is present in the CPU as a register: True

The page table is indexed using frame number: False

The page table is indexed using page number: True

The locating of the page table using PTBR also involves paging translation: False

Size of page table is always determined by the size of RAM: False

The page table is itself present in Physical memory: True

Maximum Size of page table is determined by number of bits used for page number: True

The physical address may not be of the same size (in bits) as the logical address: True

**Question 19**

Correct

Mark 2.00 out of 2.00

Given below is shared memory code with two processes sharing a memory segment.

The first process sends a user input string to second process. The second capitalizes the string. Then the first process prints the capitalized version.

Fill in the blanks to complete the code.

**// First process**

```
#define SHMSZ 27

int main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s, string[128];
    key = 5679;
    if ((shmid =
        shmget
        ✓ (key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
    if ((shm =
        shmat
        ✓ (shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }
    s = shm;
    *s = '$';
    scanf("%s", string);
    strcpy(s + 1, string);
    *s =
        @
        ✓ ';' //note the quotes
    while(*s != '
        $
        ')
        sleep(1);
        printf("%s\n", s + 1);
        exit(0);
}
```

**//Second process**

```
#define SHMSZ 27

int main()
{
    int shmid;
    key_t key;
    char *shm, *s;
    int i;
    char string[128];
    key =
        5679
```

```

✓ ;
if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
    perror("shmget");
    exit(1);
}
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat");
    exit(1);
}
s =

✓ ;
while(*s != '@')
    sleep(1);
for(i = 0; i < strlen(s + 1); i++)
    s[i + 1] = toupper(s[i + 1]);
*s = '$';
exit(0);
}

```

**Question 20**

Partially correct

Mark 0.25 out of 0.50

Map the functionality/use with function/variable in xv6 code.

return a free page, if available; 0, otherwise

Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed

Array listing the kernel memory mappings, to be used by setupkvm()

Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices

Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary

Setup kernel part of a page table, and switch to that page table

 kinit1()

 mappages()

 kmap[]

 kvmalloc()

 walkpgdir()

 setupkvm()

Your answer is partially correct.

You have correctly selected 3.

The correct answer is: return a free page, if available; 0, otherwise → kalloc(), Create page table entries for a given range of virtual and physical addresses; including page directory entries if needed → mappages(), Array listing the kernel memory mappings, to be used by setupkvm() → kmap[], Setup kernel part of a page table, mapping kernel code, data, read-only data, I/O space, devices → setupkvm(), Return address of page table entry in a given page directory, for a given virtual address; creates page table if necessary → walkpgdir(), Setup kernel part of a page table, and switch to that page table → kvmalloc()

**Question 21**

Partially correct

Mark 1.53 out of 2.50

Order events in xv6 timer interrupt code

(Transition from process P1 to P2's code.)

P2 is selected and marked RUNNING

12 ✓

Change of stack from user stack to kernel stack of P1

3 ✓

Timer interrupt occurs

2 ✓

alltraps() will call iret

17 ✗

change to context of P2, P2's kernel stack in use now

13 ✓

P2's trap() will return to alltraps

16 ✗

jump in vector.S

4 ✓

P2 will return from sched() in yield()

14 ✗

yield() is called

8 ✓

trap() is called

7 ✓

Process P2 is executing

18 ✗

P1 is marked as RUNNABLE

9 ✓

P2's yield() will return in trap()

15 ✗

Process P1 is executing

1 ✓

sched() is called,

11 ✗

change to context of the scheduler, scheduler's stack in use now

10 ✗

jump to alltraps

5 ✓

Trapframe is built on kernel stack of P1

6 ✓

Your answer is partially correct.

You have correctly selected 11.

The correct answer is: P2 is selected and marked RUNNING → 12, Change of stack from user stack to kernel stack of P1 → 3, Timer interrupt occurs → 2, alltraps() will call iret → 18, change to context of P2, P2's kernel stack in use now → 13, P2's trap() will return to alltraps → 17, jump in vector.S → 4, P2 will return from sched() in yield() → 15, yield() is called → 8, trap() is called → 7, Process P2 is executing → 14, P1 is marked as RUNNABLE → 9, P2's yield() will return in trap() → 16, Process P1 is executing → 1, sched() is called, → 10, change to context of the scheduler, scheduler's stack in use now → 11, jump to alltraps → 5, Trapframe is built on kernel stack of P1 → 6

**Question 22**

Incorrect

Mark 0.00 out of 1.00

Given that the memory access time is 200 ns, probability of a page fault is 0.7 and page fault handling time is 8 ms,  
The effective memory access time in nanoseconds is:

Answer:  ✖

The correct answer is: 5600060.00

**Question 23**

Correct

Mark 0.25 out of 0.25

Select the state that is not possible after the given state, for a process:

- New:  Running ✓
- Ready :  Waiting ✓
- Running:  None of these ✓
- Waiting:  Running ✓

**Question 24**

Partially correct

Mark 0.63 out of 1.00

Select the correct statements about sched() and scheduler() in xv6 code

- a. scheduler() switches to the selected process's context ✓
- b. When either sched() or scheduler() is called, it does not return immediately to caller ✓
- c. After call to swtch() in sched(), the control moves to code in scheduler()
- d. Each call to sched() or scheduler() involves change of one stack inside swtch() ✓
- e. After call to swtch() in scheduler(), the control moves to code in sched()
- f. When either sched() or scheduler() is called, it results in a context switch ✓
- g. sched() switches to the scheduler's context ✓
- h. sched() and scheduler() are co-routines

Your answer is partially correct.

You have correctly selected 5.

The correct answers are: sched() and scheduler() are co-routines, When either sched() or scheduler() is called, it does not return immediately to caller, When either sched() or scheduler() is called, it results in a context switch, sched() switches to the scheduler's context, scheduler() switches to the selected process's context, After call to swtch() in scheduler(), the control moves to code in sched(), After call to swtch() in sched(), the control moves to code in scheduler(), Each call to sched() or scheduler() involves change of one stack inside swtch()

**Question 25**

Correct

Mark 0.25 out of 0.25

The data structure used in kalloc() and kfree() in xv6 is

- a. Doubly linked circular list
- b. Singly linked circular list
- c. Double linked NULL terminated list
- d. Singly linked NULL terminated list



Your answer is correct.

The correct answer is: Singly linked NULL terminated list

[◀ \(Assignment\) lseek system call in xv6](#)

Jump to...

Dashboard / My courses / Computer Engineering & IT / CEIT-Even-sem-20-21 / QS-Even-sem-2020-21 / 16 May - 22 May / End Sem Exam OS-2021

Started on Saturday, 22 May 2021, 8:00 AM

State Finished

Completed on Saturday, 22 May 2021, 9:30 AM

Time taken 1 hour 30 mins

Grade 26.12 out of 40.00 (65%)

Question 1

Incorrect

Mark 0.00 out of 1.00

A 4 GB disk with 1 KB of block size would require these many number of **blocks** for its free block bitmap:

Answer: 4096 ✖

The correct answer is: 512

Question 2

Correct

Mark 1.00 out of 1.00

Given that the memory access time is 110 ns, probability of a page fault is 0.5 and page fault handling time is 12 ms,

The effective memory access time in nanoseconds is:

Answer: 6000165 ✓

The correct answer is: 6000055.00

Question 3

Incorrect

Mark 0.00 out of 1.00

The maximum size of a file in number of blocks of BSIZE in xv6 code is

(write a number only)

Answer: 268 ✖

The correct answer is: 138

Question 4

Incorrect

Mark 0.00 out of 1.00

Calculate the average waiting time using

Round Robin scheduling with time quantum of 5 time units  
for the following workload

assuming that they arrive in the order written below.

Process Burst Time

P1	5
P2	7
P3	6
P4	2

Write only a number in the answer upto two decimal points.

Answer: 40.75 ✖

The correct answer is: 10.25



**Question 5**

Correct

Mark 1.00 out of 1.00

For the reference string

4 2 5 1 0 1 2 5 4 1 2

the number of page faults, including initial ones,  
with FIFO replacement and 2 frames are :

Answer: 10 ✓

4 -

4 2

5 2

5 1

0 1

-

2 1

2 5

4 5

4 1

2 1

The correct answer is: 10

**Question 6**

Correct

Mark 1.00 out of 1.00

Assuming a 16- KB page size, what is the page number for the address 428517 reference in decimal :

(give answer also in decimal)

Answer: 27 ✓

The correct answer is: 26



## Question 7

Correct

Mark 1.00 out of 1.00

In the code below assume that each function can be executed concurrently by many threads/processes.  
Ignore syntactical issues, and focus on the semantics.

This program is an example of

```
spinlock a, b; // assume initialized
thread1() {
    spinlock(b);
    //some code;
    spinlock(a);
    //some code;
    spinunlock(b);
    spinunlock(a);
}
thread2() {
    spinlock(a);
    //some code;
    spinlock(b);
    //some code;
    spinunlock(b);
    spinunlock(a);
}
```

- a. Deadlock ✓
- b. Self Deadlock
- c. None of these
- d. Deadlock or livelock depending on actual race
- e. Livelock

Your answer is correct.

The correct answer is: Deadlock



**Question 8**

Partially correct

Mark 1.33 out of 2.00

Match the snippets of xv6 code with the core functionality they achieve, or problems they avoid.  
"..." means some code.

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
    uint result;

    // The + in "+m" denotes a read-modify-write operand.
    asm volatile("lock; xchgl %0, %1" :
                "+m" ("*addr), "=a" (result) :
                "1" (newval) :
                "cc");
    return result;
}
```

Atomic compare and swap instruction (to be expanded inline into code)



```
void
sleep(void *chan, struct spinlock *lk)
{
    ...
    if(lk != &ptable.lock){
        acquire(&ptable.lock);
        release(lk);
    }
}
```

If you don't do this, a process may be running on two processors parallelly



```
void
acquire(struct spinlock *lk)
{
    ...
    __sync_synchronize();
}
```

Tell compiler not to reorder memory access beyond this line



Your answer is partially correct.

You have correctly selected 2.

The correct answer is: static inline uint  
xchg(volatile uint \*addr, uint newval)

```
{
    uint result;
```

```
// The + in "+m" denotes a read-modify-write operand.
asm volatile("lock; xchgl %0, %1" :
            "+m" ("*addr), "=a" (result) :
            "1" (newval) :
            "cc");
return result;
} → Atomic compare and swap instruction (to be expanded inline into code), void
sleep(void *chan, struct spinlock *lk)
{
    ...
    if(lk != &ptable.lock){
        acquire(&ptable.lock);
        release(lk);
    } → Avoid a self-deadlock, void
    acquire(struct spinlock *lk)
{
    ...
    __sync_synchronize(); → Tell compiler not to reorder memory access beyond this line
```

**Question 9**

Correct

Mark 1.00 out of 1.00

Predict the output of the program given here.

Assume that all the path names for the programs are correct. For example "/usr/bin/echo" will actually run echo command.

Assume that there is no mixing of print output on screen if two of them run concurrently.

In the answer replace a new line by a single space.

For example::

good

output

should be written as good output

--

```
main() {  
    int i;  
    i = fork();  
    if(i == 0)  
        execl("/usr/bin/echo", "/usr/bin/echo", "hi", 0);  
    else  
        wait(0);  
    fork();  
    execl("/usr/bin/echo", "/usr/bin/echo", "one", 0);  
}
```

Answer: hi one one 

The correct answer is: hi one one

**Question 10**

Partially correct

Mark 1.67 out of 2.00

Select all the blocks that may need to be written back to disk (if updated, of-course), as "Yes", when an operation of deleting a file is carried out on ext2 file system.

An option has to be correct entirely to be marked "Yes"

Superblock

Yes 

One or multiple data blocks of the parent directory

No 

One or more data bitmap blocks for the parent directory

No 

Block bitmap(s) for all the blocks of the file

No 

Possibly one block bitmap corresponding to the parent directory

Yes 

Data blocks of the file

No 

Your answer is partially correct.

only one data block of parent directory. multiple blocks not possible. an entry is always contained within one single block

You have correctly selected 5.

The correct answer is: Superblock → Yes, One or multiple data blocks of the parent directory → No, One or more data bitmap blocks for the parent directory → No, Block bitmap(s) for all the blocks of the file → Yes, Possibly one block bitmap corresponding to the parent directory → Yes, Data blocks of the file → No

**Question 11**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about bootloader.

Every wrong selection will deduct marks proportional to  $1/n$  where n is total wrong choices in the question.

You will get minimum a zero.

- a. Modern Bootloaders often allow configuring the way an OS boots
- b. Bootloaders allow selection of OS to boot from
- c. Bootloader must be one sector in length
- d. The bootloader loads the BIOS
- e. LILO is a bootloader



Your answer is correct.

The correct answers are: LILO is a bootloader, Modern Bootloaders often allow configuring the way an OS boots, Bootloaders allow selection of OS to boot from



## Question 12

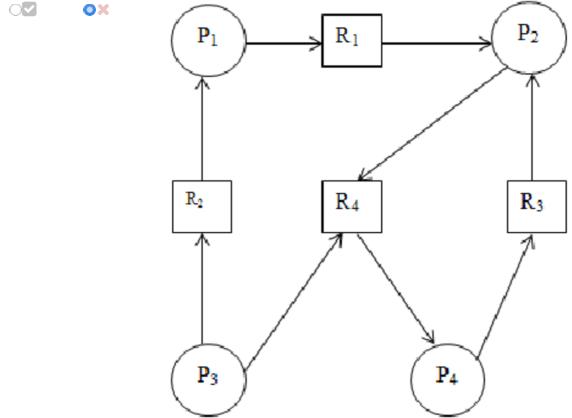
Incorrect

Mark 0.00 out of 1.00

For each of the resource allocation diagram shown,  
infer whether the graph contains at least one deadlock or not.

Yes

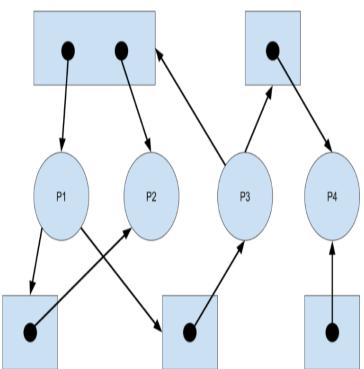
No



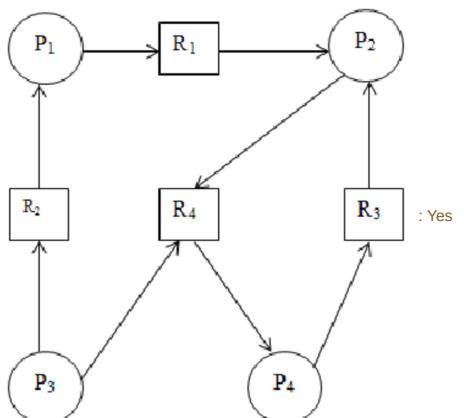
✗

✗

✓

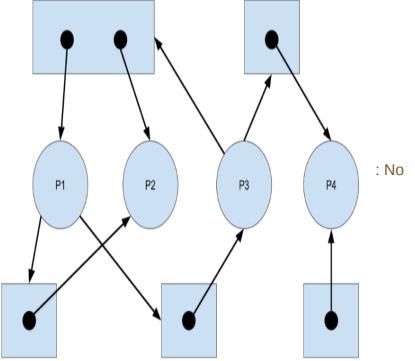


✗



: Yes





## Question 13

Partially correct

Mark 0.71 out of 1.00

Mark the statements about device drivers by marking as True or False.

True	False	
<input checked="" type="radio"/>	<input type="radio"/> ✘	It's possible that a particular hardware has multiple device drivers available for it.
<input checked="" type="radio"/>	<input type="radio"/> ✘	xv6 has device drivers for IDE disk and console.
<input checked="" type="radio"/>	<input type="radio"/> ✘	A disk driver converts OS's logical view of disk into physical locations on disk.
<input checked="" type="radio"/>	<input type="radio"/> ✘	A device driver code is specific to a hardware device
<input checked="" type="radio"/>	<input type="radio"/> ✘	All devices of the same type (e.g. 2 hard disks) can typically use the same device driver
<input checked="" type="radio"/>	<input type="radio"/> ✘	Writing a device driver mandatorily demands reading the technical documentation about the hardware.
<input type="radio"/> ✘	<input checked="" type="radio"/>	Device driver is an intermediary between the end-user and OS

It's possible that a particular hardware has multiple device drivers available for it.: True

xv6 has device drivers for IDE disk and console.: True

A disk driver converts OS's logical view of disk into physical locations on disk.: True

A device driver code is specific to a hardware device: True

All devices of the same type (e.g. 2 hard disks) can typically use the same device driver: True

Writing a device driver mandatorily demands reading the technical documentation about the hardware.: True

Device driver is an intermediary between the end-user and OS: False

**Question 14**

Partially correct

Mark 0.33 out of 1.00

Consider this program.

Some statements are identified using the // comment at the end.

Assume that `=` is an atomic operation.

```
#include <stdio.h>
#include <pthread.h>
long c = 0, c1 = 0, c2 = 0, run = 1;
void *thread1(void *arg) {
    while(run == 1) { //E
        c = 10; //A
        c1 = c2 + 5; //B
    }
}
void *thread2(void *arg) {
    while(run == 1) { //F
        c = 20; //C
        c2 = c1 + 3; //D
    }
}
int main() {
    pthread_t th1, th2;
    pthread_create(&th1, NULL, thread1, NULL);
    pthread_create(&th2, NULL, thread2, NULL);
    sleep(2);
    run = 0;
    printf(stdout, "c = %ld c1+c2 = %ld c1 = %ld c2 = %ld \n", c, c1+c2, c1, c2);
    fflush(stdout);
}
```

Which statements are part of the critical Section?

Yes	No	
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	F
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/> <input type="checkbox"/>	D
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	C
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	A
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	<input checked="" type="radio"/> <input type="checkbox"/>	B
<input checked="" type="radio"/> <input checked="" type="checkbox"/>	<input type="radio"/> <input checked="" type="checkbox"/>	E

F: No

D: Yes

C: No

A: No

B: Yes

E: No

**Question 15**

Partially correct

Mark 1.43 out of 2.00

Mark statements as T/F

All statements are in the context of preventing deadlocks.

**True****False**

<input checked="" type="radio"/>	<input type="radio"/>	A process holding one resources and waiting for just one more resource can also be involved in a deadlock.	✓
<input type="radio"/>	<input checked="" type="radio"/>	If a resource allocation graph contains a cycle then there is a guarantee of a deadlock	✗
<input type="radio"/>	<input checked="" type="radio"/>	The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order	✗
<input checked="" type="radio"/>	<input type="radio"/>	Circular wait is avoided by enforcing a lock ordering	✓
<input checked="" type="radio"/>	<input type="radio"/>	Hold and wait means a thread/process holding some locks and waiting for acquiring some.	✓
<input checked="" type="radio"/>	<input type="radio"/>	Deadlock is possible if all the conditions are met at the same time: Mutual exclusion, hold and wait, no pre-emption, circular wait.	✓
<input checked="" type="radio"/>	<input type="radio"/>	Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens	✓

A process holding one resources and waiting for just one more resource can also be involved in a deadlock.: True

If a resource allocation graph contains a cycle then there is a guarantee of a deadlock: False

The lock ordering to be followed to avoid circular wait is a code in OS that checks for compliance with decided order: False

Circular wait is avoided by enforcing a lock ordering: True

Hold and wait means a thread/process holding some locks and waiting for acquiring some.: True

Deadlock is possible if all the conditions are met at the same time: Mutual exclusion, hold and wait, no pre-emption, circular wait.: True

Mutual exclusion is a necessary condition for deadlock because it brings in locks on which deadlock happens: True

**Question 16**

Correct

Mark 1.00 out of 1.00

Match the left side use(or non-use) of a synchronization primitive with the best option on the right side.

This is the smallest primitive made available in software, using the hardware provided atomic instructions

 spinlock ✓

This tool is useful for event-wait scenarios

 semaphore ✓

This tool is more useful on multiprocessor systems

 spinlock ✓

This tool is quite attractive in solving the main bounded buffer problem

 semaphore ✓

This tool is very useful for waiting for 'something'

 condition variables ✓

Your answer is correct.

The correct answer is: This is the smallest primitive made available in software, using the hardware provided atomic instructions → spinlock, This tool is useful for event-wait scenarios → semaphore, This tool is more useful on multiprocessor systems → spinlock, This tool is quite attractive in solving the main bounded buffer problem → semaphore, This tool is very useful for waiting for 'something' → condition variables

**Question 17**

Correct

Mark 1.00 out of 1.00

The permissions -rwx--x--x on a file mean

- a. The file can be read only by the owner
- b. 'cat' on the file by owner will not work
- c. 'cat' on the file by any user will work
- d. 'rm' on the file by any user will work
- e. The file can be executed by anyone
- f. The file can be written only by the owner



Your answer is correct.

The correct answers are: The file can be executed by anyone, The file can be read only by the owner, The file can be written only by the owner, 'rm' on the file by any user will work

**Question 18**

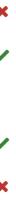
Incorrect

Mark 0.00 out of 1.00

Note: for this question you get full marks if you select all and only correct options, you get ZERO if at least one option is wrong or not selected.

Select all the correct statements about log structured file systems.

- a. a transaction is said to be committed when all operations are written to file system
- b. log may be kept on same block device or another block device
- c. file system recovery may end up losing data
- d. even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery
- e. file system recovery recovers all the lost data



Your answer is incorrect.

The correct answers are: file system recovery may end up losing data, log may be kept on same block device or another block device, even if file systems followed immediate writes (i.e. non-delayed writes), it could still require recovery

**Question 19**

Incorrect

Mark 0.00 out of 1.00

Consider the structure of directory entry in ext2, as shown in this diagram.

	inode	rec_len	file_type	name_len	name
0	21	12	1	2	.
12	22	12	2	2	.
24	53	16	5	2	h o m e
40	67	28	3	2	u s r
52	0	16	7	1	o l d f i l e
68	34	12	4	2	s b i n

Select the correct statements about the directory entry in ext2 file system.

The correct formula for rec\_len is (when entries are continuously stored)

- a.  $\text{rec\_len} = \text{sizeof(inode entry)} + \text{sizeof(name len entry)} + \text{sizeof(file type entry)} + (\text{strlen(name)} + (-1) * (\text{strlen(name)} \% 4))$
- b.  $\text{rec\_len} = \text{sizeof(inode entry)} + \text{sizeof(name len entry)} + \text{sizeof(file type entry)} + (\text{strlen(name)} + (\text{strlen(name)} - 4) \% 4)$
- c.  $\text{rec\_len} = \text{sizeof(inode entry)} + \text{sizeof(name len entry)} + \text{sizeof(file type entry)} + (\text{strlen(name)} + 4 - (\text{strlen(name)} \% 4))$  ✗
- d.  $\text{rec\_len} = \text{sizeof(inode entry)} + \text{sizeof(name len entry)} + \text{sizeof(file type entry)} + (\text{strlen(name)} + (-1) * (\text{strlen(name)} - 4))$
- e.  $\text{rec\_len} = \text{sizeof(inode entry)} + \text{sizeof(name len entry)} + \text{sizeof(file type entry)} + (\text{strlen(name)} \% 4)$
- f.  $\text{rec\_len} = \text{sizeof(inode entry)} + \text{sizeof(name len entry)} + \text{sizeof(file type entry)} + \text{strlen(name)}$

Your answer is incorrect.

The correct answer is:  $\text{rec\_len} = \text{sizeof(inode entry)} + \text{sizeof(name len entry)} + \text{sizeof(file type entry)} + (\text{strlen(name)} + (-1) * (\text{strlen(name)} - 4))$

**Question 20**

Partially correct

Mark 0.50 out of 1.00

Mark whether the given sequence of events is possible or not-possible. Also, select the reason for your answer.

For each sequence it's a not-possible sequence if some important event is not mentioned in the sequence.

Assume that the kernel code is non-interruptible and uniprocessor system.

Process P1 executing a system call  
 Timer interrupt  
 Generic interrupt handler runs  
 Scheduler runs  
 Scheduler selects P2 for execution  
 P2 returns from timer interrupt handler  
 Process p2, user code executing

This sequence of events is:  ✓

Because

✗

**Question 21**

Incorrect

Mark 0.00 out of 1.00

The given semaphore implementation faces which problem?

Assume any suitable code for signal()

Note: blocks means waits in a wait queue.

```
struct semaphore {  
    int val;  
    spinlock lk;  
};  
sem_init(semaphore *s, int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0)  
        ;  
    (s->val)--;  
    spinunlock(&(s->sl));  
}
```

- a. blocks holding a spinlock
- b. deadlock
- c. too much spinning, bounded wait not guaranteed
- d. not holding lock after unblock



Your answer is incorrect.

The correct answer is: deadlock



## Question 22

Partially correct

Mark 0.80 out of 1.00

Mark statements True/False w.r.t. change of states of a process.

Reference: The process state diagram (and your understanding of how kernel code works)

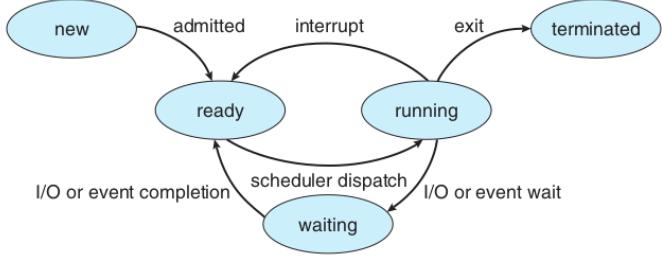


Figure 3.2 Diagram of process state.

True

False

<input type="radio"/> ✗	<input checked="" type="checkbox"/> ✖	A process in RUNNING state only can become TERMINATED because scheduler moves it to ZOMBIE state	✓
<input checked="" type="checkbox"/> ✖	<input type="radio"/> ✗	A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.	✗
<input checked="" type="checkbox"/> ✖	<input type="radio"/> ✗	A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred	✓
<input checked="" type="checkbox"/> ✖	<input type="radio"/> ✗	Every process has to go through ZOMBIE state, at least for a small duration.	✓
<input checked="" type="checkbox"/> ✖	<input type="radio"/> ✗	Only a process in READY state is considered by scheduler	✓

A process in RUNNING state only can become TERMINATED because scheduler moves it to ZOMBIE state: False

A process in READY state can not go to WAITING state because the resource on which it will WAIT will not be in use when process is in READY state.: False

A process in WAITING state can not become RUNNING because the event it's waiting for has not occurred: True

Every process has to go through ZOMBIE state, at least for a small duration.: True

Only a process in READY state is considered by scheduler: True

**Question 23**

Correct

Mark 1.00 out of 1.00

Select T/F for statements about Volume Managers.

Do pay attention to the use of the words physical partition and physical volume.

**True      False**

<input checked="" type="radio"/>	<input type="radio"/> ✗	The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.	<input checked="" type="checkbox"/>
<input checked="" type="radio"/>	<input type="radio"/> ✗	A logical volume can be extended in size but upto the size of volume group	<input checked="" type="checkbox"/>
<input checked="" type="radio"/>	<input type="radio"/> ✗	A logical volume may span across multiple physical volumes	<input checked="" type="checkbox"/>
<input checked="" type="radio"/>	<input type="radio"/> ✗	The volume manager stores additional metadata on the physical disk partitions	<input checked="" type="checkbox"/>
<input checked="" type="radio"/>	<input type="radio"/> ✗	A physical partition should be initialized as a physical volume, before it can be used by volume manager.	<input checked="" type="checkbox"/>
<input checked="" type="radio"/>	<input type="radio"/> ✗	A volume group consists of multiple physical volumes	<input checked="" type="checkbox"/>
<input checked="" type="radio"/>	<input type="radio"/> ✗	A logical volume may span across multiple physical partitions	<input checked="" type="checkbox"/> since a physical volume is made up of physical partitions, and a volume can span across multiple PVs, it can also span across multiple PP

The volume manager can create further internal sub-divisions of a physical partition for efficiency or features.: True

A logical volume can be extended in size but upto the size of volume group: True

A logical volume may span across multiple physical volumes: True

The volume manager stores additional metadata on the physical disk partitions: True

A physical partition should be initialized as a physical volume, before it can be used by volume manager.: True

A volume group consists of multiple physical volumes: True

A logical volume may span across multiple physical partitions: True

**Question 24**

Correct

Mark 1.00 out of 1.00

Map the block allocation scheme with the problem it suffers from

(Match pairs 1-1, match a scheme with the problem that it suffers from relatively the most, compared to others)

Continuous allocation	need for compaction	<input checked="" type="checkbox"/>
Linked allocation	Too many seeks	<input checked="" type="checkbox"/>
Indexed Allocation	Overhead of reading metadata blocks	<input checked="" type="checkbox"/>

Your answer is correct.

The correct answer is: Continuous allocation → need for compaction, Linked allocation → Too many seeks, Indexed Allocation → Overhead of reading metadata blocks

**Question 25**

Correct

Mark 1.00 out of 1.00

This one is not a system call:

- a. open
- b. read
- c. write
- d. scheduler



Your answer is correct.

The correct answer is: scheduler



**Question 26**

Correct

Mark 1.00 out of 1.00

Match the pairs.

This question is based on your general knowledge about operating systems/related concepts and their features.

Java threads	monitors,re-entrant locks, semaphores	✓
Linux threads	atomic-instructions, spinlocks, etc.	✓
POSIX threads	semaphore, mutex, condition variables	✓

Your answer is correct.

The correct answer is: Java threads → monitors,re-entrant locks, semaphores, Linux threads → atomic-instructions, spinlocks, etc., POSIX threads → semaphore, mutex, condition variables

**Question 27**

Correct

Mark 1.00 out of 1.00

Consider the following list of free chunks, in continuous memory management:

7k, 15k, 21k, 14k, 19k, 6k

Suppose there is a request for chunk of size 5k, then the free chunk selected under each of the following schemes will be

Best fit:	6k	✓
First fit:	7k	✓
Worst fit:	21k	✓

**Question 28**

Correct

Mark 1.00 out of 1.00

This one is not a scheduling algorithm

- a. Round Robin
- b. SJF
- c. Mergesort
- d. FCFS



Your answer is correct.

The correct answer is: Mergesort

## Question 29

Correct

Mark 1.00 out of 1.00

Mark whether the concept is related to scheduling or not.

Yes	No	
<input checked="" type="radio"/>	<input type="radio"/>	timer interrupt
<input checked="" type="radio"/>	<input type="radio"/>	context-switch
<input checked="" type="radio"/>	<input type="radio"/>	ready-queue
<input type="radio"/>	<input checked="" type="radio"/>	file-table
<input checked="" type="radio"/>	<input type="radio"/>	runnable process

timer interrupt: Yes

context-switch: Yes

ready-queue: Yes

file-table: No

runnable process: Yes



**Question 30**

Partially correct

Mark 1.00 out of 2.00

Map ext2 data structure features with their purpose

**Many copies of Superblock** Choose...**Free blocks count in superblock and group descriptor**

Redundancy to ensure the most crucial data structure is not lost

**Used directories count in group descriptor**

is redundant and helps do calculations of directory entries faster

**Combining file type and access rights in one variable**

saves 1 byte of space

**rec\_len field in directory entry**

Try to keep all the data of a directory and its file close together in a group

**File Name is padded**

aligns all memory accesses on word boundary, improving performance

**Inode bitmap is one block**

limits total number of files that can belong to a group

**Block bitmap is one block**

Limits the size of a block group, thus improvising on purpose of a group

**Mount count in superblock**

to enforce file check after certain amount of mounts at boot time

**Inode table location in Group Descriptor**

is redundant and helps do calculations of directory entries faster

**Inode table**

All inodes are kept together so that one disk read leads to reading many inodes together, effectively doing a buffering of subsequent inode reads, and to save space on disk

**A group**

Redundancy to ensure the most crucial data structure is not lost



Your answer is partially correct.

You have correctly selected 6.

The correct answer is: **Many copies of Superblock** → Redundancy to ensure the most crucial data structure is not lost, **Free blocks count in superblock and group descriptor** → Redundancy to help fsck restore consistency, **Used directories count in group descriptor** → attempt is made to evenly spread the first-level directories, this count is used there, **Combining file type and access rights in one variable** → saves 1 byte of space, **rec\_len field in directory entry** → allows holes and linking of entries in directory, File Name is padded → aligns all memory accesses on word boundary, improving performance, **Inode bitmap is one block** → limits total number of files that can belong to a group, **Block bitmap is one block** → Limits the size of a block group, thus improvising on purpose of a group, **Mount count in superblock** → to enforce file check after certain amount of mounts at boot time, **Inode table location in Group Descriptor** → Obvious, as it's per group and not per file-system, **Inode table** → All inodes are kept together so that one disk read leads to reading many inodes together, effectively doing a buffering of subsequent inode reads, and to save space on disk, **A group** → Try to keep all the data of a directory and its file close together in a group

**Question 31**

Partially correct

Mark 1.85 out of 2.00

Mark True/False

Statements about scheduling and scheduling algorithms

True	False	
<input checked="" type="radio"/>	<input type="radio"/>	The nice() system call is used to set priorities for processes
<input checked="" type="radio"/>	<input type="radio"/>	Aging is used to ensure that low-priority processes do not starve in priority scheduling.
<input type="radio"/>	<input checked="" type="radio"/>	In non-pre-emptive priority scheduling, the highest priority process is scheduled and runs until it gives up CPU.
<input checked="" type="radio"/>	<input type="radio"/>	xv6 code does not care about Processor Affinity
<input checked="" type="radio"/>	<input type="radio"/>	In pre-emptive priority scheduling, priority is implemented by assigning more time quantum to higher priority process.
<input checked="" type="radio"/>	<input type="radio"/>	A scheduling algorithm is non-preemptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.
<input checked="" type="radio"/>	<input type="radio"/>	Processor Affinity refers to memory accesses of a process being stored on cache of that processor
<input checked="" type="radio"/>	<input type="radio"/>	Response time will be quite poor on non-interruptible kernels
<input checked="" type="radio"/>	<input type="radio"/>	Shortest Remaining Time First algorithm is nothing but pre-emptive Shortest Job First algorithm
<input checked="" type="radio"/>	<input type="radio"/>	On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread
<input checked="" type="radio"/>	<input type="radio"/>	Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.
<input checked="" type="radio"/>	<input type="radio"/>	Pre-emptive scheduling leads to many race conditions in kernel code.
<input checked="" type="radio"/>	<input type="radio"/>	Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.

The nice() system call is used to set priorities for processes.: True

Aging is used to ensure that low-priority processes do not starve in priority scheduling.: True

In non-pre-emptive priority scheduling, the highest priority process is scheduled and runs until it gives up CPU.: True

xv6 code does not care about Processor Affinity: True

In pre-emptive priority scheduling, priority is implemented by assigning more time quantum to higher priority process.: True

A scheduling algorithm is non-preemptive if it does context switch only if a process voluntarily relinquishes CPU or it terminates.: True

Processor Affinity refers to memory accesses of a process being stored on cache of that processor: True

Response time will be quite poor on non-interruptible kernels: True

Shortest Remaining Time First algorithm is nothing but pre-emptive Shortest Job First algorithm: True

On Linuxes the CPU utilisation is measured as the time spent in scheduling the idle thread: True

Generally the voluntary context switches are much more than non-voluntary context switches on a Linux system.: True

Pre-emptive scheduling leads to many race conditions in kernel code.: True

Statistical observations tell us that most processes have large number of small CPU bursts and relatively smaller numbers of large CPU bursts.: True

**Question 32**

Partially correct

Mark 1.17 out of 2.00

The unix file semantics demand that changes to any open file are visible immediately to any other processes accessing that file at that point in time.

Select the data-structure/programmatic features that ensure the implementation of unix semantics. (Assume there is no mmap())

Yes	No	
<input type="radio"/> <input checked="" type="checkbox"/>	All processes accessing the same file share the file descriptor among themselves	✓
<input type="radio"/> <input checked="" type="checkbox"/>	The pointer entry in the file descriptor array entry points to the data of the file directly	✓
<input checked="" type="checkbox"/> <input type="radio"/>	There is only one global file structure per on-disk file.	✗
<input type="radio"/> <input checked="" type="checkbox"/>	All file accesses are made using only global variables	✓
<input checked="" type="checkbox"/> <input type="radio"/>	The 'file offset' is shared among all the processes that access the file.	✗
<input type="radio"/> <input checked="" type="checkbox"/>	No synchronization is implemented so that changes are made available immediately.	✓
<input checked="" type="checkbox"/> <input type="radio"/>	A single spinlock is to be used to protect the unique global 'file structure' representing the file, thus synchronizing access, and making other processes wait for earlier process to finish writing so that writes get visible immediately.	✗
<input checked="" type="checkbox"/> <input type="radio"/>	There is only one in-memory copy of the on disk file's contents in kernel memory/buffers	✓
<input checked="" type="checkbox"/> <input type="radio"/>	The file descriptors in every PCB are pointers to the same global file structure.	✗
<input type="radio"/> <input checked="" type="checkbox"/>	The file descriptor array is external to PCB and all processes that share a file, have pointers to same file-descriptors' array	✓
<input checked="" type="checkbox"/> <input type="radio"/>	All file structures representing any open file, give access to the same in-memory copy of the file's contents	✓
<input checked="" type="checkbox"/> <input type="radio"/>	The 'file offset' index is stored outside the file-structure to which file-descriptor array points	✗

All processes accessing the same file share the file descriptor among themselves: No

The pointer entry in the file descriptor array entry points to the data of the file directly: No

There is only one global file structure per on-disk file.: No

All file accesses are made using only global variables: No

The 'file offset' is shared among all the processes that access the file.: No

No synchronization is implemented so that changes are made available immediately.: No

A single spinlock is to be used to protect the unique global 'file structure' representing the file, thus synchronizing access, and making other processes wait for earlier process to finish writing so that writes get visible immediately.: No

There is only one in-memory copy of the on disk file's contents in kernel memory/buffers: Yes

The file descriptors in every PCB are pointers to the same global file structure.: No

The file descriptor array is external to PCB and all processes that share a file, have pointers to same file-descriptors' array: No

All file structures representing any open file, give access to the same in-memory copy of the file's contents: Yes

The 'file offset' index is stored outside the file-structure to which file-descriptor array points: No

**Question 33**

Partially correct

Mark 0.33 out of 2.00

Map the function in xv6's file system code, to its perceived logical layer.

namei	inode	✗
filestat()	Choose...	
dirlookup	directory	✓
ialloc	file descriptor	✗
stati	Choose...	
ideintr	buffer cache	✗
bread	Choose...	
balloc	file descriptor	✗
sys_chdir()	system call	✓
skipelem	system call	✗
commit	system call	✗
bmap	system call	✗

Your answer is partially correct.

You have correctly selected 2.

The correct answer is: namei → pathname lookup, filestat() → file descriptor, dirlookup → directory, ialloc → inode, stati → inode, ideintr → disk driver, bread → buffer cache, balloc → block allocation on disk, sys\_chdir() → system call, skipelem → pathname lookup, commit → logging, bmap → inode

[◀ Course Exit Feedback](#)[Jump to...](#)[xv6-public-master ►](#)

**Started on** Monday, 24 January 2022, 7:07:42 PM

**State** Finished

**Completed on** Monday, 24 January 2022, 8:08:11 PM

**Time taken** 1 hour

**Grade** 8.90 out of 20.00 (45%)

#### Question 1

Complete

Mark 0.80 out of 1.00

Match the register with the segment used with it.

ebp	ss
eip	cs
edi	ds
esp	ss
esi	ds

The correct answer is: ebp → ss, eip → cs, edi → es, esp → ss, esi → ds

#### Question 2

Complete

Mark 1.00 out of 1.00

```
int value = 5;
int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0) { /* child process */
        value += 15;
        return 0;
    }
    else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("%d", value); /* LINE A */
    }
    return 0;
}
```

What's the value printed here at LINE A?

Answer:

The correct answer is: 5

**Question 3**

Complete

Mark 0.50 out of 0.50

Is the command "cat README > done &" possible on xv6? (Note the & in the end)

- a. no
- b. yes

The correct answer is: yes

**Question 4**

Complete

Mark 0.00 out of 2.00

xv6.img: bootblock kernel

```
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

Consider above lines from the Makefile. Which of the following is incorrect?

- a. The xv6.img is the virtual disk that is created by combining the bootblock and the kernel file.
- b. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk.
- c. The size of xv6.img is exactly = (size of bootblock) + (size of kernel)
- d. xv6.img is the virtual processor used by the qemu emulator
- e. The size of the kernel file is nearly 5 MB
- f. Blocks in xv6.img after kernel may be all zeroes.
- g. The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies 10,000 blocks on the disk.
- h. The kernel is located at block-1 of the xv6.img
- i. The bootblock is located on block-0 of the xv6.img
- j. The bootblock may be 512 bytes or less (looking at the Makefile instruction)
- k. The size of the xv6.img is nearly 5 MB

The correct answers are: xv6.img is the virtual processor used by the qemu emulator, The xv6.img is of the size 10,000 blocks of 512 bytes each and occupies upto 10,000 blocks on the disk., The size of the kernel file is nearly 5 MB, The size of xv6.img is exactly = (size of bootblock) + (size of kernel)

**Question 5**

Complete

Mark 0.43 out of 1.00

Rank the following storage systems from slowest (first) to fastest(last)

You can drag and drop the items below/above each other.

Registers
Cache
Main memory
Nonvolatile memory
Magnetic tapes
Optical disk
Hard-disk drives

The correct order for these items is as follows:

1. Magnetic tapes
2. Optical disk
3. Hard-disk drives
4. Nonvolatile memory
5. Main memory
6. Cache
7. Registers

**Question 6**

Complete

Mark 1.00 out of 1.00

How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security) ?

Select one:

- a. It disallows hardware interrupts when a process is running
- b. It prohibits one process from accessing other process's memory
- c. It prohibits a user mode process from running privileged instructions
- d. It prohibits invocation of kernel code completely, if a user program is running

The correct answer is: It prohibits a user mode process from running privileged instructions

**Question 7**

Complete

Mark 0.00 out of 2.00

Which of the following are NOT a part of job of a typical compiler?

- a. Suggest alternative pieces of code that can be written
- b. Check the program for syntactical errors
- c. Check the program for logical errors
- d. Convert high level language code to machine code
- e. Process the # directives in a C program
- f. Invoke the linker to link the function calls with their code, extern globals with their declaration

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

**Question 8**

Complete

Mark 0.00 out of 2.00

Match the program with its output (ignore newlines in the output. Just focus on the count of the number of 'hi')

main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }

hi hi

main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }

hi hi hi

main() { int i = NULL; fork(); printf("hi\n"); }

No output

main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); }

hi

The correct answer is: main() { execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi, main() { int i = fork(); if(i == 0) execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi, main() { int i = NULL; fork(); printf("hi\n"); } → hi hi, main() { fork(); execl("/usr/bin/echo", "/usr/bin/echo", "hi\n", NULL); } → hi hi

**Question 9**

Complete

Mark 0.83 out of 2.00

Select all statements that correctly explain the use/purpose of system calls.

Select one or more:

- a. Provide an environment for process creation
- b. Switch from user mode to kernel mode
- c. Handle ALL types of interrupts
- d. Handle exceptions like division by zero
- e. Run each instruction of an application program
- f. Allow I/O device access to user processes
- g. Provide services for accessing files

The correct answers are: Switch from user mode to kernel mode, Provide services for accessing files, Allow I/O device access to user processes, Provide an environment for process creation

**Question 10**

Complete

Mark 0.50 out of 0.50

Compare multiprogramming with multitasking

- a. A multiprogramming system is not necessarily multitasking
- b. A multitasking system is not necessarily multiprogramming

The correct answer is: A multiprogramming system is not necessarily multitasking

**Question 11**

Complete

Mark 0.60 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

- a. The two modes are essential for a multitasking system
- b. Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode
- c. The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously
- d. The two modes are essential for a multiprogramming system
- e. There is an instruction like 'iret' to return from kernel mode to user mode

The correct answers are: The two modes are essential for a multiprogramming system, The two modes are essential for a multitasking system, There is an instruction like 'iret' to return from kernel mode to user mode, The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously, Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode

**Question 12**

Complete

Mark 0.50 out of 0.50

Is the terminal a part of the kernel on GNU/Linux systems?

- a. no
- b. yes

The correct answer is: no

**Question 13**

Complete

Mark 1.00 out of 1.00

Why should a program exist in memory before it starts executing ?

- a. Because the hard disk is a slow medium
- b. Because the processor can run instructions and access data only from memory
- c. Because the variables of the program are stored in memory
- d. Because the memory is volatile

The correct answer is: Because the processor can run instructions and access data only from memory

**Question 14**

Complete

Mark 1.33 out of 2.00

Which of the following instructions should be privileged?

Select one or more:

- a. Read the clock.
- b. Access memory management unit of the processor
- c. Access I/O device.
- d. Turn off interrupts.
- e. Set value of a memory location
- f. Set value of timer.
- g. Access a general purpose register
- h. Modify entries in device-status table
- i. Switch from user to kernel mode.

The correct answers are: Set value of timer., Access memory management unit of the processor, Turn off interrupts., Modify entries in device-status table, Access I/O device., Switch from user to kernel mode.

**Question 15**

Complete

Mark 0.07 out of 2.00

Select all the correct statements about calling convention on x86 32-bit.

- a. The ebp pointers saved on the stack constitute a chain of activation records
- b. The return value is either stored on the stack or returned in the eax register
- c. The two lines in the beginning of each function, "push %ebp; mov %esp, %ebp", create space for local variables
- d. Parameters may be passed in registers or on stack
- e. Return address is one location above the ebp
- f. Parameters may be passed in registers or on stack
- g. Compiler may allocate more memory on stack than needed
- h. Space for local variables is allocated by subtracting the stack pointer inside the code of the called function
- i. Parameters are pushed on the stack in left-right order
- j. during execution of a function, ebp is pointing to the old ebp
- k. Space for local variables is allocated by subtracting the stack pointer inside the code of the caller function

The correct answers are: Compiler may allocate more memory on stack than needed, Parameters may be passed in registers or on stack, Parameters may be passed in registers or on stack, Return address is one location above the ebp, during execution of a function, ebp is pointing to the old ebp, Space for local variables is allocated by subtracting the stack pointer inside the code of the called function, The ebp pointers saved on the stack constitute a chain of activation records

**Question 16**

Complete

Mark 0.33 out of 0.50

Order the following events in boot process (from 1 onwards)

Shell	3
BIOS	1
Init	4
OS	6
Login interface	5
Boot loader	2

The correct answer is: Shell → 6, BIOS → 1, Init → 4, OS → 3, Login interface → 5, Boot loader → 2

[◀ \(Task\) Compulsory xv6 task](#)

Jump to...

[\(Optional Assignment\) Shell Programming\(Conformance tests\) ▶](#)

**Started on** Wednesday, 9 February 2022, 7:00:12 PM

**State** Finished

**Completed on** Wednesday, 9 February 2022, 7:46:38 PM

**Time taken** 46 mins 26 secs

**Grade** 3.00 out of 11.00 (27%)

**Question 1**

Complete

Mark 0.00 out of 1.00

The number of GDT entries setup during boot process of xv6 is

- a. 2
- b. 3
- c. 0
- d. 256
- e. 4
- f. 255

The correct answer is: 3

**Question 2**

Complete

Mark 0.00 out of 1.00

x86 provides which of the following type of memory management options?

- a. segmentation and one level paging
- b. segmentation or one or two level paging
- c. segmentation and two level paging
- d. segmentation or paging
- e. segmentation and one or two level paging
- f. segmentation only

The correct answer is: segmentation and one or two level paging

**Question 3**

Complete

Mark 0.00 out of 1.00

which of the following is not a difference between real mode and protected mode

- a. in real mode general purpose registers are 16 bit, in protected mode they are 32 bit
- b. in real mode the addressable memory is more than in protected mode
- c. in real mode the addressable memory is less than in protected mode
- d. in real mode the segment is multiplied by 16, in protected mode segment is used as index in GDT
- e. processor starts in real mode

The correct answer is: in real mode the addressable memory is more than in protected mode

**Question 4**

Complete

Mark 0.00 out of 1.00

The kernel ELF file contains how many Program headers?

- a. 4
- b. 2
- c. 3
- d. 9
- e. 10

The correct answer is: 3

**Question 5**

Not answered

Marked out of 0.50

code line, MMU setting: Match the line of xv6 code with the MMU setup employed

Answer:

The correct answer is: inb \$0x64,%al

**Question 6**

Complete

Mark 1.00 out of 1.00

The kernel is loaded at Physical Address

- a. 0x000100000
- b. 0x00010000
- c. 0x80100000
- d. 0x800000000

The correct answer is: 0x000100000

**Question 7**

Complete

Mark 0.00 out of 1.00

Why is the code of entry() in Assembly and not in C?

- a. Because the symbol entry() is inside the ELF file
- b. Because the kernel code must begin in assembly
- c. Because it needs to setup paging
- d. There is no particular reason, it could also be in C

The correct answer is: Because it needs to setup paging

**Question 8**

Complete

Mark 1.00 out of 1.00

The ljmp instruction in general does

- a. change the CS and EIP to 32 bit mode, and jumps to next line of code
- b. change the CS and EIP to 32 bit mode, and jumps to new value of EIP
- c. change the CS and EIP to 32 bit mode
- d. change the CS and EIP to 32 bit mode, and jumps to kernel code

The correct answer is: change the CS and EIP to 32 bit mode, and jumps to new value of EIP

**Question 9**

Complete

Mark 0.00 out of 1.00

The variable \$stack in entry.S is

- a. located at the value given by %esp as setup by bootmain()
- b. located at 0x7c00
- c. a memory region allocated as a part of entry.S
- d. located at less than 0x7c00
- e. located at 0

The correct answer is: a memory region allocated as a part of entry.S

**Question 10**

Not answered

Marked out of 0.50

Match the pairs of which action is taken by whom

Answer:

The correct answer is: kernel

**Question 11**

Complete

Mark 0.00 out of 1.00

ELF Magic number is

- a. 0xFFFFFFFF
- b. 0
- c. 0xELFELFELF
- d. 0xELF
- e. 0x0x464CELF
- f. 0x464C457FL
- g. 0x464C457FU

The correct answer is: 0x464C457FU

**Question 12**

Complete

Mark 1.00 out of 1.00

The right side of line of code "entry = (void(\*)(void))(elf->entry)" means

- a. Convert the "entry" in ELF structure into void
- b. Get the "entry" in ELF structure and convert it into a function pointer accepting no arguments and returning nothing
- c. Get the "entry" in ELF structure and convert it into a function void pointer
- d. Get the "entry" in ELF structure and convert it into a void pointer

The correct answer is: Get the "entry" in ELF structure and convert it into a function pointer accepting no arguments and returning nothing

[◀ Homework questions: Basics of MM, xv6 booting](#)

Jump to...

[\(Code\) Files, redirection, dup, \(IPC\)pipe ▶](#)

**Started on** Monday, 21 February 2022, 7:00:28 PM

**State** Finished

**Completed on** Monday, 21 February 2022, 7:49:24 PM

**Time taken** 48 mins 56 secs

**Grade** 8.30 out of 10.00 (83%)

**Question 1**

Complete

Mark 0.80 out of 1.00

Match the elements of C program to their place in memory

Local Static variables	Data
Global variables	Data
Code of main()	Code
Function code	Code
Arguments	Stack
Mallocoed Memory	Heap
#include files	No Memory needed
#define MACROS	No memory needed
Local Variables	Stack
Global Static variables	Data

The correct answer is: Local Static variables → Data, Global variables → Data, Code of main() → Code, Function code → Code, Arguments → Stack, Mallocoed Memory → Heap, #include files → No memory needed, #define MACROS → No Memory needed, Local Variables → Stack, Global Static variables → Data

**Question 2**

Complete

Mark 1.00 out of 1.00

What will be the output of this program

```
int main() {
    int fd;
    printf("%d ", open("/etc/passwd", O_RDONLY));
    close(1);
    fd = printf("%d ", open("/etc/passwd", O_RDONLY));
    close(fd);
    fd = printf("%d ", open("/etc/passwd", O_RDONLY));
}
```

- a. 3 1 2
- b. 3 3 3
- c. 3 1 1
- d. 1 1 1
- e. 3 4 5
- f. 2 2 2

The correct answer is: 3 1 1

**Question 3**

Complete

Mark 1.00 out of 1.00

Arrange in correct order, the files involved in execution of system call

vectors.S	2
trap.c	4
usys.S	1
trapasm.S	3

The correct answer is: vectors.S → 2, trap.c → 4, usys.S → 1, trapasm.S → 3

**Question 4**

Complete

Mark 1.00 out of 1.00

The "push 0" in vectors.S is

- a. A placeholder to match the size of struct trapframe
- b. Place for the error number value
- c. To indicate that it's a system call and not a hardware interrupt
- d. To be filled in as the return value of the system call

The correct answer is: Place for the error number value

**Question 5**

Complete

Mark 0.00 out of 1.00

A process blocks itself means

- a. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler
- b. The application code calls the scheduler
- c. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler
- d. The kernel code of system call calls scheduler

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

**Question 6**

Complete

Mark 1.00 out of 1.00

Select the odd one out

- a. Kernel stack of new process to Process stack of new process
- b. Process stack of running process to kernel stack of running process
- c. Kernel stack of running process to kernel stack of scheduler
- d. Kernel stack of scheduler to kernel stack of new process
- e. Kernel stack of new process to kernel stack of scheduler

The correct answer is: Kernel stack of new process to kernel stack of scheduler

**Question 7**

Complete

Mark 0.50 out of 0.50

Match the File descriptors to their meaning

0 Standard Input

2 Standard error

1 Standard output

The correct answer is: 0 → Standard Input, 2 → Standard error, 1 → Standard output

**Question 8**

Complete

Mark 1.00 out of 1.00

Which of the following is not a task of the code of swtch() function

- a. Switch stacks
- b. Change the kernel stack location
- c. Load the new context
- d. Jump to next context EIP
- e. Save the return value of the old context code
- f. Save the old context

The correct answers are: Save the return value of the old context code, Change the kernel stack location

**Question 9**

Complete

Mark 0.50 out of 0.50

Match the names of PCB structures with kernel

linux struct task\_struct

xv6 struct proc

The correct answer is: linux → struct task\_struct, xv6 → struct proc

**Question 10**

Complete

Mark 0.50 out of 0.50

Match the MACRO with its meaning

KERNBASE	2 GB
KERNLINK	2.224 GB
PHYSTOP	224 MB

The correct answer is: KERNBASE → 2 GB, KERNLINK → 2.224 GB, PHYSTOP → 224 MB

**Question 11**

Complete

Mark 1.00 out of 1.00

The trapframe, in xv6, is built by the

- a. hardware, trapasm.S
- b. hardware, vectors.S
- c. hardware, vectors.S, trapasm.S, trap()
- d. hardware, vectors.S, trapasm.S
- e. vectors.S, trapasm.S

The correct answer is: hardware, vectors.S, trapasm.S

**Question 12**

Complete

Mark 0.00 out of 0.50

Which of the following state transitions are not possible?

- a. Running -> Waiting
- b. Ready -> Waiting
- c. Waiting -> Terminated
- d. Ready -> Terminated

The correct answers are: Ready -&gt; Terminated, Waiting -&gt; Terminated, Ready -&gt; Waiting

[◀ Description of some possible course mini projects](#)

Jump to...

[\(Code\) mmap related programs ▶](#)

**Started on** Saturday, 26 February 2022, 5:18:30 PM

**State** Finished

**Completed on** Saturday, 26 February 2022, 6:30:44 PM

**Time taken** 1 hour 12 mins

**Grade** 8.55 out of 15.00 (57%)

**Question 1**

Complete

Mark 0.50 out of 0.50

Map the technique with its feature/problem

static linking	large executable file
dynamic loading	allocate memory only if needed
dynamic linking	small executable file
static loading	wastage of physical memory

The correct answer is: static linking → large executable file, dynamic loading → allocate memory only if needed, dynamic linking → small executable file, static loading → wastage of physical memory

**Question 2**

Complete

Mark 0.00 out of 1.00

Calculate the EAT in NANO-seconds (upto 2 decimal points) w.r.t. a page fault, given

Memory access time = 299 ns

Average page fault service time = 6 ms

Page fault rate = 0.8

Answer:

The correct answer is: 4800059.80

**Question 3**

Complete

Mark 1.00 out of 1.00

Given six memory partitions of 300 KB , 600 KB , 350 KB , 200 KB , 750 KB , and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB and 500 KB (in order)?

best fit 115 KB	125 KB
best fit 500 KB	600 KB
worst fit 500 KB	635 KB
worst fit 115 KB	750 KB
first fit 500 KB	600 KB
first fit 115 KB	300 KB

The correct answer is: best fit 115 KB → 125 KB, best fit 500 KB → 600 KB, worst fit 500 KB → 635 KB, worst fit 115 KB → 750 KB, first fit 500 KB → 600 KB, first fit 115 KB → 300 KB

**Question 4**

Complete

Mark 0.29 out of 1.00

Compare paging with demand paging and select the correct statements.

Select one or more:

- a. TLB hit ration has zero impact in effective memory access time in demand paging.
- b. Both demand paging and paging support shared memory pages.
- c. Calculations of number of bits for page number and offset are same in paging and demand paging.
- d. Demand paging requires additional hardware support, compared to paging.
- e. The meaning of valid-invalid bit in page table is different in paging and demand-paging.
- f. Paging requires NO hardware support in CPU
- g. With paging, it's possible to have user programs bigger than physical memory.
- h. With demand paging, it's possible to have user programs bigger than physical memory.
- i. Paging requires some hardware support in CPU
- j. Demand paging always increases effective memory access time.

The correct answers are: Demand paging requires additional hardware support, compared to paging., Both demand paging and paging support shared memory pages., With demand paging, it's possible to have user programs bigger than physical memory., Demand paging always increases effective memory access time., Paging requires some hardware support in CPU, Calculations of number of bits for page number and offset are same in paging and demand paging., The meaning of valid-invalid bit in page table is different in paging and demand-paging.

**Question 5**

Complete

Mark 0.36 out of 0.50

Map the parts of a C code to the memory regions they are related to

local variables	stack
global un-initialized variables	bss
static variables	data
global initialized variables	data
function arguments	stack
malloced memory	stack
functions	stack

The correct answer is: local variables → stack, global un-initialized variables → bss, static variables → data, global initialized variables → data, function arguments → stack, malloced memory → heap, functions → code

**Question 6**

Complete

Mark 0.75 out of 1.00

which of the following, do you think, are valid concerns for making the kernel pageable?

- a. The kernel must have some dedicated frames for it's own work
- b. No data structure of kernel should be pageable
- c. The kernel's own page tables should not be pageable
- d. The disk driver and disk interrupt handler should not be pageable
- e. No part of kernel code should be pageable.
- f. The page fault handler should not be pageable

The correct answers are: The kernel's own page tables should not be pageable, The page fault handler should not be pageable, The kernel must have some dedicated frames for it's own work, The disk driver and disk interrupt handler should not be pageable

## Question 7

Complete

Mark 0.75 out of 1.00

W.r.t the figure given below, mark the given statements as True or False.

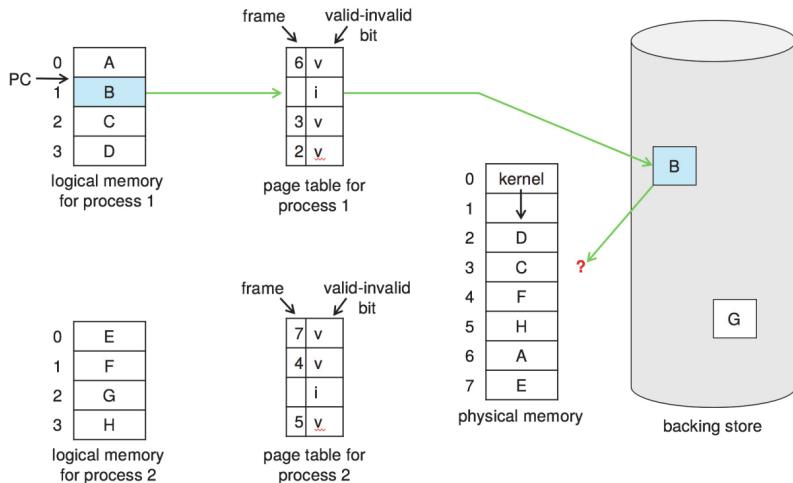


Figure 10.9 Need for page replacement.

True      False

- Kernel occupies two page frames
- Handling this scenario demands two disk I/Os
- Local replacement means chose any of the frames 2, 3, 6
- Local replacement means chose any of the frame from 2 to 7
- Global replacement means chose any of the frame from 0 to 7
- Global replacement means chose any of the frame from 2 to 7
- The kernel's pages can not used for replacement if kernel is not pageable.
- Page 1 of process 1 needs a replacement

Kernel occupies two page frames: True

Handling this scenario demands two disk I/Os: True

Local replacement means chose any of the frames 2, 3, 6: True

Local replacement means chose any of the frame from 2 to 7: False

Global replacement means chose any of the frame from 0 to 7: False

Global replacement means chose any of the frame from 2 to 7: True

The kernel's pages can not used for replacement if kernel is not pageable.: True

Page 1 of process 1 needs a replacement: True

**Question 8**

Complete

Mark 0.67 out of 1.00

Shared memory is possible with which of the following memory management schemes ?

Select one or more:

- a. paging
- b. segmentation
- c. continuous memory management
- d. demand paging

The correct answers are: paging, segmentation, demand paging

**Question 9**

Complete

Mark 0.60 out of 1.00

Given below is the "maps" file for a particular instance of "vim.basic" process.

Mark the given statements as True or False, w.r.t. the contents of the map file.

55a43501b000-55a435049000 r--p 00000000 103:05 917529	/usr/bin/vim.basic
55a435049000-55a435248000 r-xp 0002e000 103:05 917529	/usr/bin/vim.basic
55a435248000-55a4352b6000 r--p 0022d000 103:05 917529	/usr/bin/vim.basic
55a4352b7000-55a4352c5000 r--p 0029b000 103:05 917529	/usr/bin/vim.basic
55a4352c5000-55a4352e2000 rw-p 002a9000 103:05 917529	/usr/bin/vim.basic
55a4352e2000-55a4352f0000 rw-p 00000000 00:00 0	[heap]
55a436bc9000-55a436e5b000 rw-p 00000000 00:00 0	/usr/lib/x86_64-linux-
7f275b0a3000-7f275b0a6000 r--p 00000000 103:05 917901	/usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so	gnu/libnss_files-2.31.so
7f275b0a6000-7f275b0ad000 r-xp 00003000 103:05 917901	/usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so	gnu/libnss_files-2.31.so
7f275b0ad000-7f275b0af000 r--p 0000a000 103:05 917901	/usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so	gnu/libnss_files-2.31.so
7f275b0af000-7f275b0b0000 r--p 0000b000 103:05 917901	/usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so	gnu/libnss_files-2.31.so
7f275b0b0000-7f275b0b1000 rw-p 0000c000 103:05 917901	/usr/lib/x86_64-linux-
gnu/libnss_files-2.31.so	gnu/libnss_files-2.31.so
7f275b0b1000-7f275b0b7000 rw-p 00000000 00:00 0	
7f275b0b7000-7f275b8f5000 r--p 00000000 103:05 925247	/usr/lib/locale/locale-archive
7f275b8f5000-7f275b8fa000 rw-p 00000000 00:00 0	
7f275b8fa000-7f275b8fc000 r--p 00000000 103:05 924216	/usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4	gnu/libogg.so.0.8.4
7f275b8fc000-7f275b901000 r-xp 00002000 103:05 924216	/usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4	gnu/libogg.so.0.8.4
7f275b901000-7f275b904000 r--p 00007000 103:05 924216	/usr/lib/x86_64-linux-
gnu/libogg.so.0.8.4	gnu/libogg.so.0.8.4
7f275b904000-7f275b905000 ---p 0000a000 103:05 924216	
gnu/libogg.so.0.8.4	gnu/libogg.so.0.8.4
7f275b905000-7f275b906000 r--p 0000a000 103:05 924216	
gnu/libogg.so.0.8.4	gnu/libogg.so.0.8.4
7f275b906000-7f275b907000 rw-p 0000b000 103:05 924216	
gnu/libogg.so.0.8.4	gnu/libogg.so.0.8.4
7f275b907000-7f275b90a000 r--p 00000000 103:05 924627	
gnu/libvorbis.so.0.4.8	gnu/libvorbis.so.0.4.8
7f275b90a000-7f275b921000 r-xp 00003000 103:05 924627	
gnu/libvorbis.so.0.4.8	gnu/libvorbis.so.0.4.8
7f275b921000-7f275b932000 r--p 0001a000 103:05 924627	
gnu/libvorbis.so.0.4.8	gnu/libvorbis.so.0.4.8
7f275b932000-7f275b933000 ---p 0002b000 103:05 924627	
gnu/libvorbis.so.0.4.8	gnu/libvorbis.so.0.4.8
7f275b933000-7f275b934000 r--p 0002b000 103:05 924627	
gnu/libvorbis.so.0.4.8	gnu/libvorbis.so.0.4.8
7f275b934000-7f275b935000 rw-p 0002c000 103:05 924627	
gnu/libvorbis.so.0.4.8	gnu/libvorbis.so.0.4.8
7f275b935000-7f275b937000 rw-p 00000000 00:00 0	
7f275b937000-7f275b938000 r--p 00000000 103:05 917914	/usr/lib/x86_64-linux-
gnu/libutil-2.31.so	gnu/libutil-2.31.so
7f275b938000-7f275b939000 r-xp 00001000 103:05 917914	/usr/lib/x86_64-linux-
gnu/libutil-2.31.so	gnu/libutil-2.31.so
7f275b939000-7f275b93a000 r--p 00002000 103:05 917914	/usr/lib/x86_64-linux-
gnu/libutil-2.31.so	gnu/libutil-2.31.so
7f275b93a000-7f275b93b000 r--p 00002000 103:05 917914	/usr/lib/x86_64-linux-
gnu/libutil-2.31.so	gnu/libutil-2.31.so
7f275b93b000-7f275b93c000 rw-p 00003000 103:05 917914	/usr/lib/x86_64-linux-

```

gnu/libutil-2.31.so
7f275b93c000-7f275b93e000 r--p 00000000 103:05 915906      /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b93e000-7f275b94f000 r-xp 00002000 103:05 915906      /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b94f000-7f275b955000 r--p 00013000 103:05 915906      /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b955000-7f275b956000 ---p 00019000 103:05 915906      /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b956000-7f275b957000 r--p 00019000 103:05 915906      /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b957000-7f275b958000 rw-p 0001a000 103:05 915906      /usr/lib/x86_64-linux-
gnu/libz.so.1.2.11
7f275b958000-7f275b95c000 r--p 00000000 103:05 923645      /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b95c000-7f275b978000 r-xp 00004000 103:05 923645      /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b978000-7f275b982000 r--p 00020000 103:05 923645      /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b982000-7f275b983000 ---p 0002a000 103:05 923645      /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b983000-7f275b985000 r--p 0002a000 103:05 923645      /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b985000-7f275b986000 rw-p 0002c000 103:05 923645      /usr/lib/x86_64-linux-
gnu/libexpat.so.1.6.11
7f275b986000-7f275b988000 r--p 00000000 103:05 924057      /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b988000-7f275b98d000 r-xp 00002000 103:05 924057      /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b98d000-7f275b98f000 r--p 00007000 103:05 924057      /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b98f000-7f275b990000 r--p 00008000 103:05 924057      /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b990000-7f275b991000 rw-p 00009000 103:05 924057      /usr/lib/x86_64-linux-
gnu/libltdl.so.7.3.1
7f275b991000-7f275b995000 r--p 00000000 103:05 921934      /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b995000-7f275b9a3000 r-xp 00004000 103:05 921934      /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9a3000-7f275b9a9000 r--p 00012000 103:05 921934      /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9a9000-7f275b9aa000 r--p 00017000 103:05 921934      /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9aa000-7f275b9ab000 rw-p 00018000 103:05 921934      /usr/lib/x86_64-linux-
gnu/libtdb.so.1.4.3
7f275b9ab000-7f275b9ad000 rw-p 00000000 00:00 0
7f275b9ad000-7f275b9af000 r--p 00000000 103:05 924631      /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9af000-7f275b9b4000 r-xp 00002000 103:05 924631      /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b4000-7f275b9b5000 r--p 00007000 103:05 924631      /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b5000-7f275b9b6000 ---p 00008000 103:05 924631      /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b6000-7f275b9b7000 r--p 00008000 103:05 924631      /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b7000-7f275b9b8000 rw-p 00009000 103:05 924631      /usr/lib/x86_64-linux-
gnu/libvorbisfile.so.3.3.7
7f275b9b8000-7f275b9ba000 r--p 00000000 103:05 924277      /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0
7f275b9ba000-7f275ba1e000 r-xp 00002000 103:05 924277      /usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0

```

7f275ba1e000-7f275ba46000 r--p 00066000 103:05 924277	/usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0	
7f275ba46000-7f275ba47000 r--p 0008d000 103:05 924277	/usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0	
7f275ba47000-7f275ba48000 rw-p 0008e000 103:05 924277	/usr/lib/x86_64-linux-
gnu/libpcre2-8.so.0.9.0	
7f275ba48000-7f275ba6d000 r--p 00000000 103:05 917893	/usr/lib/x86_64-linux-
gnu/libc-2.31.so	
7f275ba6d000-7f275bbe5000 r-xp 00025000 103:05 917893	/usr/lib/x86_64-linux-
gnu/libc-2.31.so	
7f275bbe5000-7f275bc2f000 r--p 0019d000 103:05 917893	/usr/lib/x86_64-linux-
gnu/libc-2.31.so	
7f275bc2f000-7f275bc30000 ---p 001e7000 103:05 917893	/usr/lib/x86_64-linux-
gnu/libc-2.31.so	
7f275bc30000-7f275bc33000 r--p 001e7000 103:05 917893	/usr/lib/x86_64-linux-
gnu/libc-2.31.so	
7f275bc33000-7f275bc36000 rw-p 001ea000 103:05 917893	/usr/lib/x86_64-linux-
gnu/libc-2.31.so	
7f275bc36000-7f275bc3a000 rw-p 00000000 00:00 0	
7f275bc3a000-7f275bc41000 r--p 00000000 103:05 917906	/usr/lib/x86_64-linux-
gnu/libpthread-2.31.so	
7f275bc41000-7f275bc52000 r-xp 00007000 103:05 917906	/usr/lib/x86_64-linux-
gnu/libpthread-2.31.so	
7f275bc52000-7f275bc57000 r--p 00018000 103:05 917906	/usr/lib/x86_64-linux-
gnu/libpthread-2.31.so	
7f275bc57000-7f275bc58000 r--p 0001c000 103:05 917906	/usr/lib/x86_64-linux-
gnu/libpthread-2.31.so	
7f275bc58000-7f275bc59000 rw-p 0001d000 103:05 917906	/usr/lib/x86_64-linux-
gnu/libpthread-2.31.so	
7f275bc59000-7f275bc5d000 rw-p 00000000 00:00 0	
7f275bc5d000-7f275bcce000 r--p 00000000 103:05 917016	/usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0	
7f275bcce000-7f275bf29000 r-xp 00071000 103:05 917016	/usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0	
7f275bf29000-7f275c142000 r--p 002cc000 103:05 917016	/usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0	
7f275c142000-7f275c143000 ---p 004e5000 103:05 917016	/usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0	
7f275c143000-7f275c149000 r--p 004e5000 103:05 917016	/usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0	
7f275c149000-7f275c190000 rw-p 004eb000 103:05 917016	/usr/lib/x86_64-linux-
gnu/libpython3.8.so.1.0	
7f275c190000-7f275c1b3000 rw-p 00000000 00:00 0	
7f275c1b3000-7f275c1b4000 r--p 00000000 103:05 917894	/usr/lib/x86_64-linux-
gnu/libdl-2.31.so	
7f275c1b4000-7f275c1b6000 r-xp 00001000 103:05 917894	/usr/lib/x86_64-linux-
gnu/libdl-2.31.so	
7f275c1b6000-7f275c1b7000 r--p 00003000 103:05 917894	/usr/lib/x86_64-linux-
gnu/libdl-2.31.so	
7f275c1b7000-7f275c1b8000 r--p 00003000 103:05 917894	/usr/lib/x86_64-linux-
gnu/libdl-2.31.so	
7f275c1b8000-7f275c1b9000 rw-p 00004000 103:05 917894	/usr/lib/x86_64-linux-
gnu/libdl-2.31.so	
7f275c1b9000-7f275c1bb000 rw-p 00000000 00:00 0	
7f275c1bb000-7f275c1c0000 r-xp 00000000 103:05 923815	/usr/lib/x86_64-linux-
gnu/libgpm.so.2	
7f275c1c0000-7f275c3bf000 ---p 00005000 103:05 923815	/usr/lib/x86_64-linux-
gnu/libgpm.so.2	
7f275c3bf000-7f275c3c0000 r--p 00004000 103:05 923815	/usr/lib/x86_64-linux-
gnu/libgpm.so.2	
7f275c3c0000-7f275c3c1000 rw-p 00005000 103:05 923815	/usr/lib/x86_64-linux-
gnu/libgpm.so.2	

```

7f275c3c1000-7f275c3c3000 r--p 00000000 103:05 923315          /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3c3000-7f275c3c8000 r-xp 00002000 103:05 923315          /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3c8000-7f275c3ca000 r--p 00007000 103:05 923315          /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3ca000-7f275c3cb000 r--p 00008000 103:05 923315          /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3cb000-7f275c3cc000 rw-p 00009000 103:05 923315          /usr/lib/x86_64-linux-
gnu/libacl.so.1.1.2253
7f275c3cc000-7f275c3cf000 r--p 00000000 103:05 923446          /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3cf000-7f275c3d9000 r-xp 00003000 103:05 923446          /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3d9000-7f275c3dd000 r--p 0000d000 103:05 923446          /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3dd000-7f275c3de000 r--p 00010000 103:05 923446          /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3de000-7f275c3df000 rw-p 00011000 103:05 923446          /usr/lib/x86_64-linux-
gnu/libcanberra.so.0.2.5
7f275c3df000-7f275c3e5000 r--p 00000000 103:05 924431          /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c3e5000-7f275c3fe000 r-xp 00006000 103:05 924431          /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c3fe000-7f275c405000 r--p 0001f000 103:05 924431          /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c405000-7f275c406000 ---p 00026000 103:05 924431          /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c406000-7f275c407000 r--p 00026000 103:05 924431          /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c407000-7f275c408000 rw-p 00027000 103:05 924431          /usr/lib/x86_64-linux-
gnu/libselinux.so.1
7f275c408000-7f275c40a000 rw-p 00000000 00:00 0
7f275c40a000-7f275c418000 r--p 00000000 103:05 924540          /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c418000-7f275c427000 r-xp 0000e000 103:05 924540          /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c427000-7f275c435000 r--p 0001d000 103:05 924540          /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c435000-7f275c439000 r--p 0002a000 103:05 924540          /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c439000-7f275c43a000 rw-p 0002e000 103:05 924540          /usr/lib/x86_64-linux-
gnu/libtinfo.so.6.2
7f275c43a000-7f275c449000 r--p 00000000 103:05 917895          /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c449000-7f275c4f0000 r-xp 0000f000 103:05 917895          /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c4f0000-7f275c587000 r--p 000b6000 103:05 917895          /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c587000-7f275c588000 r--p 0014c000 103:05 917895          /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c588000-7f275c589000 rw-p 0014d000 103:05 917895          /usr/lib/x86_64-linux-
gnu/libm-2.31.so
7f275c589000-7f275c58b000 rw-p 00000000 00:00 0
7f275c5ae000-7f275c5af000 r--p 00000000 103:05 917889          /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5af000-7f275c5d2000 r-xp 00001000 103:05 917889          /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5d2000-7f275c5da000 r--p 00024000 103:05 917889          /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5db000-7f275c5dc000 r--p 0002c000 103:05 917889          /usr/lib/x86_64-linux-gnu/ld-
2.31.so

```

```
7f275c5dc000-7f275c5dd000 rw-p 0002d000 103:05 917889          /usr/lib/x86_64-linux-gnu/ld-
2.31.so
7f275c5dd000-7f275c5de000 rw-p 00000000 00:00 0
7ffd22d2f000-7ffd22d50000 rw-p 00000000 00:00 0          [stack]
7ffd22db0000-7ffd22db4000 r--p 00000000 00:00 0          [vvar]
7ffd22db4000-7ffd22db6000 r-xp 00000000 00:00 0          [vdso]
ffffffff600000-ffffffff601000 --xp 00000000 00:00 0          [vsyscall]
```

**True      False**

- The size of the heap is one page
- This is a virtual memory map (not physical memory map)
- vim.basic uses the math library
- The 5th entry 55a4352c5000-55a4352e2000 may correspond to "data" of the vim.basic
- The size of the stack is one page

The size of the heap is one page: False

This is a virtual memory map (not physical memory map): True

vim.basic uses the math library: True

The 5th entry 55a4352c5000-55a4352e2000 may correspond to "data" of the vim.basic: True

The size of the stack is one page: False

**Question 10**

Complete

Mark 0.00 out of 1.00

Select all the correct statements, w.r.t. Copy on Write

- a. Fork() used COW technique to improve performance of new process creation.
- b. Vfork() assumes that there will be no write, but rather exec()
- c. COW helps us save memory
- d. If either parent or child modifies a COW-page, then a copy of the page is made and page table entry is updated
- e. use of COW during fork() is useless if child called exit()
- f. use of COW during fork() is useless if exec() is called by the child

The correct answers are: Fork() used COW technique to improve performance of new process creation., If either parent or child modifies a COW-page, then a copy of the page is made and page table entry is updated, COW helps us save memory, Vfork() assumes that there will be no write, but rather exec()

**Question 11**

Complete

Mark 1.00 out of 1.00

Assuming a 8- KB page size, what is the page numbers for the address 1093943 reference in decimal :  
 (give answer also in decimal)

Answer: 

The correct answer is: 134

**Question 12**

Complete

Mark 0.00 out of 1.00

Order the following events, related to page fault handling, in correct order

1. Page fault handler detects that it's a page fault and not illegal memory access
2. Disk interrupt handler runs
3. MMU detects that a page table entry is marked "invalid"
4. Page faulted process is moved to ready-queue
5. Empty frame is found
6. Page fault interrupt is generated
7. Other processes scheduled by scheduler
8. Page table of page faulted process is updated
9. Disk Interrupt occurs
10. Disk read is issued
11. Page fault handler in kernel starts executing
12. Page faulting process is made to wait in a queue

The correct order for these items is as follows:

1. MMU detects that a page table entry is marked "invalid"
2. Page fault interrupt is generated
3. Page fault handler in kernel starts executing
4. Page fault handler detects that it's a page fault and not illegal memory access
5. Empty frame is found
6. Disk read is issued
7. Page faulting process is made to wait in a queue
8. Other processes scheduled by scheduler
9. Disk Interrupt occurs
10. Disk interrupt handler runs
11. Page table of page faulted process is updated
12. Page faulted process is moved to ready-queue

**Question 13**

Complete

Mark 1.00 out of 1.00

Page sizes are a power of 2 because

Select one:

- a. Certain bits are reserved for offset in logical address. Hence page size =  $2^{(\text{no.of offset bits})}$
- b. Power of 2 calculations are highly efficient
- c. Certain bits are reserved for offset in logical address. Hence page size =  $2^{(32 - \text{no.of offset bits})}$
- d. operating system calculations happen using power of 2
- e. MMU only understands numbers that are power of 2

The correct answer is: Certain bits are reserved for offset in logical address. Hence page size =  $2^{(\text{no.of offset bits})}$

**Question 14**

Complete

Mark 1.00 out of 1.00

Given below is the output of the command "ps -eo min\_flt,maj\_flt,cmd" on a Linux Desktop system. Select the statements that are consistent with the output

```
626729 482768 /usr/lib/firefox/firefox -contentproc -parentBuildID 20220202182137 -prefsLen 9256 -prefMapSize 264738 -appDir /usr/lib/firefox/browser 6094 true rdd
2167 687 /usr/sbin/apache2 -k start
1265185 222 /usr/bin/gnome-shell
102648 111 /usr/sbin/mysqld
9813 0 bash
15497 370 /usr/bin/gedit --gapplication-service
```

- a. All of the processes here exhibit some good locality of reference
- b. Firefox has likely been running for a large amount of time
- c. The bash shell is mostly busy doing work within a particular locality
- d. Apache web-server has not been doing much work

The correct answers are: Firefox has likely been running for a large amount of time, Apache web-server has not been doing much work, The bash shell is mostly busy doing work within a particular locality, All of the processes here exhibit some good locality of reference

**Question 15**

Complete

Mark 0.14 out of 1.00

Suppose two processes share a library between them. The library consists of 5 pages, and these 5 pages are mapped to frames 9, 15, 23, 4, 7 respectively. Process P1 has got 6 pages, first 3 of which consist of process's own code/data and 3 correspond to library's pages 0, 2, 4. Process P2 has got 7 pages, first 3 of which consist of process's own code/data and remaining 4 correspond to library's pages 0, 1, 3, 4. Fill in the blanks for page table entries of P1 and P2.

Page table of P1, Page 5	<input type="text" value="23"/>
Page table of P1, Page 3	<input type="text" value="9"/>
Page table of P2, Page 0	<input type="text" value="6"/>
Page table of P2, Page 3	<input type="text" value="7"/>
Page table of P2, Page 4	<input type="text" value="9"/>
Page table of P2, Page 1	<input type="text" value="5"/>
Page table of P1, Page 4	<input type="text" value="9"/>

The correct answer is: Page table of P1, Page 5 → 7, Page table of P1, Page 3 → 9, Page table of P2, Page 0 → 9, Page table of P2, Page 3 → 4, Page table of P2, Page 4 → 7, Page table of P2, Page 1 → 15, Page table of P1, Page 4 → 23

**Question 16**

Complete

Mark 0.50 out of 1.00

For the reference string

3 4 3 5 2

the number of page faults (including initial ones) using

FIFO replacement and 2 page frames is :

FIFO replacement and 3 page frames is :

◀ (Code) mmap related programs

Jump to...

Points from Mid-term feedback ►

**Started on** Monday, 7 March 2022, 7:00:12 PM

**State** Finished

**Completed on** Monday, 7 March 2022, 8:00:04 PM

**Time taken** 59 mins 52 secs

**Grade** 9.78 out of 15.00 (65%)

**Question 1**

Complete

Mark 1.00 out of 1.00

Why is there a call to kinit2? Why is it not merged with knit1?

- a. call to seginit() makes it possible to actually use PHYSTOP in argument to kinit2()
- b. When kinit1() is called there is a need for few page frames, but later kinit2() is called to serve need of more page frames
- c. Because there is a limit on the values that the arguments to kinit1() can take.
- d. kinit2 refers to virtual addresses beyond 4MB, which are not mapped before kalloc() is called

The correct answer is: kinit2 refers to virtual addresses beyond 4MB, which are not mapped before kalloc() is called

**Question 2**

Complete

Mark 1.50 out of 1.50

Arrange the following in the correct order of execution (w.r.t. 'init')

initcode() returns in forkret()

6

'initcode' process is marked RUNNABLE

3

'initcode' struct proc is created

2

userinit() is called

1

mpmain() calls scheduler()

4

initcode() calls exec("/init", ...)

8

initcode() returns from trapret()

7

scheduler() schedules initcode() process

5

The correct answer is: initcode() returns in forkret() → 6, 'initcode' process is marked RUNNABLE → 3, 'initcode' struct proc is created → 2, userinit() is called → 1, mpmain() calls scheduler() → 4, initcode() calls exec("/init", ...) → 8, initcode() returns from trapret() → 7, scheduler() schedules initcode() process → 5

**Question 3**

Complete

Mark 0.00 out of 2.00

exec() does this: curproc->tf->eip = elf.entry, but userinit() does this: p->tf->eip = 0; Select all the statements from below, that collectively explain this

- a. the 'entry' in initcode is anyways 0
- b. exec() loads from ELF file and the address of first instruction to be executed is given by 'entry'
- c. the initcode is created using objcopy, which discards all relocation information and symbols (like entry)
- d. elf.entry is anyways 0, so both statements mean the same
- e. In userinit() the function inituvm() has mapped the code of 'initcode' to be starting at virtual address 0
- f. the code of 'initcode' is loaded at physical address 0

The correct answers are: exec() loads from ELF file and the address of first instruction to be executed is given by 'entry', In userinit() the function inituvm() has mapped the code of 'initcode' to be starting at virtual address 0, the initcode is created using objcopy, which discards all relocation information and symbols (like entry)

**Question 4**

Complete

Mark 1.00 out of 1.00

What does userinit() do ?

- a. initializes the users
- b. sets up the 'initcode' process to start execution in forkret()
- c. sets up the 'initcode' process to start execution in forkret()
- d. sets up the 'initcode' process to start execution in trapret()
- e. sets up the 'init' process to start execution in forkret()
- f. initializes the process 'init' and starts executing it

The correct answer is: sets up the 'initcode' process to start execution in forkret()

**Question 5**

Complete

Mark 0.67 out of 1.00

Which of the following is done by mappages()?

- a. allocate page directory if required
- b. allocate page frame if required
- c. allocate page table if required
- d. create page table mappings for the range given by "va" and "va + size"
- e. create page table mappings to the range given by "pa" and "pa + size"

The correct answers are: create page table mappings for the range given by "va" and "va + size", allocate page table if required, create page table mappings to the range given by "pa" and "pa + size"

**Question 6**

Complete

Mark 0.00 out of 1.00

Select the statement that most correctly describes what setupkvm() does

- a. creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global array
- b. creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global array and makes kpgdir point to it
- c. creates a 1-level page table for the use by the kernel, as specified in kmap[] global array
- d. creates a 2-level page table for the use of the kernel, as specified in gdtdesc

The correct answer is: creates a 2-level page table setup with virtual->physical mappings specified in the kmap[] global array

**Question 7**

Complete

Mark 0.00 out of 1.00

The approximate number of page frames created by kinit1 is

- a. 4
- b. 16
- c. 4000
- d. 3000
- e. 2000
- f. 1000
- g. 10

The correct answer is: 3000

**Question 8**

Complete

Mark 1.20 out of 1.50

Which of the following is DONE by allocproc() ?

- a. setup kernel memory mappings for the process
- b. Select an UNUSED struct proc for use
- c. ensure that the process starts in trapret()
- d. allocate kernel stack for the process
- e. allocate PID to the process
- f. ensure that the process starts in forkret()
- g. setup the trapframe and context pointers appropriately
- h. setup the contents of the trapframe of the process properly

The correct answers are: Select an UNUSED struct proc for use, allocate PID to the process, allocate kernel stack for the process, setup the trapframe and context pointers appropriately, ensure that the process starts in forkret()

**Question 9**

Complete

Mark 1.00 out of 1.00

Map the virtual address to physical address in xv6

KERNLINK	0x100000
0xFE000000	0xFE000000
80108000	0x108000
KERNBASE	0

The correct answer is: KERNLINK → 0x100000, 0xFE000000 → 0xFE000000, 80108000 → 0x108000, KERNBASE → 0

**Question 10**

Complete

Mark 0.42 out of 1.00

Select all the correct statements about initcode

- a. code of initcode is loaded at virtual address 0
- b. The data and stack of initcode is mapped to one single page in userinit()
- c. code of 'initcode' is loaded along with the kernel during booting
- d. the size of 'initcode' is 2c
- e. code of initcode is loaded in memory by the kernel during userinit()
- f. initcode is the 'init' process
- g. initcode essentially calls exec("/init",...)

The correct answers are: code of 'initcode' is loaded along with the kernel during booting, the size of 'initcode' is 2c, The data and stack of initcode is mapped to one single page in userinit(), initcode essentially calls exec("/init",...)

**Question 11**

Complete

Mark 1.00 out of 1.00

The variable 'end' used as argument to kinit1 has the value

- a. 801154a8
- b. 81000000
- c. 80110000
- d. 80102da0
- e. 8010a48c
- f. 80000000

The correct answer is: 801154a8

**Question 12**

Complete

Mark 1.00 out of 1.00

What does seginit() do?

- a. Nothing significant, just repetition of earlier GDT setup but with 2-level paging setup done
- b. Nothing significant, just repetition of earlier GDT setup but with free frames list created now
- c. Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 0
- d. Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 3
- e. Nothing significant, just repetition of earlier GDT setup but with kernel page table allocated now

The correct answer is: Adds two additional entries to GDT corresponding to Code and Data segments, but to be used in privilege level 3

**Question 13**

Complete

Mark 1.00 out of 1.00

Does exec() code around clearptau() lead to wastage of one page frame?

- a. no
- b. yes

The correct answer is: yes

[◀ Questions for test on kalloc/kfree/kvmalloc, etc.](#)

Jump to...

[\(Optional Assignment\) Slab allocator in xv6 ►](#)

---

**Started on** Tuesday, 16 January 2024, 5:05 PM

**State** Finished

---

**Completed on** Tuesday, 16 January 2024, 5:58 PM

**Time taken** 53 mins 16 secs

---

**Grade** **10.02** out of 15.00 (**66.78%**)

Question 1

Incorrect

Mark 0.00 out of 1.00

Select the sequence of events that are NOT possible, assuming a non-interruptible kernel code

(Note: non-interruptible kernel code means, if the kernel code is executing, then interrupts will be disabled).

Note: A possible sequence may have some missing steps in between. An impossible sequence will have n and n+1th steps such that n+1th step can not follow n'th step.

Select one or more:

- a. P1 running  
keyboard hardware interrupt

keyboard interrupt handler running

interrupt handler returns

P1 running

P1 makes system call

system call returns

P1 running

timer interrupt

Scheduler

P2 running

- b. P1 running  
P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again

- c. P1 running  
P1 makes system call

system call returns

P1 running

timer interrupt

Scheduler running

P2 running

- d.  
P1 running  
P1 makes system call

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again

- e. P1 running ✗

P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P3 running

Hardware interrupt

Interrupt unblocks P1

Interrupt returns

P3 running

Timer interrupt

Scheduler

P1 running

- f. P1 running

P1 makes system call

timer interrupt

Scheduler

P2 running

timer interrupt

Scheuler

P1 running

P1's system call return

Your answer is incorrect.

The correct answers are: P1 running

P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again, P1 running

P1 makes system call

timer interrupt

Scheduler

P2 running

timer interrupt

Scheuler

P1 running

P1's system call return,

P1 running

P1 makes system call

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again

**Question 2**

Incorrect

Mark 0.00 out of 1.00

Write the possible contents of the file /tmp/xyz after this program.

In the answer if you want to mention any non-text character, then write \0. For example abc\0\0 means abc followed by any two non-text characters

```
int main(int argc, char *argv[]) {  
    int fd1, fd2, n, i;  
    char buf[128];  
  
    fd1 = open("/tmp/xyz", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);  
    write(fd1, "hello", 5);  
    fd2 = open("/tmp/xyz", O_WRONLY, S_IRUSR|S_IWUSR);  
    write(fd2, "bye", 3);  
    close(fd1);  
    close(fd2);  
    return 0;  
}
```

Answer: bye\0\0\0



The correct answer is: byelo

**Question 3**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about bootloader.

Every wrong selection will deduct marks proportional to  $1/n$  where n is total wrong choices in the question.

You will get minimum a zero.

- a. Bootloader must be one sector in length
- b. The bootloader loads the BIOS
- c. Bootloaders allow selection of OS to boot from ✓
- d. Modern Bootloaders often allow configuring the way an OS boots ✓
- e. LILO is a bootloader ✓

Your answer is correct.

The correct answers are: LILO is a bootloader, Modern Bootloaders often allow configuring the way an OS boots, Bootloaders allow selection of OS to boot from

**Question 4**

Correct

Mark 1.00 out of 1.00

How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security) ?

Select one:

- a. It prohibits a user mode process from running privileged instructions ✓
- b. It prohibits invocation of kernel code completely, if a user program is running
- c. It prohibits one process from accessing other process's memory
- d. It disallows hardware interrupts when a process is running

Your answer is correct.

The correct answer is: It prohibits a user mode process from running privileged instructions

**Question 5**

Correct

Mark 0.50 out of 0.50

Select all the correct statements about bootloader.

Every wrong selection will deduct marks proportional to  $1/n$  where n is total wrong choices in the question.

You will get minimum a zero.

- a. Bootloaders allow selection of OS to boot from ✓
- b. LILO is a bootloader ✓
- c. Bootloader must be one sector in length
- d. The bootloader loads the BIOS
- e. Modern Bootloaders often allow configuring the way an OS boots ✓

Your answer is correct.

The correct answers are: LILO is a bootloader, Modern Bootloaders often allow configuring the way an OS boots, Bootloaders allow selection of OS to boot from

Question **6**

Incorrect

Mark 0.00 out of 1.00

What will this program do?

```
int main() {  
    fork();  
    execl("/bin/ls", "/bin/ls", NULL);  
    printf("hello");  
}
```

- a. run ls twice and print hello twice, but output will appear in some random order
- b. one process will run ls, another will print hello
- c. run ls once X
- d. run ls twice and print hello twice
- e. run ls twice

Your answer is incorrect.

The correct answer is: run ls twice

Question **7**

Correct

Mark 0.50 out of 0.50

Compare multiprogramming with multitasking

- a. A multiprogramming system is not necessarily multitasking ✓
- b. A multitasking system is not necessarily multiprogramming

The correct answer is: A multiprogramming system is not necessarily multitasking

**Question 8**

Partially correct

Mark 0.75 out of 1.00

Given below is the output of "ps -eaf".

Answer the questions based on it.

UID	PID	PPID	C	S	TIME	TTY	TIME	CMD
root	1	0	0	Jan05	? 00:01:08	/sbin/init	splash	
root	2	0	0	Jan05	? 00:00:00	[kthreadd]		
root	3	2	0	Jan05	? 00:00:00	[rcu_gp]		
root	4	2	0	Jan05	? 00:00:00	[rcu_par_gp]		
root	9	2	0	Jan05	? 00:00:00	[mm_percpu_wq]		
root	10	2	0	Jan05	? 00:00:00	[rcu_tasks_rude_]		
root	11	2	0	Jan05	? 00:00:00	[rcu_tasks_trace]		
root	12	2	0	Jan05	? 00:00:22	[ksoftirqd/0]		
root	13	2	0	Jan05	? 00:06:29	[rcu_sched]		
root	14	2	0	Jan05	? 00:00:02	[migration/0]		
root	15	2	0	Jan05	? 00:00:00	[idle_inject/0]		
root	16	2	0	Jan05	? 00:00:00	[cpuhp/0]		
root	17	2	0	Jan05	? 00:00:00	[cpuhp/1]		
root	18	2	0	Jan05	? 00:00:00	[idle_inject/1]		
root	19	2	0	Jan05	? 00:00:03	[migration/1]		
root	20	2	0	Jan05	? 00:00:13	[ksoftirqd/1]		
root	22	2	0	Jan05	? 00:00:00	[kworker/1:0H-events_highpri]		
root	23	2	0	Jan05	? 00:00:00	[cpuhp/2]		
root	24	2	0	Jan05	? 00:00:00	[idle_inject/2]		
root	25	2	0	Jan05	? 00:00:01	[migration/2]		
root	26	2	0	Jan05	? 00:00:09	[ksoftirqd/2]		
root	28	2	0	Jan05	? 00:00:00	[kworker/2:0H-kblockd]		
root	29	2	0	Jan05	? 00:00:00	[cpuhp/3]		
root	30	2	0	Jan05	? 00:00:00	[idle_inject/3]		
root	31	2	0	Jan05	? 00:00:02	[migration/3]		
root	32	2	0	Jan05	? 00:00:07	[ksoftirqd/3]		
root	34	2	0	Jan05	? 00:00:00	[kworker/3:0H-events_highpri]		
root	35	2	0	Jan05	? 00:00:00	[cpuhp/4]		
root	36	2	0	Jan05	? 00:00:00	[idle_inject/4]		
root	37	2	0	Jan05	? 00:00:02	[migration/4]		
root	38	2	0	Jan05	? 00:00:06	[ksoftirqd/4]		
root	40	2	0	Jan05	? 00:00:00	[kworker/4:0H-events_highpri]		
root	41	2	0	Jan05	? 00:00:00	[cpuhp/5]		
root	42	2	0	Jan05	? 00:00:00	[idle_inject/5]		
root	43	2	0	Jan05	? 00:00:02	[migration/5]		
root	44	2	0	Jan05	? 00:00:05	[ksoftirqd/5]		
root	46	2	0	Jan05	? 00:00:00	[kworker/5:0H-events_highpri]		
root	47	2	0	Jan05	? 00:00:00	[cpuhp/6]		
root	48	2	0	Jan05	? 00:00:00	[idle_inject/6]		
root	49	2	0	Jan05	? 00:00:02	[migration/6]		
root	50	2	0	Jan05	? 00:00:05	[ksoftirqd/6]		
root	52	2	0	Jan05	? 00:00:00	[kworker/6:0H-events_highpri]		
root	53	2	0	Jan05	? 00:00:00	[cpuhp/7]		
root	54	2	0	Jan05	? 00:00:00	[idle_inject/7]		
root	55	2	0	Jan05	? 00:00:02	[migration/7]		
root	56	2	0	Jan05	? 00:00:06	[ksoftirqd/7]		
root	58	2	0	Jan05	? 00:00:00	[kworker/7:0H-events_highpri]		
root	59	2	0	Jan05	? 00:00:00	[kdevtmpfs]		
root	60	2	0	Jan05	? 00:00:00	[netns]		
root	61	2	0	Jan05	? 00:00:00	[inet_frag_wq]		
root	62	2	0	Jan05	? 00:00:00	[kaudittd]		
root	63	2	0	Jan05	? 00:00:00	[khungtaskd]		
root	64	2	0	Jan05	? 00:00:00	[oom_reaper]		
root	65	2	0	Jan05	? 00:00:00	[writeback]		
root	66	2	0	Jan05	? 00:01:58	[kcompactd0]		
root	67	2	0	Jan05	? 00:00:00	[ksmd]		
root	68	2	0	Jan05	? 00:00:04	[khugepaged]		
root	115	2	0	Jan05	? 00:00:00	[kintegrityd]		

root 116 2 0 Jan05 ? 00:00:00 [kblockd]  
root 117 2 0 Jan05 ? 00:00:00 [blkcg\_punt\_bio]  
root 118 2 0 Jan05 ? 00:00:00 [tpm\_dev\_wq]  
root 119 2 0 Jan05 ? 00:00:00 [ata\_sff]  
root 120 2 0 Jan05 ? 00:00:00 [md]  
root 121 2 0 Jan05 ? 00:00:00 [edac-poller]  
root 122 2 0 Jan05 ? 00:00:00 [devfreq\_wq]  
root 123 2 0 Jan05 ? 00:00:00 [watchdogd]  
root 129 2 0 Jan05 ? 00:00:00 [irq/25-AMD-Vi]  
root 131 2 0 Jan05 ? 00:04:33 [kswapd0]  
root 132 2 0 Jan05 ? 00:00:00 [ecryptfs-kthrea]  
root 134 2 0 Jan05 ? 00:00:00 [kthrotld]  
root 135 2 0 Jan05 ? 00:00:00 [irq/27-pciehp]  
root 139 2 0 Jan05 ? 00:00:00 [acpi\_thermal\_pm]  
root 140 2 0 Jan05 ? 00:00:00 [vfio-irqfd-clea]  
root 141 2 0 Jan05 ? 00:00:00 [ipv6\_addrconf]  
root 144 2 0 Jan05 ? 00:00:03 [kworker/6:1H-kblockd]  
root 151 2 0 Jan05 ? 00:00:00 [kstrp]  
root 154 2 0 Jan05 ? 00:00:00 [zswap-shrink]  
root 162 2 0 Jan05 ? 00:00:00 [charger\_manager]  
root 164 2 0 Jan05 ? 00:00:03 [kworker/4:1H-kblockd]  
root 197 2 0 Jan05 ? 00:00:03 [kworker/3:1H-kblockd]  
root 213 2 0 Jan05 ? 00:00:03 [kworker/7:1H-kblockd]  
root 215 2 0 Jan05 ? 00:00:00 [nvme-wq]  
root 216 2 0 Jan05 ? 00:00:00 [nvme-reset-wq]  
root 217 2 0 Jan05 ? 00:00:00 [nvme-delete-wq]  
root 223 2 0 Jan05 ? 00:00:00 [irq/42-ELAN2513]  
root 224 2 0 Jan05 ? 00:04:20 [irq/41-ELAN071B]  
root 225 2 0 Jan05 ? 00:00:00 [cryptd]  
root 226 2 0 Jan05 ? 00:00:00 [amd\_iommu\_v2]  
root 249 2 0 Jan05 ? 00:00:00 [ttm\_swap]  
root 250 2 0 Jan05 ? 00:20:29 [gfx]  
root 251 2 0 Jan05 ? 00:00:00 [comp\_1.0.0]  
root 252 2 0 Jan05 ? 00:00:00 [comp\_1.1.0]  
root 253 2 0 Jan05 ? 00:00:00 [comp\_1.2.0]  
root 254 2 0 Jan05 ? 00:00:00 [comp\_1.3.0]  
root 255 2 0 Jan05 ? 00:00:00 [comp\_1.0.1]  
root 256 2 0 Jan05 ? 00:00:00 [comp\_1.1.1]  
root 257 2 0 Jan05 ? 00:00:00 [comp\_1.2.1]  
root 258 2 0 Jan05 ? 00:00:00 [comp\_1.3.1]  
root 259 2 0 Jan05 ? 00:00:27 [sdma0]  
root 260 2 0 Jan05 ? 00:00:00 [vcn\_dec]  
root 261 2 0 Jan05 ? 00:00:00 [vcn\_enc0]  
root 262 2 0 Jan05 ? 00:00:00 [vcn\_encl]  
root 263 2 0 Jan05 ? 00:00:00 [jpeg\_dec]  
root 265 2 0 Jan05 ? 00:00:00 [card0-crtc0]  
root 266 2 0 Jan05 ? 00:00:00 [card0-crtc1]  
root 267 2 0 Jan05 ? 00:00:00 [card0-crtc2]  
root 268 2 0 Jan05 ? 00:00:00 [card0-crtc3]  
root 271 2 0 Jan05 ? 00:00:03 [kworker/5:1H-kblockd]  
root 277 2 0 Jan05 ? 00:00:03 [kworker/1:1H-kblockd]  
root 320 2 0 Jan05 ? 00:00:00 [raid5wq]  
root 380 2 0 Jan05 ? 00:00:11 [jbd2/nvme0n1p5-]  
root 381 2 0 Jan05 ? 00:00:00 [ext4-rsv-conver]  
root 432 2 0 Jan05 ? 00:00:03 [kworker/2:1H-kblockd]  
root 446 1 0 Jan05 ? 00:01:16 /lib/systemd/systemd-journald  
root 470 2 0 Jan05 ? 00:00:00 [rpciod]  
root 473 2 0 Jan05 ? 00:00:00 [xpriod]  
root 474 2 0 Jan05 ? 00:00:00 bpfilter\_umh  
root 503 1 0 Jan05 ? 00:00:07 /lib/systemd/systemd-udevd  
root 517 2 0 Jan05 ? 00:00:00 [loop0]  
root 554 2 0 Jan05 ? 00:00:00 [loop1]  
root 563 2 0 Jan05 ? 00:00:00 [loop2]  
root 586 2 0 Jan05 ? 00:00:00 [loop3]  
root 588 2 0 Jan05 ? 00:00:00 [loop4]  
root 589 2 0 Jan05 ? 00:00:00 [loop5]

root 612 2 0 Jan05 ? 00:00:00 [loop6]  
root 613 2 0 Jan05 ? 00:00:00 [loop7]  
root 629 2 0 Jan05 ? 00:00:00 [loop8]  
root 637 2 0 Jan05 ? 00:00:00 [cfg80211]  
root 676 2 0 Jan05 ? 00:05:00 [irq/75-iwlwifi:]  
root 678 2 0 Jan05 ? 00:01:07 [irq/76-iwlwifi:]  
root 682 2 0 Jan05 ? 00:01:27 [irq/77-iwlwifi:]  
root 688 2 0 Jan05 ? 00:00:49 [irq/78-iwlwifi:]  
root 695 2 0 Jan05 ? 00:01:39 [irq/79-iwlwifi:]  
root 700 2 0 Jan05 ? 00:01:22 [irq/80-iwlwifi:]  
root 703 2 0 Jan05 ? 00:01:13 [irq/81-iwlwifi:]  
root 704 2 0 Jan05 ? 00:01:38 [irq/82-iwlwifi:]  
root 708 2 0 Jan05 ? 00:00:44 [irq/83-iwlwifi:]  
root 713 2 0 Jan05 ? 00:00:00 [loop9]  
root 715 2 0 Jan05 ? 00:00:00 [irq/84-iwlwifi:]  
root 782 2 0 Jan05 ? 00:00:00 [loop10]  
root 797 2 0 Jan05 ? 00:00:00 [loop11]  
root 811 2 0 Jan05 ? 00:00:00 [loop12]  
root 838 2 0 Jan05 ? 00:00:00 [loop13]  
root 847 2 0 Jan05 ? 00:00:00 [loop14]  
root 879 2 0 Jan05 ? 00:00:00 [loop15]  
root 884 2 0 Jan05 ? 00:00:00 [loop16]  
root 885 2 0 Jan05 ? 00:00:00 [loop17]  
root 945 2 0 Jan05 ? 00:00:00 [loop18]  
root 946 2 0 Jan05 ? 00:00:00 [loop19]  
root 947 2 0 Jan05 ? 00:00:00 [loop20]  
root 1012 2 0 Jan05 ? 00:00:00 [jbd2/nvme0n1p8-]  
root 1013 2 0 Jan05 ? 00:00:00 [ext4-rsv-conver]  
root 1015 2 0 Jan05 ? 00:01:09 [jbd2/nvme0n1p7-]  
root 1016 2 0 Jan05 ? 00:00:00 [ext4-rsv-conver]  
\_rpc 1062 1 0 Jan05 ? 00:00:00 /sbin/rpcbind -f -w  
systemd+ 1063 1 0 Jan05 ? 00:01:24 /lib/systemd/systemd-resolved  
systemd+ 1064 1 0 Jan05 ? 00:00:00 /lib/systemd/systemd-timesyncd  
root 1144 1 0 Jan05 ? 00:00:46 /usr/sbin/acpid  
avahi 1146 1 0 Jan05 ? 00:00:06 avahi-daemon: running [abhijit-laptop.local]  
root 1149 1 0 Jan05 ? 00:00:01 /usr/lib/bluetooth/bluetoothd  
message+ 1150 1 0 Jan05 ? 00:04:21 /usr/bin/dbus-daemon --system --address=systemd: --nofork --  
nopidfile --systemd-activation --syslog-only  
root 1152 1 0 Jan05 ? 00:03:12 /usr/sbin/NetworkManager --no-daemon  
root 1157 1 0 Jan05 ? 00:01:02 /usr/sbin/iio-sensor-proxy  
root 1159 1 0 Jan05 ? 00:00:27 /usr/sbin/irqbalance --foreground  
root 1162 1 0 Jan05 ? 00:00:01 /usr/bin/lxcsfs /var/lib/lxcsfs  
root 1165 1 0 Jan05 ? 00:00:00 /usr/bin/python3 /usr/bin/networkd-dispatcher --run-startup-  
triggers  
root 1170 1 0 Jan05 ? 00:00:29 /usr/lib/policykit-1/polkitd --no-debug  
syslog 1175 1 0 Jan05 ? 00:00:20 /usr/sbin/rsyslogd -n -iNONE  
root 1182 1 0 Jan05 ? 00:01:13 /usr/lib/snapd/snapd  
root 1187 1 0 Jan05 ? 00:00:12 /usr/lib/accounts-service/accounts-daemon  
root 1192 1 0 Jan05 ? 00:00:00 /usr/sbin/cron -f  
root 1198 1 0 Jan05 ? 00:00:00 /usr/libexec/switcheroo-control  
root 1201 1 0 Jan05 ? 00:00:12 /lib/systemd/systemd-logind  
root 1202 1 0 Jan05 ? 00:00:07 /lib/systemd/systemd-machined  
root 1203 1 0 Jan05 ? 00:01:28 /usr/lib/udisks2/udisksd  
root 1204 1 0 Jan05 ? 00:00:15 /sbin/wpa\_supplicant -u -s -O /run/wpa\_supplicant  
daemon 1209 1 0 Jan05 ? 00:00:00 /usr/sbin/atd -f  
avahi 1216 1146 0 Jan05 ? 00:00:00 avahi-daemon: chroot helper  
docker-+ 1279 1 0 Jan05 ? 00:00:22 /usr/bin/docker-registry serve /etc/docker/registry/config.yml  
root 1282 1 0 Jan05 ? 00:00:00 /usr/bin/python3 /usr/bin/twistd3 --nodaemon --pidfile= epoptes  
jenkins 1285 1 0 Jan05 ? 00:15:01 /usr/bin/java -Djava.awt.headless=true -jar  
/usr/share/java/jenkins.war --webroot=/var/cache/jenkins/war --httpPort=8080  
root 1296 1 0 Jan05 ? 00:00:18 php-fpm: master process (/etc/php/7.4/fpm/php-fpm.conf)  
vnstat 1314 1 0 Jan05 ? 00:00:15 /usr/sbin/vnstatd -n  
root 1326 1 0 Jan05 ? 00:00:02 /usr/sbin/ModemManager  
root 1327 1 0 Jan05 ? 00:00:31 /usr/bin/anydesk --service  
colord 1359 1 0 Jan05 ? 00:00:01 /usr/libexec/colord  
root 1374 1 0 Jan05 ? 00:00:00 /usr/sbin/gdm3

```
root      1420      1  0 Jan05 ?      00:00:14 /usr/sbin/apache2 -k start
www-data  1436    1296  0 Jan05 ?      00:00:00 php-fpm: pool www
www-data  1437    1296  0 Jan05 ?      00:00:00 php-fpm: pool www
mysql     1461      1  0 Jan05 ?      00:38:52 /usr/sbin/mysqld
root      1490      1  0 Jan05 ?      00:00:04 /usr/sbin/libvиртd
root      1491      1  0 Jan05 ?      00:00:00 /usr/bin/python3 /usr/share/unattended-upgrades/unattended-
upgrade-shutdown --wait-for-signal
rtkit     1593      1  0 Jan05 ?      00:00:07 /usr/libexec/rtkit-daemon
libvirt+  1766      1  0 Jan05 ?      00:00:00 /usr/sbin/dnsmasq --conf-
file=/var/lib/libvirt/dnsmasq/default.conf --leasefile-ro --dhcp-script=/usr/lib/libvirt/libvirt_leaseshelper
root      1767    1766  0 Jan05 ?      00:00:00 /usr/sbin/dnsmasq --conf-
file=/var/lib/libvirt/dnsmasq/default.conf --leasefile-ro --dhcp-script=/usr/lib/libvirt/libvirt_leaseshelper
root      1859      1  0 Jan05 ?      00:00:18 /usr/lib/upower/upowerd
root      1995      1  0 Jan05 ?      00:00:00 /opt/saltstack/salt/run/run minion
root      2041    1995  0 Jan05 ?      00:05:18 /opt/saltstack/salt/run/run minion MultiMinionProcessManager
MinionProcessManager
root      2278      1  0 Jan05 ?      00:00:50 /usr/bin/dockerd -H fd:// --
containerd=/run/containerd/containerd.sock
root      2282      1  0 Jan05 ?      00:00:17 /usr/sbin/inetd
whoopsie  2302      1  0 Jan05 ?      00:00:01 /usr/bin/whoopsie -f
kernoops  2330      1  0 Jan05 ?      00:00:19 /usr/sbin/kerneloops --test
kernoops  2341      1  0 Jan05 ?      00:00:19 /usr/sbin/kerneloops
root      2366      2  0 Jan05 ?      00:00:00 [iprt-VBoxWQueue]
lxc-dns+  2370      1  0 Jan05 ?      00:00:00 dnsmasq --conf-file=/dev/null -u lxc-dnsmasq --strict-order --
bind-interfaces --pid-file=/run/lxc/dnsmasq.pid --listen-address 10.0.3.1 --dhcp-range 10.0.3.2,10.0.3.254 --dhcp-
lease-max=253 --dhcp-no-override --except-interface=lo --interface=lxcbr0 --dhcp-
leasefile=/var/lib/misc/dnsmasq.lxcbr0.leases --dhcp-authoritative
root      2404      2  0 Jan05 ?      00:00:00 [iprt-VBoxTscThr]
root      3508      1  0 Jan05 ?      00:00:01 /usr/lib/postfix/sbin/master -w
root      3615    1374  0 Jan05 ?      00:00:01 gdm-session-worker [pam/gdm-password]
abhijit   3629      1  0 Jan05 ?      00:00:17 /lib/systemd/systemd --user
abhijit   3630    3629  0 Jan05 ?      00:00:00 (sd-pam)
abhijit   3636    3629  1 Jan05 ?      04:31:21 /usr/bin/pulseaudio --daemonize=no --log-target=journal
abhijit   3638    3629  0 Jan05 ?      00:20:35 /usr/libexec/tracker-miner-fs
abhijit   3642    3629  0 Jan05 ?      00:01:11 /usr/bin/dbus-daemon --session --address(systemd: --nofork --
nopidfile --systemd-activation --syslog-only
abhijit   3644      1  0 Jan05 ?      00:00:03 /usr/bin/gnome-keyring-daemon --daemonize --login
abhijit   3662    3629  0 Jan05 ?      00:00:01 /usr/libexec/gvfsd
abhijit   3667    3629  0 Jan05 ?      00:00:00 /usr/libexec/gvfsd-fuse /run/user/1000/gvfs -f -o big_writes
abhijit   3673    3629  0 Jan05 ?      00:00:02 /usr/libexec/gvfs-udisks2-volume-monitor
abhijit   3681    3629  0 Jan05 ?      00:00:01 /usr/libexec/gvfs-mtp-volume-monitor
abhijit   3685    3629  0 Jan05 ?      00:00:11 /usr/libexec/gvfs-afc-volume-monitor
abhijit   3690    3629  0 Jan05 ?      00:00:01 /usr/libexec/gvfs-gphoto2-volume-monitor
abhijit   3695    3629  0 Jan05 ?      00:00:01 /usr/libexec/gvfs-goa-volume-monitor
abhijit   3700    3629  0 Jan05 ?      00:00:01 /usr/libexec/goa-daemon
root      3701      2  0 Jan05 ?      00:00:00 [krfcomm]
abhijit   3708    3629  0 Jan05 ?      00:00:04 /usr/libexec/goa-identity-service
abhijit   3724    3615  tty2      00:00:00 /usr/lib/gdm3/gdm-x-session --run-script env
GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --systemd --session=ubuntu
abhijit   3726    3724  1 Jan05 tty2      02:47:57 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth
/run/user/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3
abhijit   3747    3724  0 Jan05 tty2      00:00:00 /usr/libexec/gnome-session-binary --systemd --systemd --
session=ubuntu
abhijit   3816    3747  0 Jan05 ?      00:00:01 /usr/bin/ssh-agent /usr/bin/im-launch env
GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --systemd --session=ubuntu
abhijit   3845    3629  0 Jan05 ?      00:00:00 /usr/libexec/at-spi-bus-launcher
abhijit   3850    3845  0 Jan05 ?      00:00:07 /usr/bin/dbus-daemon --config-file=/usr/share/defaults/at-
spi2/accessibility.conf --nofork --print-address 3
abhijit   3869    3629  0 Jan05 ?      00:00:00 /usr/libexec/gnome-session-ctl --monitor
abhijit   3876    3629  0 Jan05 ?      00:00:04 /usr/libexec/gnome-session-binary --systemd-service --
session=ubuntu
abhijit   3890    3629  1 Jan05 ?      03:43:47 /usr/bin/gnome-shell
abhijit   3927    3890  0 Jan05 ?      00:31:49 ibus-daemon --panel disable --xim
abhijit   3931    3927  0 Jan05 ?      00:00:00 /usr/libexec/ibus-dconf
abhijit   3932    3927  0 Jan05 ?      00:01:51 /usr/libexec/ibus-extension-gtk3
abhijit   3934    3629  0 Jan05 ?      00:00:04 /usr/libexec/ibus-x11 --kill-daemon
```

abhijit 3937 3629 0 Jan05 ? 00:00:02 /usr/libexec/ibus-portal  
abhijit 3949 3629 0 Jan05 ? 00:00:27 /usr/libexec/at-spi2-registryd --use-gnome-session  
abhijit 3953 3629 0 Jan05 ? 00:00:00 /usr/libexec/xdg-permission-store  
abhijit 3955 3629 0 Jan05 ? 00:00:01 /usr/libexec/gnome-shell-calendar-server  
abhijit 3964 3629 0 Jan05 ? 00:00:00 /usr/libexec/evolution-source-registry  
abhijit 3973 3629 0 Jan05 ? 00:00:02 /usr/libexec/evolution-calendar-factory  
abhijit 3986 3629 0 Jan05 ? 00:00:01 /usr/libexec/dconf-service  
abhijit 3992 3629 0 Jan05 ? 00:00:01 /usr/libexec/evolution-addressbook-factory  
abhijit 4007 3629 0 Jan05 ? 00:00:00 /usr/bin/gjs /usr/share/gnome-shell/org.gnome.Shell.Notifications  
abhijit 4023 3629 0 Jan05 ? 00:00:00 /usr/libexec/gsd-ally-settings  
abhijit 4025 3629 0 Jan05 ? 00:00:08 /usr/libexec/gsd-color  
abhijit 4029 3629 0 Jan05 ? 00:00:00 /usr/libexec/gsd-datetime  
abhijit 4032 3629 0 Jan05 ? 00:00:21 /usr/libexec/gsd-housekeeping  
abhijit 4033 3629 0 Jan05 ? 00:00:05 /usr/libexec/gsd-keyboard  
abhijit 4036 3629 0 Jan05 ? 00:00:12 /usr/libexec/gsd-media-keys  
abhijit 4037 3629 0 Jan05 ? 00:00:11 /usr/libexec/gsd-power  
abhijit 4038 3629 0 Jan05 ? 00:00:00 /usr/libexec/gsd-print-notifications  
abhijit 4039 3629 0 Jan05 ? 00:00:01 /usr/libexec/gsd-rfkill  
abhijit 4041 3629 0 Jan05 ? 00:00:01 /usr/libexec/gsd-screensaver-proxy  
abhijit 4042 3876 0 Jan05 ? 00:01:06 /usr/lib/x86\_64-linux-gnu/libexec/kdeconnectd  
abhijit 4045 3629 0 Jan05 ? 00:00:35 /usr/libexec/gsd-sharing  
abhijit 4047 3629 0 Jan05 ? 00:00:00 /usr/libexec/gsd-smartcard  
abhijit 4051 3629 0 Jan05 ? 00:00:00 /usr/libexec/gsd-sound  
abhijit 4057 3629 0 Jan05 ? 00:00:00 /usr/libexec/gsd-usb-protection  
abhijit 4063 3629 0 Jan05 ? 00:00:05 /usr/libexec/gsd-wacom  
abhijit 4071 3629 0 Jan05 ? 00:00:00 /usr/libexec/gsd-wwan  
abhijit 4072 3876 0 Jan05 ? 00:01:00 baloo\_file  
abhijit 4075 3876 0 Jan05 ? 00:00:00 /usr/libexec/gsd-disk-utility-notify  
abhijit 4076 3629 0 Jan05 ? 00:00:08 /usr/libexec/gsd-xsettings  
abhijit 4078 3876 0 Jan05 ? 00:00:10 /usr/bin/python3 /usr/bin/blueman-applet  
abhijit 4082 3876 0 Jan05 ? 00:00:14 /usr/bin/anydesk --tray  
abhijit 4108 3876 0 Jan05 ? 00:00:00 /usr/lib/x86\_64-linux-gnu/indicator-messages/indicator-messages-service  
abhijit 4109 3876 0 Jan05 ? 00:00:06 /usr/libexec/evolution-data-server/evolution-alarm-notify  
abhijit 4129 3629 0 Jan05 ? 00:00:50 /snap/snap-store/959/usr/bin/snap-store --application-service  
abhijit 4191 3629 0 Jan05 ? 00:00:00 /usr/libexec/gsd-printer  
abhijit 4219 3629 0 Jan05 ? 00:00:03 /usr/libexec/xdg-document-portal  
abhijit 4265 3927 0 Jan05 ? 00:03:20 /usr/libexec/ibus-engine-simple  
abhijit 4291 3629 0 Jan05 ? 00:00:10 /usr/bin/python3 /usr/bin/blueman-tray  
abhijit 4301 3629 0 Jan05 ? 00:00:00 /usr/lib/bluetooth/obexd  
abhijit 4377 3662 0 Jan05 ? 00:00:02 /usr/libexec/gvfsd-trash --spawner :1.3 /org/gtk/gvfs/exec\_spaw/0  
abhijit 4395 3629 0 Jan05 ? 00:00:12 /usr/libexec/xdg-desktop-portal  
abhijit 4399 3629 0 Jan05 ? 00:01:08 /usr/libexec/xdg-desktop-portal-gtk  
abhijit 4480 3629 0 Jan05 ? 00:00:21 /usr/libexec/gvfsd-metadata  
abhijit 5687 3662 0 Jan05 ? 00:00:00 /usr/libexec/gvfsd-network --spawner :1.3  
/org/gtk/gvfs/exec\_spaw/1  
abhijit 5701 3662 0 Jan05 ? 00:00:01 /usr/libexec/gvfsd-dnssd --spawner :1.3 /org/gtk/gvfs/exec\_spaw/3  
root 6228 2 0 Jan05 ? 00:00:00 [kdmflush]  
root 6236 2 0 Jan05 ? 00:00:00 [kcryptd\_io/253:]  
root 6237 2 0 Jan05 ? 00:00:00 [kcryptd/253:0]  
root 6238 2 0 Jan05 ? 00:00:23 [dmcrypt\_write/2]  
root 6260 2 0 Jan05 ? 00:00:24 [jbd2/dm-0-8]  
root 6261 2 0 Jan05 ? 00:00:00 [ext4-rsv-conver]  
abhijit 6421 3927 0 Jan05 ? 00:00:36 /usr/lib/ibus/ibus-engine-m17n --ibus  
abhijit 6434 3876 0 Jan05 ? 00:00:13 update-notifier  
abhijit 30565 3629 0 Jan05 ? 00:08:56 /usr/libexec/gnome-terminal-server  
abhijit 30576 30565 0 Jan05 pts/0 00:00:00 bash  
abhijit 131017 364845 1 Jan09 ? 03:12:27 /usr/lib/virtualbox/VirtualBoxVM --comment ubuntu 18.04 --startvm 45993a5c-3ded-452f-941e-4579d12c1ad9 --no-startvm-errormsgbox  
abhijit 159668 30565 0 Jan09 pts/10 00:00:00 bash  
abhijit 161637 30565 0 Jan05 pts/1 00:00:01 bash  
abhijit 171109 159668 0 Jan09 pts/10 00:00:33 evince.....  
abhijit 171114 3629 0 Jan09 ? 00:00:00 /usr/libexec/evinced  
abhijit 204055 3629 0 Jan10 ? 00:00:13 /usr/bin/python3 /usr/bin/update-manager --no-update --no-focus-on-map  
abhijit 270190 3629 0 Jan05 ? 00:56:34 /usr/lib/x86\_64-linux-gnu/libexec/kactivitymanagerd

abhijit 270199 3629 0 Jan05 ? 00:00:24 /usr/bin/kglobalaccel5  
abhijit 270207 3629 0 Jan05 ? 00:00:00 kdeinit5: Running...  
abhijit 270208 270207 0 Jan05 ? 00:00:19 /usr/lib/x86\_64-linux-gnu/libexec/kf5/klauncher --fd=8  
abhijit 279971 3629 0 Jan05 ? 01:20:12 telegram-desktop  
abhijit 280011 279971 0 Jan05 ? 00:00:00 sh -c /usr/lib/x86\_64-linux-gnu/libproxy/0.4.15/pxgsettings  
org.gnome.system.proxy org.gnome.system.proxy.http org.gnome.system.proxy.https org.gnome.system.proxy.ftp  
org.gnome.system.proxy.socks  
abhijit 280015 280011 0 Jan05 ? 00:00:00 /usr/lib/x86\_64-linux-gnu/libproxy/0.4.15/pxgsettings  
org.gnome.system.proxy org.gnome.system.proxy.http org.gnome.system.proxy.https org.gnome.system.proxy.ftp  
org.gnome.system.proxy.socks  
abhijit 325756 3629 0 Jan12 ? 00:01:09 /usr/libexec/tracker-store  
root 326592 1 0 Jan12 ? 00:00:00 sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups  
abhijit 347912 3644 0 Jan06 ? 00:00:00 /usr/bin/ssh-agent -D -a /run/user/1000/keyring/.ssh  
root 348716 1 0 Jan12 ? 00:00:28 /usr/bin/containerd  
abhijit 351306 3629 3 Jan12 ? 03:32:42 /usr/lib/firefox/firefox  
abhijit 351429 351306 0 Jan12 ? 00:00:00 /usr/lib/firefox/firefox -contentproc -parentBuildID  
20240108143603 -prefsLen 37272 -prefMapSize 247458 -appDir /usr/lib/firefox/browser {83364ade-74ec-4bcc-94f8-  
9f3779a869f1} 351306 true socket  
abhijit 351458 351306 0 Jan12 ? 00:29:34 /usr/lib/firefox/firefox -contentproc -childID 1 -isForBrowser -  
prefsLen 37337 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -  
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {4dbb54ee-e1a2-41e4-b10b-587526532b78}  
351306 true tab  
abhijit 351495 351306 0 Jan12 ? 00:04:06 /usr/lib/firefox/firefox -contentproc -childID 2 -isForBrowser -  
prefsLen 38090 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -  
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {02f1a4e7-d831-4a5f-a9fd-59a809bb03e8}  
351306 true tab  
abhijit 351717 351306 0 Jan12 ? 00:01:14 /usr/lib/firefox/firefox -contentproc -parentBuildID  
20240108143603 -sandboxingKind 0 -prefsLen 42823 -prefMapSize 247458 -appDir /usr/lib/firefox/browser {ale46009-cfd1-  
46c8-ac9c-19ba8bf3f46a} 351306 true utility  
abhijit 351844 351306 0 Jan12 ? 00:07:23 /usr/lib/firefox/firefox -contentproc -parentBuildID  
20240108143603 -prefsLen 43306 -prefMapSize 247458 -appDir /usr/lib/firefox/browser {91008bef-b6d6-452f-b4a6-  
e78d887b3569} 351306 true rdd  
abhijit 353500 3662 0 Jan06 ? 00:00:00 /usr/libexec/gvfsd-http --spawner :1.3 /org/gtk/gvfs/exec\_spaw/4  
abhijit 364809 3890 0 Jan06 ? 00:19:20 /usr/lib/virtualbox/VirtualBox  
abhijit 364837 3629 0 Jan06 ? 00:16:49 /usr/lib/virtualbox/VBoxXPComIPCD  
abhijit 364845 3629 0 Jan06 ? 00:29:44 /usr/lib/virtualbox/VBoxSVC --auto-shutdown  
root 364957 2 0 Jan06 ? 00:00:00 [dio/dm-0]  
abhijit 369749 30565 0 Jan06 pts/3 00:00:00 bash  
abhijit 369879 30565 0 Jan06 pts/4 00:00:00 bash  
abhijit 379620 30565 0 Jan06 pts/6 00:00:00 bash  
abhijit 381917 3890 0 Jan06 ? 00:00:46 flameshot  
root 386201 2 0 Jan06 ? 00:00:02 [kworker/0:2H-acpi\_thermal\_pm]  
postfix 389291 3508 0 Jan06 ? 00:00:00 qmgr -l -t unix -u  
root 486171 2 0 Jan12 ? 00:00:01 [kworker/0:0H-kblockd]  
abhijit 527395 3629 0 Jan12 ? 00:00:49 /usr/bin/gedit --gapplication-service  
abhijit 551299 351306 0 Jan12 ? 00:01:54 /usr/lib/firefox/firefox -contentproc -childID 438 -isForBrowser -  
prefsLen 35109 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -  
appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {c4ebd70d-bad7-4e7c-9066-39827103c84e}  
351306 true tab  
abhijit 552002 3890 0 Jan12 ? 00:03:32 /opt/Signal/signal-desktop --no-sandbox  
abhijit 552005 552002 0 Jan12 ? 00:00:00 /opt/Signal/signal-desktop --type=zygote --no-zygote-sandbox --no-  
sandbox  
abhijit 552006 552002 0 Jan12 ? 00:00:00 /opt/Signal/signal-desktop --type=zygote --no-sandbox  
abhijit 552037 552005 0 Jan12 ? 00:03:28 /opt/Signal/signal-desktop --type=gpu-process --no-sandbox --  
enable-crash-reporter=18887fa1-4d37-46f0-bf9f-b37513c81cf9,no\_channel --user-data-dir=/home/abhijit/.config/Signal --  
gpu-  
preferences=WAAAAAAAAAAGAAAAAaaaaaaaaaaaaAABgAAAAAAA4AAAAAAAAAAAAAAAABAAAAGAAAAAAAAYAA/-  
--use-gl=angle --use-angle=swiftshader-webgl --shared-files --field-trial-  
handle=0,i,3213315393136307567,15155041764863120512,262144 --disable-  
features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess  
abhijit 552045 552002 0 Jan12 ? 00:00:04 /opt/Signal/signal-desktop --type=utility --utility-sub-  
type=network.mojom.NetworkService --lang=en-GB --service-sandbox-type=none --no-sandbox --enable-crash-  
reporter=18887fa1-4d37-46f0-bf9f-b37513c81cf9,no\_channel --user-data-dir=/home/abhijit/.config/Signal --shared-  
files=v8\_context\_snapshot\_data:100 --field-trial-handle=0,i,3213315393136307567,15155041764863120512,262144 --disable-  
features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess  
abhijit 552103 552002 1 Jan12 ? 00:57:51 /opt/Signal/signal-desktop --type=renderer --enable-crash-

reporter=18887fa1-4d37-46f0-bf9f-b37513c81cf9,no\_channel --user-data-dir=/home/abhijit/.config/Signal --app-path=/opt/Signal/resources/app.asar --no-sandbox --no-zygote --enable-blink-features=CSSPseudoDir,CSSLogical --disable-blink-features=Accelerated2dCanvas,AcceleratedSmallCanvases --first-renderer-process --no-sandbox --disable-gpu-compositing --lang=en-GB --num-raster-threads=4 --enable-main-frame-before-activation --renderer-client-id=4 --time-ticks-at-unix-epoch=-1704847690836396 --launch-time-ticks=218630770368 --shared-files=v8\_context\_snapshot\_data:100 --field-trial-handle=0,i,3213315393136307567,15155041764863120512,262144 --disable-features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess  
abhijit 552149 552002 0 Jan12 ? 00:00:12 /opt/Signal/signal-desktop --type=utility --utility-sub-type=audio.mojom.AudioService --lang=en-GB --service-sandbox-type=none --no-sandbox --enable-crash-reporter=18887fa1-4d37-46f0-bf9f-b37513c81cf9,no\_channel --user-data-dir=/home/abhijit/.config/Signal --shared-files=v8\_context\_snapshot\_data:100 --field-trial-handle=0,i,3213315393136307567,15155041764863120512,262144 --disable-features=HardwareMediaKeyHandling,SpareRendererForSitePerProcess  
abhijit 554660 351306 0 Jan13 ? 00:04:20 /usr/lib/firefox/firefox -contentproc -childID 442 -isForBrowser -prefsLen 35109 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {2cffa6f3-6f8c-4a3f-880d-01cc71baf983}  
351306 true tab  
abhijit 554855 351306 4 Jan13 ? 03:24:06 /usr/lib/firefox/firefox -contentproc -childID 445 -isForBrowser -prefsLen 35109 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {abcba8e9-5b97-4281-ad64-1356841dcf34}  
351306 true tab  
abhijit 556349 351306 0 Jan13 ? 00:01:18 /usr/lib/firefox/firefox -contentproc -childID 451 -isForBrowser -prefsLen 35109 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {6318d453-643c-4b85-9f0f-bfd0a429cd41}  
351306 true tab  
abhijit 557600 3629 0 Jan13 ? 00:00:08 /usr/lib/speech-dispatcher-modules/sd\_espeak-ng /etc/speech-dispatcher/modules/espeak-ng.conf  
abhijit 557607 3629 0 Jan13 ? 00:00:08 /usr/lib/speech-dispatcher-modules/sd\_dummy /etc/speech-dispatcher/modules/dummy.conf  
abhijit 557610 3629 0 Jan13 ? 00:00:08 /usr/lib/speech-dispatcher-modules/sd\_generic /etc/speech-dispatcher/modules/mary-generic.conf  
abhijit 557613 3629 0 Jan13 ? 00:00:00 /usr/bin/speech-dispatcher --spawn --communication-method unix\_socket --socket-path /run/user/1000/speech-dispatcher/speechd.sock  
abhijit 571320 351306 0 Jan13 ? 00:01:16 /usr/lib/firefox/firefox -contentproc -childID 486 -isForBrowser -prefsLen 35110 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {71966a59-b1df-4c44-975c-62d7cd41188c}  
351306 true tab  
abhijit 571410 351306 0 Jan13 ? 00:01:34 /usr/lib/firefox/firefox -contentproc -childID 488 -isForBrowser -prefsLen 35110 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {bfbc15ff-6ca5-4837-b97e-334e0bfc8855}  
351306 true tab  
abhijit 571582 351306 0 Jan13 ? 00:03:32 /usr/lib/firefox/firefox -contentproc -childID 492 -isForBrowser -prefsLen 35110 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {04e791e5-9cde-4e03-9211-141bdb440139}  
351306 true tab  
abhijit 591339 4139495 0 Jan13 pts/9 00:00:00 vi mh-pyt.txt  
root 594356 2 0 Jan13 ? 00:00:00 [dio/nvme0n1p7]  
abhijit 594726 30565 0 Jan13 pts/5 00:00:00 bash  
abhijit 791421 351306 1 Jan14 ? 00:40:27 /usr/lib/firefox/firefox -contentproc -childID 839 -isForBrowser -prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {b6f69be3-7ebe-43c9-9d55-c72161180b5e}  
351306 true tab  
abhijit 794226 351306 0 Jan14 ? 00:02:03 /usr/lib/firefox/firefox -contentproc -childID 848 -isForBrowser -prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {28fddc81-ea39-4496-a1b6-b70d9a8a9c31}  
351306 true tab  
abhijit 797871 351306 0 Jan14 ? 00:02:41 /usr/lib/firefox/firefox -contentproc -childID 851 -isForBrowser -prefsLen 35226 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {b254a3c7-e1f7-4245-9fcf-820074a4e69f}  
351306 true tab  
abhijit 799607 30565 0 Jan14 pts/11 00:00:00 bash  
abhijit 803503 30565 0 Jan14 pts/12 00:00:00 bash  
abhijit 815513 799607 0 Jan15 pts/11 00:00:00 vi timetable-todo2  
abhijit 821738 351306 0 Jan15 ? 00:07:30 /usr/lib/firefox/firefox -contentproc -childID 995 -isForBrowser -prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {90cad852-941e-44b3-8fa4-0d44cbcfda7a}  
351306 true tab

abhijit 895193 351306 0 Jan15 ? 00:00:04 /usr/lib/firefox/firefox -contentproc -childID 1131 -isForBrowser  
-prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja  
-appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {218262bd-4f9e-423a-9368-4d2e44f33125}  
351306 true tab

User	Process ID	Time	Command
root	942398	1 0 10:40 ?	00:00:00 /usr/sbin/cupsd -l
root	942399	1 0 10:40 ?	00:00:00 /usr/sbin/cups-browsed
www-data	942465	1420 0 10:40 ?	00:00:00 /usr/sbin/apache2 -k start
www-data	942466	1420 0 10:40 ?	00:00:00 /usr/sbin/apache2 -k start
www-data	942468	1420 0 10:40 ?	00:00:00 /usr/sbin/apache2 -k start
www-data	942469	1420 0 10:40 ?	00:00:00 /usr/sbin/apache2 -k start
www-data	942470	1420 0 10:40 ?	00:00:00 /usr/sbin/apache2 -k start
www-data	942471	1420 0 10:40 ?	00:00:00 /usr/sbin/apache2 -k start
abhijit	954109	30565 0 11:21 pts/2	00:00:00 bash
abhijit	961628	351306 3 12:36 ?	00:05:34 /usr/lib/firefox/firefox -contentproc -childID 1704 -isForBrowser -prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {cb82b978-ef9d-4a76-b59c-5b13b652a0ec}
351306	true	tab	
root	961877	2 0 12:36 ?	00:00:02 [kworker/u32:5-events_unbound]
root	962113	2 0 12:39 ?	00:00:08 [kworker/u33:3-hci0]
abhijit	968060	351306 0 13:09 ?	00:00:05 /usr/lib/firefox/firefox -contentproc -childID 1749 -isForBrowser -prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {c0ba3673-966e-48a6-b029-91346e348342}
351306	true	tab	
root	968299	2 0 13:11 ?	00:00:02 [kworker/u32:4-events_unbound]
root	969560	2 0 13:23 ?	00:00:01 [kworker/5:1-cgroup_destroy]
root	969608	2 0 13:24 ?	00:00:00 [kworker/u32:7-events_unbound]
root	969648	2 0 13:24 ?	00:00:01 [kworker/2:0-inet_frag_wq]
abhijit	969715	379620 0 13:25 pts/6	00:00:00 ssh root@10.1.101.41
root	970435	2 0 13:27 ?	00:00:01 [kworker/3:2-rcu_gp]
root	971921	2 0 13:33 ?	00:00:00 [kworker/1:2-rcu_gp]
root	972048	2 0 13:35 ?	00:00:00 [kworker/7:2-rcu_gp]
root	972127	2 0 13:37 ?	00:00:01 [kworker/0:1-events_long]
root	972207	2 0 13:37 ?	00:00:01 [kworker/4:1-events]
root	972378	2 0 13:39 ?	00:00:00 [kworker/1:0-cgroup_destroy]
root	972435	2 0 13:39 ?	00:00:00 [kworker/6:2-events]
root	972964	2 0 13:41 ?	00:00:00 [kworker/7:0-events]
root	973166	2 0 13:41 ?	00:00:00 [kworker/u32:0-events_unbound]
root	973193	2 0 13:42 ?	00:00:00 [kworker/2:1-events]
root	973282	2 0 13:43 ?	00:00:00 [kworker/4:2-events]
root	973384	2 0 13:44 ?	00:00:00 [kworker/3:0-rcu_gp]
root	973389	2 0 13:44 ?	00:00:01 [kworker/u33:0-hci0]
root	973391	2 0 13:44 ?	00:00:00 [kworker/6:0-rcu_gp]
root	973392	2 0 13:44 ?	00:00:00 [kworker/5:2-events]
root	973655	2 0 13:45 ?	00:00:00 [kworker/1:1-events]
root	973664	2 0 13:46 ?	00:00:00 [kworker/7:1-events]
root	973965	2 0 13:47 ?	00:00:00 [kworker/2:2-events]
root	974126	2 0 13:48 ?	00:00:00 [kworker/u32:1-kcryptd/253:0]
root	974189	2 0 13:48 ?	00:00:00 [kworker/4:0-events]
root	974190	2 0 13:48 ?	00:00:00 [kworker/0:2-events]
root	974369	2 0 13:49 ?	00:00:00 [kworker/u32:2-nvme-wq]
root	974539	2 0 13:49 ?	00:00:00 [kworker/u32:3-events_unbound]
root	974655	2 0 13:50 ?	00:00:00 [kworker/6:1-events]
root	974690	2 0 13:50 ?	00:00:00 [kworker/5:0-events]
root	974740	2 0 13:50 ?	00:00:00 [kworker/7:3-events]
root	974742	2 0 13:50 ?	00:00:00 [kworker/3:1-pm]
root	974743	2 0 13:50 ?	00:00:00 [kworker/u32:6-events_unbound]
root	974744	2 0 13:50 ?	00:00:00 [kworker/u32:8-kcryptd/253:0]
root	974745	2 0 13:50 ?	00:00:00 [kworker/u32:9-events_unbound]
root	974746	2 0 13:50 ?	00:00:00 [kworker/u32:10-events_unbound]
root	974747	2 0 13:50 ?	00:00:00 [kworker/u32:11-events_unbound]
root	974748	2 0 13:50 ?	00:00:00 [kworker/u32:12-kcryptd/253:0]
root	974749	2 0 13:50 ?	00:00:00 [kworker/u32:13-events_unbound]
root	974750	2 0 13:50 ?	00:00:00 [kworker/u32:14-events_unbound]
root	974751	2 0 13:50 ?	00:00:00 [kworker/u32:15-events_unbound]
root	974752	2 0 14:18 ?	00:00:00 [kworker/u32:16-events_unbound]
root	974753	2 0 14:18 ?	00:00:00 [kworker/u32:17-events_unbound]

```

root      974754      2  0 14:18 ?          00:00:00 [kworker/u32:18-events_unbound]
root      974755      2  0 14:18 ?          00:00:00 [kworker/u32:19-events_unbound]
root      974756      2  0 14:18 ?          00:00:00 [kworker/u32:20-kcryptd/253:0]
root      974757      2  0 14:18 ?          00:00:00 [kworker/u32:21-events_unbound]
root      974758      2  0 14:18 ?          00:00:00 [kworker/u32:22-events_unbound]
root      974759      2  0 14:18 ?          00:00:00 [kworker/u32:23-events_unbound]
root      974760      2  0 14:18 ?          00:00:00 [kworker/u32:24-events_unbound]
root      974761      2  0 14:18 ?          00:00:00 [kworker/u32:25-events_unbound]
root      974762      2  0 14:18 ?          00:00:00 [kworker/u32:26-events_unbound]
root      974763      2  0 14:18 ?          00:00:00 [kworker/u32:27-kcryptd/253:0]
root      974764      2  0 14:18 ?          00:00:00 [kworker/u32:28-kcryptd/253:0]
root      974765      2  0 14:18 ?          00:00:00 [kworker/u32:29-events_unbound]
root      974766      2  0 14:18 ?          00:00:00 [kworker/u32:30+events_unbound]
root      974767      2  0 14:18 ?          00:00:00 [kworker/u32:31-events_unbound]
root      974768      2  0 14:18 ?          00:00:00 [kworker/u32:32-events_unbound]
root      974769      2  0 14:18 ?          00:00:00 [kworker/u32:33-events_unbound]
root      974770      2  0 14:18 ?          00:00:00 [kworker/u32:34-events_unbound]
root      974771      2  0 14:18 ?          00:00:00 [kworker/u32:35-events_unbound]
root      974772      2  0 14:18 ?          00:00:00 [kworker/u32:36-events_unbound]
root      974773      2  0 14:18 ?          00:00:00 [kworker/u32:37-events_unbound]
root      974774      2  0 14:18 ?          00:00:00 [kworker/u32:38-events_unbound]
root      974775      2  0 14:18 ?          00:00:00 [kworker/u32:39-events_unbound]
root      974776      2  0 14:18 ?          00:00:00 [kworker/u32:40-kcryptd/253:0]
root      974778      2  0 14:18 ?          00:00:00 [kworker/u32:42-events_unbound]
root      974779      2  0 14:18 ?          00:00:00 [kworker/3:3-events]
root      974780      2  0 14:18 ?          00:00:00 [kworker/3:4-events]
root      974798      2  0 14:18 ?          00:00:00 [kworker/3:5-events]
root      974995      2  0 14:18 ?          00:00:00 [kworker/u33:2-rb_allocator]
root      975505      2  0 14:20 ?          00:00:00 [kworker/6:3-events]
root      975656      2  0 14:20 ?          00:00:00 [kworker/5:3-events]
root      975657      2  0 14:29 ?          00:00:00 [kworker/1:3-pm]
root      975658      2  0 14:29 ?          00:00:00 [kworker/1:4-events]
abhijit   975722  351306  0 14:29 ?          00:00:02 /usr/lib/firefox/firefox -contentproc -childID 1836 -isForBrowser
-prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja
-appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {d9ea12ea-5b3f-477c-9c08-9a4196ea8d04}
351306 true tab
root      976242      2  0 15:16 ?          00:00:00 [kworker/0:0-events]
root      976243      2  0 15:16 ?          00:00:00 [kworker/0:3-events]
root      976244      2  0 15:16 ?          00:00:00 [kworker/0:4-events]
postfix   976248  3508  0 15:16 ?          00:00:00 pickup -l -t unix -u -c
abhijit   976770  351306  0 15:18 ?          00:00:00 /usr/lib/firefox/firefox -contentproc -childID 1840 -isForBrowser
-prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja
-appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {cb567d37-1c03-46db-966c-632f4b708354}
351306 true tab
abhijit   976836  351306  0 15:19 ?          00:00:00 /usr/lib/firefox/firefox -contentproc -childID 1841 -isForBrowser
-prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja
-appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {b62e1986-221e-481e-bda4-62fb37caa67}
351306 true tab
abhijit   977138  351306  0 15:20 ?          00:00:00 /usr/lib/firefox/firefox -contentproc -childID 1842 -isForBrowser
-prefsLen 35227 -prefMapSize 247458 -jsInitLen 229864 -parentBuildID 20240108143603 -greomni /usr/lib/firefox/omni.ja
-appomni /usr/lib/firefox/browser/omni.ja -appDir /usr/lib/firefox/browser {989da731-7bd5-44ce-abd0-200ca4c84b9b}
351306 true tab
abhijit   977184  594726  0 15:21 pts/5    00:00:00 ps -eaf
abhijit   1664880  3629  0 Jan07 ?          00:00:06 /usr/bin/gnome-calendar --application-service
abhijit   1665340  3629  0 Jan07 ?          00:00:00 /usr/bin/gpg-agent --supervised
abhijit   3872409  3629  0 Jan07 ?          00:00:05 /usr/bin/seahorse --application-service
root      3873244      1  0 Jan07 ?          00:00:55 /sbin/mount.ntfs /dev/nvme0n1p3 /media/abhijit/windows -o
rw,nodev,nosuid,windows_names,uid=1000,gid=1000,uhelper=udisks2
abhijit   3884359  30565  0 Jan07 pts/7    00:00:00 bash
abhijit   4108623  30565  0 Jan08 pts/8    00:00:00 bash
abhijit   4136834  3890  0 Jan08 ?          00:00:00 /usr/lib/libreoffice/program/oosplash --calc
abhijit   4136869  4136834  0 Jan08 ?          00:32:11 /usr/lib/libreoffice/program/soffice.bin --calc
abhijit   4139495  30565  0 Jan08 pts/9    00:00:00 bash

```

The two processes which were created by kernel have PIDs (in increasing order)  ✓ and  ✓

The process that created most of the "graphical" processes is having PID  ✗

Question 9

Partially correct

Mark 0.67 out of 1.00

Select all statements that correctly explain the use/purpose of system calls.

Select one or more:

- a. Handle ALL types of interrupts
- b. Provide an environment for process creation ✓
- c. Handle exceptions like division by zero ✗
- d. Run each instruction of an application program
- e. Allow I/O device access to user processes ✓
- f. Provide services for accessing files ✓
- g. Switch from user mode to kernel mode ✓

Your answer is partially correct.

You have selected too many options.

The correct answers are: Switch from user mode to kernel mode, Provide services for accessing files, Allow I/O device access to user processes, Provide an environment for process creation

**Question 10**

Correct

Mark 1.00 out of 1.00

Predict the output of the program given here.

Assume that all the path names for the programs are correct. For example "/usr/bin/echo" will actually run echo command.

Assume that there is no mixing of printf output on screen if two of them run concurrently.

In the answer replace a new line by a single space.

For example::

good  
output  
should be written as good output

--

```
main() {  
    int i;  
    i = fork();  
    if(i == 0)  
        execl("/usr/bin/echo", "/usr/bin/echo", "hi", 0);  
    else  
        wait(0);  
    fork();  
    execl("/usr/bin/echo", "/usr/bin/echo", "one", 0);  
}
```

Answer: hi one one



The correct answer is: hi one one

**Question 11**

Correct

Mark 0.50 out of 0.50

When you turn your computer ON, on BIOS based systems, you are often shown an option like "Press F9 for boot options". What does this mean?

- a. The choice of which OS to boot from
- b. The choice of the boot loader (e.g. GRUB or Windows-Loader)
- c. The choice of booting slowly or fast
- d. The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded

The correct answer is: The BIOS allows us to choose the boot device, the device from which the boot loader will be loaded

**Question 12**

Partially correct

Mark 0.60 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

- a. The two modes are essential for a multitasking system
- b. The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously✓
- c. There is an instruction like 'iret' to return from kernel mode to user mode✓
- d. The two modes are essential for a multiprogramming system
- e. Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode✓

Your answer is partially correct.

You have correctly selected 3.

The correct answers are: The two modes are essential for a multiprogramming system, The two modes are essential for a multitasking system, There is an instruction like 'iret' to return from kernel mode to user mode, The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously, Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode

**Question 13**

Incorrect

Mark 0.00 out of 1.00

Select all the correct statements about the process init on Linuxes/Unixes.

Select one or more:

- a. init is created by kernel 'by hand'
- b. init is created by kernel by forking itself✗
- c. only a process run by 'root' user can exec 'init'✓
- d. init typically has a pid=1✓
- e. init can not be killed with SIGKILL
- f. any user process can fork and exec init✗
- g. no process can exec 'init'

Your answer is incorrect.

The correct answers are: init is created by kernel 'by hand', init typically has a pid=1, init can not be killed with SIGKILL, only a process run by 'root' user can exec 'init'

**Question 14**

Correct

Mark 1.00 out of 1.00

Select all the correct statements about two modes of CPU operation

Select one or more:

- a. Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode ✓
- b. The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously ✓
- c. There is an instruction like 'iret' to return from kernel mode to user mode ✓
- d. The two modes are essential for a multitasking system ✓
- e. The two modes are essential for a multiprogramming system ✓

Your answer is correct.

The correct answers are: The two modes are essential for a multiprogramming system, The two modes are essential for a multitasking system, There is an instruction like 'iret' to return from kernel mode to user mode, The software interrupt instructions change the mode from user mode to kernel mode and jumps to predefined location simultaneously, Some instructions are allowed to run only in user mode, while all instructions can run in kernel mode

**Question 15**

Correct

Mark 1.00 out of 1.00

Order the following events in boot process (from 1 onwards)

Boot loader	2	✓
Login interface	5	✓
Init	4	✓
Shell	6	✓
BIOS	1	✓
OS	3	✓

Your answer is correct.

The correct answer is: Boot loader → 2, Login interface → 5, Init → 4, Shell → 6, BIOS → 1, OS → 3

**Question 16**

Correct

Mark 1.00 out of 1.00

Consider the following programs

**exec1.c**

```
#include <unistd.h>
#include <stdio.h>
int main() {
    exec("./exec2", "./exec2", NULL);
}
```

**exec2.c**

```
#include <unistd.h>
#include <stdio.h>
int main() {
    exec("/bin/ls", "/bin/ls", NULL);
    printf("hello\n");
}
```

Compiled as

```
cc  exec1.c -o exec1
cc  exec2.c -o exec2
```

And run as

```
$ ./exec1
```

Explain the output of the above command (./exec1)

Assume that /bin/ls , i.e. the 'ls' program exists.

Select one:

- a. Execution fails as one exec can't invoke another exec
- b. Execution fails as the call to exec() in exec1 fails
- c. Execution fails as the call to exec() in exec2 fails
- d. Program prints hello
- e. "ls" runs on current directory ✓

Your answer is correct.

The correct answer is: "ls" runs on current directory

**Question 17**

Correct

Mark 0.50 out of 0.50

Is the terminal a part of the kernel on GNU/Linux systems?

- a. no ✓ wrong
- b. yes

The correct answer is: no

[◀ Surprise Quiz - 1 \(pre-requisites\)](#)

Jump to...

[Surprise Quiz - 3 \(processes, memory management, event driven kernel\), compilation-linking-loading ▶](#)

**Started on** Wednesday, 7 February 2024, 6:06 PM

**State** Finished

**Completed on** Wednesday, 7 February 2024, 6:56 PM

**Time taken** 50 mins 14 secs

**Grade** 15.14 out of 20.00 (75.7%)

Question 1

Correct

Mark 1.00 out of 1.00

Which of the following are NOT a part of job of a typical compiler?

- a. Process the # directives in a C program
- b. Check the program for syntactical errors
- c. Check the program for logical errors ✓
- d. Invoke the linker to link the function calls with their code, extern globals with their declaration
- e. Suggest alternative pieces of code that can be written ✓
- f. Convert high level language code to machine code

The correct answers are: Check the program for logical errors, Suggest alternative pieces of code that can be written

**Question 2**

Correct

Mark 2.00 out of 2.00

Consider the two programs given below to implement the command (ignore the fact that error checks are not done on return values of functions)

```
$ ls ./tmp/asdfksdf >/tmp/ddd 2>&1
```

**Program 1**

```
int main(int argc, char *argv[]) {  
    int fd, n, i;  
    char buf[128];  
  
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);  
    close(1);  
    dup(fd);  
    close(2);  
    dup(fd);  
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);  
}
```

**Program 2**

```
int main(int argc, char *argv[]) {  
    int fd, n, i;  
    char buf[128];  
  
    close(1);  
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);  
    close(2);  
    fd = open("/tmp/ddd", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);  
    execl("/bin/ls", "/bin/ls", ".", "/tmp/asldjfaldfs", NULL);  
}
```

Select all the correct statements about the programs

Select one or more:

- a. Only Program 1 is correct ✓
- b. Program 1 ensures 2>&1 and does not ensure > /tmp/ddd
- c. Only Program 2 is correct
- d. Program 1 makes sure that there is one file offset used for '2' and '1' ✓
- e. Program 2 is correct for > /tmp/ddd but not for 2>&1
- f. Program 2 ensures 2>&1 and does not ensure > /tmp/ddd
- g. Both programs are correct
- h. Both program 1 and 2 are incorrect
- i. Program 2 does 1>&2
- j. Program 2 makes sure that there is one file offset used for '2' and '1'
- k. Program 1 is correct for > /tmp/ddd but not for 2>&1
- l. Program 1 does 1>&2

The correct answers are: Only Program 1 is correct, Program 1 makes sure that there is one file offset used for '2' and '1'

**Question 3**

Partially correct

Mark 0.25 out of 1.00

Select all the correct statements about MMU and its functionality (on a non-demand paged system)

Select one or more:

- a. The operating system interacts with MMU for every single address translation
- b. MMU is a separate chip outside the processor
- c. Illegal memory access is detected in hardware by MMU and a trap is raised
- d. MMU is inside the processor ✓
- e. Illegal memory access is detected by operating system
- f. The Operating system sets up relevant CPU registers to enable proper MMU translations
- g. Logical to physical address translations in MMU are done with specific machine instructions
- h. Logical to physical address translations in MMU are done in hardware, automatically

The correct answers are: MMU is inside the processor, Logical to physical address translations in MMU are done in hardware, automatically, The Operating system sets up relevant CPU registers to enable proper MMU translations, Illegal memory access is detected in hardware by MMU and a trap is raised

**Question 4**

Correct

Mark 1.00 out of 1.00

Select the state that is not possible after the given state, for a process:

- New: Running ✓
- Ready : Waiting ✓
- Running: None of these ✓
- Waiting: Running ✓

**Question 5**

Correct

Mark 1.00 out of 1.00

Mark the statements as True/False w.r.t. the basic concepts of memory management.

True	False	
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The kernel refers to the page table for converting each virtual address to physical address.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	When a process is executing, each virtual address is converted into physical address by the kernel directly.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.
<input checked="" type="radio"/> ✗	<input type="radio"/> ✗	The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.
<input type="radio"/> ✗	<input checked="" type="radio"/> ✗	The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.

The kernel refers to the page table for converting each virtual address to physical address.: False

The compiler generates the address references for code/data/stack/heap in the executable file as per the memory management schema chosen by the compiler itself, and then the kernel ensures that program is executed with this schema.: False

When a process is executing, each virtual address is converted into physical address by the kernel directly.: False

The kernel ensures that the MMU is setup before scheduling a process and then the CPU/MMU ensures that the address translation takes place.: True

When a process is executing, each virtual address is converted into physical address by the CPU hardware directly.: True

The compiler generates address references for code/data/stack/heap in the executable file, depending on the MM architecture provided by CPU and kernel.: True

The compiler interacts with the kernel continuously while compiling a program and obtains the correct set of memory addresses for code/stack/heap/data and then generates the machine code file.: False

Question 6

Correct

Mark 1.00 out of 1.00

Select all the correct statements about zombie processes

Select one or more:

- a. A zombie process occupies space in OS data structures ✓ ✓
- b. If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent
- c. init() typically keeps calling wait() for zombie processes to get cleaned up ✓
- d. A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it ✓
- e. Zombie processes are harmless even if OS is up for long time
- f. A zombie process remains zombie forever, as there is no way to clean it up
- g. A process becomes zombie when its parent finishes
- h. A process can become zombie if it finishes, but the parent has finished before it ✓

The correct answers are: A process becomes zombie when it finishes, and remains zombie until parent calls wait() on it, A process can become zombie if it finishes, but the parent has finished before it, A zombie process occupies space in OS data structures, If the parent of a process finishes, before the process itself, then after finishing the process is typically attached to 'init' as parent, init() typically keeps calling wait() for zombie processes to get cleaned up

**Question 7**

Partially correct

Mark 0.71 out of 1.00

Consider the following code and MAP the file to which each fd points at the end of the code. Assume that files/folders exist when needed with proper permissions and open() calls work.

```
int main(int argc, char *argv[]) {
    int fd1, fd2 = 1, fd3 = 1, fd4 = 1;

    fd1 = open("/tmp/1", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
    fd2 = open("/tmp/2", O_RDONLY);
    fd3 = open("/tmp/3", O_WRONLY | O_CREAT, S_IRUSR|S_IWUSR);
    close(0);
    close(1);
    dup(fd2);
    dup(fd3);
    close(fd3);
    dup2(fd2, fd4);
    printf("%d %d %d %d\n", fd1, fd2, fd3, fd4);
    return 0;
}
```

0	stdin	✗
fd2	/tmp/2	✓
2	stderr	✓
1	stdout	✗
fd1	/tmp/1	✓
fd3	closed	✓
fd4	/tmp/2	✓

The correct answer is: 0 → /tmp/2, fd2 → /tmp/2, 2 → stderr, 1 → /tmp/3, fd1 → /tmp/1, fd3 → closed, fd4 → /tmp/2

**Question 8**

Correct

Mark 1.00 out of 1.00

Select the compiler's view of the process's address space, for each of the following MMU schemes:  
(Assume that each scheme,e.g. paging/segmentation/etc is effectively utilised)

Segmentation	many continuous chunks of variable size	✓
Paging	one continuous chunk	✓
Relocation + Limit	one continuous chunk	✓
Segmentation, then paging	many continuous chunks of variable size	✓

The correct answer is: Segmentation → many continuous chunks of variable size, Paging → one continuous chunk, Relocation + Limit → one continuous chunk, Segmentation, then paging → many continuous chunks of variable size

**Question 9**

Correct

Mark 2.00 out of 2.00

Order the events that occur on a timer interrupt:

Select another process for execution

5	✓
7	✓
1	✓
6	✓
3	✓
4	✓
2	✓

Execute the code of the new process

Change to kernel stack of currently running process

Set the context of the new process

Save the context of the currently running process

Jump to scheduler code

Jump to a code pointed by IDT

The correct answer is: Select another process for execution → 5, Execute the code of the new process → 7, Change to kernel stack of currently running process → 1, Set the context of the new process → 6, Save the context of the currently running process → 3, Jump to scheduler code → 4, Jump to a code pointed by IDT → 2

**Question 10**

Partially correct

Mark 1.80 out of 2.00

Match the elements of C program to their place in memory

#include files

No Memory needed



Local Variables

Stack



Global Static variables

Data



Global variables

Data



Local Static variables

Data



Function code

Code



Code of main()

Code



Arguments

Stack



#define MACROS

No Memory needed



Allocated Memory

Heap



The correct answer is: #include files → No memory needed, Local Variables → Stack, Global Static variables → Data, Global variables → Data, Local Static variables → Data, Function code → Code, Code of main() → Code, Arguments → Stack, #define MACROS → No Memory needed, Allocated Memory → Heap

Question 11

Incorrect

Mark 0.00 out of 1.00

A process blocks itself means

- a. The application code calls the scheduler
- b. The kernel code of an interrupt handler, moves the process to a waiting queue and calls scheduler X
- c. The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler
- d. The kernel code of system call calls scheduler

The correct answer is: The kernel code of system call, called by the process, moves the process to a waiting queue and calls scheduler

Question 12

Partially correct

Mark 0.25 out of 1.00

Predict the output of the program given here.

Assume that there is no mixing of printf output on screen if two of them run concurrently.

In the answer replace a new line by a single space.

For example::

good

output

should be written as good output

--

```
int main() {  
    int pid;  
    printf("hi\n");  
    pid = fork();  
    if(pid == 0) {  
        exit(0);  
    }  
    printf("bye\n");  
    fork();  
    printf("ok\n");  
}
```

Answer: hi bye ok



The correct answer is: hi bye ok ok

Question 13

Incorrect

Mark 0.00 out of 1.00

Select the sequence of events that are NOT possible, assuming an interruptible kernel code

Select one or more:

- a. P1 running ✖

keyboard hardware interrupt  
keyboard interrupt handler running  
interrupt handler returns  
P1 running  
P1 makes system call  
system call returns  
P1 running  
timer interrupt  
scheduler  
P2 running

- b. P1 running ✖

P1 makes system call and blocks  
Scheduler  
P2 running  
P2 makes system call and blocks  
Scheduler  
P3 running  
Hardware interrupt  
Interrupt unblocks P1  
Interrupt returns  
P3 running  
Timer interrupt  
Scheduler  
P1 running

- c. P1 running

P1 makes system call  
timer interrupt  
Scheduler  
P2 running  
timer interrupt  
Scheduler  
P1 running  
P1's system call return

- d.

P1 running  
P1 makes system call  
Scheduler  
P2 running  
P2 makes system call and blocks  
Scheduler  
P1 running again

- e. P1 running

P1 makes system call  
system call returns  
P1 running  
timer interrupt  
Scheduler running  
P2 running

- f. P1 running

P1 makes system call and blocks  
Scheduler

P2 running  
P2 makes system call and blocks  
Scheduler  
P1 running again

The correct answers are: P1 running

P1 makes system call and blocks

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again,

P1 running

P1 makes system call

Scheduler

P2 running

P2 makes system call and blocks

Scheduler

P1 running again

**Question 14**

Correct

Mark 1.00 out of 1.00

Select the order in which the various stages of a compiler execute.

- |                              |                |   |
|------------------------------|----------------|---|
| Intermediate code generation | 3              | ✓ |
| Loading                      | does not exist | ✓ |
| Linking                      | 4              | ✓ |
| Syntactical Analysis         | 2              | ✓ |
| Pre-processing               | 1              | ✓ |

The correct answer is: Intermediate code generation → 3, Loading → does not exist, Linking → 4, Syntactical Analysis → 2, Pre-processing → 1

Question 15

Partially correct

Mark 0.25 out of 1.00

Select the correct statements about paging (not demand paging) mechanism

Select one or more:

- a. Page table is accessed by the MMU as part of execution of an instruction ✓
- b. The PTBR is loaded by the OS ✓
- c. Page table is accessed by the OS as part of execution of an instruction ✗
- d. An invalid entry on a page means, either it was illegal memory reference or the page was not present in memory. ✗
- e. An invalid entry on a page means, it was an illegal memory reference
- f. OS creates the page table for every process ✓
- g. User process can update its own PTBR
- h. User process can update its own page table entries

The correct answers are: OS creates the page table for every process, The PTBR is loaded by the OS, Page table is accessed by the MMU as part of execution of an instruction, An invalid entry on a page means, it was an illegal memory reference

**Question 16**

Partially correct

Mark 0.88 out of 1.00

Consider the image given below, which explains how paging works.

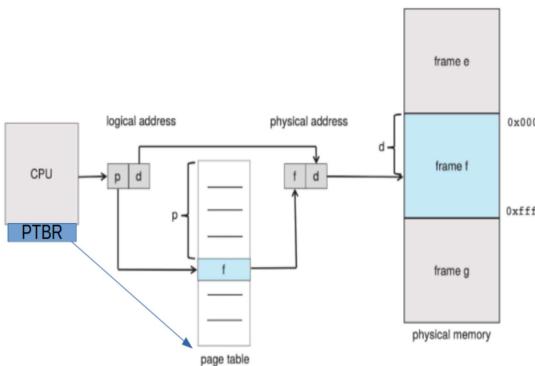


Figure 9.8 Paging hardware.

Mention whether each statement is True or False, with respect to this image.

True	False
<input checked="" type="radio"/>	<input type="radio"/>
<input checked="" type="radio"/>	<input type="radio"/>
<input type="radio"/>	<input checked="" type="radio"/>
<input checked="" type="radio"/>	<input type="radio"/>
<input type="radio"/>	<input checked="" type="radio"/>
<input checked="" type="radio"/>	<input type="radio"/>
<input checked="" type="radio"/>	<input type="radio"/>
<input type="radio"/>	<input checked="" type="radio"/>
<input checked="" type="radio"/>	<input type="radio"/>
<input type="radio"/>	<input checked="" type="radio"/>
<input checked="" type="radio"/>	<input type="radio"/>

The page table is itself present in Physical memory

✓

Maximum Size of page table is determined by number of bits used for page number

✓

The page table is indexed using frame number

✓

Size of page table is always determined by the size of RAM

✗

The PTBR is present in the CPU as a register

✓

The physical address may not be of the same size (in bits) as the logical address

✓

The locating of the page table using PTBR also involves paging translation

✓

The page table is indexed using page number

✓

The page table is itself present in Physical memory: True

Maximum Size of page table is determined by number of bits used for page number: True

The page table is indexed using frame number: False

Size of page table is always determined by the size of RAM: False

The PTBR is present in the CPU as a register: True

The physical address may not be of the same size (in bits) as the logical address: True

The locating of the page table using PTBR also involves paging translation: False

The page table is indexed using page number: True

Question 17

Correct

Mark 1.00 out of 1.00

which of the following is not a difference between real mode and protected mode

- a. in real mode the addressable memory is less than in protected mode
- b. processor starts in real mode
- c. in real mode general purpose registers are 16 bit, in protected mode they are 32 bit
- d. in real mode the segment is multiplied by 16, in protected mode segment is used as index in GDT
- e. in real mode the addressable memory is more than in protected mode ✓

The correct answer is: in real mode the addressable memory is more than in protected mode

[◀ Surprise Quiz - 2](#)

Jump to...

[Questions for test on kalloc/kfree/kvmalloc, etc. ►](#)

# **Introduction to Applications, Files, Linux commands and Basics of System Calls**

Abhijit A. M.  
[abhijit.comp@coep.ac.in](mailto:abhijit.comp@coep.ac.in)

# **Why GNU/Linux ?**

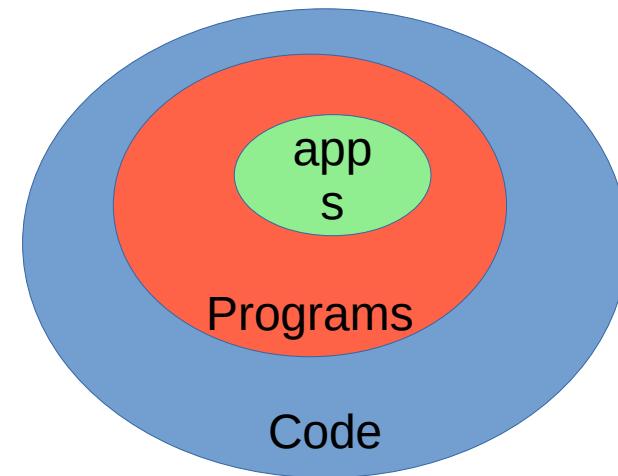
# Few Key Concepts

- **What is a file?**

- File is a “dumb” sequence of bytes lying on the hard disk (or SSD or CD or PD, etc)
- It has a name, owner, size, etc.
- Does not do anything on its own ! Just stays there !

# Few Key Concepts

- **What is an application?**
  - Application is a program that runs in an “environment” created by the operating system, under the control of the operating system
    - Hence also called “User Applications”
  - Words: Program, Application, Code.
    - Code: Any piece of complete/incomplete programming language
    - Program: Any piece of “complete” code (OS, device drivers, applications, ...)
    - Application: as above



# Few Key Concepts

- ***Files don't open themselves***
  - Always some application/program open()s a file.
- ***Files don't display themselves***
  - A file is displayed by the program which opens it. Each program has its own way of handling files
  - It's possible NOT TO HAVE an application to display/show a file

# Few Key Concepts

- **Programs don't run themselves**
  - You click on a program, or run a command --> equivalent to request to Operating System to run it. The OS runs your program
- **Users (humans) request OS to run programs, using Graphical or Command line interface**
  - and programs open files

# Path names

- Tree like directory structure
  - Root directory called /
  - Programs need to identify files using path names
    - A absolute/complete path name for a file.
      - /home/student/a.c
    - Relative path names, . and .. notation
      - concept: every running program has a *current working directory*
      - . current directory
      - .. parent directory
- E.g. ./Desktop/xyz/..../p.c

# A command

- **Name of an executable file**
  - For example: 'ls' is actually “/bin/ls”
- **Command takes arguments**
  - E.g. ls /tmp/
- **Command takes options**
  - E.g. ls -a

# A command

- Command can take both arguments and options
  - E.g. `ls -a /tmp/`
- Options and arguments are basically `argv[]` of the `main()` of that program

```
int main(int argc, char *argv[]) {  
    int i;  
    for(i = 0; i < argc; i++)  
        printf("%s\n", argv[i]);  
}
```

- `$ ./a.out hi hello 123 /a/b/c.c ./m/a.c`

# Basic Navigation Commands

- `pwd`
- `ls`
  - `ls -l`
  - `ls -l /tmp/`
  - `ls -l /home/student/Desktop`
  - `ls -l ./Desktop`
  - `ls -a`
  - `\ls -F`
- `cd`
  - `cd /tmp/`
  - `cd`
  - `cd /home/student/Desktop`
- **notation: ~**
  - `cd ~`
  - `cd ~/Desktop`
  - `ls ~/Desktop`

Map these commands  
to navigation using a  
graphical file browser

# Before the command line, the concept of Shell and System calls

- **System Call**
  - *Applications* often need to tasks involving hardware
    - Reading input, printing on screen, reading from network, etc.
  - They are not permitted to do it *directly* and compelled to do it using functionality given by OS
    - How is this done? We'll learn in later few lectures.
  - This functionality is called “system calls”

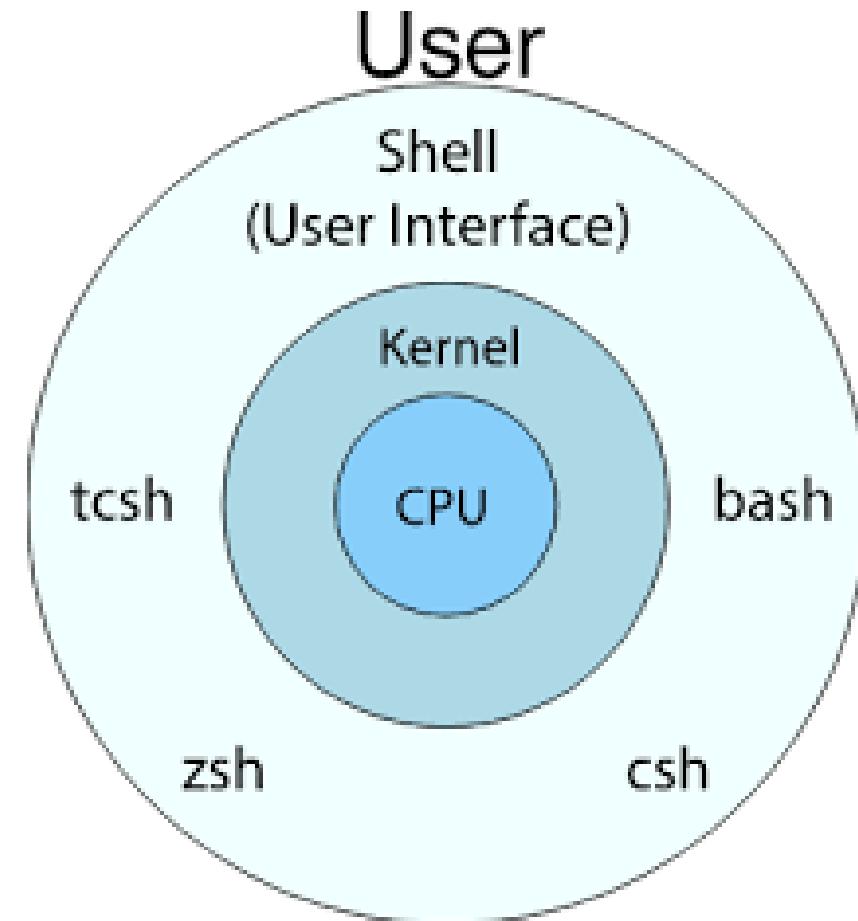
# Before the command line, the concept of Shell and System calls

- **System Call**

- A function from OS code
- Does specific operations with hardware (e.g. reading from keyboard, writing to screen, *opening* a file from disk, etc.)
- Applications can't access hardware directly, they have to request the OS using system calls
- Examples
  - `open("/x/y", ...)`
  - `read(fd, &a, ...);`
  - `fork()`
  - `exec("/usr/bin/ls" ...)`

# The Shell

- **Shell = Cover**
- **Covers some of the Operating System's "System Calls" (mainly fork+exec) for the Applications**
- **Talks with Users and Applications and does some talk with OS**



Not a very accurate diagram !

# The Shell

Shell waits for user's input

Requests the OS to run a program which the user  
has asked to run

Again waits for user's input

GUI is a Shell !

# Let's Understand fork() and exec()

```
#include <unistd.h>  
  
int main() {  
    fork();  
    printf("hi\n");  
    return 0;  
}
```

```
#include <unistd.h>  
  
int main() {  
    printf("hi\n");  
    execl("/bin/ls",  
          "ls", NULL);  
    printf("bye\n");  
    return 0;  
}
```

# A simple shell

```
#include <stdio.h>
#include <unistd.h>
int main() {
    char string[128];
    int pid;
    while(1) {
        printf("prompt>");
        scanf("%s", string);
        pid = fork();
        if(pid == 0) {
            execl(string, string, NULL);
        } else {
            wait(0);
        }
    }
}
```

# Users on Linux

- **Root and others**
  - **root**
    - **superuser, can do (almost) everything**
    - **Uid = 0**
  - **Other users**
    - **Uid != 0**
    - **UID, GID understood by kernel**
- **Groups**
  - **Set of users is a group**
  - **Any number of groups is possible**
- **users/groups data in Text files: /etc/passwd /etc/shadow /etc/group ...**

# File Permissions on Linux

- **3 sets of 3 permission**
  - Octal notation: Read = 4, Write = 2, Execute = 1
  - 644 means
    - Read-Write for owner, Read for Group, Read for others
- **chmod command, used to change permissions, uses these notations**
  - It calls the `chmod()` system call
- **Permissions are for processes started by the user, but in common language often we say “permissions are for the user”**

# File Permissions on Linux

-rw-r--r-- 1 abhijit abhijit 1183744 May 16 12:48 01\_linux\_basics.ppt

-rw-r--r-- 1 abhijit abhijit 341736 May 17 10:39 Debian Family Tree.svg

drwxr-xr-x 2 abhijit abhijit 4096 May 17 11:16 fork-exec

-rw-r--r-- 1 abhijit abhijit 7831341 May 11 12:13 foss.odp

3 sets of 3 permissions

3 sets = user (owner),  
group, others

3 permissions = read,  
write, execute

Owner

size

name

last-modification

hard link count

# File Permissions on Linux

- **r on a file : can read the file**
  - `open(... O_RDONLY)` works
- **w on a file: can modify the file**
  - `open(... O_WRONLY)` works
- **x on a file: can ask the os to run the file as an executable program**
  - `exec(...)` works
- **r on a directory: can do 'ls'**
- **w on a directory: can add/remove files from that directory (even without 'r'!)**
- **x on a directory: can 'cd' to that directory**

# Access rights examples

- **- rW- r-- r--**  
Readable and writable for file owner (actually a process started by the owner!), only readable for others
- **- rW- r- - - -**  
Readable and writable for file owner, only readable for users belonging to the file group.
- **drwx- - - - -**  
Directory only accessible by its owner
- **- - - - - r- x**  
File executable by others but neither by your friends nor by yourself.  
Nice protections for a trap...

# Permissions: more !

- **Setuid/setgid bit**

```
$ ls -l /usr/bin/passwd
```

```
-rwsr-xr-x 1 root root 68208 Nov 29 17:23 /usr/bin/passwd
```

- **How to set the s bit?**

- chmod u+s <filename>

- **What does this mean?**

- Any user can run this process, but the process itself runs as if run by the owner of the file
    - passwd runs as if run by “root” even if you run it

# Man Pages (self study)

- **Manpage**
  - \$ man ls
  - \$ man 2 mkdir
  - \$ man man
  - \$ man -k mkdir
- **Manpage sections**
  - **1 User-level cmds and apps**
    - /bin/mkdir
  - **2 System calls**
    - int mkdir(const char \*, ...);
  - **3 Library calls**
    - int printf(const char \*, ...);
- **4 Device drivers and network protocols**
  - /dev/tty
- **5 Standard file formats**
  - /etc/hosts
- **6 Games and demos**
  - /usr/games/fortune
- **7 Misc. files and docs**
  - man 7 locale
- **8 System admin. Cmds**
  - /sbin/reboot

# GNU / Linux filesystem structure

Not imposed by the system. Can vary from one system to the other, even between two GNU/Linux installations!

/	Root directory
/bin/	Basic, essential system commands
/boot/	Kernel images, initrd and configuration files
/dev/	Files representing devices /dev/hda: first IDE hard disk
/etc/	System and application configuration files
/home/	User directories
/lib/	Basic system shared libraries

# GNU / Linux filesystem structure (self study)

/lost+found	Corrupt files the system tried to recover
/media	Mount points for removable media: /media/usbdisk, /media/cdrom
/mnt/ filesystems	Mount points for temporarily mounted
/opt/	Specific tools installed by the sysadmin /usr/local/ often used instead
/proc/	Access to system information /proc/cpuinfo, /proc/version ...
/root/	root user home directory
/sbin/	Administrator-only commands
/sys/	System and device controls (cpu frequency, device power, etc.)

# GNU / Linux filesystem structure (self study)

/tmp/

Temporary files

/usr/

Regular user tools (not essential to the system)

/usr/bin/, /usr/lib/, /usr/sbin...

/usr/local/

Specific software installed by the sysadmin  
(often preferred to /opt/)

/var/

Data used by the system or system servers

/var/log/, /var/spool/mail (incoming  
mail), /var/spool/lpd (print jobs)...

# Files: cut, copy, paste, remove, (self study)

- **cat <filenames>**
  - cat /etc/passwd
  - cat fork.c
  - cat <filename1> <filename2>
- **cp <source> <target>**
  - cp a.c b.c
  - cp a.c /tmp/
  - cp a.c /tmp/b.c
  - cp -r ./folder1 /tmp/
  - cp -r ./folder1 /tmp/folder2
- **mv <source> <target>**
  - mv a.c b.c
  - mv a.c /tmp/
  - mv a.c /tmp/b.c
- **rm <filename>**
  - rm a.c
  - rm a.c b.c c.c
  - rm -r /tmp/a
- **mkdir**
  - mkdir /tmp/a /tmp/b
- **rmdir**
  - rmdir /tmp/a /tmp/b

# Useful Commands (self study)

- **echo**
  - echo hi
  - echo hi there
  - echo "hi there"
  - j=5; echo \$j
- **sort**
  - sort
  - sort < /etc/passwd
- **firefox**
- **libreoffice**
- **grep**
  - grep bash /etc/passwd
  - grep -i display /etc/passwd
  - egrep -i 'a|b' /etc/passwd
- **less <filename>**
- **head <filename>**
  - head -5 <filename>
  - tail -10 <filename>

# Useful Commands (self study)

- **alias**  
`alias ll='ls -l'`
- **tar**  
`tar cvf folder.tar folder`
- **gzip**  
`gzip a.c`
- **touch**  
`touch xy.txt`  
`touch a.c`
- **strings**  
`strings a.out`
- **adduser**  
`sudo adduser test`
- **su**  
`su administrator`

# Useful Commands (self study)

- **df**

**df -h**

- **du**

**du -hs .**

- **bc**

- **time**

- **date**

- **diff**

- **wc**

- **dd**

# Network Related Commands (self study)

- **ifconfig**
- **ssh**
- **scp**
- **telnet**
- **ping**
- **w**
- **last**
- **whoami**

# Unix job control

- Start a background process:
  - gedit a.c &
  - gedit
    - hit ctrl-z
  - bg
- Where did it go?
  - jobs
  - ps
- Terminate the job: kill it
  - kill *%jobid*
  - kill *pid*
- Bring it back into the foreground
  - fg %1

# Configuration Files

- Most applications have configuration files in TEXT format
- Most of them are in `/etc`
- `/etc/passwd` and `/etc/shadow`
  - Text files containing user accounts
- `/etc/resolv.conf`
  - DNS configuration
- `/etc/network/interfaces`
  - *Network configuration*
- `/etc/hosts`
  - Local database of Hostname-IP mappings
- `/etc/apache2/apache2.conf`
  - Apache webserver configuration

# **~/.bashrc file (self study)**

- **~/.bashrc**  
**Shell script read each time a bash shell is started**
- **You can use this file to define**
  - Your default environment variables (**PATH, EDITOR...**).
  - Your aliases.
  - Your prompt (see the **bash** manual for details).
  - A greeting message.
- **Also ~/.bash\_history**

# **Mounting**

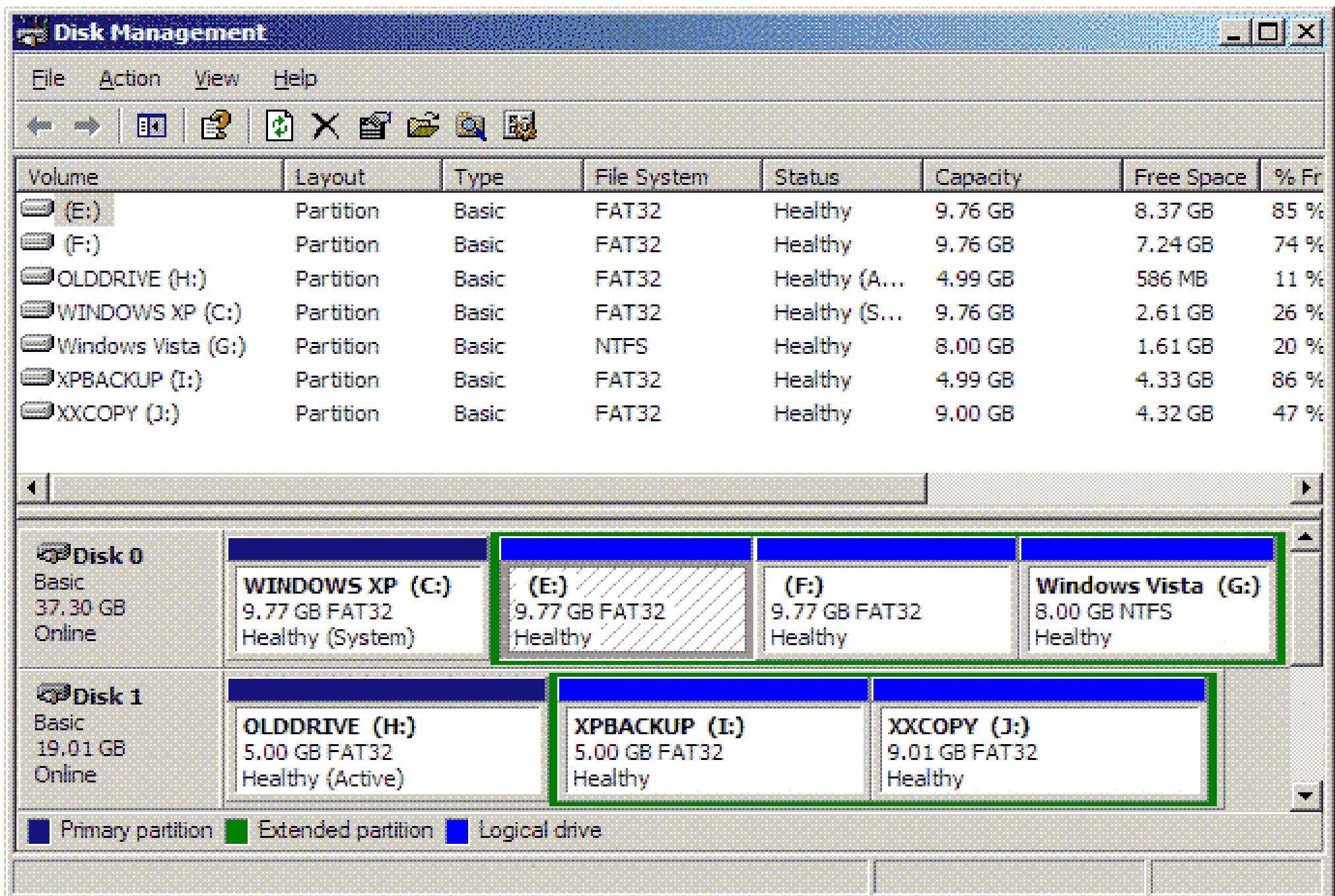
# Partition

- **What is C:\ , D:\, E:\ etc on your computer ?**
  - “Drive” is the popular term
  - Typically one of them represents a CD/DVD RW
- **What do the others represent ?**
  - They are “partitions” of your “hard disk”

# Partition

- Your hard disk is one contiguous chunk of storage
  - Lot of times we need to “logically separate” our storage
  - Partition is a “logical division” of the storage
  - Every “drive” is a partition
- A logical chunk of storage is partition
  - Hard disk partitions (C:, D:), CD-ROM, Pen drive, ...

# Partitions



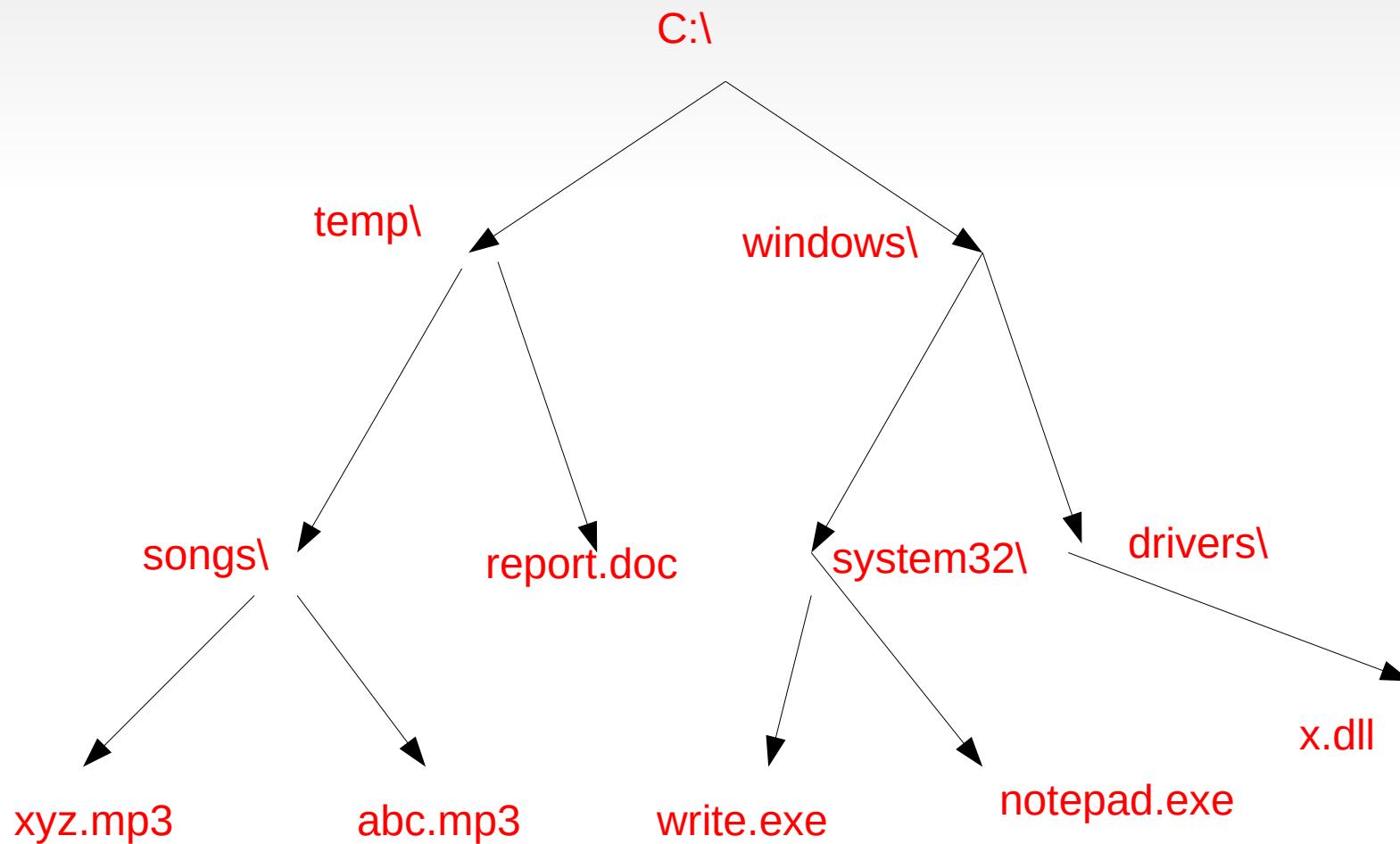
# Managing partitions and hard drives

- **System → Administration → Disk Utility**
- **Use gparted or fdisk to partition drives on Linux**
- **Had drive partition names on Linux**
  - `/dev/sda` → Entire hard drive
  - `/dev/sda1, /dev/sda2, /dev/sda3, ....` Different partitions of the hard drive
  - Each partition has a *type* – ext4, ext3, ntfs, fat32, etc.
- **Formatting: creating an empty layout on disk, layout capable of storing the tree of files/folders**
  - There are different layouts named ext4, ext2, ntfs, etc.

# Windows Namespace

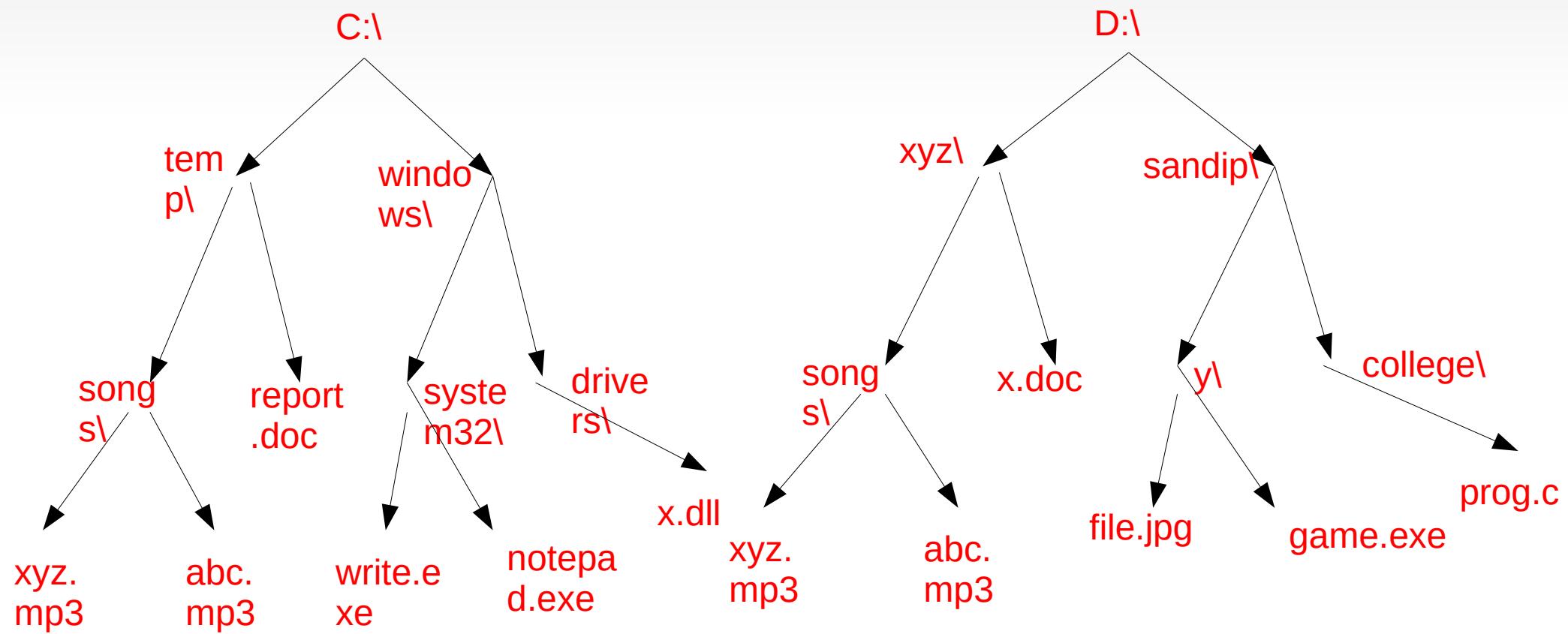
c:\temp\songs\xyz.mp3

- Root is C:\ or D:\ etc
- Separator is also “\”



# Windows Namespace

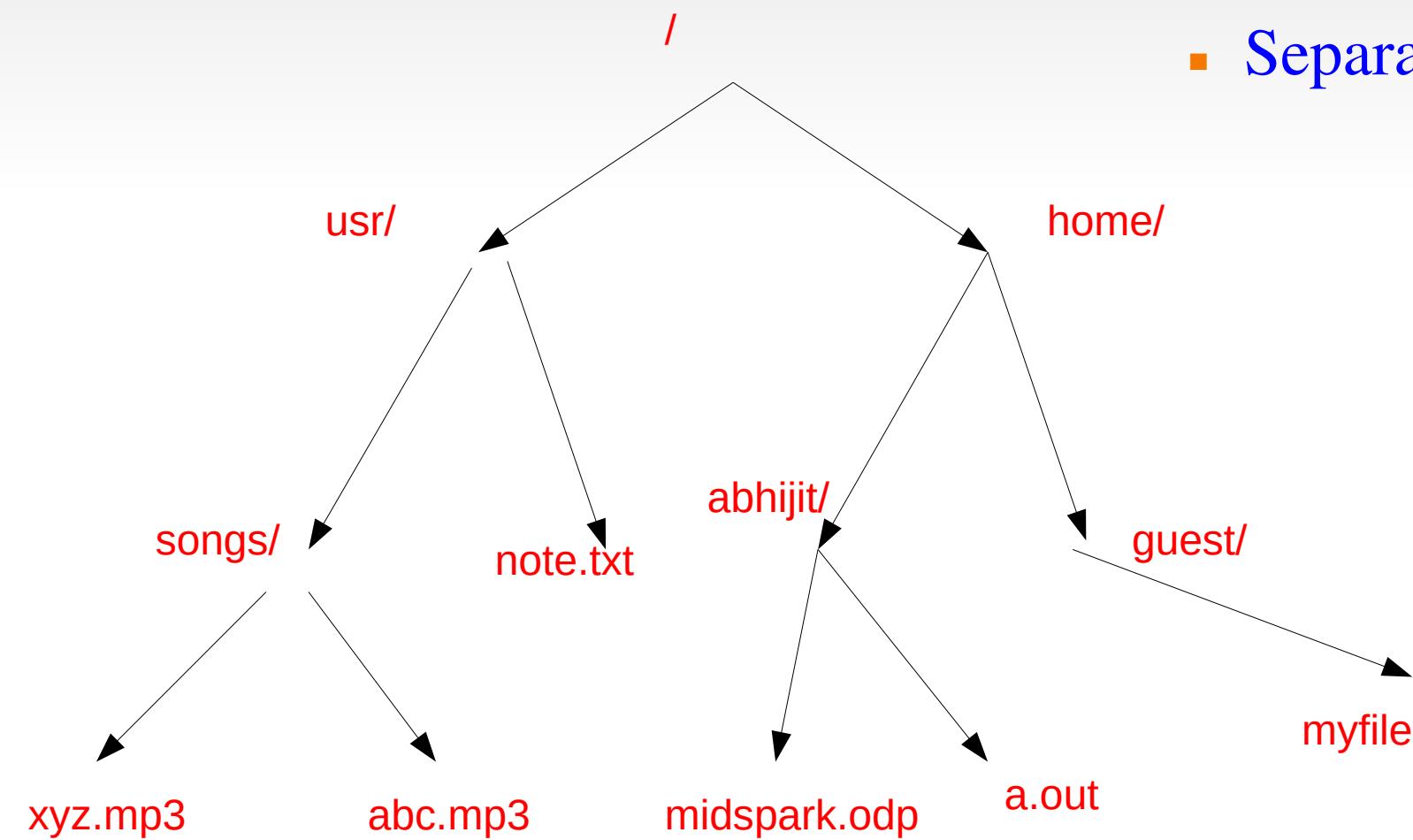
- C:\ D:\ Are partitions of the disk drive
- Typical convention: C: contains programs, D: contains data
- One “tree” per partition
  - Together they make a “forest”



# Linux Namespace: On a partition

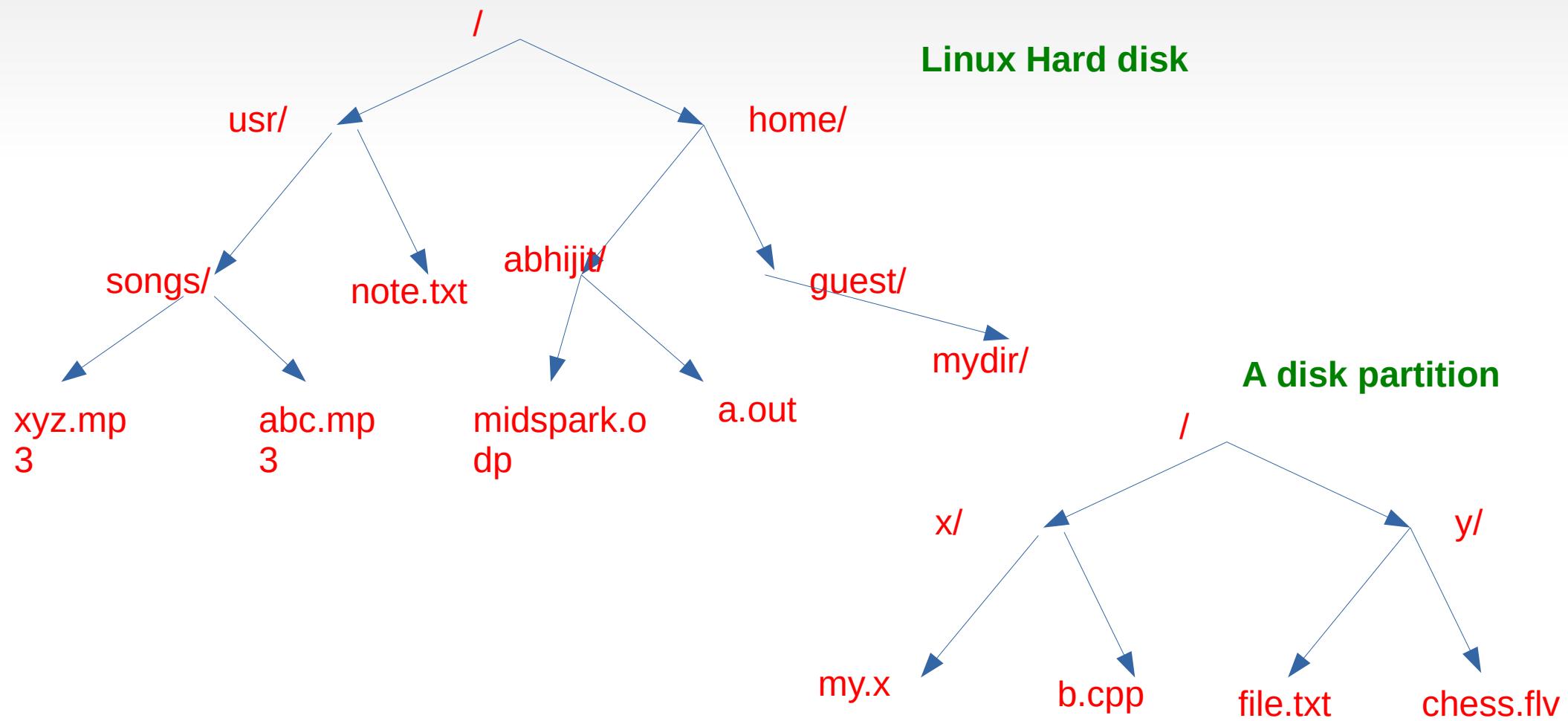
/usr/songs/xyz.mp3

- On every partition:
  - Root is “/”
  - Separator is also “/”

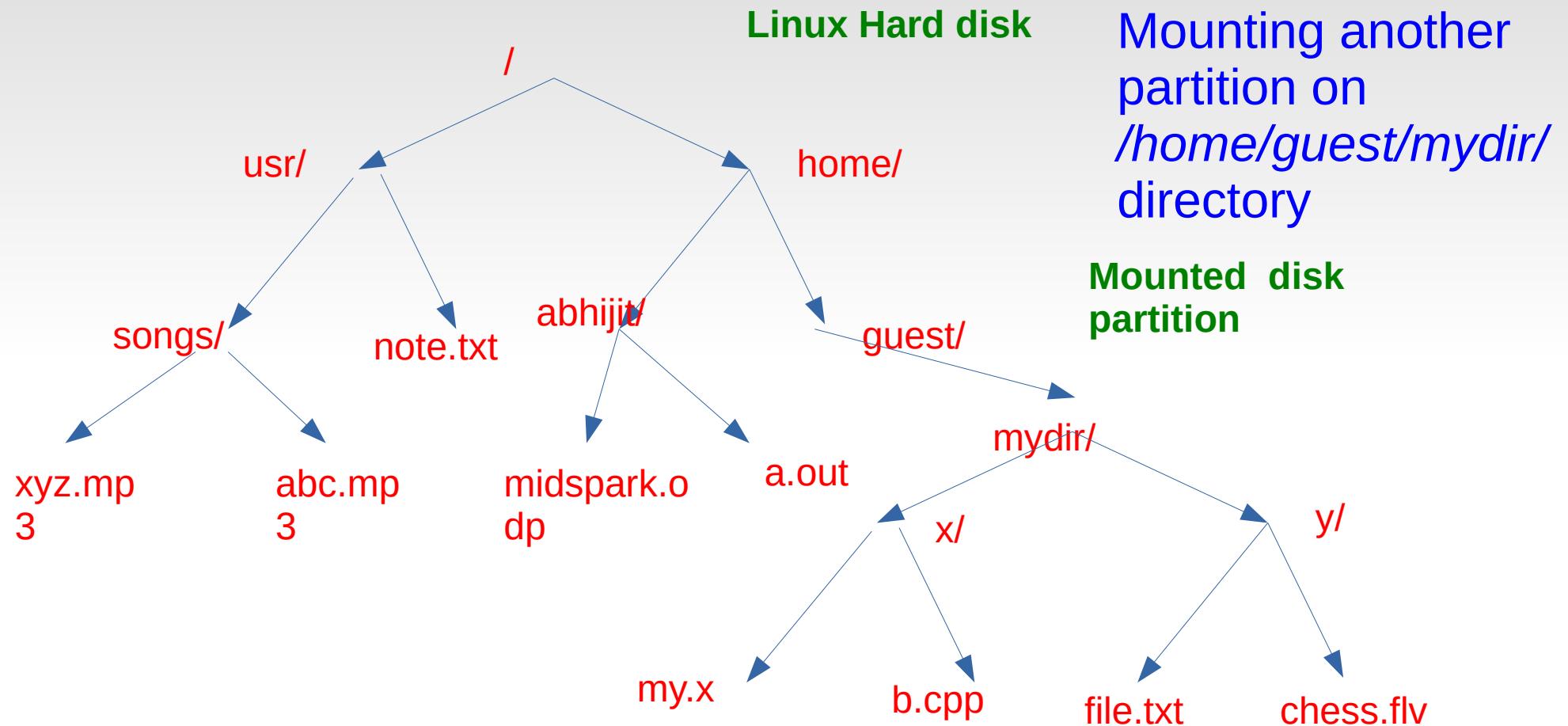


# Linux namespace: Mount

- Linux namespace is a single “tree” and not a “forest” like Windows
- Combining of multiple trees is done through “mount”



# Linux namespace Mounting a partition



/home/guest/mydir/x/b.cpp → way to access the file on the other disk partition

# Mounting across network!

Using Network File System (NFS)

`sudo apt install nfs-common`

`$ sudo mount 172.16.1.75:/mnt/data /myfolder`

**Files that are not regular/directory**

# Special devices (1)

## Device files with a special behavior or contents

- **/dev/null**

The data sink! Discards all data written to this file.

Useful to get rid of unwanted output, typically log information:

```
mplayer black_adder_4th.avi &> /dev/null
```

- **/dev/zero**

Reads from this file always return \0 characters

Useful to create a file filled with zeros:

```
dd if=/dev/zero of=disk.img bs=1k count=2048
```

See `man null` or `man zero` for details

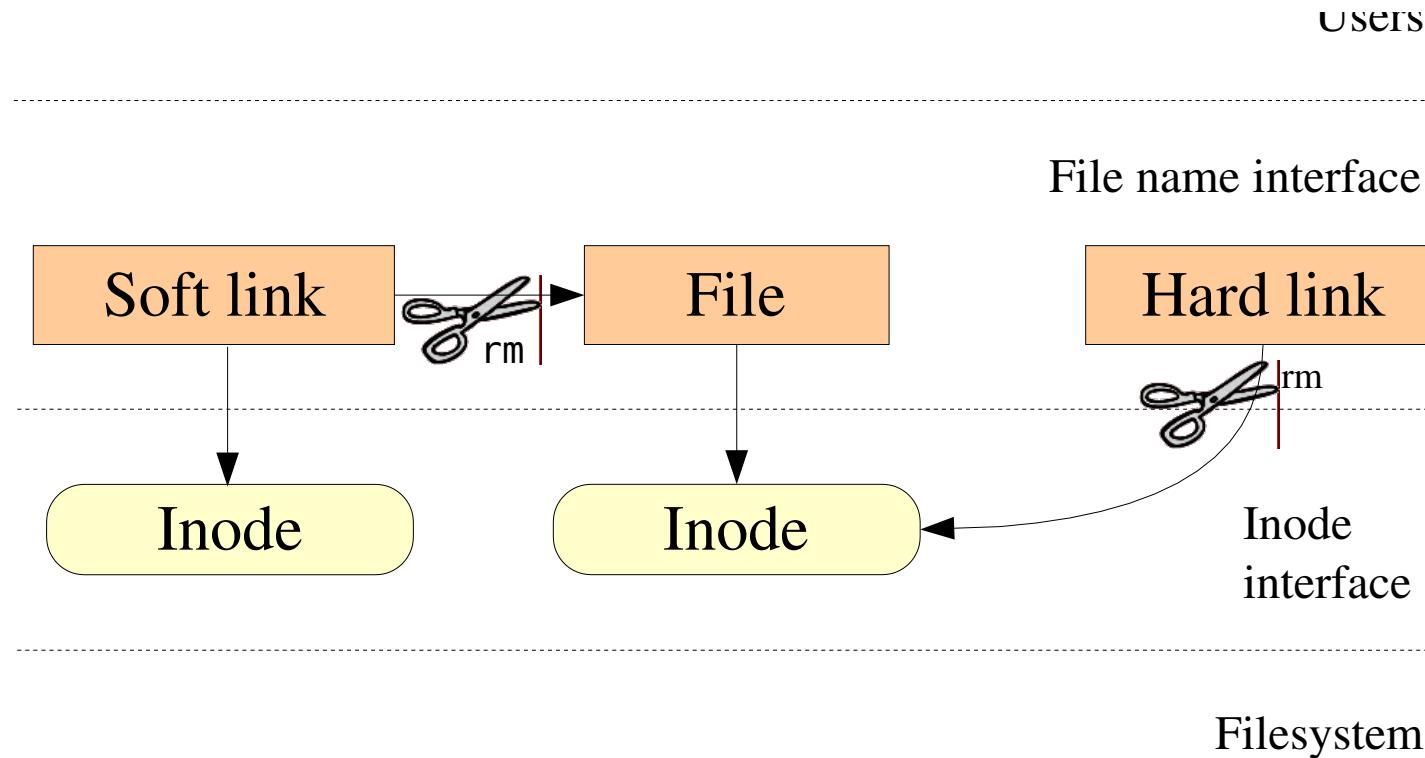
# Special devices (2)

- **/dev/random**  
Returns random bytes when read. Mainly used by cryptographic programs. Uses interrupts from some device drivers as sources of true randomness (“entropy”).  
Reads can be blocked until enough entropy is gathered.
- **/dev/urandom**  
For programs for which pseudo random numbers are fine.  
Always generates random bytes, even if not enough entropy is available (in which case it is possible, though still difficult, to predict future byte sequences from past ones).

See `man random` for details.

# Files names and inodes

## Hard Links Vs Soft Links



# Creating “links”

- Hard link

```
$ touch m  
$ ls -l m  
-rw-rw-r-- 1 abhijit abhijit 0 Jan  5 16:18 m  
$ ln m mm  
$ ls -l m mm  
-rw-rw-r-- 2 abhijit abhijit 0 Jan  5 16:18 m  
-rw-rw-r-- 2 abhijit abhijit 0 Jan  5 16:18 mm  
$ ln mm mmm  
$ ls -l m mm mmm  
-rw-rw-r-- 3 abhijit abhijit 0 Jan  5 16:18 m  
-rw-rw-r-- 3 abhijit abhijit 0 Jan  5 16:18 mm  
-rw-rw-r-- 3 abhijit abhijit 0 Jan  5 16:18 mmm  
$ echo Hi > m  
$ ls -l m mm mmm  
-rw-rw-r-- 3 abhijit abhijit 3 Jan  5 16:18 m  
-rw-rw-r-- 3 abhijit abhijit 3 Jan  5 16:18 mm  
-rw-rw-r-- 3 abhijit abhijit 3 Jan  5 16:18 mmm  
$ cat m  
Hi  
$ cat mm  
Hi
```

- Soft Link

```
$ ln -s m y  
$ ls -l m y  
-rw-rw-r-- 3 abhijit abhijit 3 Jan  5 16:18 m  
lrwxrwxrwx 1 abhijit abhijit 1 Jan  5 16:18 y -> m
```

# **System Calls, fork(), exec() Event Driven Kernel, Multi-tasking OS**

Abhijit A. M.  
[abhijit.comp@coep.ac.in](mailto:abhijit.comp@coep.ac.in)

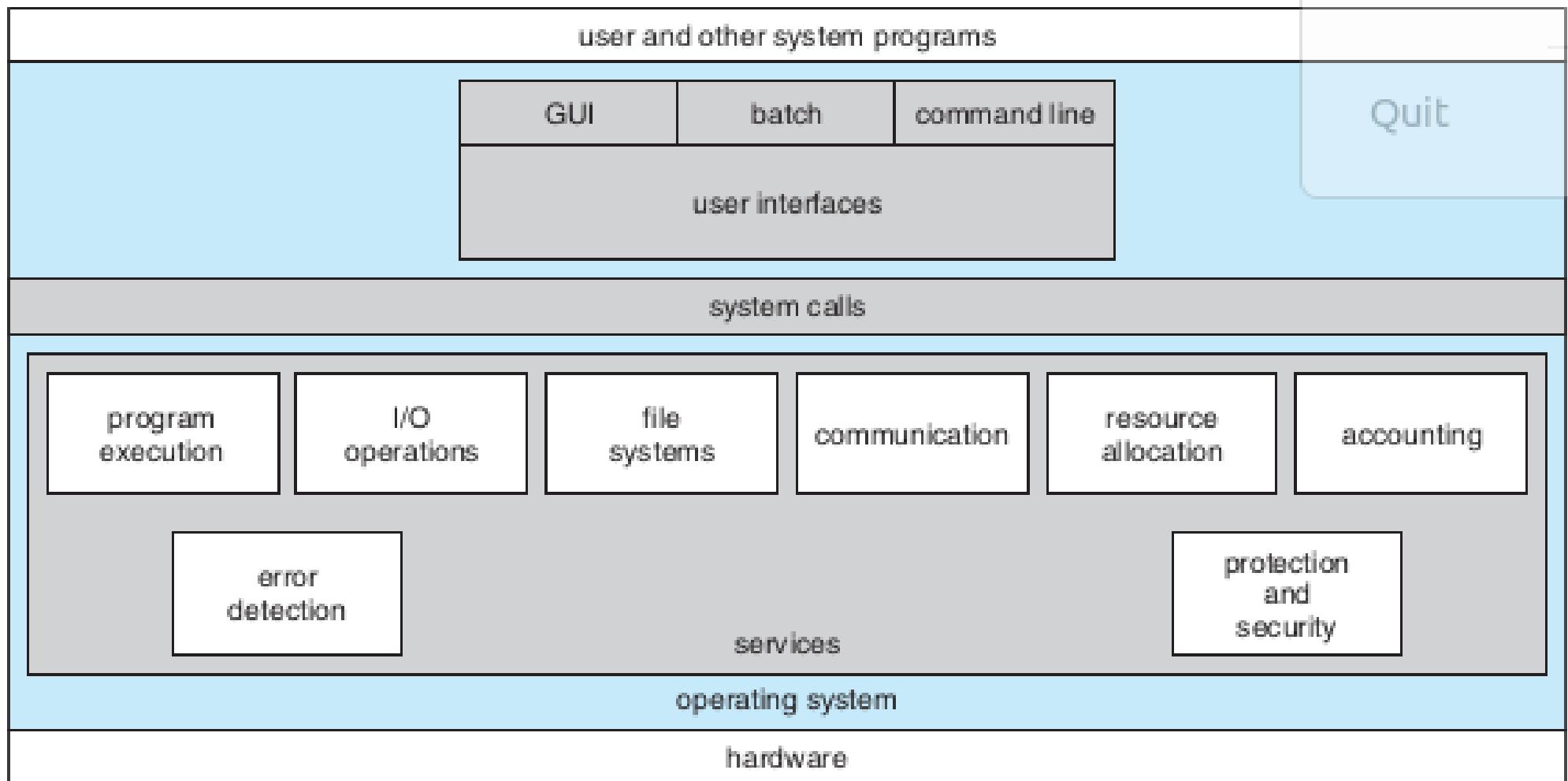
(C) Abhijit A.M.

Available under Creative Commons Attribution-ShareAlike License V3.0+

Credits: Slides of “OS Book” ed10.

# Building an “OS”

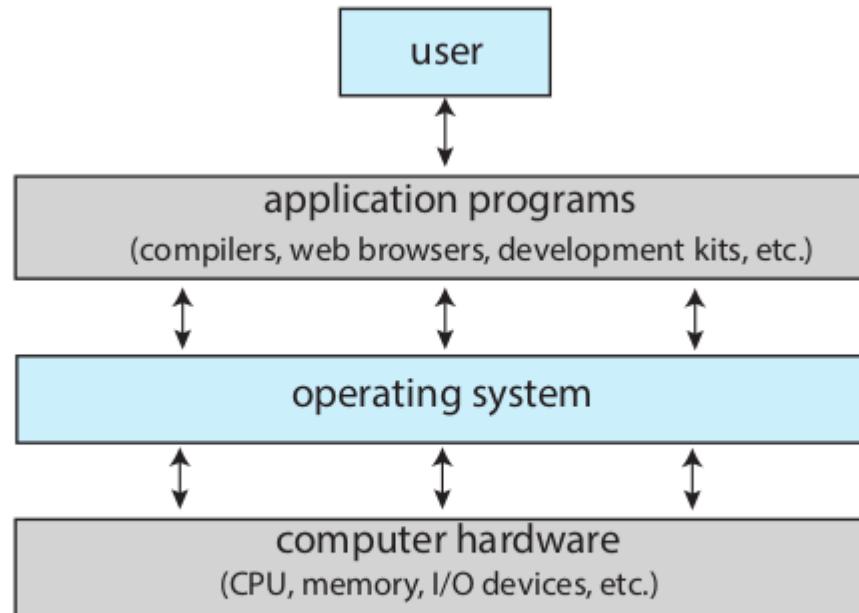
- E.g. Debian
  - A collection of thousands of applications, libraries, system programs, ... and Linux kernel !
  - Linux kernel is at the heart, but heart is not a human without the body !
  - Job of “Debian Developers”
    - Collect the source code of all things you want
    - Create an “Environment” for compiling things : bootstrap challenge
    - Compile everything
    - Ensure that things work “with each other”
      - Very difficult! Versions, dependencies !
    - “Package” things (e.g. make .deb files )



**Figure 2.1** A view of operating system services.

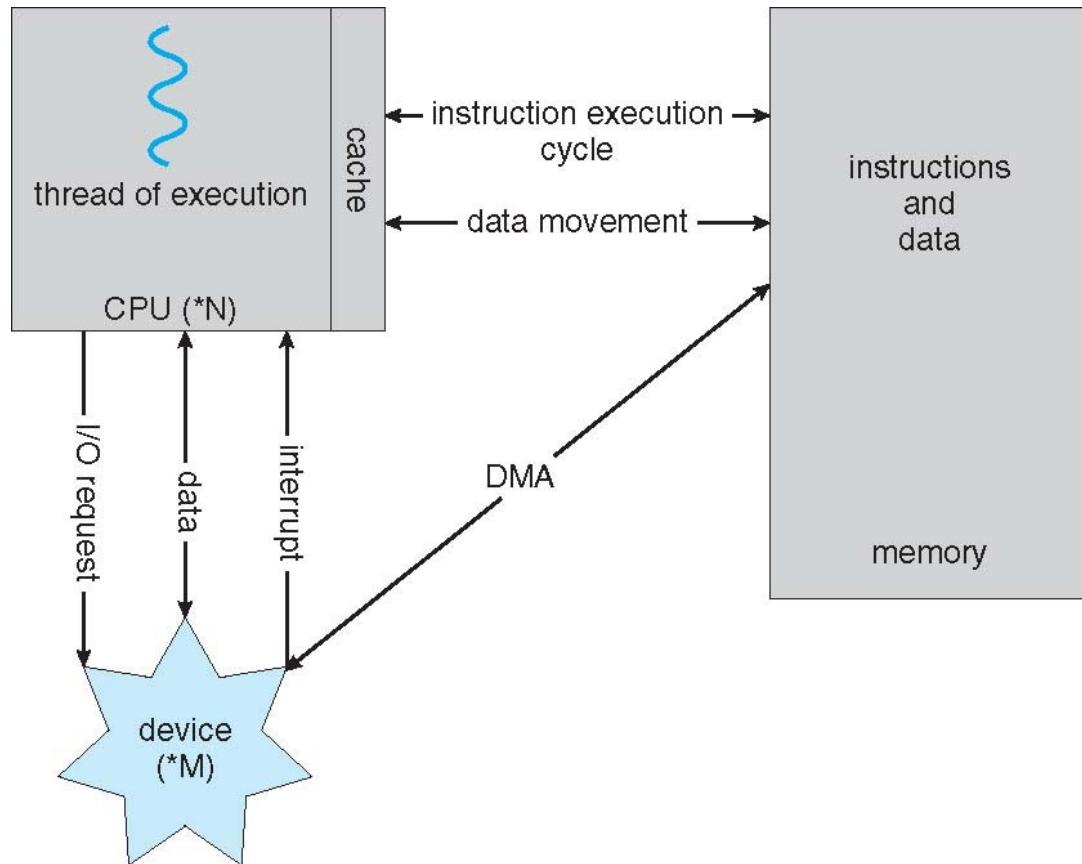
# Components of a computer system

## Abstract view



**Figure 1.1** Abstract view of the components of a computer system.

# Von Neumann architecture



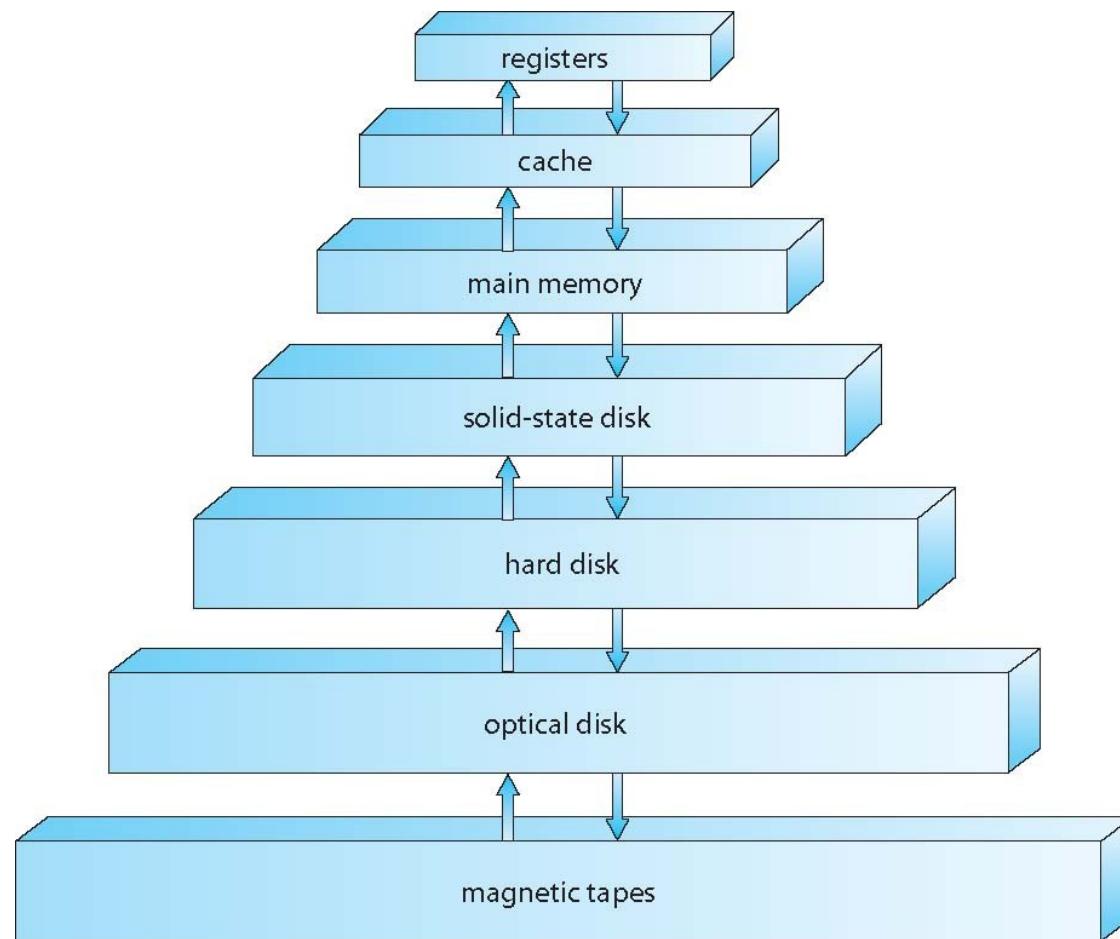
- Processor
  - Fetch
  - Decode
  - Execute
- Repeat

*A von Neumann architecture*

# A key to “understanding”

- Everything happens on processor
- Processor is always running some instruction
- We should be able to tell possible execution sequences on processor

# Memory hierarchy



# Memory hierarchy

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

# System Calls

- **Services provided by operating system to applications**
  - Essentially available to applications by calling the particular software interrupt application
    - All system calls essentially involve the “INT 0x80” on x86 processors + Linux
    - Different arguments specified in EAX register inform the kernel about different system calls
- **The C library has wrapper functions for each of the system calls**
  - E.g. open(), read(), write(), fork(), mmap(), etc.

# Types of System Calls

- **File System Related**
  - Open(), read(), write(), close(), etc.
- **Processes Related**
  - Fork(), exec(), ...
- **Memory management related**
  - Mmap(), shm\_open(), ...
- **Device Management**
- **Information maintainance – time,date**
- **Communication between processes (IPC)**
- **Read man syscalls**

[https://linuxhint.com/list\\_of\\_linux\\_syscalls/](https://linuxhint.com/list_of_linux_syscalls/)

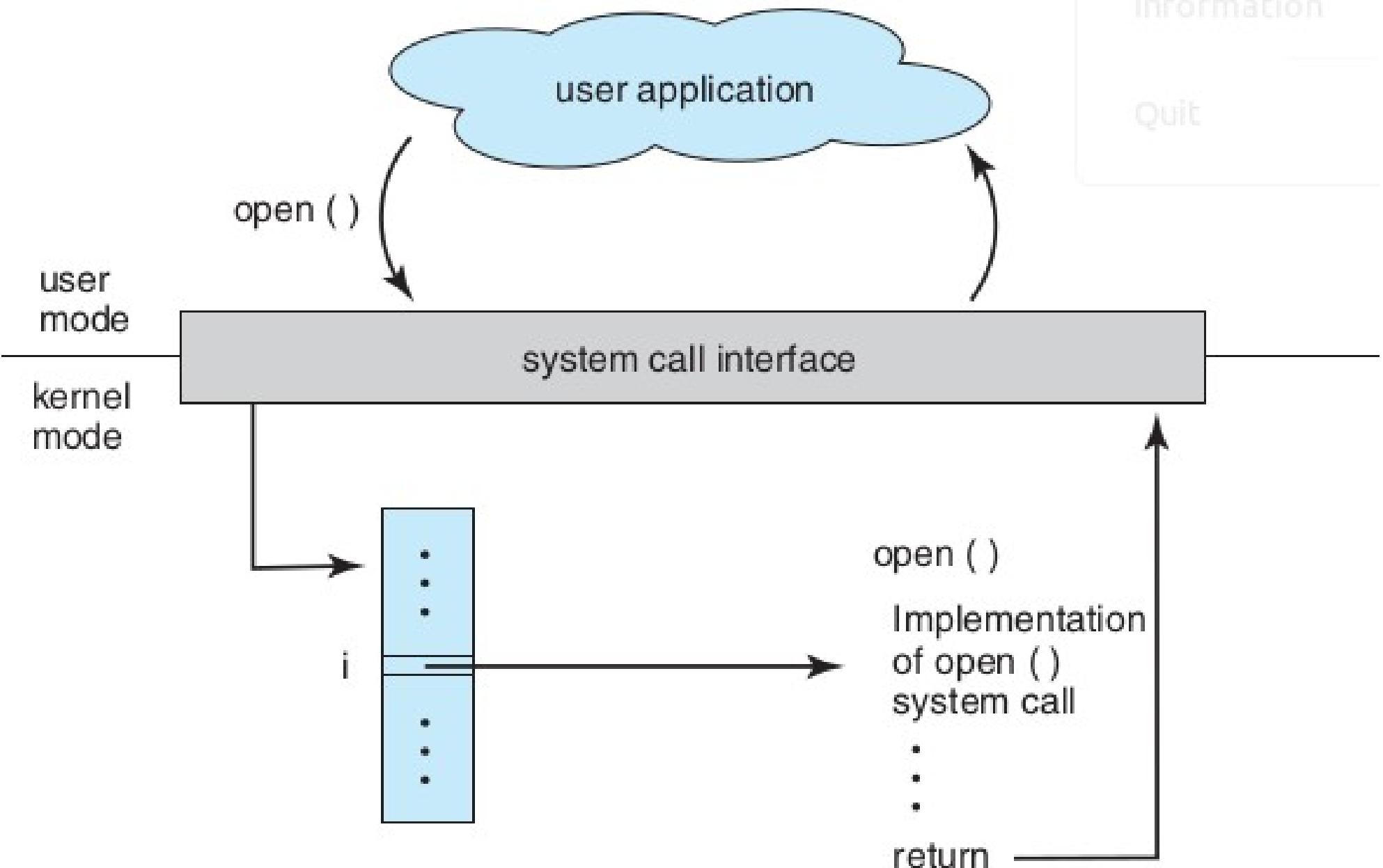
## EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
<b>Process Control</b>	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
<b>File Manipulation</b>	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
<b>Device Manipulation</b>	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
<b>Information Maintenance</b>	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
<b>Communication</b>	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
<b>Protection</b>	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

```
int main() {
    int a = 2;
    printf("hi\n");
}
-----
C Library
-----
int printf("void *a, ...) {
    ...
    write(1, a, ...);
}
int write(int fd, char *, int len) {
    int ret;
    ...
    mov $5, %eax,
    mov ... %ebx,
    mov ..., %ecx
    int $0x80
    __asm__("movl %eax, -4(%ebp)");
    # -4ebp is ret
    return ret;
}
```

# Code schematic

-----user-kernel-mode-boundary-----  
//OS code  
**int sys\_write(int fd, char \*, int len) {**  
 figure out location on disk  
 where to do the write and  
 carry out the operation,  
 etc.  
**}**



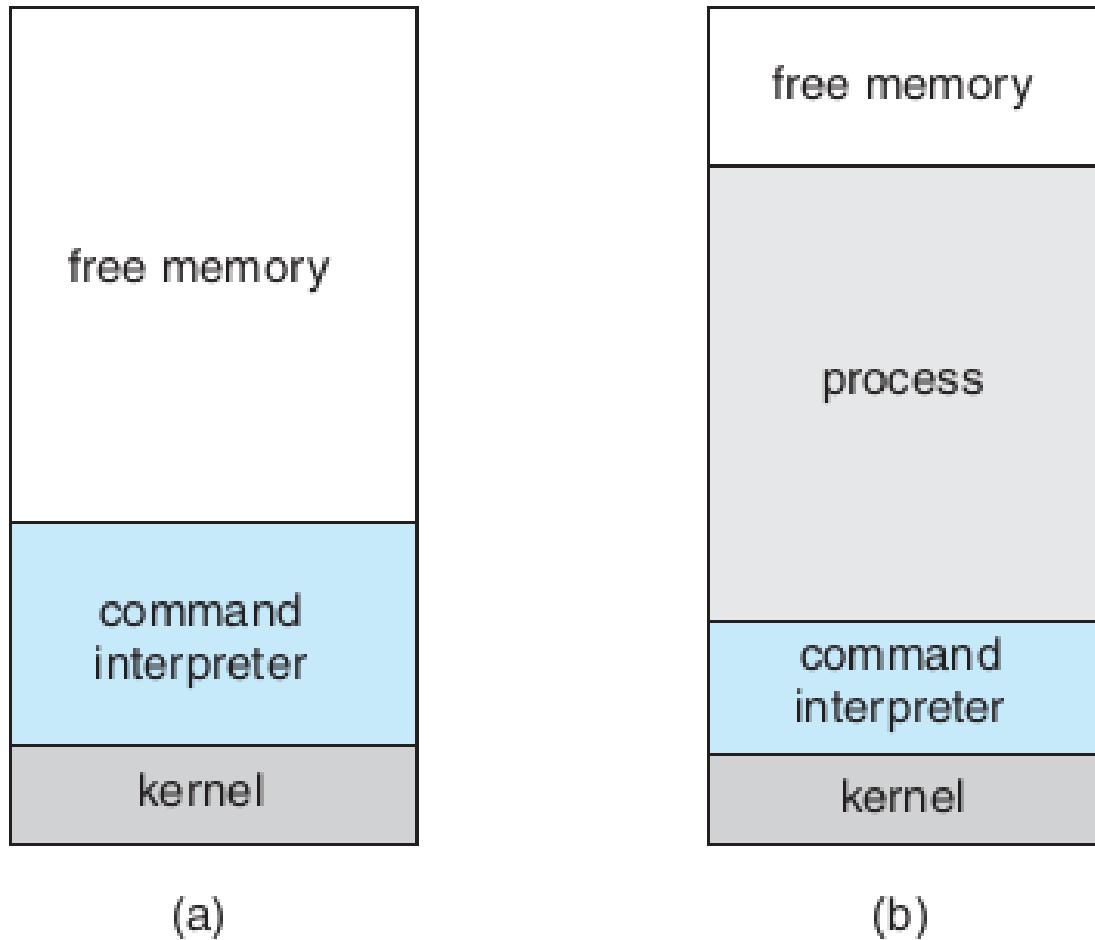
**Figure 2.6** The handling of a user application invoking the `open()` system call.

# Process

- **A program in execution**
- **Exists in RAM**
- **Scheduled by OS**
  - In a timesharing system, intermittantly scheduled by allocating a time quantum, e.g. 20 microseconds
- **The “ps” command on Linux**

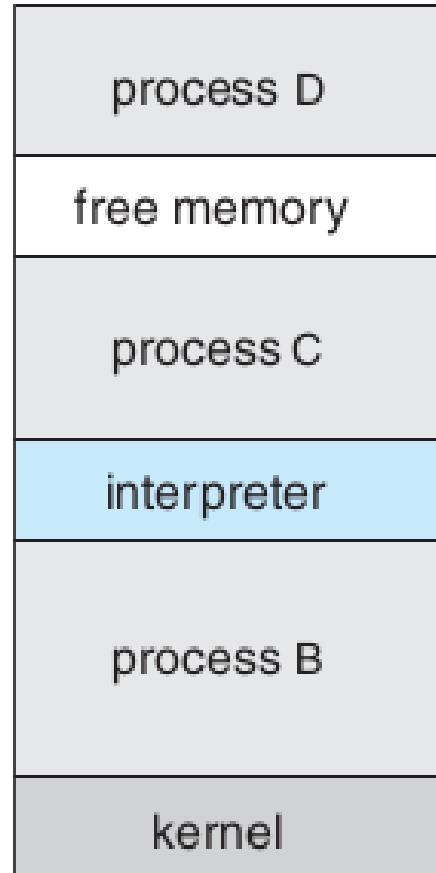
# Process in RAM

- **Memory is required to store the following components of a process**
  - **Code**
  - **Global variables (data)**
  - **Stack (stores local variables of functions)**
  - **Heap (stores malloced memory)**
  - **Shared libraries (e.g. code of printf, etc)**
  - **Few other things, may be**



**Figure 2.9** MS-DOS execution. (a) At system startup. (b) Running a program.

MS-DOS: a single tasking operating system  
Only one program in RAM at a time, and only one program can run at a time



A multi tasking system  
With multiple programs loaded in memory  
Along with kernel

(A very simplified conceptual diagram. Things are more complex in reality)

# **fork()**

- **A running process creates it's duplicate!**
- **After call to fork() is over**
  - Two processes are running
  - Identical
  - The calling function returns in two places!
  - Caller is called parent, and the new process is called child
  - PID is returned to parent and 0 to child

# **exec()**

- **Variants: execvp(), execl(), etc.**
- **Takes the path name of an executable as an argument**
- **Overwrites the existing process using the code provided in the executable**
- **The original process is OVER ! Vanished!**
- **The new program starts running overwritting the existing process!**

# Let's Understand fork() and exec()

```
#include <unistd.h>
int main() {
    fork();
    printf("hi\n");
    return 0;
}
```

```
#include <unistd.h>
int main() {
    printf("hi\n");
    execl("/bin/ls",
    "ls", NULL);
    printf("bye\n");
    return 0;
}
```

# A simple shell

```
#include <stdio.h>
#include <unistd.h>
int main() {
    char string[128];
    int pid;
    while(1) {
        printf("prompt>");
        scanf("%s", string);
        pid = fork();
        if(pid == 0) {
            execl(string, string, NULL);
        } else {
            wait(0);
        }
    }
}
```

# Shell using fork and exec

- Demo
- The only way a process can be created on Unix/Linux is using `fork() + exec()`
- All processes that you see were started by some other process using `fork() + exec()` , except the initial “*init*” process
- *When you click on “firefox” icon, the user-interface program does a `fork() + exec()` to start firefox; same with a command line shell program*
- The “bash” shell you have been using is nothing but an advanced version of the shell code shown during the demo
- See the process tree starting from “*init*”
- Your next assignment

# The boot process

- **BIOS/UEFI**

- The firmware. Runs “Automatically”.
- CPU is hardwired to start running instructions stored at a fixed address.
- The Motherboard manufacturers, ensure that the BIOS/UEFI is at this address

- **Boot loader**

- BIOS reads the code from sector 0 of the “Boot device”, this is called “Boot Loader”, and loads it in RAM
- PC changed to Boot Loader code
- GRUB is an example

- **kernel**

- The Boot Loader shows the choice of running an OS, user selects a choice
- The Boot loader loads the code of kernel from disk into RAM
- PC changed to kernel code
- **Kernel initializes hardware, its own data structures, etc etc**
- Init created by kernel by Hand
- **Kernel schedules init (the only process)**
- **Init fork-exec's some programs (user mode)**
  - Now these programs will be scheduled by OS
- **Init -> GUI -> terminal -> shell**
  - One of the typical parent-child relationships

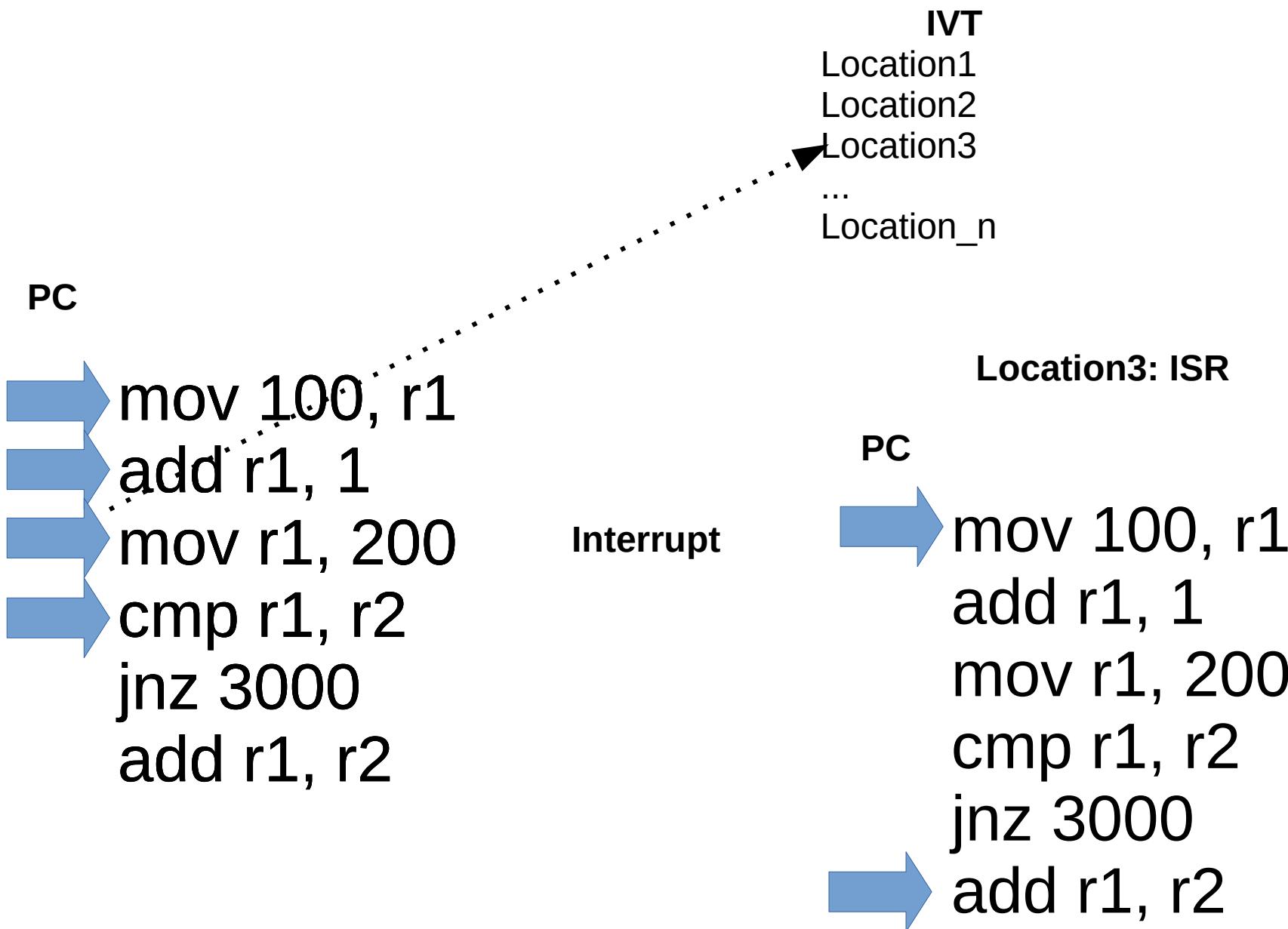
# **Event Driven kernel Multi-tasking, Multi-programming**

# **Understanding hardware interrupts**

- **Hardware devices (keyboard, mouse, hard disk, etc) can raise “hardware interrupts”**
- **Basically create an electrical signal on some connection to CPU (/bus)**
- **This is notified to CPU (in hardware)**
- **Now CPU’s normal execution is interrupted!**
  - What’s the normal execution?
  - CPU will not continue doing the fetch, decode, execute, change PC cycle !
  - What happens then?

# Understanding hardware interrupts

- **On Hardware interrupt**
  - The PC changes to a location pre-determined by CPU manufacturers!
  - Now CPU resumes normal execution
    - What's normal?
    - Same thing: Fetch, Decode, Execute, Change PC, repeat!
    - But...
  - But what's there at this pre-determined address?
  - OS! How's that ?(in a few minutes)



# Hardware interrupts and OS

- **When OS starts running initially**
  - It copies relevant parts of it's own code at all possible memory addresses that a hardware interrupt can lead to!
  - If there is an IVT in the hardware, OS sets up the IVT also!
  - Intelligent, isn't' it?
- **Now what?**
  - Whenever there is a hardware interrupt – what will happen?
  - The PC will change to predetermined location, and control will jump into OS code
- **So remember: whenever there is a hardware interrupt, OS code will run!**
- **This is “taking control of hardware”**
-

# Key points

- **Understand the interplay of hardware features + OS code + clever combination of the two to achieve OS control over hardware**
- **Most features of computer systems / operating systems are derived from hardware features**
  - We will keep learning this throughout the course
  - Hardware support is needed for many OS features

# Time Shared CPU

- Timesharing happens after the OS has been loaded and Desktop environment is running
- The OS and different application programs keep executing on the CPU alternatively (more about this later)
  - The CPU is time-shared between different applications and OS itself
- How is this done?
  - kernel sets up the “timer register”
  - kernel changes PC to address of a process (now process runs)
    - Timer register decremented automatically in hardware
  - When timer is “up”, a hardware interrupt occurs!
  - Now OS runs again! This OS code is called “scheduler”

# Multiprogramming

- **Program**
  - Just a binary (machine code) file lying on the **hard drive**. E.g. `/bin/ls`
  - Does not do anything!
- **Process**
  - A program that is executing
  - Must **exist in RAM** before it executes. **Exec()** does this.
  - One program can run as multiple processes. What does that mean?

# Multiprogramming

- **Multiprogramming**
  - A system where multiple processes(!) exist at the same time in the RAM
  - But only one runs at a time!
    - Because there is only one CPU
- **Multi tasking**
  - Time sharing between multiple processes in a multi-programming system
  - Timer interrupt used to achieve this.

# Question

- **Select the correct one**
  - 1) A multiprogramming system is not necessarily multitasking
  - 2) A multitasking system is not necessarily multiprogramming

# Events , that interrupt CPU's functioning

- Three types of “traps” : Events that make the CPU run code at a pre-defined address (given by IVT)
  - 1) Hardware interrupts
  - 2) Software interrupt instructions (trap)  
E.g. instruction “INT”
  - 3) Exceptions
    - e.g. a machine instruction that does division by zero
    - Illegal instruction, etc.
    - Some are called “faults”, e.g. “page fault”, recoverable
    - some are called “aborts”, e.g. division by zero, non-recoverable
- The kernel code occupies all memory locations corresponding to the PC values related to all the above events, at the time of boot!  
It also sets up the IVT if there is one.

# Multi tasking requirements

- **Two processes should not be**
  - Able to steal time of each other
  - See data/code of each other
  - Modify data/code of each other
  - Etc.
- **The OS ensures all these things. How?**
  - To be seen later.

**But the OS is “always” “running”  
“in the background”  
Isn’t it?**

**Absolutely No!**

**Let's understand  
What kind of  
Hardware, OS interplay  
makes  
Multitasking possible**

# Two types of CPU instructions and two modes of CPU operation

- CPU instructions can be divided into two types
- Normal instructions
  - mov, jmp, add, etc.
- Privileged instructions
  - Normally related to hardware devices
  - E.g.
    - IN, OUT # write to I/O memory locations
    - INTR # software interrupt, etc.

# Two types of CPU instructions and two modes of CPU operation

- CPUs have a mode bit (can be 0 or 1)
- The two values of the mode bit are called: User mode and Kernel mode
- If the bit is in user mode
  - Only the normal instructions can be executed by CPU
- If the bit is in kernel mode
  - Both the normal and privileged instructions can be executed by CPU
- If the CPU is “made” to execute privileged instruction when the mode bit is in “User mode”
  - It results in a illegal instruction execution
  - Again jumping to kernel code, using IVT!

# **Two types of CPU instructions and two modes of CPU operation**

- **The operating system code runs in kernel mode.**
  - How? Wait for that!
- **The application code runs in user mode**
  - How? Wait !
  - So application code can not run privileged hardware instructions
- **Transition from user mode to kernel mode and vice-versa**
  - Special instruction called “software interrupt” instructions
  - E.g. INT instruction on x86

# Software interrupt instruction

- E.g. INT on x86 processors
- Does two things at the same time!
  - Changes mode from user mode to kernel mode in CPU
    - + Jumps to a pre-defined location! Basically changes PC to a pre-defined value.
      - Close to the way a hardware interrupt works. Isn't it?
  - Why two things together?
  - What's there are the pre-defined location?
    - Obviously, OS code. OS occupied these locations in Memory, at Boot time.

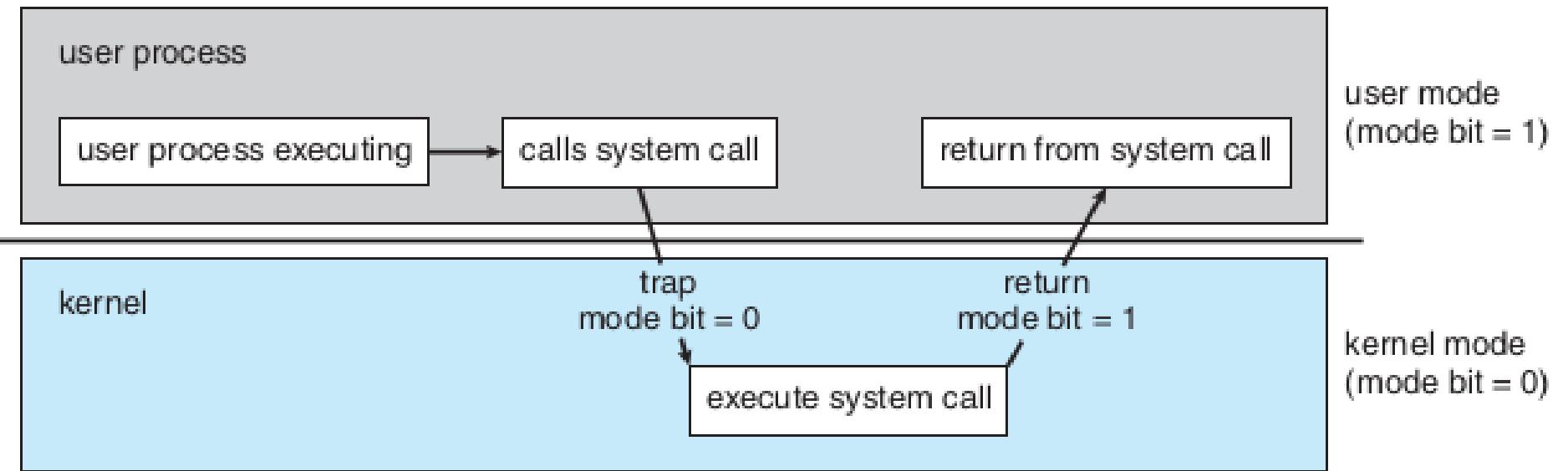
# Software interrupt instruction

- **What's the use of these type of instructions?**
  - An application code running INT 0x80 on x86 will now cause
    - Change of mode
    - Jump into OS code
  - Effectively a request by application code to OS to do a particular task!
  - E.g. read from keyboard or write to screen !
  - OS providing hardware services to applications !

```
int main() {
    int a = 2;
    printf("hi\n");
}
-----
C Library
-----
int printf("void *a, ...) {
    ...
    write(1, a, ...);
}
int write(int fd, char *, int len) {
    int ret;
    ...
    mov $5, %eax,
    mov ... %ebx,
    mov ..., %ecx
    int $0x80
    __asm__("movl %eax, -4(%ebp)");
    # -4ebp is ret
    return ret;
}
```

# Code schematic

-----user-kernel-mode-boundary-----  
//OS code  
**int sys\_write(int fd, char \*, int len) {**  
 figure out location on disk  
 where to do the write and  
 carry out the operation,  
 etc.  
**}**



**Figure 1.10** Transition from user to kernel mode.

# Software interrupt instruction

- **How does application code run INT instruction?**
  - C library functions like `printf()`, `scanf()` which do I/O requests contain the INT instruction!
  - Control flow
    - Application code -> `printf` -> INT -> OS code -> back to `printf` code -> back to application code

# Example: C program

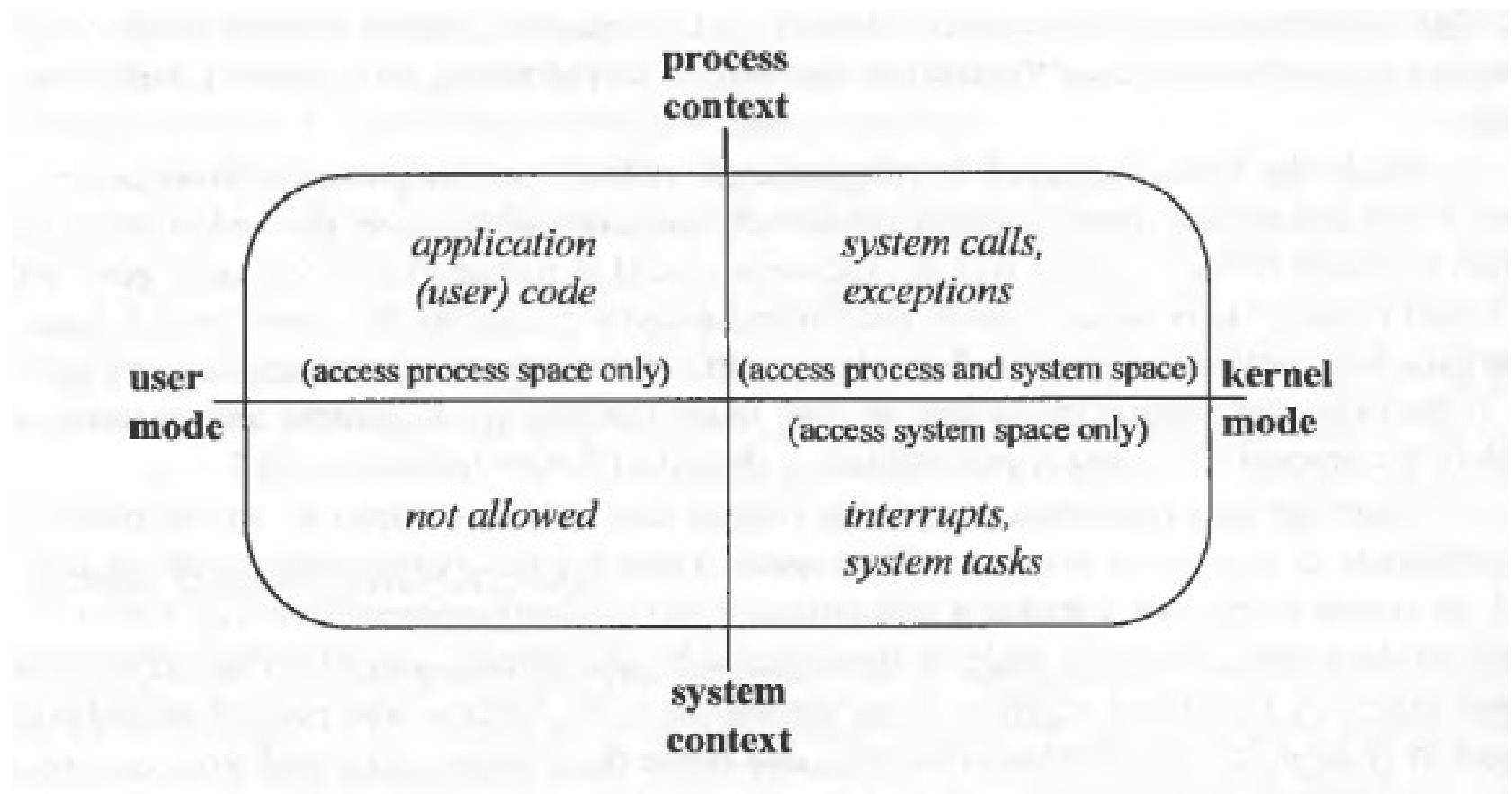
```
int main() {  
    int i, j, k;  
    k = 20;  
    scanf("%d", &i); // This jumps into OS and returns back  
    j = k + i;  
    printf("%d\n", j); // This jumps into OS and returns back  
    return 0;  
}
```

# **Interrupt driven OS code**

- **OS code is sitting in memory , and runs intermittantly . When?**
  - On a software or hardware interrupt or exception!
  - Event/Interrupt driven OS!
  - Hardware interrupts and exceptions occur asynchronously (un-predictedly) while software interrupts are caused by application code

# What runs on the processor ?

- 4 possibilities.



# Some system calls related to files

Abhijit A M

abhijit.comp@coep.ac.in

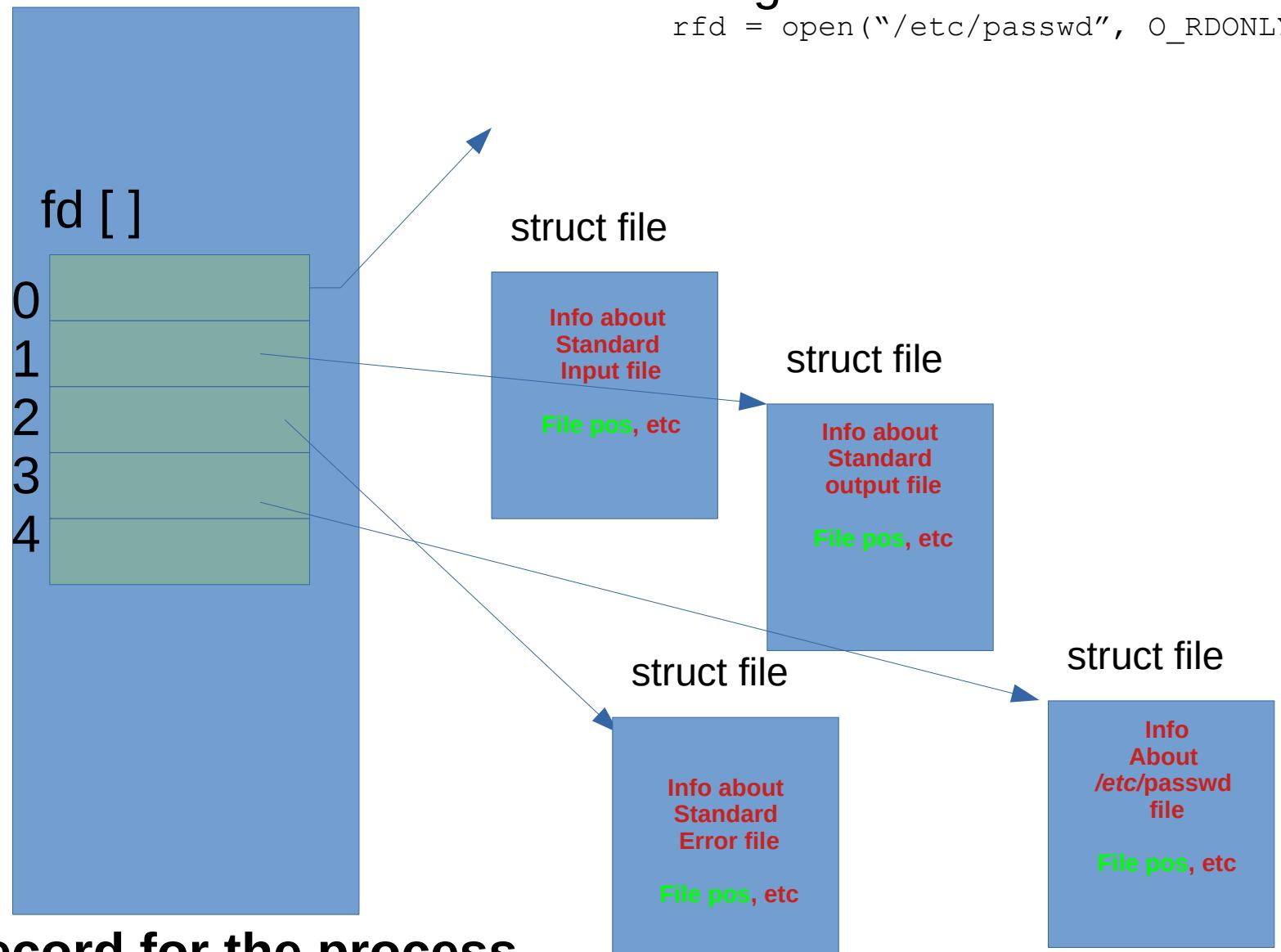
# System Calls

- Kernel provided functions
- Run in Kernel mode
- Essentially invoked through the Software Interrupt instruction (“INT”)
  - INT -> Lookup in IVT -> jump to kernel code

# List of open files

Program did

```
rfd = open("/etc/passwd", O_RDONLY)
```



In kernel, record for the process

# File system related system calls

- To get permission to “access” the file
  - open()
- To read sequentially from a file that has been open()
  - read()
- To write sequentially to a file that has been open()
  - write()
- To change “file position” anywhere
  - lseek()
- To release access to the file
  - close()
- More: dup(), dup3(), fcntl(), flock(), lockf(), ...

## Example: cat program

```
int main(int argc, char *argv[]) {
    int fd;
    char ch;
    fd = open(argv[1], O_RDONLY);
    if(fd == -1) {
        perror("mycat: ");
        exit(errno);
    }
    while(read(fd, &ch, 1))
        putchar(ch);
    return 0;
}
```

# Example: cp program

```
int main(int argc, char *argv[]) {
    int fd, fdw;
    char ch;
    fd = open(argv[1], O_RDONLY);
    if(fd == -1) {
        perror("open failed:");
        return errno;
    }
    fdw = open(argv[2], O_WRONLY | O_CREAT, S_IRUSR);
    if(fdw == -1) {
        perror("open failed:");
        return errno;
    }
    while(read(fd, &ch, 1))
        write(fdw, &ch, 1);
    return 0;
}
```

# Standard file descriptors

- `stdin` (0), `stdout`(1), `stderr`(2)
- Already open when a process begins
- Can be closed!
- `stdin`
  - Read from keyboard
- `stdout`, `stderr`
  - Write to screen, but two different “streams”

# Standard file descriptors

- Stdin(0)  
`ch= getchar();`  
*is equivalent to*  
`read(0, &ch, 1);`
- Stdout(1)  
`printf("hello")`  
*is equivalent to*  
`write(1, "hello", 5);`
- Stderr(2)  
`fprintf(stderr,  
'hello')`  
*is equivalent to*  
`write(2, "hello", 5);`

# Redirection

- Output redirection

```
close(1);
```

```
fd = open(..., O_WRONLY);
```

- Input redirection

```
close(0);
```

```
fd = open(..., O_RDONLY);
```

# **dup()**

- Duplicates a file descriptor
  - Essentially the “struct file \*” in the kernel fdarray is copied !
- Example

```
fd = open(..., O_RDONLY);  
close(0);  
dup(fd);
```

# Processes

Abhijit A M

abhijit.comp@coep.ac.in

# Process related data structures in kernel code

- Kernel needs to maintain following types of data structures for managing processes
  - List of all processes
  - Memory management details for each, files opened by each etc.
  - Scheduling information about the process
  - Status of the process
  - List of processes “waiting” for different events to occur,
  - Etc.

# Process Control Block

- A record representing a process in operating system's data structures
- OS maintains a “list” of PCBs, one for each process
- Called “`struct task_struct`” in Linux kernel code and “`struct proc`” in xv6 code

## Fields in PCB



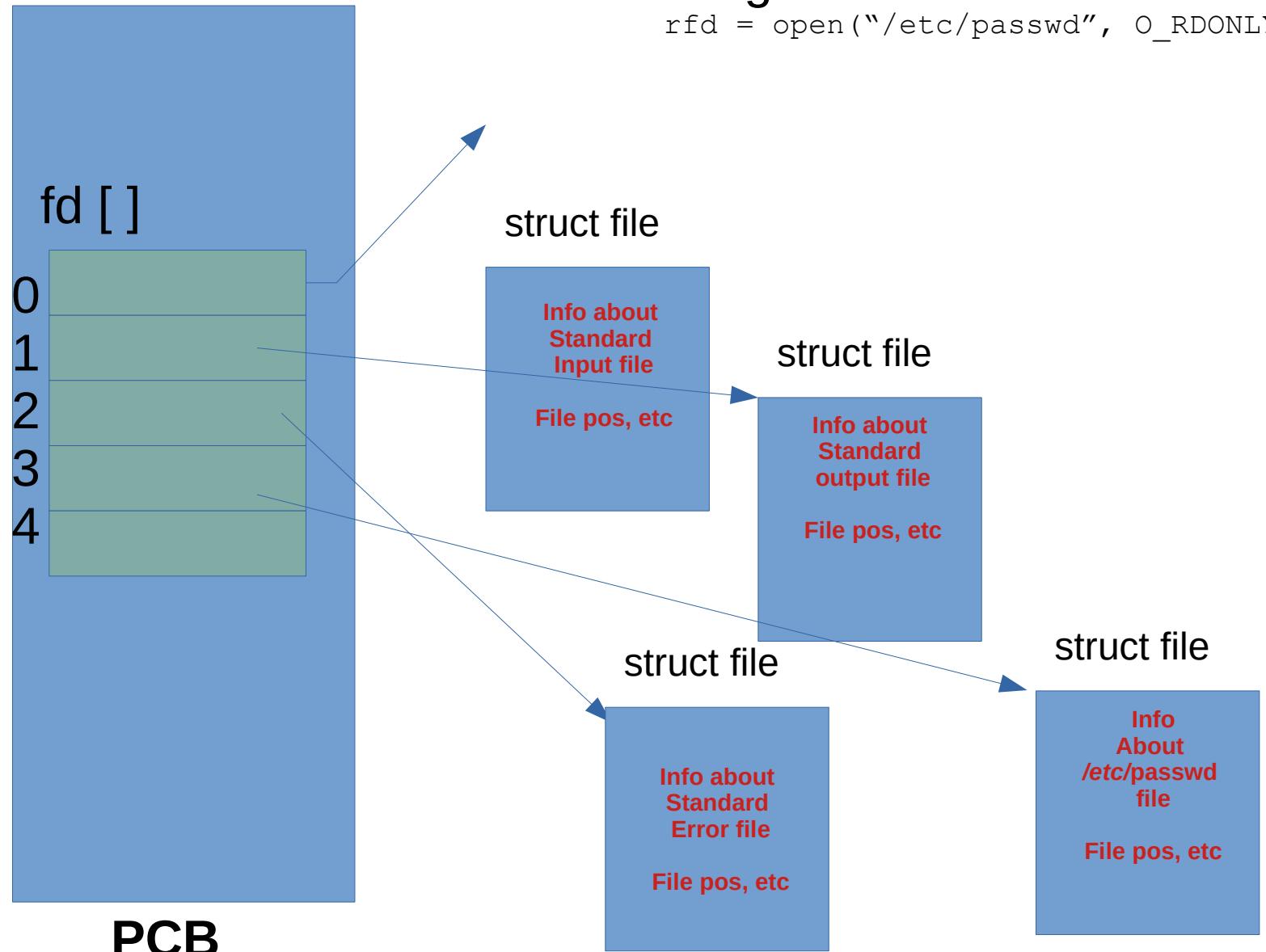
Figure 3.3 Process control block (PCB).

- Process ID (PID)
- Process State
- Program counter
- Registers (copy)
- Memory limits of the process
- Accounting information
- I/O status
- Scheduling information
- array of file descriptors (list of open files)
- ...etc

# List of open files

Program did

```
rfd = open("/etc/passwd", O_RDONLY)
```



# List of open files

- The PCB contains an array of pointers, called file descriptor array (fd[ ]), pointers to structures representing files
- When open() system call is made
  - A new file structure is created and relevant information is stored in it
  - Smallest available of fd [ ] pointers is made to point to this new struct file
  - The index of this fd [ ] pointer is returned by open
- When subsequent calls are made to read(fd, ....) or write(fd, ...), etc.
  - The kernel gets the “fd” as an index in the fd[ ] array and is able to locate the file structure for that file

```
// XV6 Code : Per-process state
enum procstate { UNUSED, EMBRYO, SLEEPING,
RUNNABLE, RUNNING, ZOMBIE };

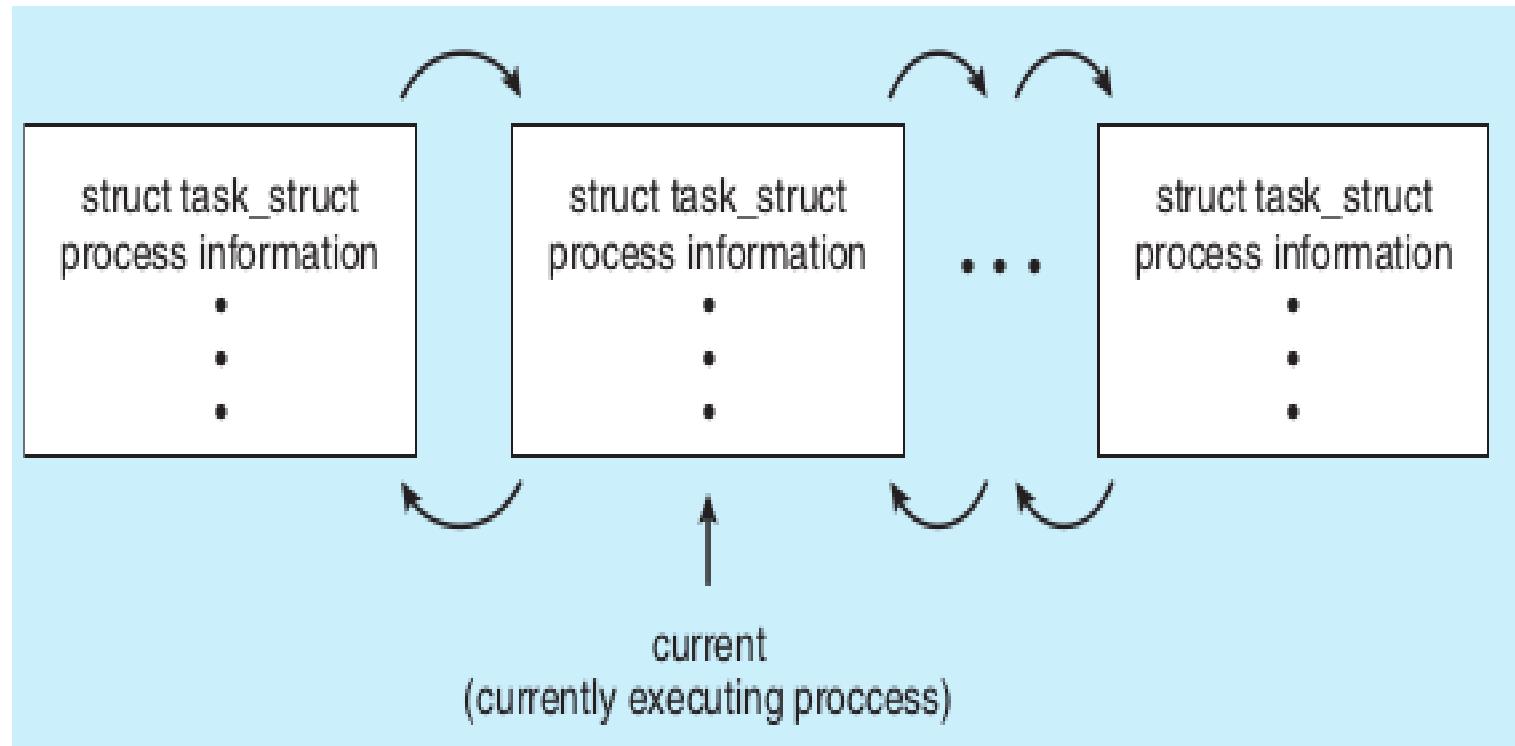
struct proc {
    uint sz;                      // Size of process memory (bytes)
    pde_t* pgdir;                 // Page table
    char *kstack;                 // Bottom of kernel stack for this
process
    enum procstate state;         // Process state
    int pid;                      // Process ID
    struct proc *parent;          // Parent process
    struct trapframe *tf;         // Trap frame for current syscall
    struct context *context;      // swtch() here to run process
    void *chan;                   // If non-zero, sleeping on chan
    int killed;                   // If non-zero, have been killed
    struct file *ofile[NFILE];    // Open files
    struct inode *cwd;            // Current directory
    char name[16];                // Process name (debugging)
};

struct {
    struct spinlock lock;
    struct proc proc[NPROC];
} ptable;
```

```
struct file {
    enum { FD_NONE,
FD_PIPE, FD_INODE } type;
    int ref; // reference count
    char readable;
    char writable;
    struct pipe *pipe;
    struct inode *ip;
    uint off;
};
```

# Process Queues/Lists inside OS

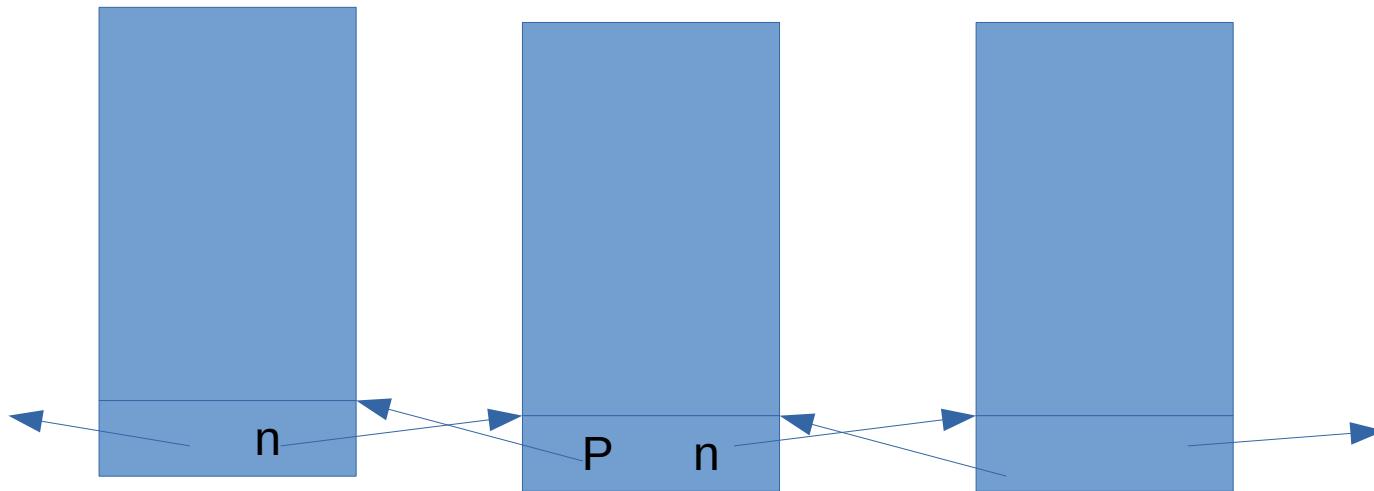
- Different types of queues/lists can be maintained by OS for the processes
  - A queue of processes which need to be scheduled
  - A queue of processes which have requested input/output to a device and hence need to be put on hold/wait
  - List of processes currently running on multiple CPUs
  - Etc.



## // Linux data structure

```
struct task_struct {  
    long state; /* state of the process */  
    struct sched_entity se; /* scheduling information */  
    struct task_struct *parent; /* this process's parent */  
    struct list_head children; /* this process's children */  
    struct files_struct *files; /* list of open files */  
    struct mm_struct *mm; /* address space */
```

```
struct list_head {  
    struct list_head  
    *next, *prev;  
};
```



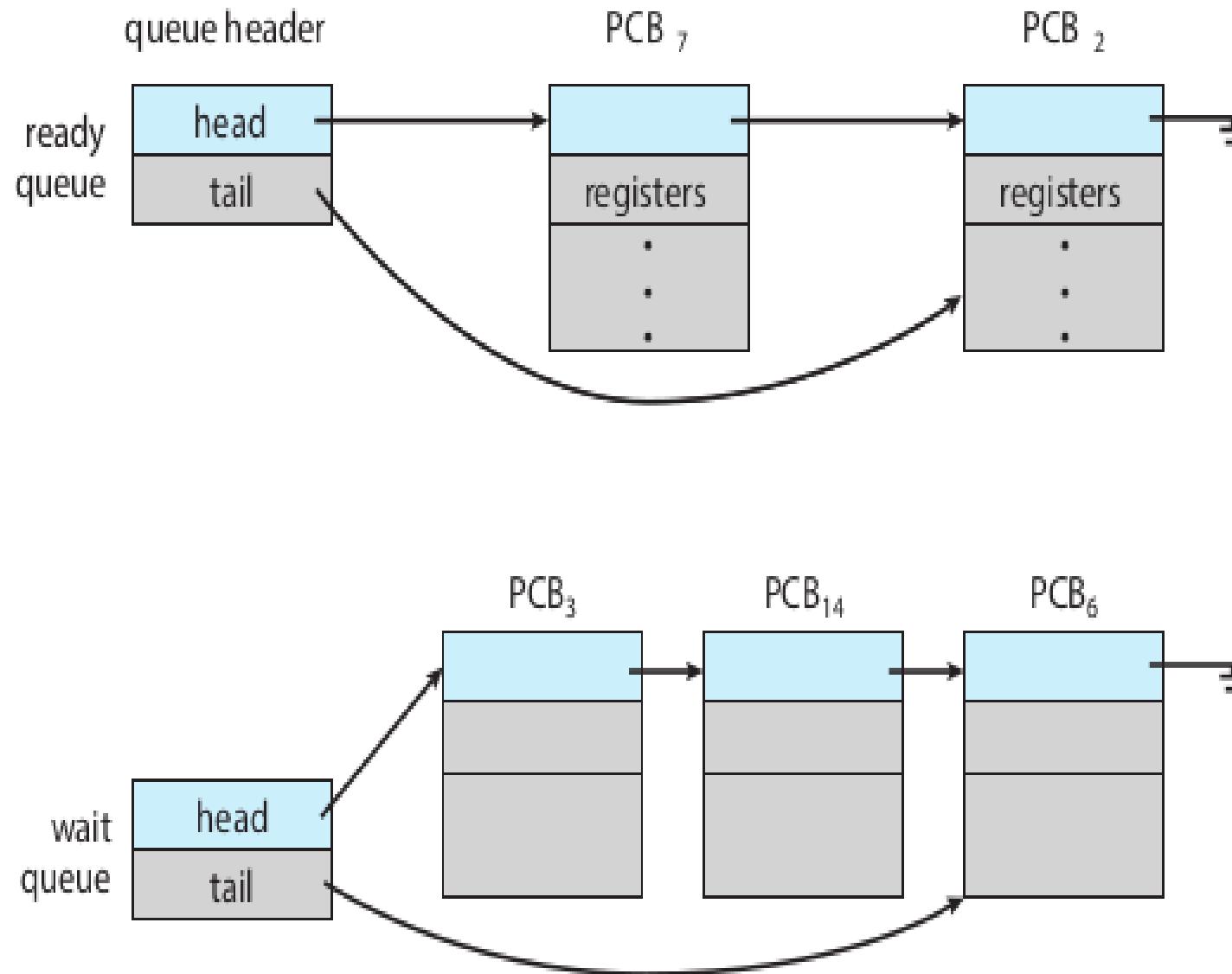
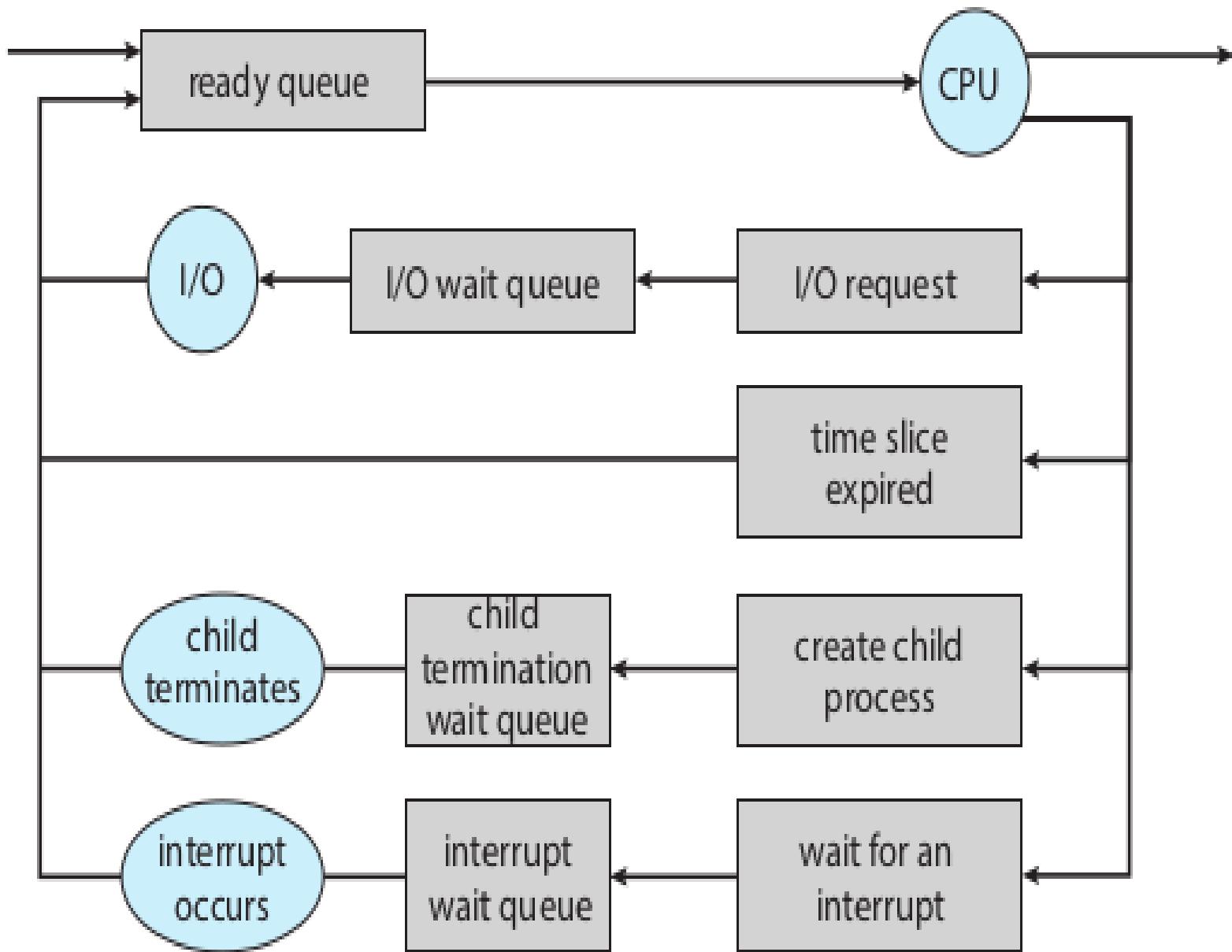
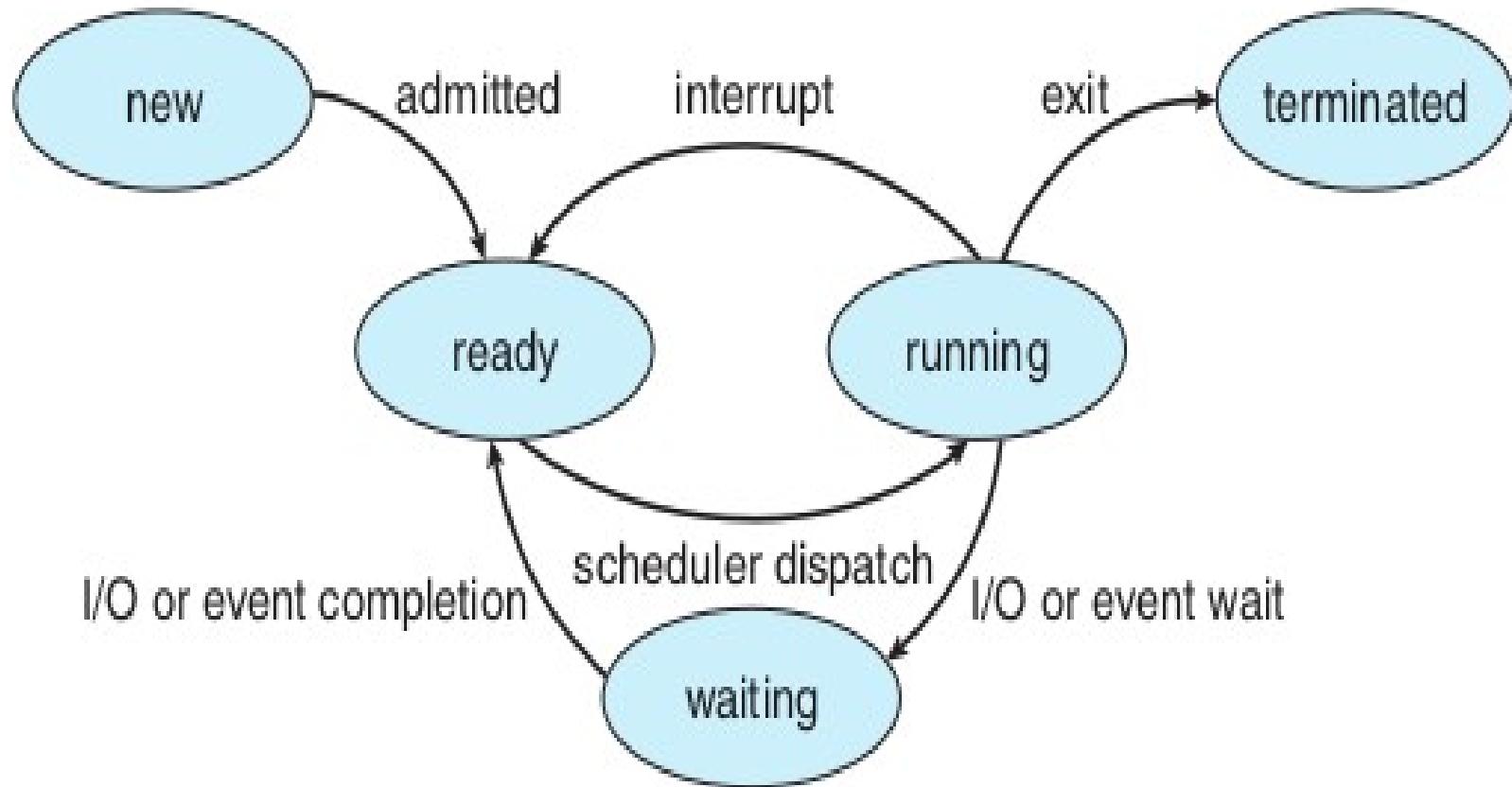


Figure 3.4 The ready queue and wait queues.



**Figure 3.5** Queueing-diagram representation of process scheduling.



**Figure 3.2** Diagram of process state.

**Conceptual diagram**

See state in output of  
ps axu  
(BSD style options,  
without -)

On Linux

# “Giving up” CPU by a process or blocking

```
int main() {  
    i = j + k;  
    scanf("%d", &k);  
}  
  
int scanf(char *x, ...) {  
    ...  
    read(0, ..., ...);  
}  
  
int read(int fd, char *buf, int len) {  
    ...  
    __asm__ { "int 0x80..." }  
    ...  
}
```

## OS Syscall

```
sys_read(int fd, char *buf, int len) {  
    file f = current->fdarray[fd];  
    int offset = f->position;  
    ...  
    disk_read(..., offset, ...);  
    // Do what now?  
    //asynchronous read  
    //Interrupt will occur when the disk read is complete  
    // Move the process from ready queue to a wait queue and call scheduler!  
    // This is called "blocking"  
    Return the data read ;  
}  
disk_read(..., offset, ... ) {  
    __asm__("outb PORT ..");  
    return;  
}
```

# **“Giving up” CPU by a process or blocking**

The relevant code in xv6 is in

Sleep()

The wakeup code is in wakeup() and wakeup1()

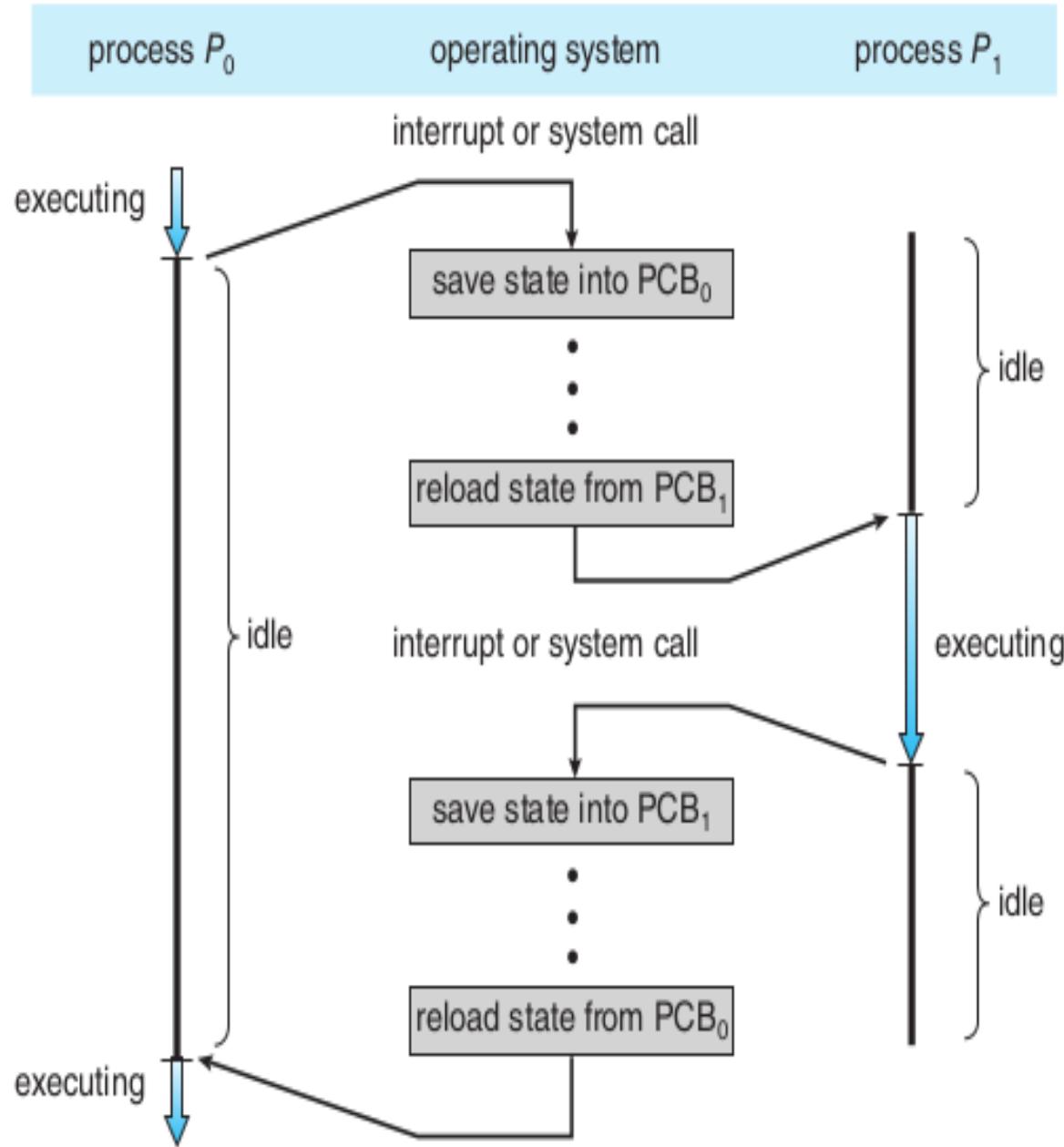
To be seen later

# Context Switch

- Context
  - Execution context of a process
  - CPU registers, process state, memory management information, all configurations of the CPU that are specific to execution of a process/kernel
- Context Switch
  - Change the context from one process/OS to OS/another process
  - Need to save the old context and load new context
  - Where to save? --> PCB of the process

# Context Switch

- Is an overhead
- No useful work happening while doing a context switch
- Time can vary from hardware to hardware
- Special instructions may be available to save a set of registers in one go



**Figure 3.6** Diagram showing context switch from process to process.

# Peculiarity of context switch

- When a process is running, the function calls work in LIFO fashion
  - Made possible due to calling convention (a protocol for using processor stack for passing parameters and returning values, used by compilers to generate machine code)
- When an interrupt occurs
  - It can occur anytime
  - Context switch can happen in the middle of execution of any function
- After context switch
  - One process takes place of another
  - This “switch” is obviously not going to happen using calling convention, as no “call” is happening
  - Code for context switch must be in assembly!

# Compilation, Linking, Loading

Abhijit A M

# Review of last few lectures

Boot sequence: BIOS, boot-loader, kernel

- Boot sequence: Process world
  - kernel->init -> many forks+execs() -> ....
- Hardware interrupts, system calls, exceptions
- Event driven kernel
- System calls
  - Fork, exec, ... open, read, ...

# What are compiler, assembler, linker and loader, and C library

## System Programs/Utilities

Most essential to make a kernel really  
usable

# Standard C Library

- A collection of some of the most frequently needed functions for C programs
  - `scanf`, `printf`, `getchar`, system-call wrappers (`open`, `read`, `fork`, `exec`, etc.), ...
- An machine/object code file containing the machine code of all these functions
  - Not a source code! Neither a header file. More later.
- Where is the C library on your computer?
  - `/usr/lib/x86_64-linux-gnu/libc-2.31.so`

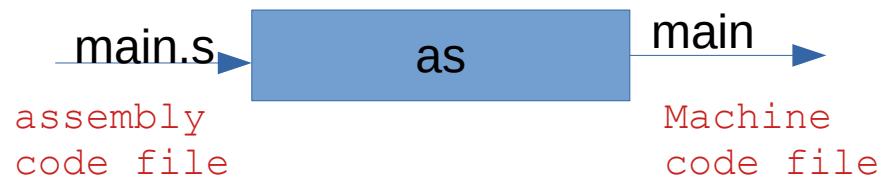
# Compiler

- application program, which converts one (programming) language to another
  - Most typically compilers convert a high level language like C, C++, etc. to Machine code language
- E.g. GCC /usr/bin/gcc
  - Usage: e.g.
  - \$ gcc main.c -o main
  - Here main.c is the C code, and "main" is the object/machine code file generated
- Input is a file and output is also a file.
- Other examples: g++ (for C++), javac (for java)

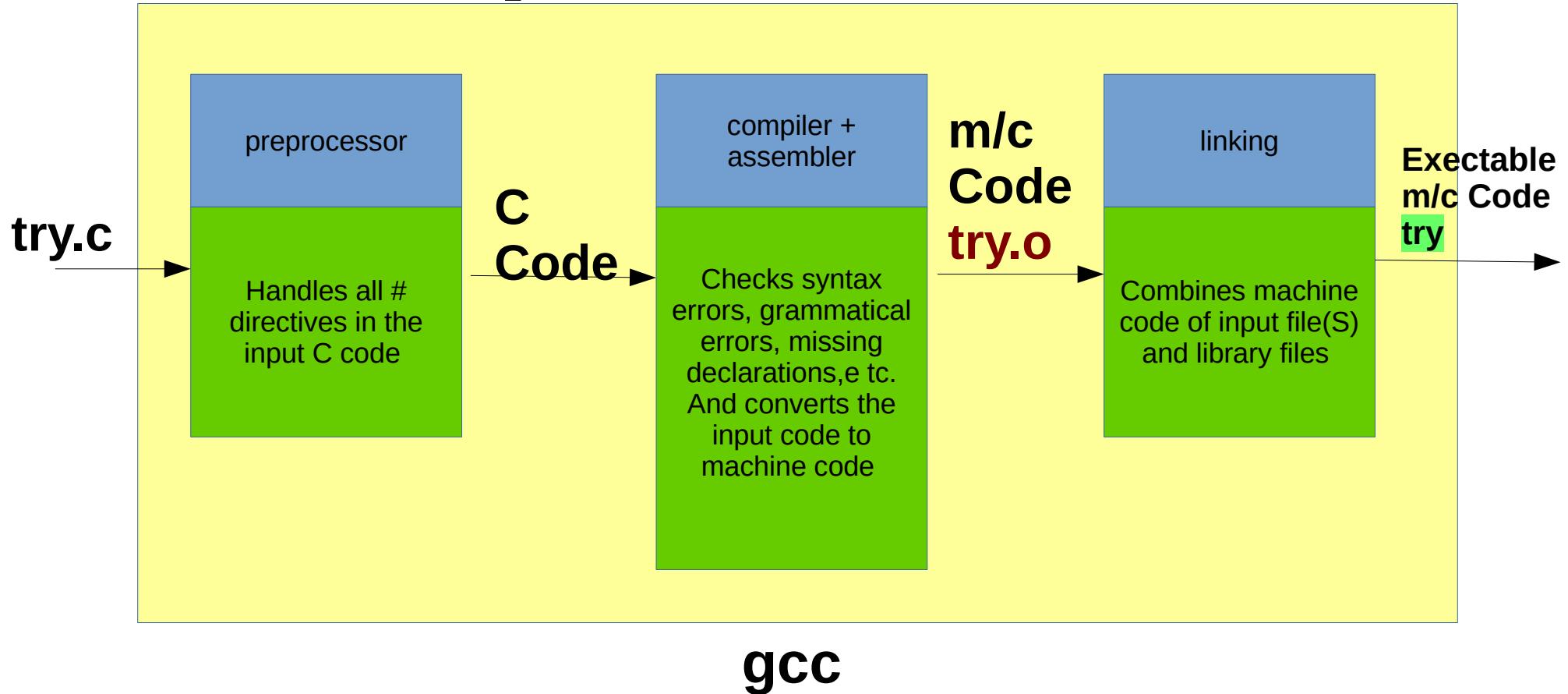


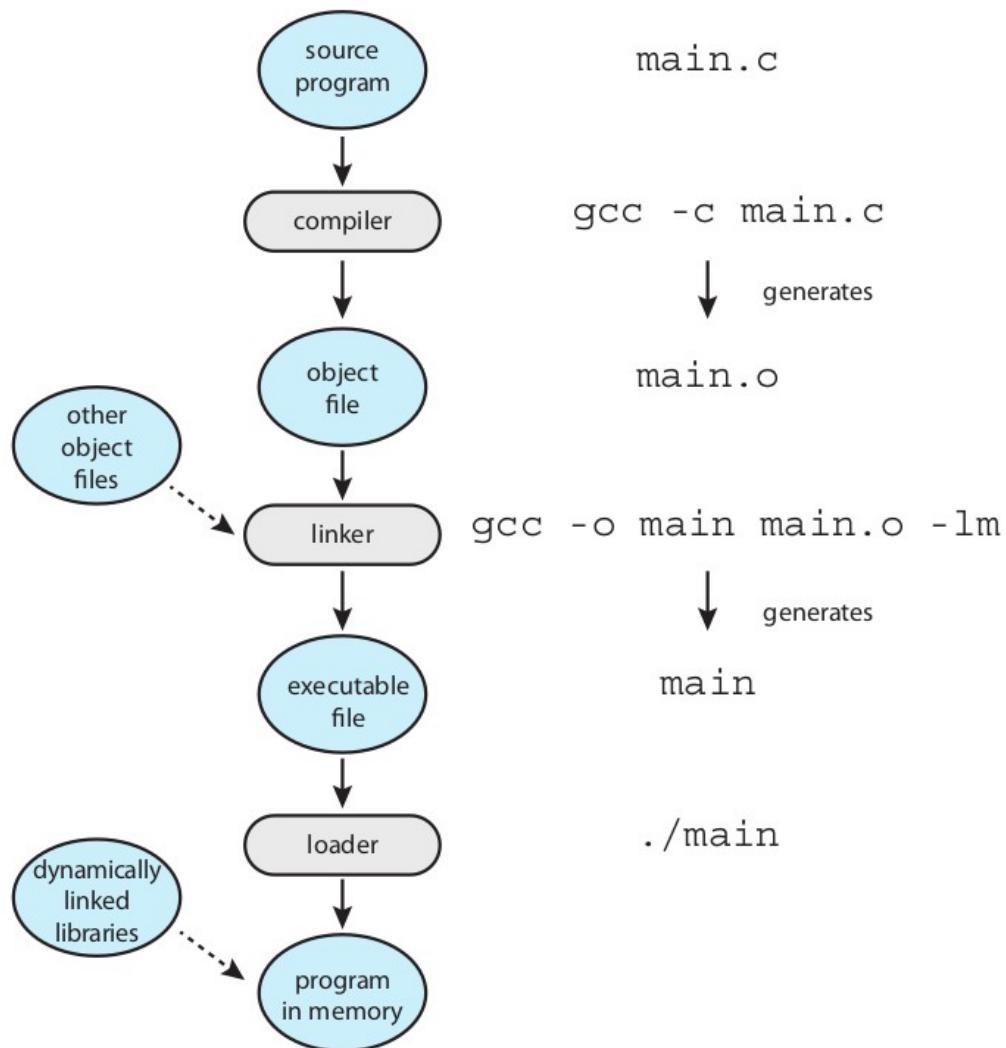
# Assembler

- application program, converts assembly code into machine code
- What is assembly language?
  - Human readable machine code language.
- E.g. x86 assembly code
  - mov 50, r1
  - add 10, r1
  - mov r1, 500
- Usage. eg..
  - \$ as something.s -o something



# Compilation Process





- From the textbook

**Figure 2.11** The role of the linker and loader.

# Example

**try.c**

```
#include <stdio.h>
#define MAX 30
int f(int, int);
int main() {
    int i, j, k;
    scanf("%d%d", &i, &j);
    k = f(i, j) + MAX;
    printf("%d\n", k);
    return 0;
}
```

**f.c**

```
int g(int);
#define ADD(a, b) (a + b)
int f(int m, int n) {
    return ADD(m,n) + g(10);
}
```

**g.c**

```
int g(int x) {
    return x + 10;
}
```

Try these commands, observe the output/errors/warnings, and try to understand what is happening

```
$ gcc try.c
$ gcc -c try.c
$ gcc -c f.c
$ gcc -c g.c
$ gcc try.o f.o g.o -o try
$ gcc -E try.c
$ gcc -E f.c
```

# More about the steps

- Pre-processor
  - `#define ABC XYZ`
    - cut ABC and paste XYZ
  - `# include <stdio.h>`
    - copy-paste the file stdio.h
    - There is no CODE in stdio.h, only typedefs, #includes, #define, #ifdef, etc.
- Linking
  - Normally links with the standard C-library by default
  - To link with other libraries, use the -l option of gcc
    - `cc main.c -lm -lncurses -o main` # links with libm.so and libncurses.so

# Using gcc itself to understand the process

- Run only the preprocessor
  - `cc -E test.c`
  - Shows the output on the screen
- Run only till compilation (no linking)
  - `cc -c test.c`
  - Generates the “test.o” file , runs compilation + assembler
  - `gcc -S main.c`
  - One step before machine code generation, stops at assembly code
- Combine multiple .o files (only linking part)  
`cc test.o main.o try.o -o something`

# Linking steps

- Linker is an application program
  - On linux, it's the "ld" program
  - E.g. you can run commands like \$ ld a.o b.o -o c.o
  - Normally you have to specify some options to ld to get a proper executable file.
- When you run gcc
  - \$ cc main.o f.o g.o -o try
  - the CC will internally invoke "ld" . ld does the job of linking

# Linking steps

- The resultant file "try" here, will contain the codes of all the functions and linkages also.
- **What is linking?**
  - "connecting" the call of a function with the code of the function.
- What happens with the code of printf()
  - The linker or CC will automatically pick up code from the libc.so.6 file for the functions.

# Executable file format

- An executable file needs to execute in an environment created by OS and on a particular processor
  - Contains machine code + other information for OS
  - Need for a structured-way of storing machine code in it
- Different OS demand different formats
  - Windows: PE, Linux: ELF, Old Unixes: a.out, etc.
- ELF : The format on Linux.
- Try this
  - `$ file /bin/ls`
  - `$ file /usr/lib/x86_64-linux-gnu/libc-2.31.so`

# Exec() and ELF

- When you run a program
  - \$ ./try
  - Essentially there will be a fork() and exec("./try", ...)
  - So the kernel has to read the file "./try" and understand it.
  - So each kernel will demand its own object code file format.
  - Hence ELF, EXE, etc. Formats
- ELF is used not only for executable (complete machine code) programs, but also for partially compiled files e.g. main.o and library files like libc.so.6
- What is a.out?
  - "a.out" was the name of a format used on earlier Unixes.
  - It so happened that the early compiler writers, also created executable with default name 'a.out'

# Utilities to play with object code files

- **objdump**
  - `$ objdump -D -x /bin/ls`
  - Shows all disassembled machine instructions and “headers”
- **hexdump**
  - `$ hexdump /bin/ls`
  - Just shows the file in hexadecimal
- **readelf**
  - Alternative to objdump
- **ar**
  - To create a “statically linked” library file
  - `$ ar -crs libmine.a one.o two.o`
- **Gcc to create shared library**
  - `$ gcc hello.o -shared -o libhello.so`
- To see how gcc invokes as, ld, etc; do this
  - `$ gcc -v hello.c -o hello`
  - `/*  
https://stackoverflow.com/questions/1170809/how-to-get-gcc-linker-command  
*/`

# Linker, Loader, Link-Loader

- Linker or linkage-editor or link-editor
  - The “ld” program. Does linking.
- Loader
  - The exec(). It loads an executable in the memory.
- Link-Loader
  - Often the linker is called link-loader in literature. Because where were days when the linker and loader's jobs were quite over-lapping.

# Static, dynamic / linking, loading

- Both linking and loading can be
  - Static or dynamic
  - More about this when we learn memory management
- An important fundamental:
  - memory management features of processor, memory management architecture of kernel, executable/object-code file format, output of linker and job of loader, are all interdependent and in-separable.
  - They all should fit into each other to make a system work
  - That's why the phrase “system programs”

# Cross-compiler

- Compiler on system-A, but generate object-code file for system-B (target system)
  - E.g. compile on Ubuntu, but create an EXE for windows
- Normally used when there is no compiler available on target system
  - see gcc -m option
- See [https://wiki.osdev.org/GCC\\_Cross-Compiler](https://wiki.osdev.org/GCC_Cross-Compiler)

# **Calling Convention**

Abhijit A M

# The need for calling convention

## An essential task of the compiler

Generates object code (file) for given source code (file)

Processors provide simple features

Registers, machine instructions (add, mov, jmp, call, etc.), imp registers like stack-pointer, etc; ability to do byte/word sized operations

No notion of data types, functions, variables, etc.

But languages like C provide high level features

Data types, variables, functions, recursion, etc

Compiler needs to map the features of C into processor's features, and then generate machine code

In reality, the language designers design a language feature only after answering the question of conversion into machine code

# The need for calling convention

## Examples of some of the challenges before the compiler

“call” + “ret” does not make a C-function call!

A “call” instruction in processor simply does this

Pushes IP(that is PC) on stack + Jumps to given address

This is not like calling a C-function !

Unsolved problem: How to handle parameters, return value?

Processor does not understand data types!

Although it has instructions for byte, word sized data and can differentiate between integers and reals (mov, movw, addl, addf, etc. )

# Compiler and Machine code generation

## Example, code inside a function

```
int a, b, c;  
c = a + b;
```

## What kind of code is generated by compiler for this?

```
sub 12, <esp> #normally local variables are located on stack, make space  
mov <location of a in memory>, r1 #location is on stack, e.g. -4(esp)  
mov <location of b in memory>, r2  
add r1, r2 # result in r1  
mov r1, <location of c in memory>
```

# Compiler and Machine code generation

## Across function calls

```
int f(int m, n) {  
    int x = m, y = n;  
    return g(x, y);  
}  
  
int x(int a) {  
    return g(a, a+1);  
}  
  
int g(int p, int q) {  
    p = p * q + p;  
    return p  
}
```

**g() may be called from f()  
or from x()**

**Sequence of function calls  
can NOT be predicted by  
compiler**

**Compiler has to generate  
machine code for each  
function assuming  
nothing about the caller**

# Compiler and Machine code generation

## Machine code generation for functions

Mapping C language features to existing machine code instructions.

Typical examples

a = 100 ; ==> mov instruction

a = b + c; ==> mov, add instructions

while(a < 5) { j++; } ==> mov, cmp, jlt, add, etc. Instruction

## Where are the local variables in memory?

The only way to store them is on a stack.

Why?

# Function calls

## LIFO

Last in First Out

Must need a “stack” like feature to implement them

## Processor Stack

Processors provide a stack pointer

%esp on x86

Instructions like push and pop are provided by hardware

They automatically increment/decrement the stack pointer. On x86 stack grows downwards (subtract from esp!)

Unlike a “stack data type” data structure, this “stack” is simply implemented with only the esp (as the “top”). The entire memory can be treated as the “array”.

# Function calls

## System stack, compilers, Languages

Processor provides us with esp (stack) and push/pop instructions.

The esp pointer is initialized to a proper value at the time of fork-exec by the OS for each process. Then process runs.

Knowing the above, compilers go ahead with generating machine code using the *esp*.

This means, Langauges like C which provide for function calls, and recursion can only run on processors which support a system stack.

## Convention needed

How to use the stack for effective implementation of function calls ?

## What goes on stack?

Local variables

Function Parameters

Return address of instruction which called a function !

# Activation Record

**Local Vars + parameters + return address**

**When functions call each other**

One activation record is built on stack for each function call

On function return, the record is destroyed

**On x86**

ebp and esp pointers are used to denote the activation record.

How? We will see soon. You may start exploring with "gcc -S" output assembly code.

# X86 instructions

## leave

Equivalent to

```
mov %ebp, %esp # esp = ebp  
pop %ebp
```

## ret

Equivalent to

```
pop %ecx  
Jmp %ecx
```

## call x

Equivalent to

```
push %eip  
jmp x
```

# X86 instructions

## **endbr64**

Normally a NOP

# Let's see some examples now

## Let's compile using

```
gcc -S
```

See code and understand

# simple.c and simple.s

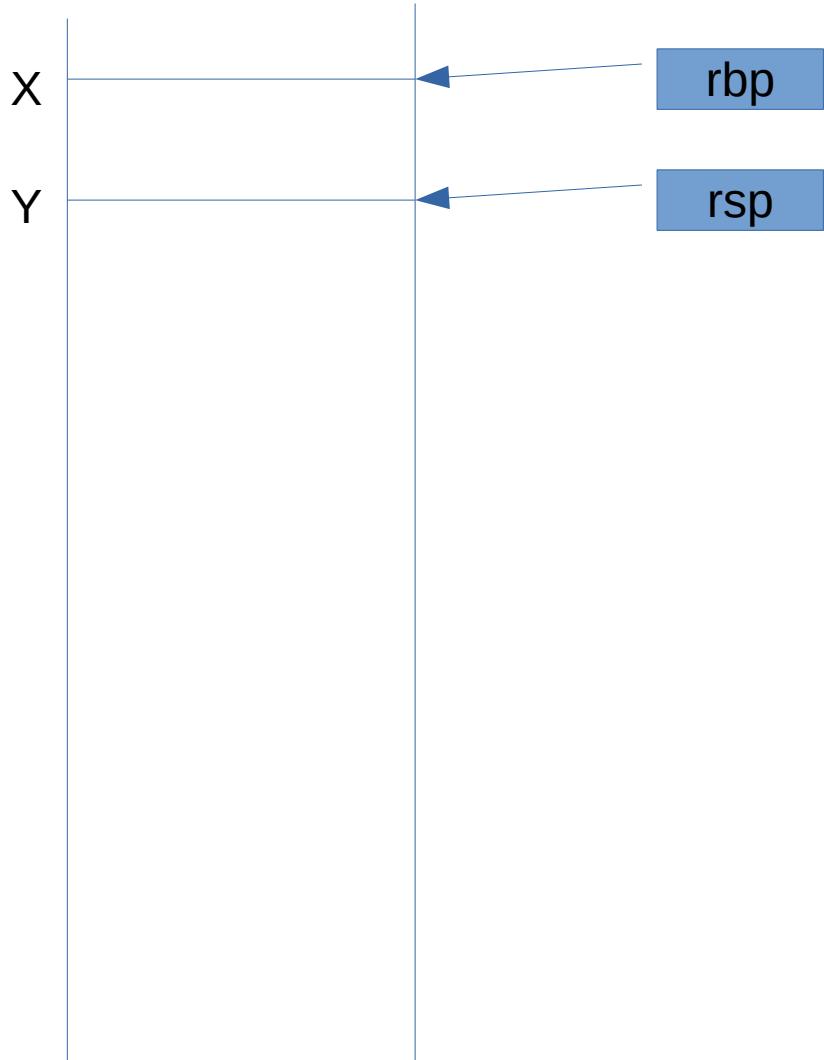
```
int f(int x) {  
    int y;  
    y = x + 3;  
    return y;  
}  
  
int main() {  
    int a = 20, b = 30;  
    b = f(a);  
    return b;  
}
```

main:

```
    endbr64  
    pushq  %rbp  
    movq   %rsp, %rbp  
    subq   $16, %rsp  
    movl   $20, -8(%rbp)  
    movl   $30, -4(%rbp)  
    movl   -8(%rbp), %eax  
    movl   %eax, %edi  
    call   f  
    movl   %eax, -4(%rbp)  
    movl   -4(%rbp), %eax  
    leave  
    ret
```

f:

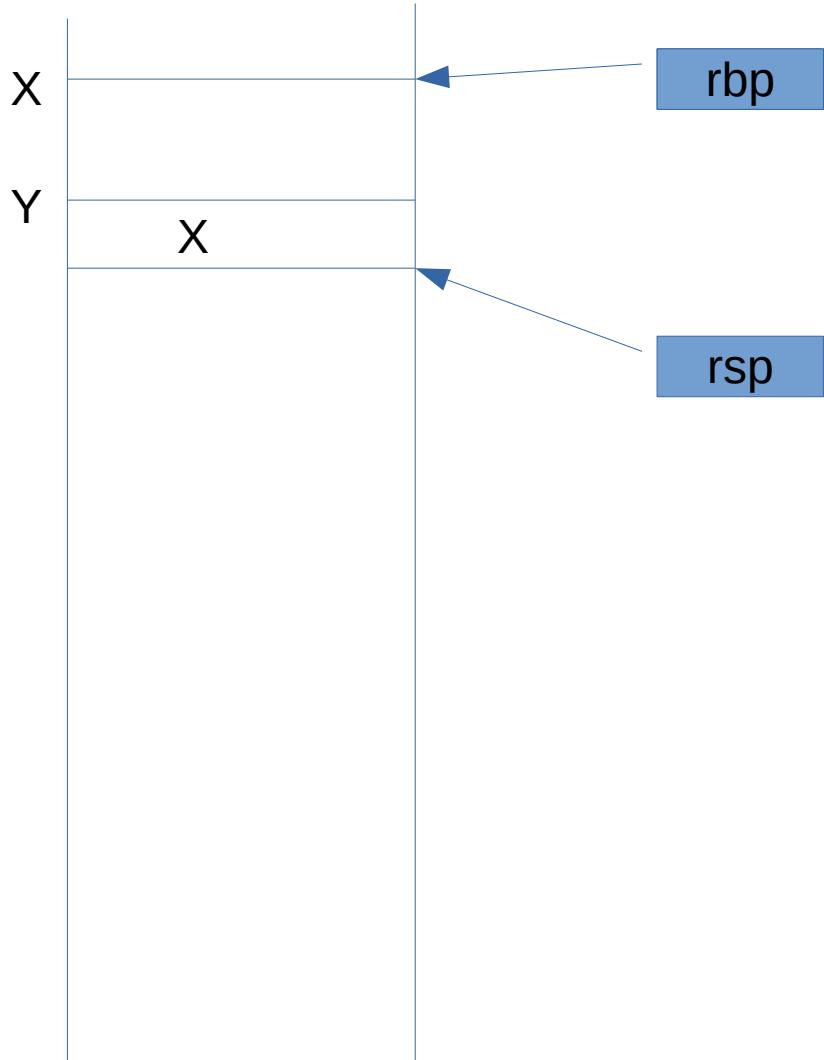
```
    endbr64  
    pushq  %rbp  
    movq   %rsp, %rbp  
    movl   %edi, -20(%rbp)  
    movl   -20(%rbp), %eax  
    addl   $3, %eax  
    movl   %eax, -4(%rbp)  
    movl   -4(%rbp), %eax  
    popq   %rbp  
    ret
```



main:

```
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
ret
```

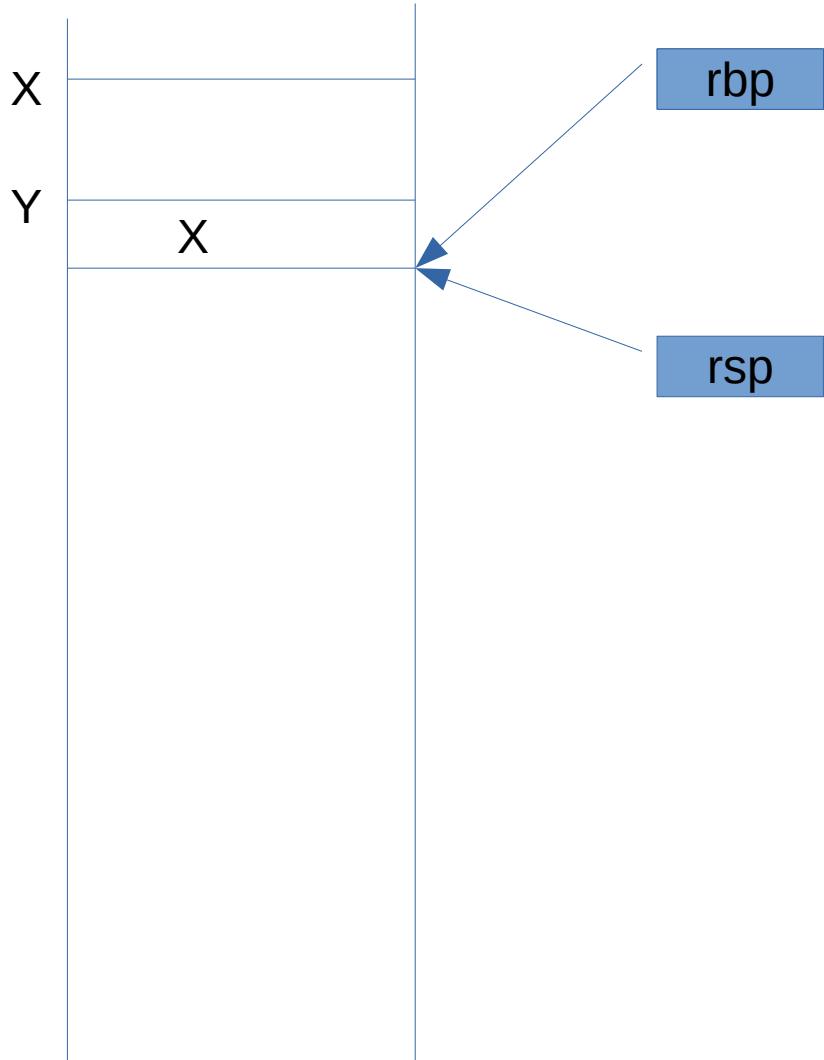
```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



main:

```
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
ret
```

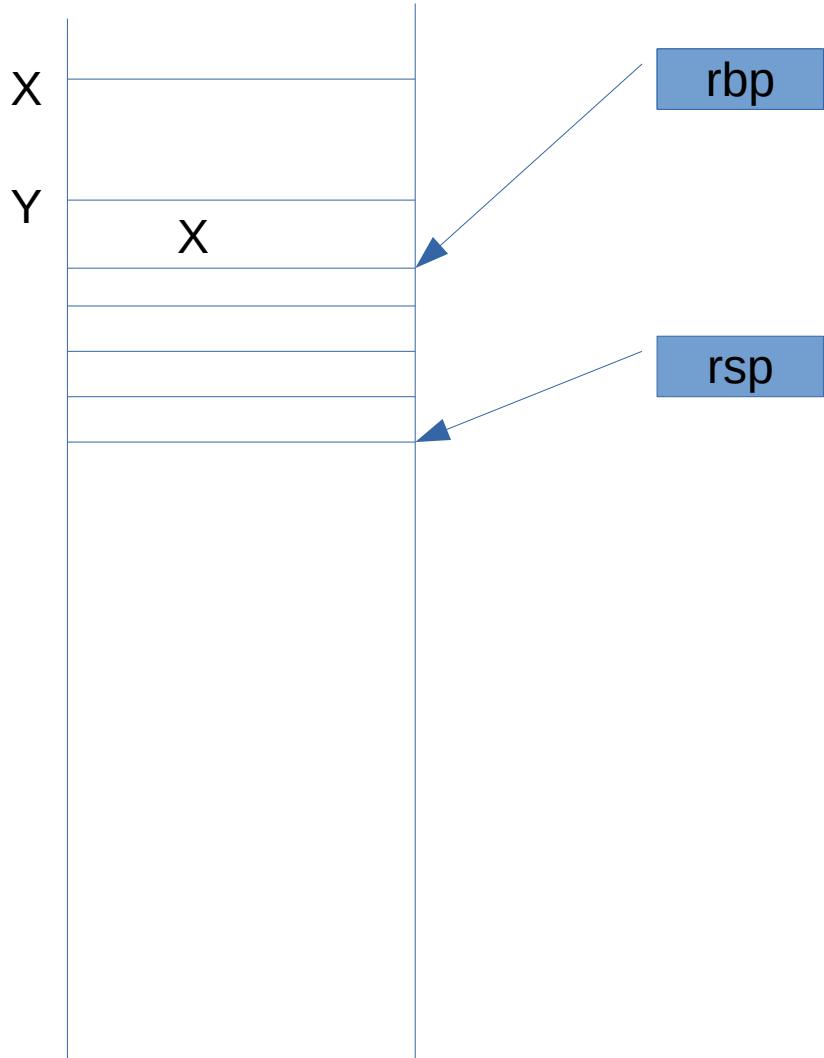
```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



main:

```
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
ret
```

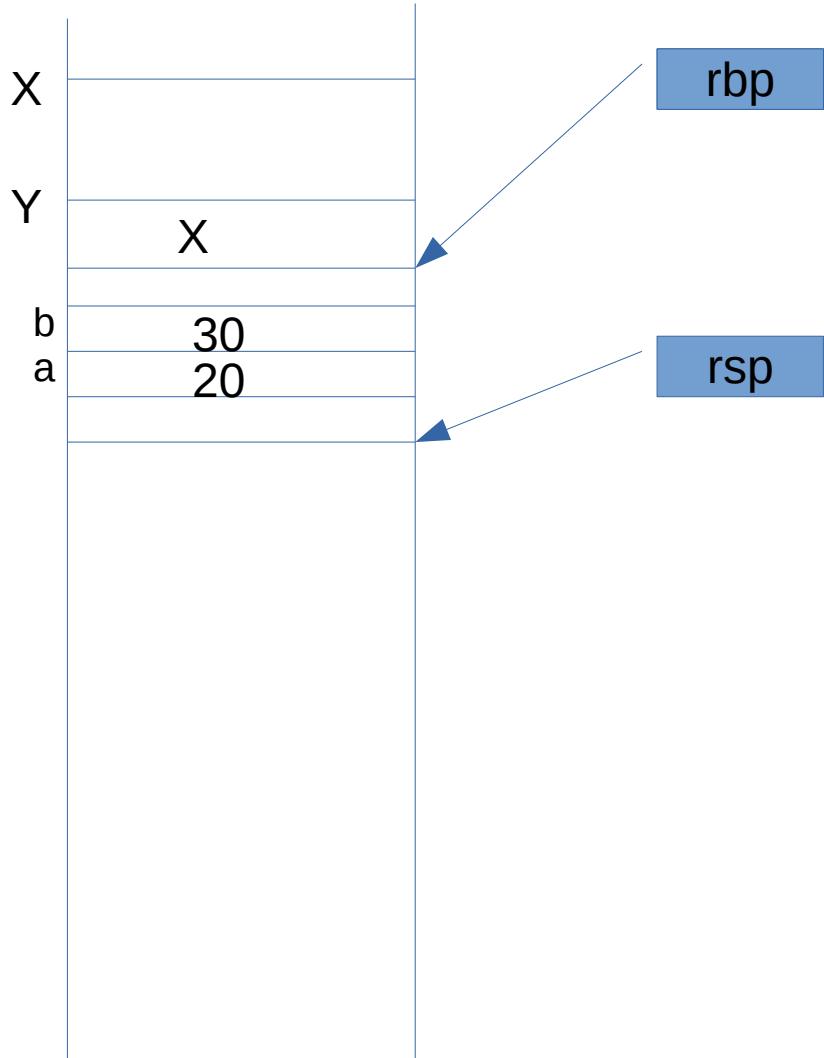
```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



main:

```
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
ret
```

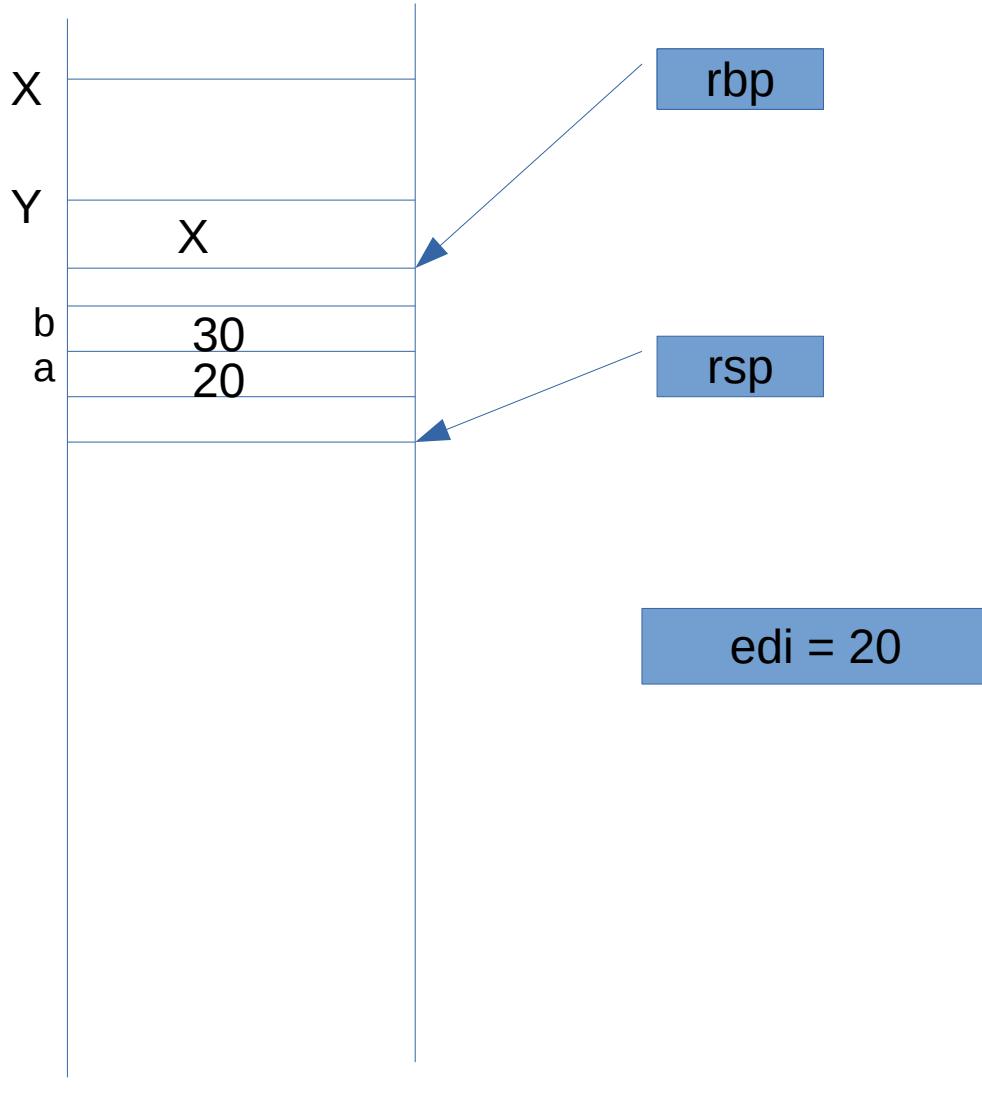
```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



main:

```
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
ret
```

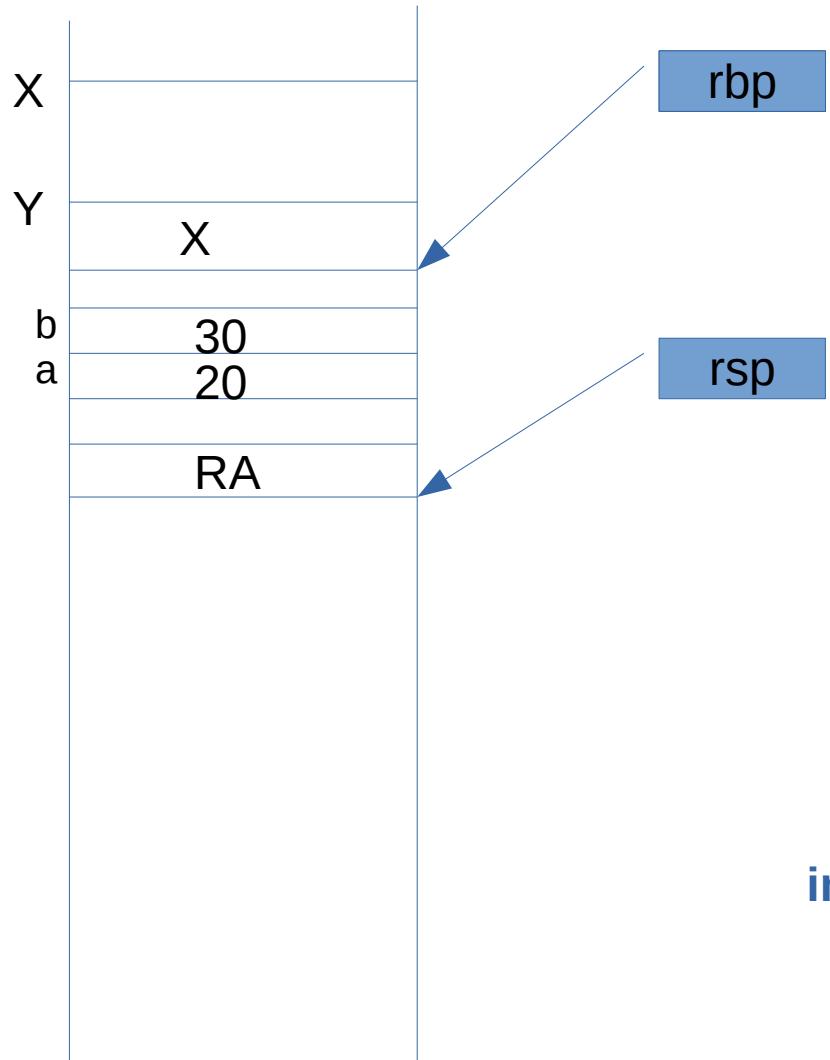
```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



main:

```
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
ret
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



```

main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret

int f(int x) {
    int y;
    y = x + 3;
    return y;
}

```

```

f:
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret

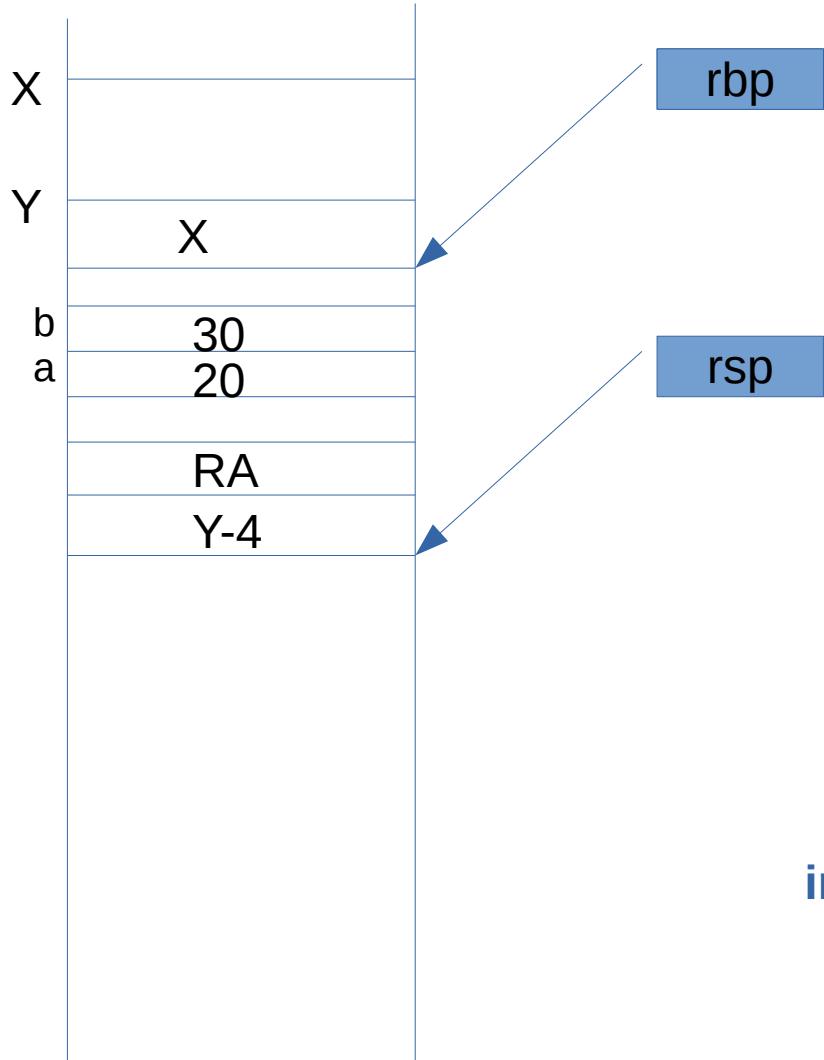
```

`edi = 20`

```

int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}

```



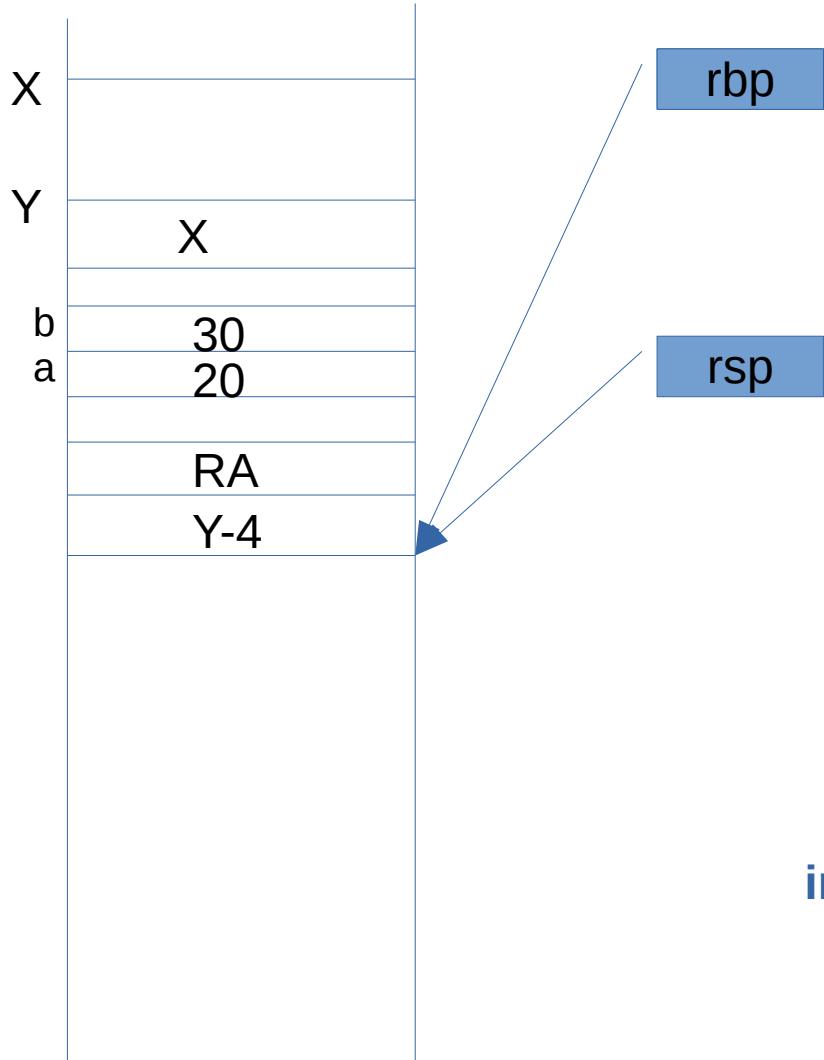
```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

f:

```
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret
```

edi = 20



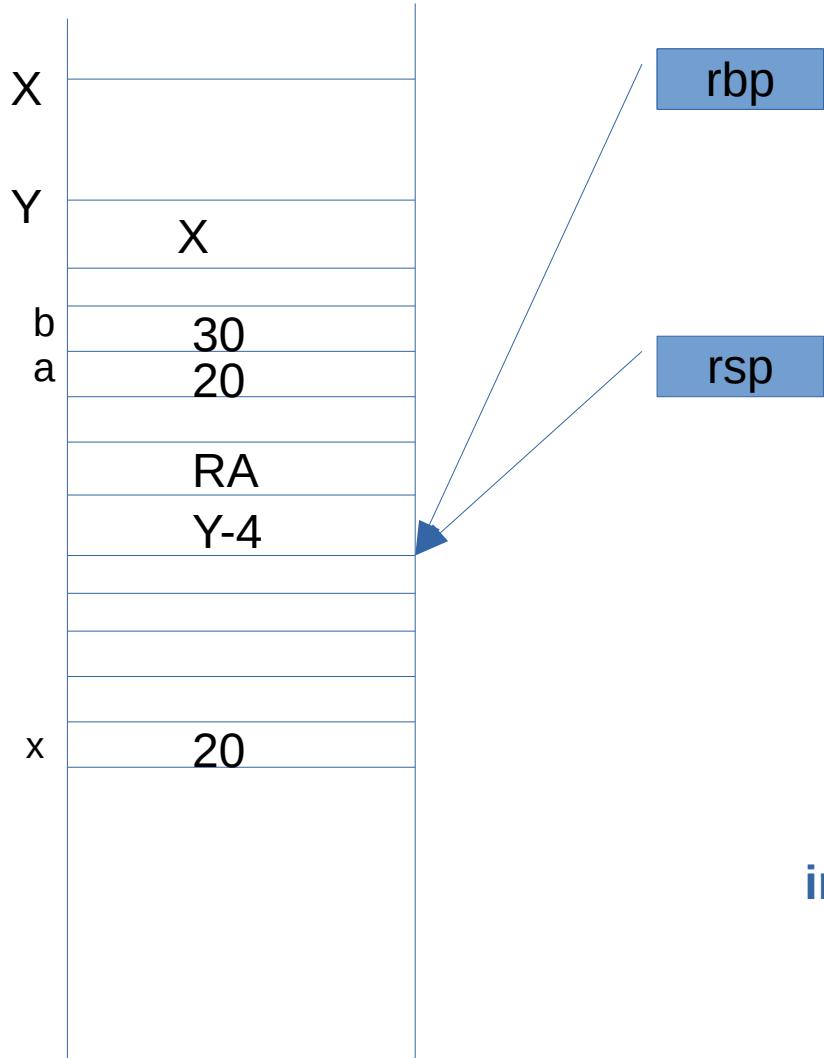
```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

```
f:
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret
```

edi = 20

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



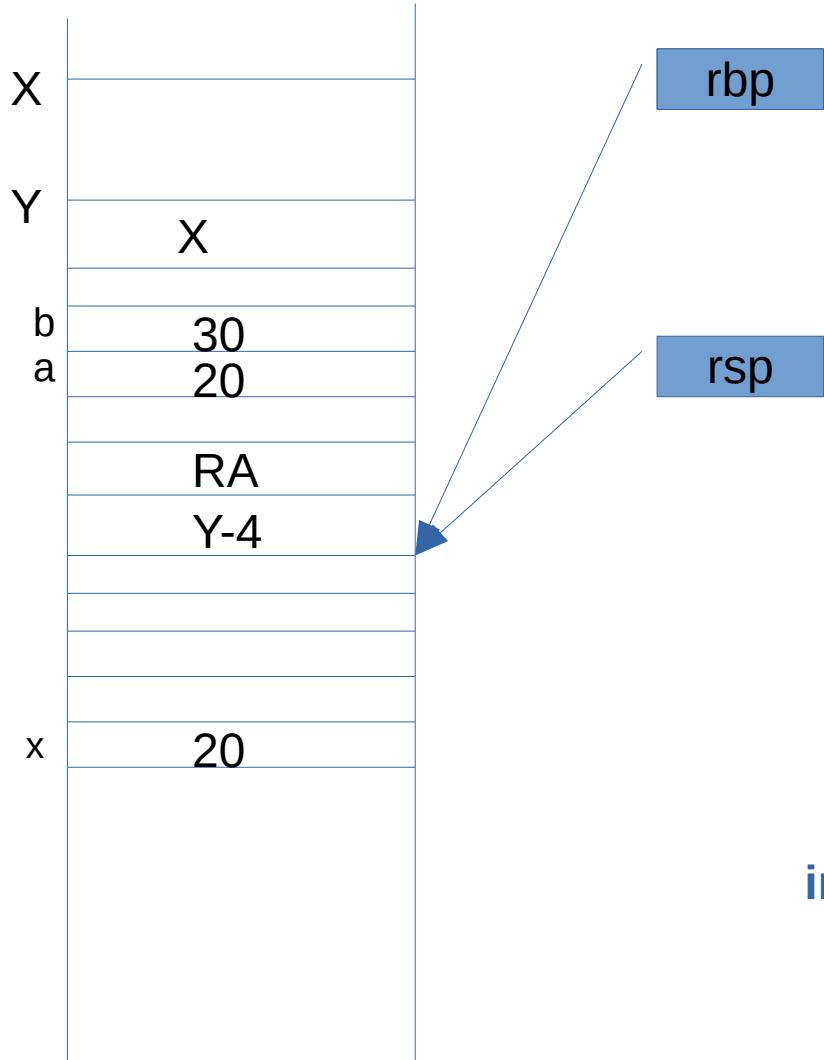
```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

```
f:
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret
```

edi = 20

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



```

main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
    
```

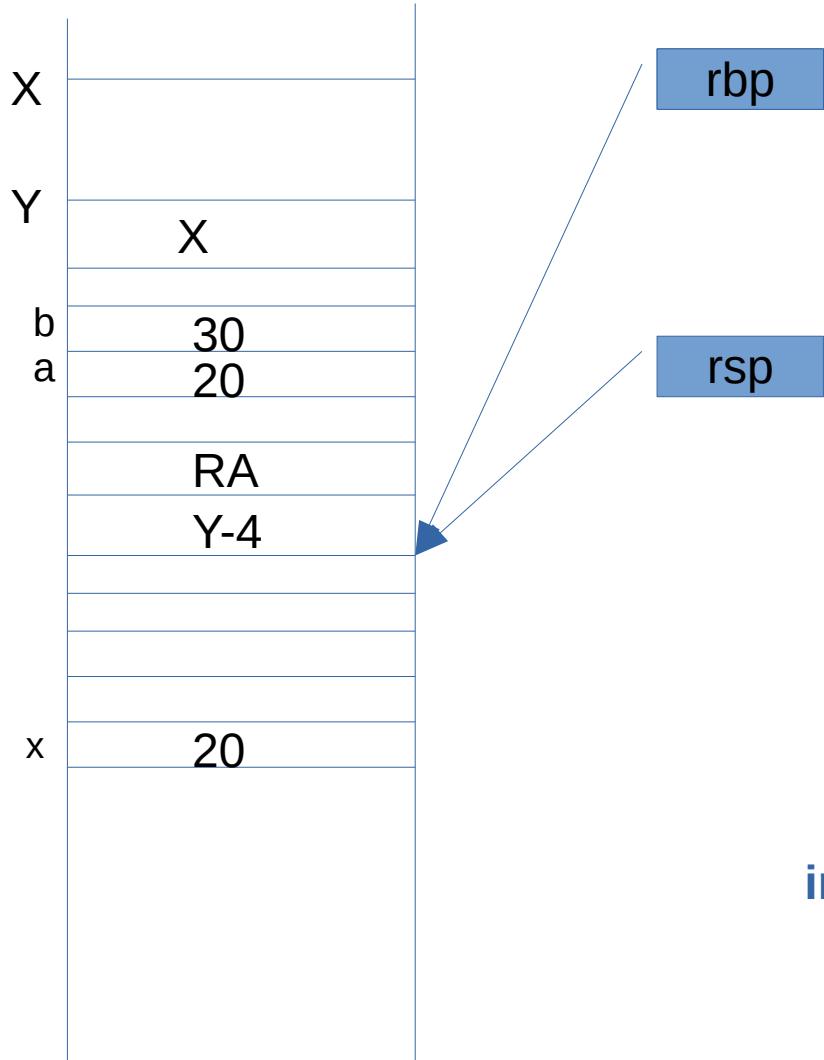
f:

```

endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret
    
```

edi = 20

eax = 20



```

main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
    
```

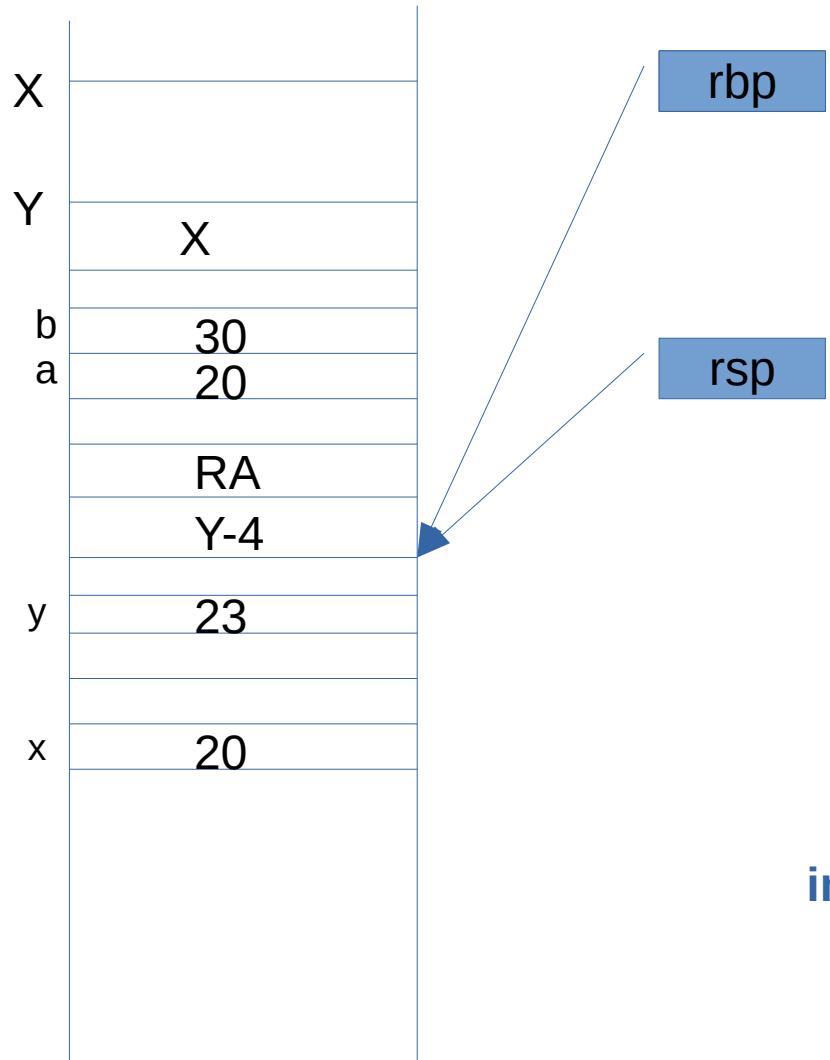
f:

```

endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret
    
```

edi = 20

eax = 23



```

main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}

```

f:

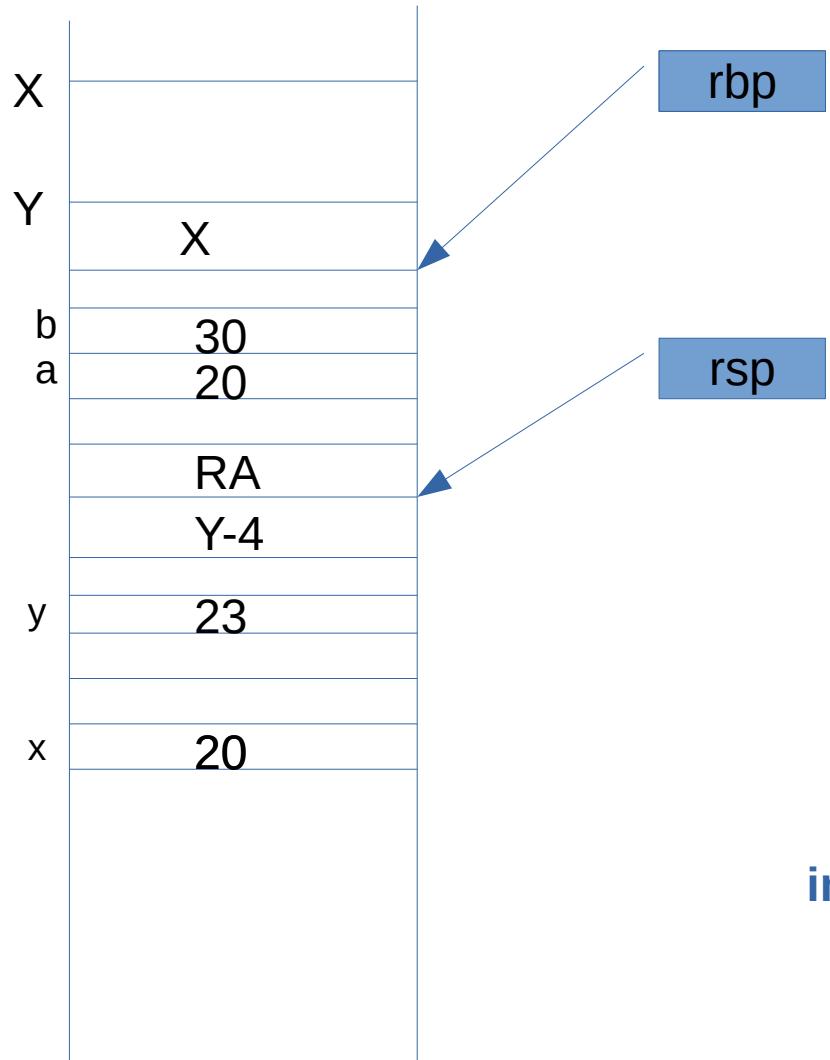
```

endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret

```

edi = 20

eax = 23



```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

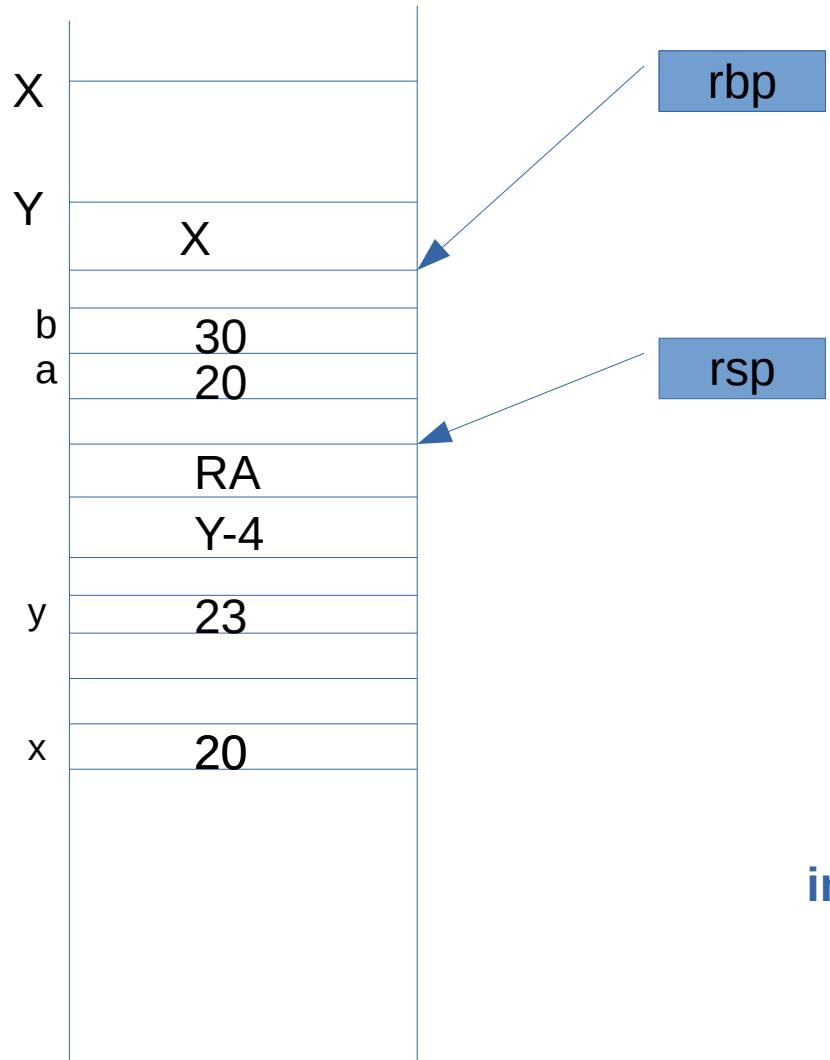
```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

f:

```
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret
```

edi = 20

Eax = 23



```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

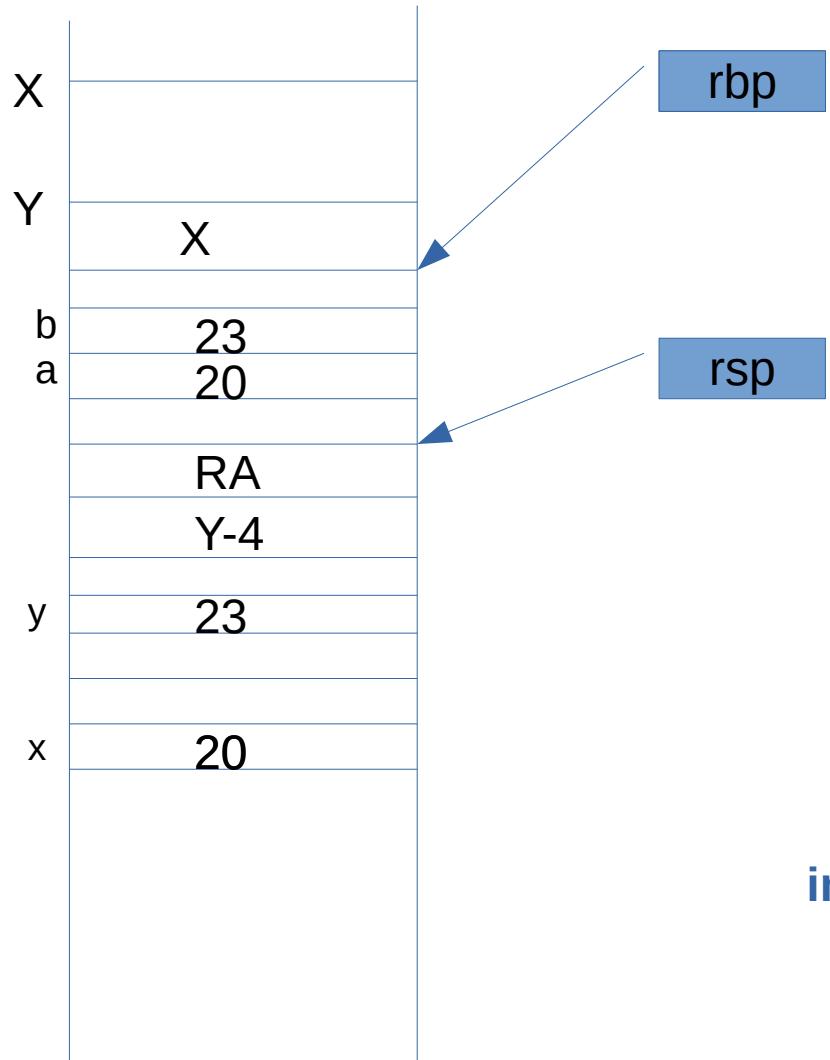
```
f:
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret
```

edi = 20

eax = 23

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```

eip = RA



```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
ret
```

f:

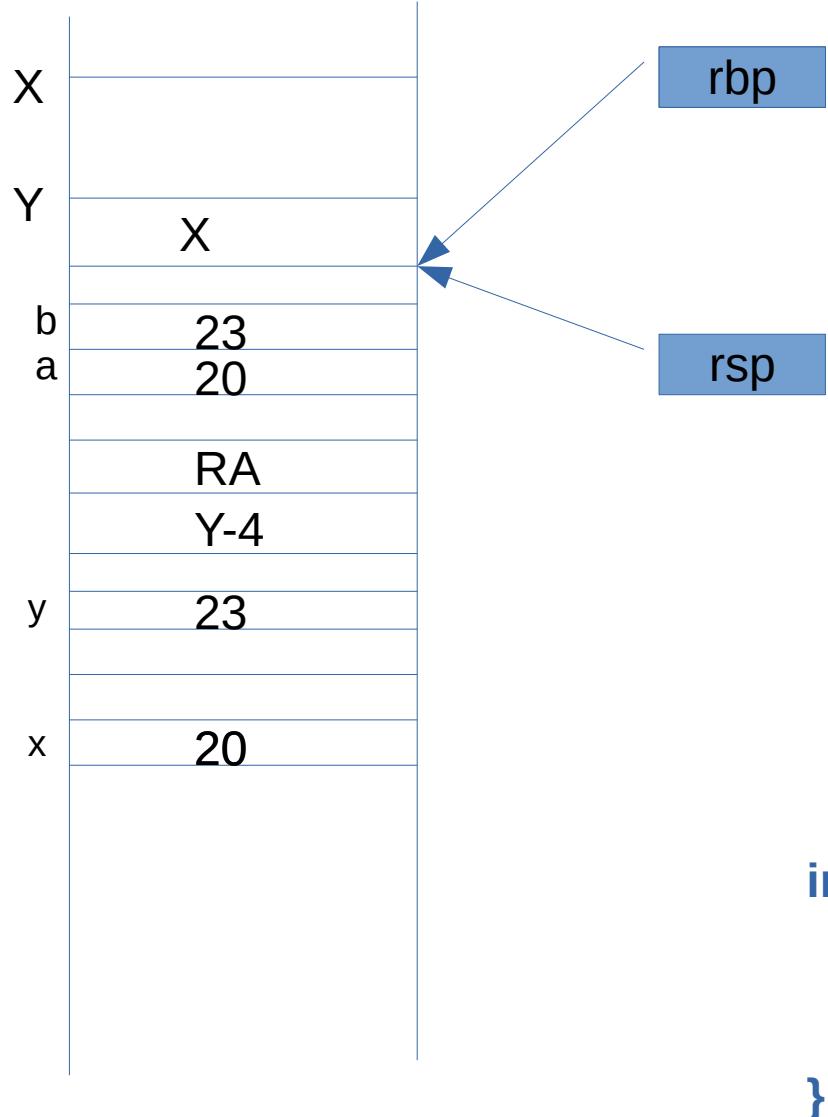
```
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret
```

edi = 20

eax = 23

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```

eip = RA



```

int f(int x) {
  int y;
  y = x + 3;
  return y;
}
  
```

```

main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
# mov rbp rsp; pop rbp
ret
  
```

```

int main() {
  int a = 20, b = 30;
  b = f(a);
  return b;
}
  
```

f:

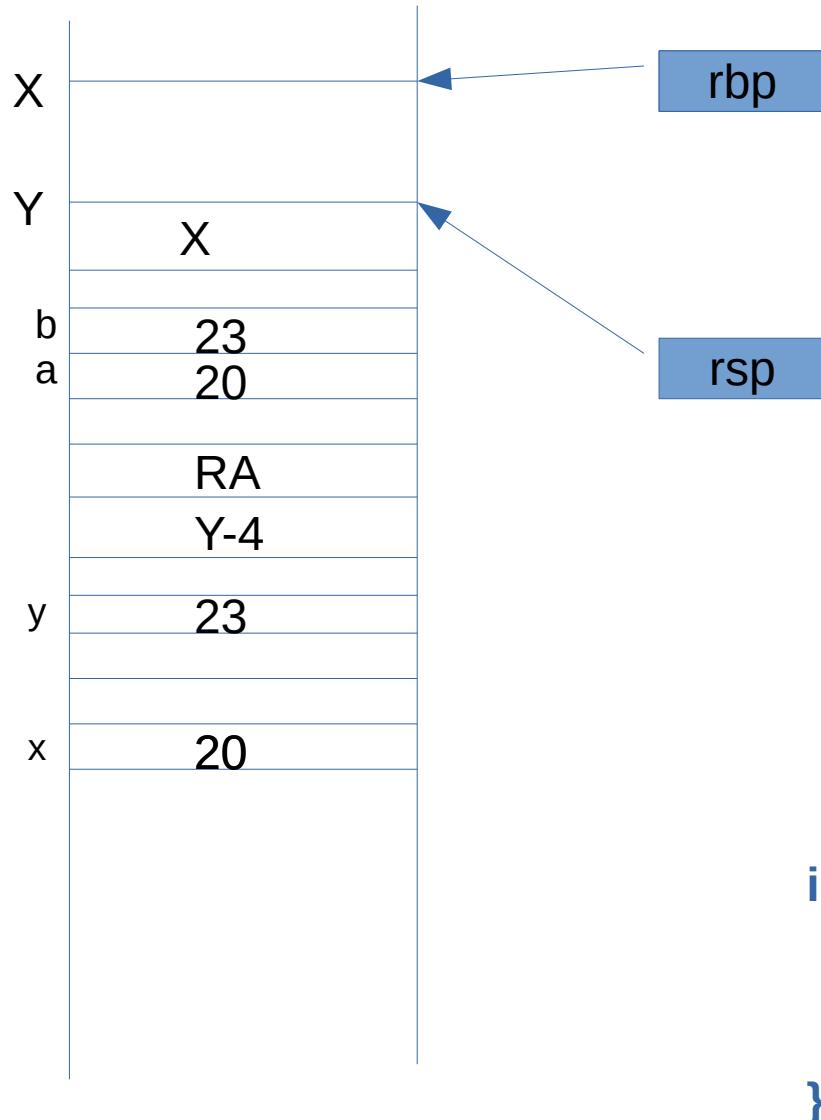
```

endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret
  
```

edi = 20

eax = 23

eip = RA



main:  
endbr64

```

pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
# mov ebp esp; pop ebp
ret
  
```

```

int f(int x) {
  int y;
  y = x + 3;
  return y;
}
  
```

```

int main() {
  int a = 20, b = 30;
  b = f(a);
  return b;
}
  
```

f:

```

endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret
  
```

edi = 20

eax = 23

eip = RA

# **Further on calling convention**

**This was a simple program**

**The parameter was passed in a register!**

What if there were many parameters?

CPUs have different numbers of registers.

**More parameters, more functions demand a  
more sophisticated convention**

May be slightly different on different processors, or 32-bit, 64-bit variants also.

# Caller save and Callee save registers

## Local variables

Are visible only within the function

Recursion: different copies of variables

Stored on "stack"

## Registers

Are only one copy

Are within the CPU

## Local Variables & Registers conflict

Compiler's dilemma: While generating code for a function, which registers to use?

The register might have been in use in earlier function call

# **Caller save and Callee save registers**

## **Caller Save registers**

Which registers need to be saved by caller function . They can be used by the callee function!

The caller function will push them (if already in use, otherwise no need) on the stack

## **Callee save registers**

Will be pushed on to the stack by called (callee) function

## **How to return values?**

On the stack itself – then caller will have to pop

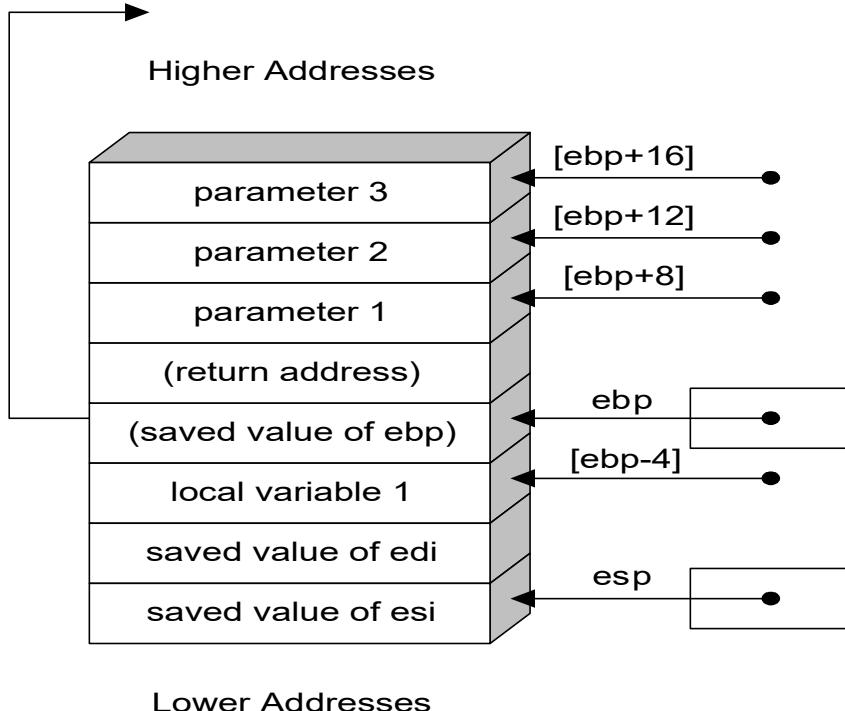
In a register, e.g. eax

# X86 convention - caller, callee saved 32 bit

The **caller-saved registers** are EAX, ECX, EDX.

The **callee-saved registers** are EBX, EDI, and ESI

# Activation record looks like this



**F() called g()**

**Parameters-i refers to parameters passed by f() to g()**

**Local variable is a variable in g()**

**Return address is the location in f() where call should go back**

# X86 caller and callee rules(32 bit)

## Caller rules on call

Push caller saved registers on stack

Push parameters on the stack – in reverse order. Why?

Substract esp, copy data

call f() // push + jmp

## Caller rules on return

return value is in eax

remove parameters from stack : Add to esp.

Restore caller saved registers (if any)

# X86 caller and callee rules

## Callee rules on call

1) push ebp

mov ebp, esp

ebp(+/-offset) normally used to locate local vars and parameters on stack

ebp holds a copy of esp

Ebp is pushed so that it can be later popped while returnig

2) Allocate local variables

3) Save callee-saved registers

# X86 caller and callee rules

## Callee rules on return

- 1) Leave return value in eax
- 2) Restore callee saved registers
- 3) Deallocate local variables
- 4) restore the ebp
- 5) return

# **32 bit vs 64 bit calling convention**

**Registers are used for passing parameters in 64 bit , to a large extent**

Upto 6 parameters

More parameters pushed on stack

**See**

<https://aaronbloomfield.github.io/pdr/book/x86-64bit-ccc-chapter.pdf>

# Beware

**When you read assembly code generated using**

gcc -S

**You will find**

More complex instructions

But they will essentially follow the convention mentioned

# Comparison

	<b>MIPS</b>	<b>x86</b>
<b>Arguments:</b>	First 4 in %a0–%a3, remainder on stack	Generally all on stack
<b>Return values:</b>	%v0–%v1	%eax
<b>Caller-saved registers:</b>	%t0–%t9	%eax, %ecx, & %edx
<b>Callee-saved registers:</b>	%s0–%s9	Usually none

**Figure 6.2:** A comparison of the calling conventions of MIPS and x86

From the textbook by Misruada

# simple3.c and simple3.s

```
int f(int x1, int x2, int x3, int x4, int x5, int x6, int x7, int x8, int x9, int x10) {  
    int h;  
    h = x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10 + 3;  
    return h;  
}  
int main() {  
    int a1 = 10, a2 = 20, a3 = 30, a4 = 40, a5 = 50, a6 = 60, a7 = 70, a8 = 80, a9 = 90, a10 = 100;  
    int b;  
    b = f(a1, a2, a3, a4, a5, a6, a7, a8, a9, a10);  
    return b;  
}
```

# simple3.c and simple3.s

main:

endbr64	movl -24(%rbp), %r9d	
pushq %rbp	movl -28(%rbp), %r8d	movl %eax, %edi
movq %rsp, %rbp	movl -32(%rbp), %ecx	call f
subq \$48, %rsp	movl -36(%rbp), %edx	addq \$32, %rsp
movl \$10, -44(%rbp)	movl -40(%rbp), %esi	movl %eax, -4(%rbp)
movl \$20, -40(%rbp)	movl -44(%rbp), %eax	movl -4(%rbp), %eax
movl \$30, -36(%rbp)	movl -8(%rbp), %edi	leave
movl \$40, -32(%rbp)	pushq %rdi	ret
movl \$50, -28(%rbp)	movl -12(%rbp), %edi	
movl \$60, -24(%rbp)	pushq %rdi	
movl \$70, -20(%rbp)	movl -16(%rbp), %edi	
movl \$80, -16(%rbp)	pushq %rdi	
movl \$90, -12(%rbp)	movl -20(%rbp), %edi	
movl \$100, -8(%rbp)	pushq %rdi	

# simple3.c and simple3.s

f:

endbr64		
pushq %rbp	addl %eax, %edx	
movq %rsp, %rbp	movl -36(%rbp), %eax	movl 40(%rbp), %eax
movl %edi, -20(%rbp)	addl %eax, %edx	addl %edx, %eax
movl %esi, -24(%rbp)	movl -40(%rbp), %eax	addl \$3, %eax
movl %edx, -28(%rbp)	addl %eax, %edx	movl %eax, -4(%rbp)
movl %ecx, -32(%rbp)	movl 16(%rbp), %eax	movl -4(%rbp), %eax
movl %r8d, -36(%rbp)	addl %eax, %edx	popq %rbp
movl %r9d, -40(%rbp)	movl 24(%rbp), %eax	ret
movl -20(%rbp), %edx	addl %eax, %edx	
movl -24(%rbp), %eax	movl 32(%rbp), %eax	
addl %eax, %edx	addl %eax, %edx	
movl -28(%rbp), %eax		
addl %eax, %edx		
movl -32(%rbp), %eax		

Let's see a demo of how the stack is built and destroyed during function calls, on a Linux machine using GCC.

### Consider this C code

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

### Translated to assembly as:

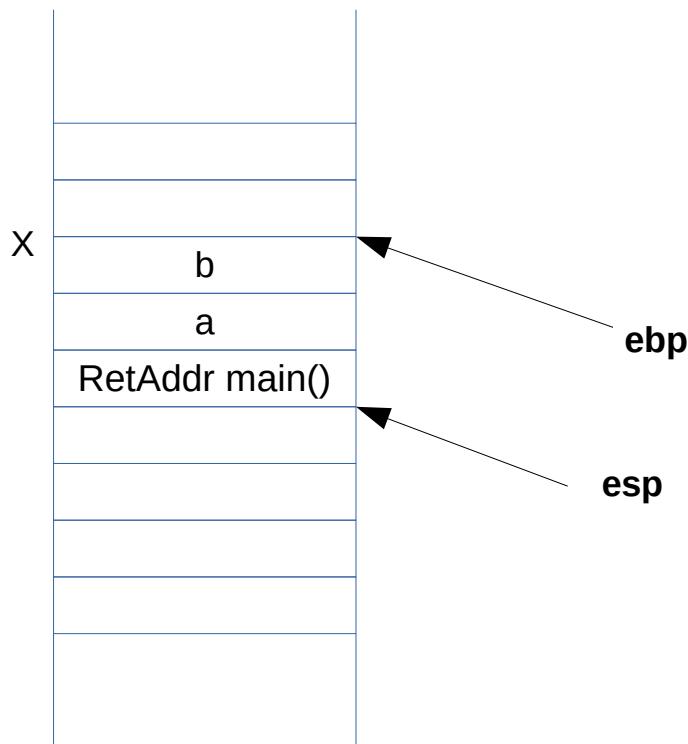
#### add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax  
leave  
ret
```

#### mult:

```
pushl %ebp  
movl %esp, %ebp  
subl $24, %esp  
movl $20, -24(%ebp)  
movl $30, -20(%ebp)  
subl $8, %esp  
pushl -20(%ebp)  
pushl -24(%ebp)  
call add  
addl $16, %esp  
movl %eax, -16(%ebp)  
movl 8(%ebp), %eax  
imull 12(%ebp), %eax  
movl %eax, %edx  
movl -16(%ebp), %eax  
addl %edx, %eax  
movl %eax, -12(%ebp)  
movl -12(%ebp), %eax  
leave  
ret
```

Stack



*/\* Control is here \*/*

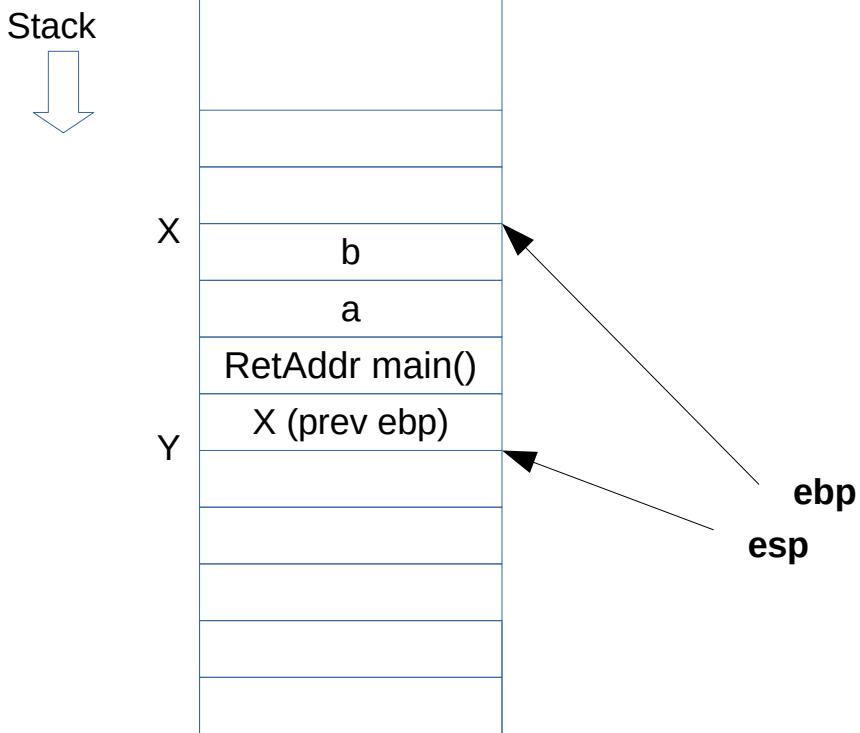
```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}
```

```
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

*\* Control is here \**

mult:

```
pushl %ebp  
movl %esp, %ebp  
subl $24, %esp  
movl $20, -24(%ebp)  
movl $30, -20(%ebp)  
subl $8, %esp  
pushl -20(%ebp)  
pushl -24(%ebp)  
call add
```



```

int mult(int a, int b) {
    int c, d = 20, e = 30, f;
    f = add(d, e);
    c = a * b + f;
    return c;
}
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}

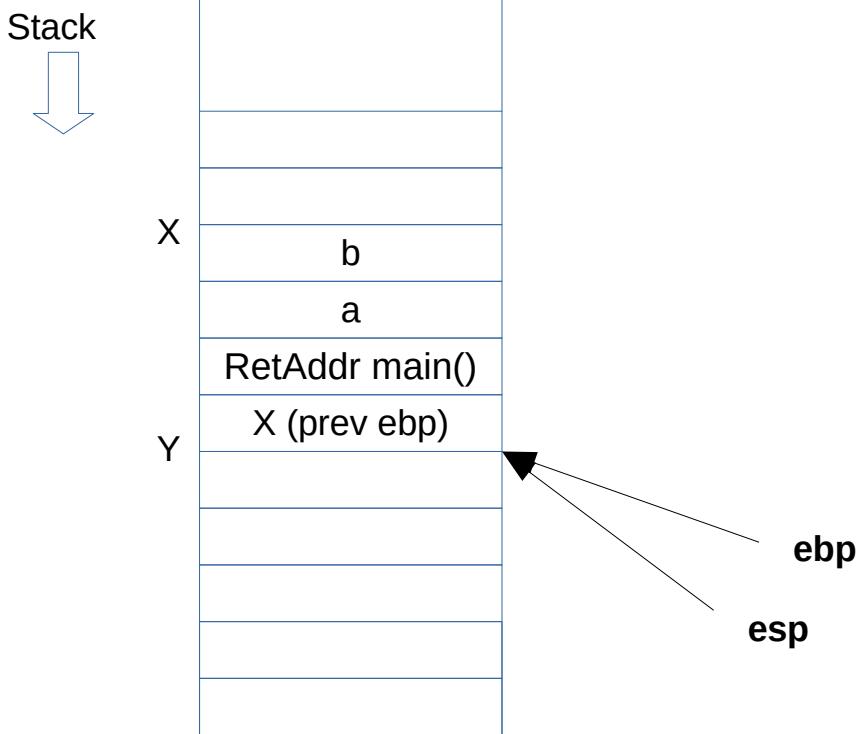
```

### mult:

```

pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $20, -24(%ebp)
movl $30, -20(%ebp)
subl $8, %esp
pushl -20(%ebp)
pushl -24(%ebp)
call add

```



```

int mult(int a, int b) {
    int c, d = 20, e = 30, f;
    f = add(d, e);
    c = a * b + f;
    return c;
}
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}

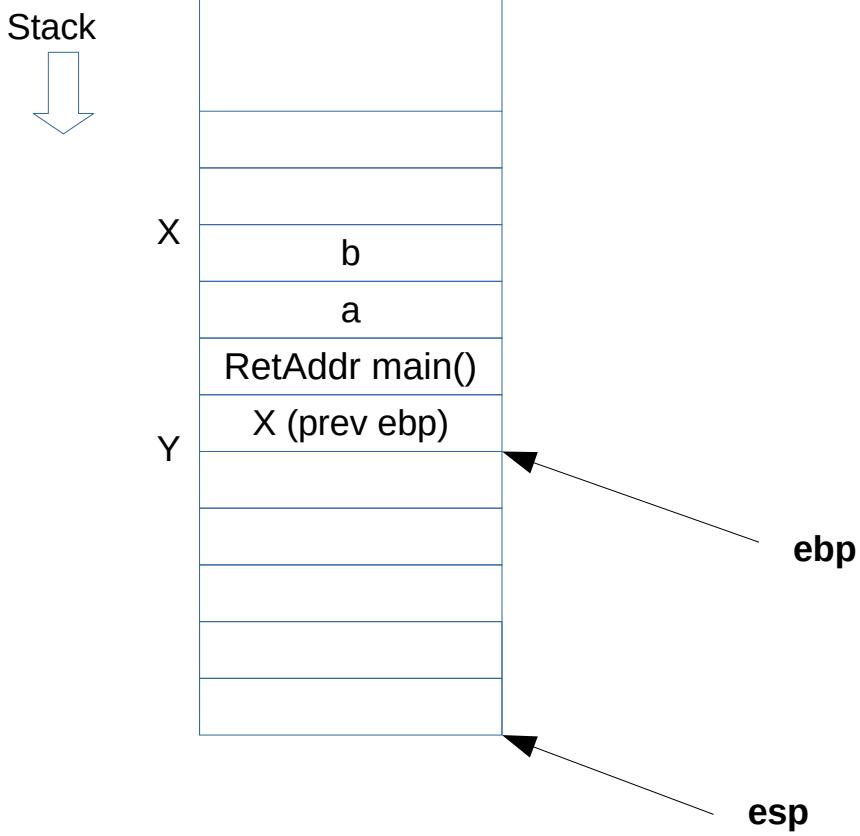
```

### mult:

```

pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $20, -24(%ebp)
movl $30, -20(%ebp)
subl $8, %esp
pushl -20(%ebp)
pushl -24(%ebp)
call add

```



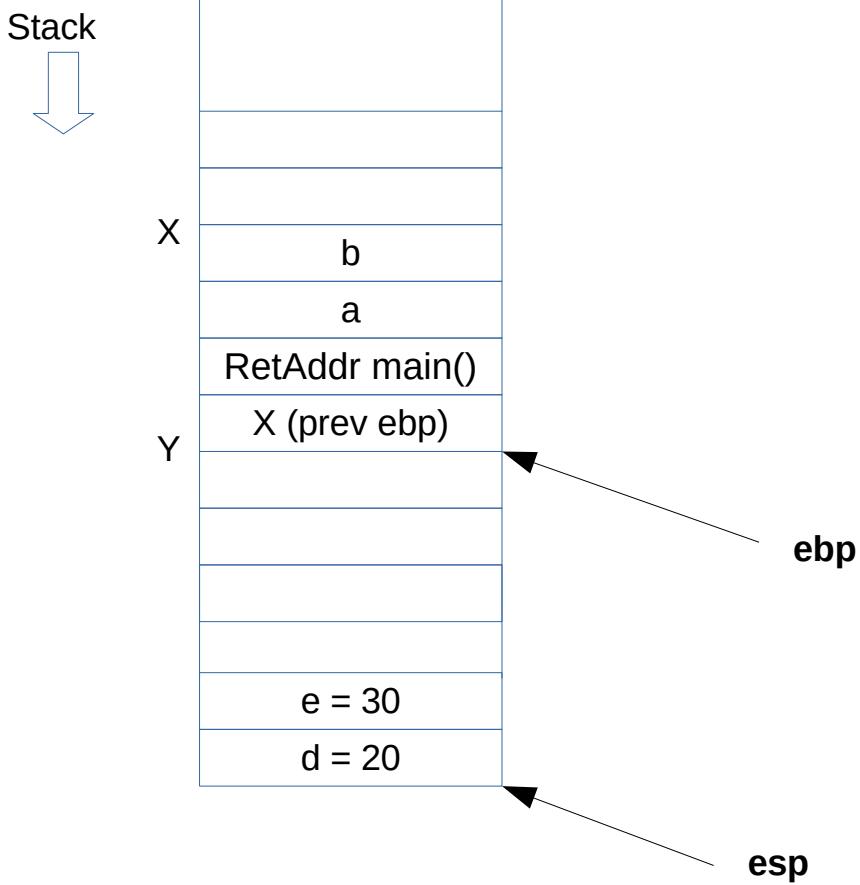
```

int mult(int a, int b) {
    int c, d = 20, e = 30, f;
    f = add(d, e);
    c = a * b + f;
    return c;
}
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}
  
```

### mult:

```

pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $20, -24(%ebp)
movl $30, -20(%ebp)
subl $8, %esp
pushl -20(%ebp)
pushl -24(%ebp)
call add
  
```



```

int mult(int a, int b) {
    int c, d = 20, e = 30, f;
    f = add(d, e);
    c = a * b + f;
    return c;
}
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}

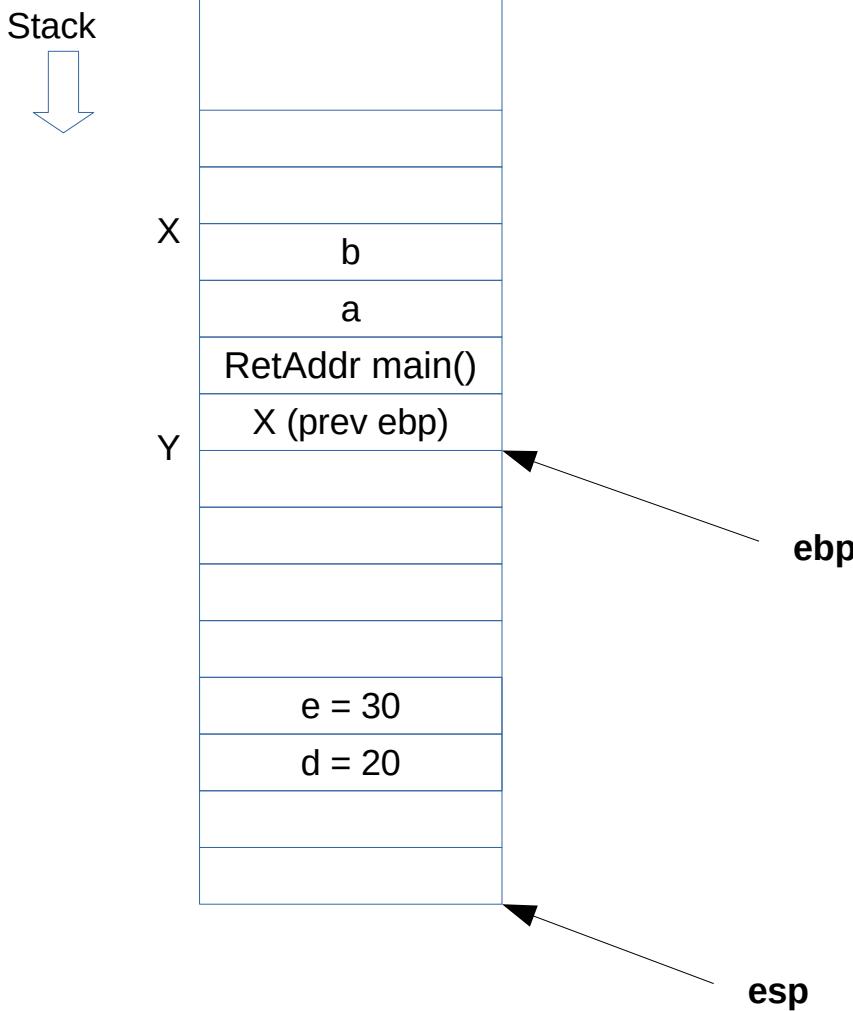
```

### mult:

```

pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $20, -24(%ebp)
movl $30, -20(%ebp)
subl $8, %esp
pushl -20(%ebp)
pushl -24(%ebp)
call add

```



```

int mult(int a, int b) {
    int c, d = 20, e = 30, f;
    f = add(d, e);
    c = a * b + f;
    return c;
}
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}

```

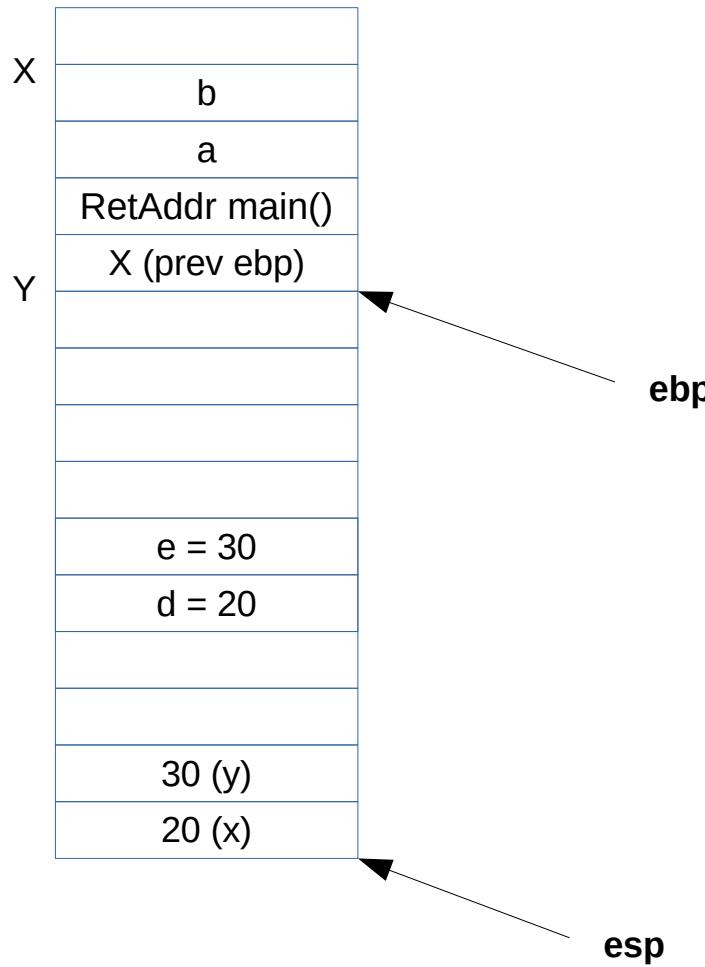
### mult:

```

pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $20, -24(%ebp)
movl $30, -20(%ebp)
subl $8, %esp
pushl -20(%ebp)
pushl -24(%ebp)
call add

```

Stack

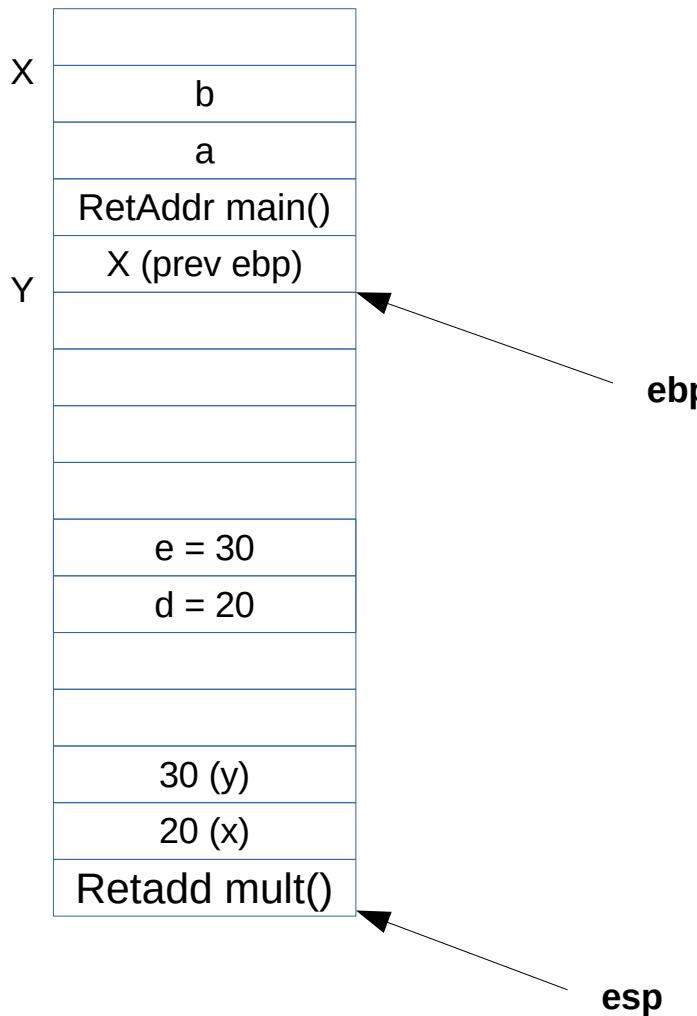


```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

mult:

```
pushl %ebp  
movl %esp, %ebp  
subl $24, %esp  
movl $20, -24(%ebp)  
movl $30, -20(%ebp)  
subl $8, %esp  
pushl -20(%ebp)  
pushl -24(%ebp)  
call add
```

Stack



```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

mult:

```
pushl %ebp  
movl %esp, %ebp  
subl $24, %esp  
movl $20, -24(%ebp)  
movl $30, -20(%ebp)  
subl $8, %esp  
pushl -20(%ebp)  
pushl -24(%ebp)  
call add
```

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
e = 30	
d = 20	
30 (y)	
20 (x)	
Retaddr mult()	
	Y(prev ebp)



ebp



esp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax  
leave  
ret
```

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
e = 30	
d = 20	
30 (y)	
20 (x)	
Retadd mult()	
Y(prev ebp)	

edx = 20  
eax = 30  
eax = eax + edx = 50

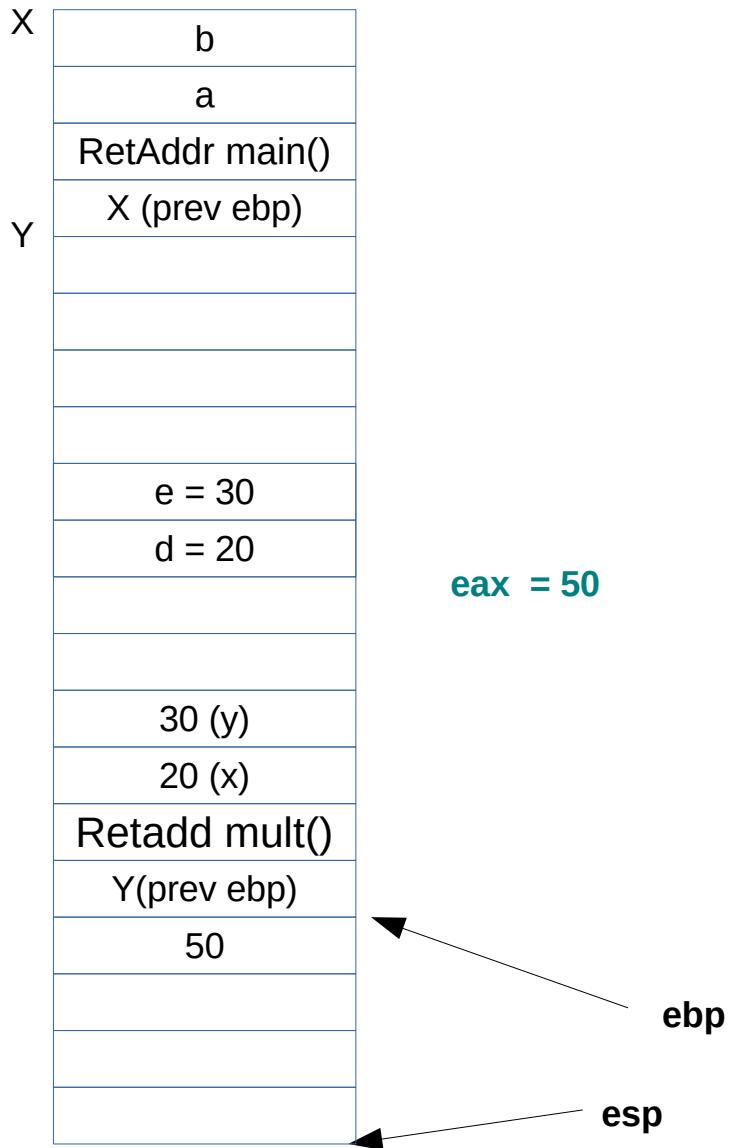
ebp  
esp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax  
leave  
ret
```

Stack



```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax  
leave  
ret
```

Some redundant code generated here.  
Before "leave". Result is in eax

Stack



X	b
	a
RetAddr main()	
X (prev ebp)	
e = 30	
d = 20	
30 (y)	
20 (x)	
Retadd mult()	
Y(prev ebp)	
50	

## leave: step 1

eax = 50

esp

ebp



```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax
```

**leave # # Set ESP to EBP,  
then pop EBP.**  
ret

Stack



X	b
	a
Y	RetAddr main()
	X (prev ebp)
	e = 30
	d = 20
	30 (y)
	20 (x)
	Retadd mult()
	Y(prev ebp)
	50

## leave: step 2

eax = 50

ebp

esp

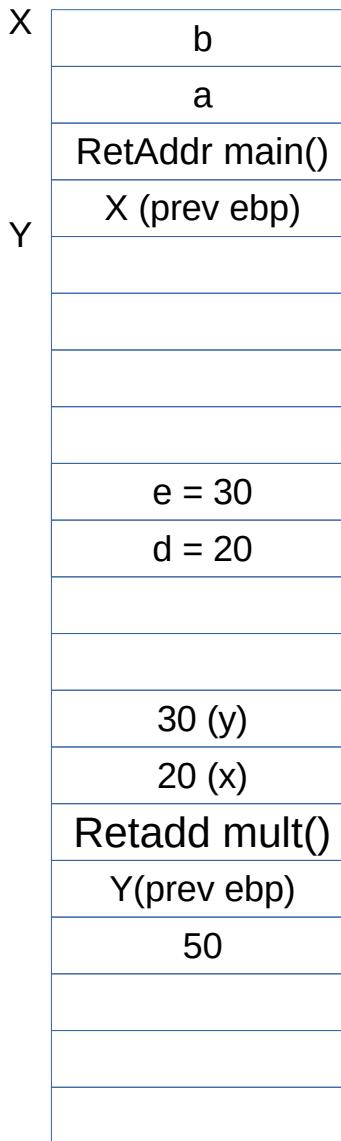
```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax
```

**leave # # Set ESP to EBP,  
then pop EBP.**  
**ret**

Stack



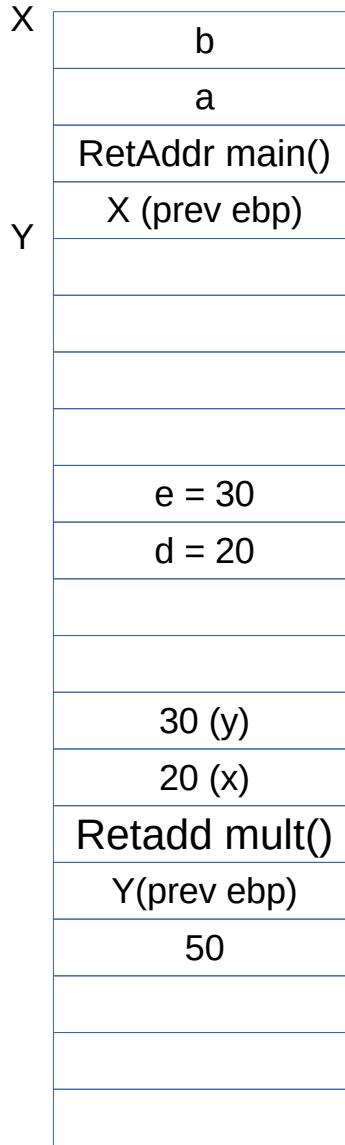
```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e); // here  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax
```

**leave #** # Set ESP to EBP,  
then pop EBP.  
**ret**

Stack



```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e); // here  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

Mult:

```
....  
call  add  
addl $16, %esp  
movl %eax, -16(%ebp)  
movl 8(%ebp), %eax  
imull 12(%ebp), %eax  
movl %eax, %edx  
movl -16(%ebp), %eax  
addl %edx, %eax  
movl %eax, -12(%ebp)  
movl -12(%ebp), %eax  
leave  
ret
```

Stack



X	b
	a
Y	RetAddr main()
	X (prev ebp)
	<b>f = 50 (eax)</b>
	e = 30
	d = 20
	30 (y)
	20 (x)
Y	Retadd mult()
	Y(prev ebp)
	50

**eax = 50**

**esp**

**ebp**

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

Mult:

```
....  
call  add  
addl $16, %esp  
movl %eax, -16(%ebp)  
movl 8(%ebp), %eax  
imull 12(%ebp), %eax  
movl %eax, %edx  
movl -16(%ebp), %eax  
addl %edx, %eax  
movl %eax, -12(%ebp)  
movl -12(%ebp), %eax  
leave  
ret
```

Stack



X	b
	a
Y	RetAddr main()
	X (prev ebp)
	<b>f = 50</b>
	e = 30
	d = 20
	30 (y)
	20 (x)
Y	Retadd mult()
	Y(prev ebp)
	50

eax = a  
eax = eax \* b  
edx = eax  
eax = f  
eax = edx + eax  
*// eax = a\*b + f*

ebp  
esp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

Mult:

```
....  
call  add  
addl $16, %esp  
movl %eax, -16(%ebp)  
movl 8(%ebp), %eax  
imull 12(%ebp), %eax  
movl %eax, %edx  
movl -16(%ebp), %eax  
addl %edx, %eax  
movl %eax, -12(%ebp)  
movl -12(%ebp), %eax  
leave  
ret
```

Stack



X	b
	a
Y	RetAddr main()
	X (prev ebp)
	c = eax
	f = 50
	e = 30
	d = 20
	30 (y)
	20 (x)
	Retaddr mult()
	Y(prev ebp)
	50

// eax = a\*b + f

Again some redundant code

ebp  
esp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

Mult:

```
....  
call  add  
addl $16, %esp  
movl %eax, -16(%ebp)  
movl 8(%ebp), %eax  
imull 12(%ebp), %eax  
movl %eax, %edx  
movl -16(%ebp), %eax  
addl %edx, %eax  
movl %eax, -12(%ebp)  
movl -12(%ebp), %eax  
leave  
ret
```

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
	<b>c = eax</b>
	<b>f = 50</b>
	e = 30
	d = 20
	30 (y)
	20 (x)
	Retadd mult()
	Y(prev ebp)
	50

After leave  
// eax = a\*b + f

ebp

esp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

Mult:

```
....  
call  add  
addl $16, %esp  
movl %eax, -16(%ebp)  
movl 8(%ebp), %eax  
imull 12(%ebp), %eax  
movl %eax, %edx  
movl -16(%ebp), %eax  
addl %edx, %eax  
movl %eax, -12(%ebp)  
movl -12(%ebp), %eax  
leave  
ret
```

# Lessons

- **Calling function (caller)**
  - Pushes arguments on stack , copies values
- **On call**
  - Return IP is pushed
- **Initially in called function (callee)**
  - Old ebp is pushed
  - ebp = stack
  - Stack is decremented to make space for local variables

# Lessons

- **Before Return**
  - Ensure that result is in ‘eax’
- **On Return**
  - stack = ebp
  - Pop ebp (ebp = old ebp)
- **On ‘ret’**
  - Pop ‘return IP’ and go back in old function

# Lessons

- **This was a demonstration for a**
  - User program, compiled with GCC, On Linux
  - Followed the conventions we discussed earlier
- **Applicable to**
  - C programs which work using LIFO function calls
- **Compiler can't be used to generate code using this mechanism for**
  - Functions like fork(), exec(), scheduler(), etc.
  - Boot code of OS

# **Memory Management Basics**

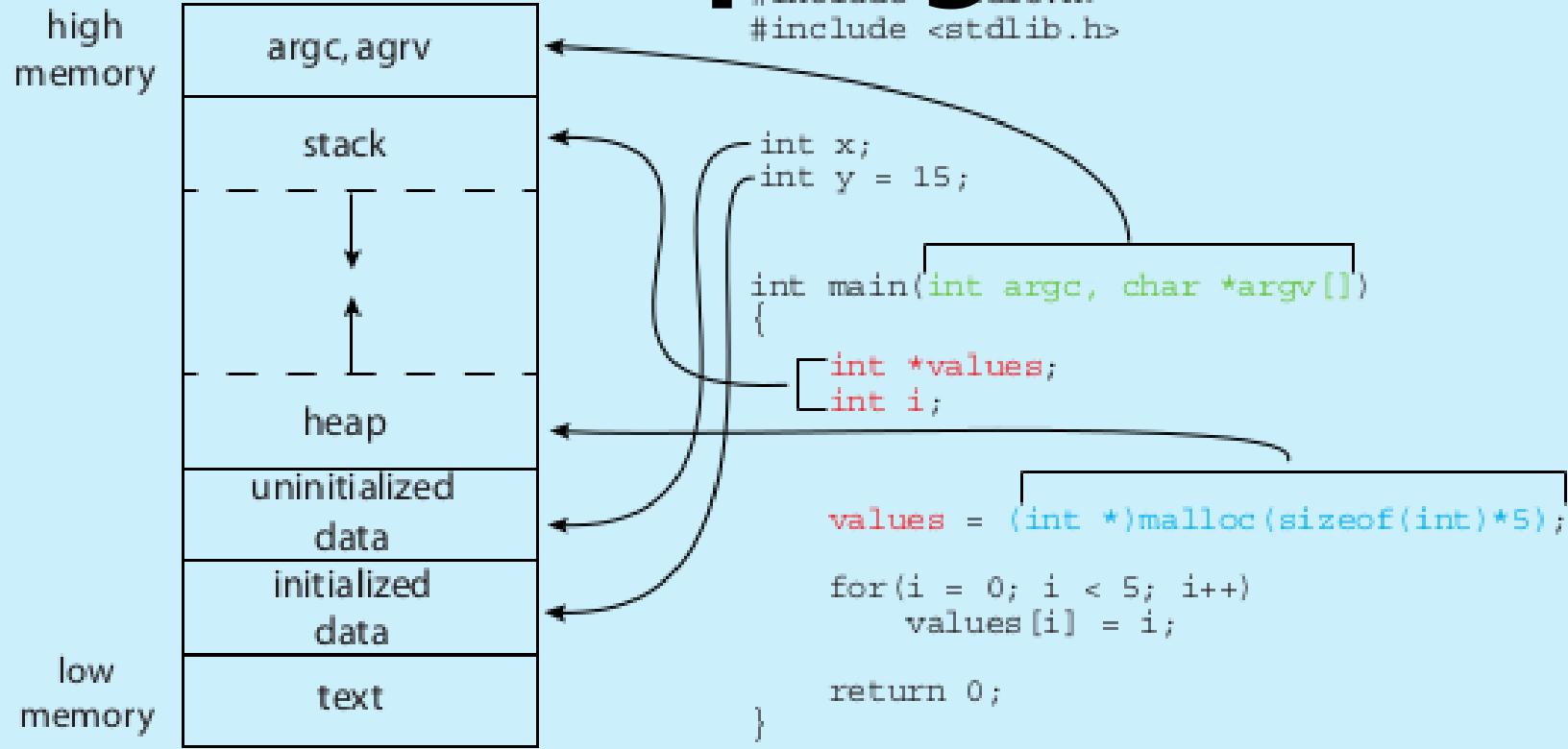
# Summary

- Understanding how the processor architecture drives the memory management features of OS and system programs (compilers, linkers)
- Understanding how different hardware designs lead to different memory management schemes by operating systems

# Addresses issued by CPU

- During the entire ‘on’ time of the CPU
  - Addresses are “issued” by the CPU on address bus
  - One address to fetch instruction from location specified by PC
  - Zero or more addresses depending on instruction
    - e.g. mov \$0x300, r1 # move contents of address 0x300 to r1 --> one extra address issued on address bus

# Memory layout of a C program



This “layout” shows

- (a) which parts of a C program occupy memory, when the program is running
- (b) A typical conceptual layout assumed by many compilers for calculating addresses in the generated machine code. This does not mean that other layouts are not possible

\$ size /bin/ls

text	data	bss	dec	hex	filename
128069	4688	4824	137581	2196d	/bin/ls

# Terminology

- Text
  - The machine code of the program : machine code for functions
- Data
  - Initialized global variables
  - Do not get confused with the generic word “data” in English
- BSS
  - Uninitialized global variables
- Heap
  - Region from where malloc() gets memory
- Stack
  - Region from where the local variables and formal parameters are allocated space

# Desired from a multi-tasking system

- Multiple processes in RAM at the same time (multi-programming)
- Processes should not be able to see/touch each other's code, data (globals), stack, heap, etc.
- Further advanced requirements
  - Process could reside anywhere in RAM
  - Process need not be continuous in RAM
  - Parts of process could be moved anywhere in RAM

# Different ‘times’

- Different actions related to memory management for a program are taken at different times. So let's know the different ‘times’
- Compile time
  - When compiler is compiling your C code
- Load time
  - When you execute “./myprogram” and it's getting loaded in RAM by loader i.e. exec()
- Run time
  - When the process is alive, and getting scheduled by the OS

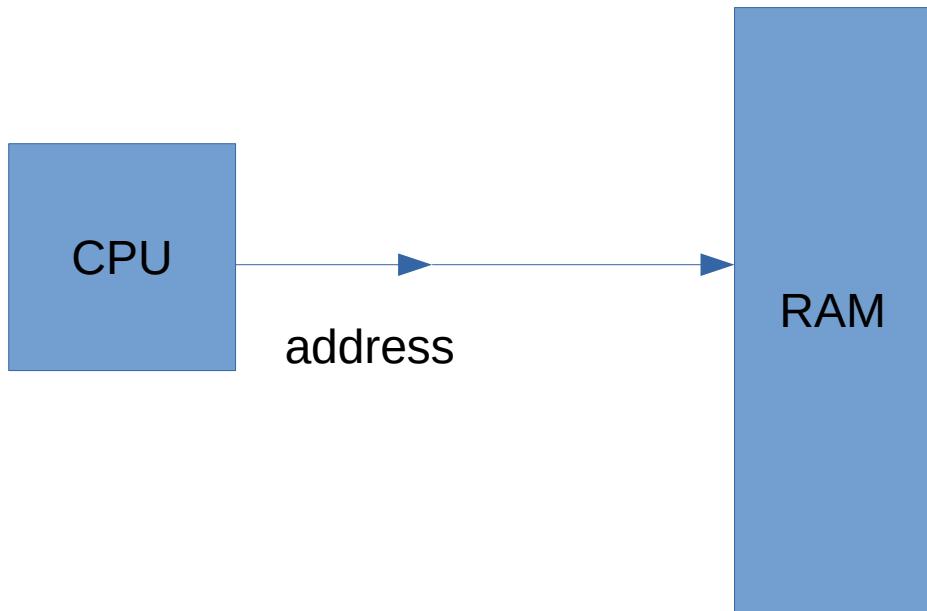
# The sequence

- Do not forget this
  - Machine code is typically generated by compiler
  - This machine code is put in RAM by the Loader (part of kernel) , that is exec() , when it's requested to run that program
  - The CPU's PIPELINE will issue addresses on address bus as seen in the executable file
- So question arises
  - How does compiler put in addresses (for code, data, bss, local-variables, etc) in the executable file ?

# Different types of Address binding

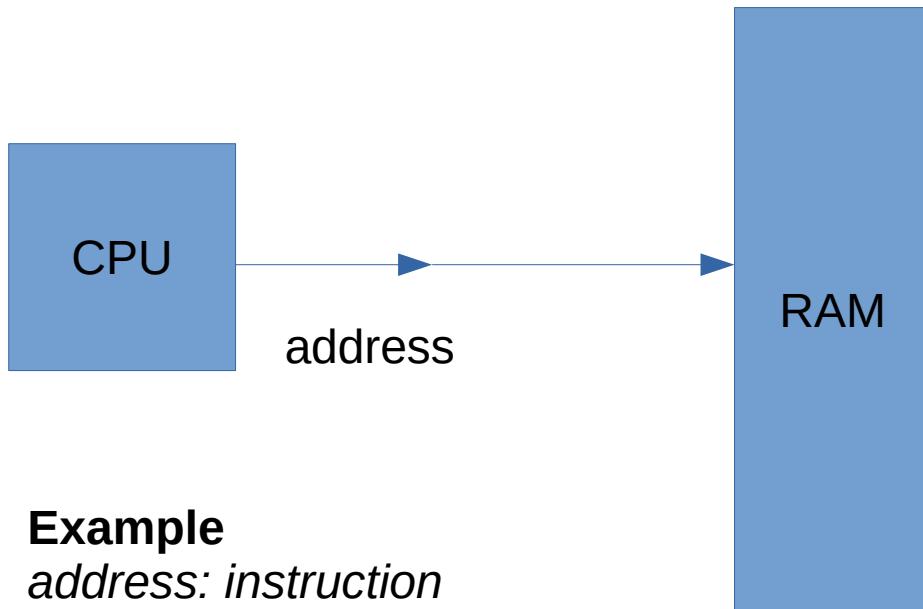
- Compile time address binding
    - Address of code/variables is fixed by compiler
    - Very rigid scheme
    - Location of process in RAM can not be changed ! Non-relocatable code.
  - Load time address binding
    - Address of code/variables is fixed by loader
    - Location of process in RAM is decided at load time, but can't be changed later
    - Flexible scheme, relocatable code
  - Run time address binding
    - Address of code/variables is fixed at the time of executing the code
    - Very flexible scheme , highly relocatable code
    - Location of process in RAM is decided at load time, but CAN be changed later also
- Which binding is actually used, is mandated by processor features + OS

# Simplest case



- Suppose the address issued by CPU reaches the RAM controller directly

# Simplest case



## Example

*address: instruction*

1000: mov \$0x300, r1

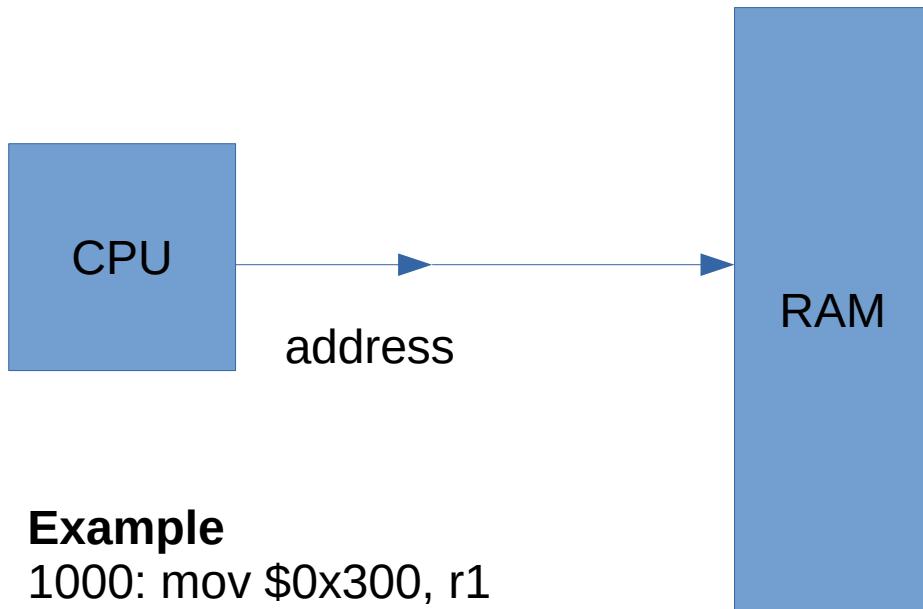
1004: add r1, -3

1008: jnz 1000

Sequence of addressed sent by CPU: 1000, 0x300, 1004, 1008, 1000, 0x300, ...

- How does this impact the compiler and OS ?
- When a process is running the addresses issued by it, will reach the RAM directly
- So exact addresses of globals, addresses in “jmp” and “call” must be part the machine instructions generated by compiler
  - How will the compiler know the addresses, at “compile time” ?

# Simplest case

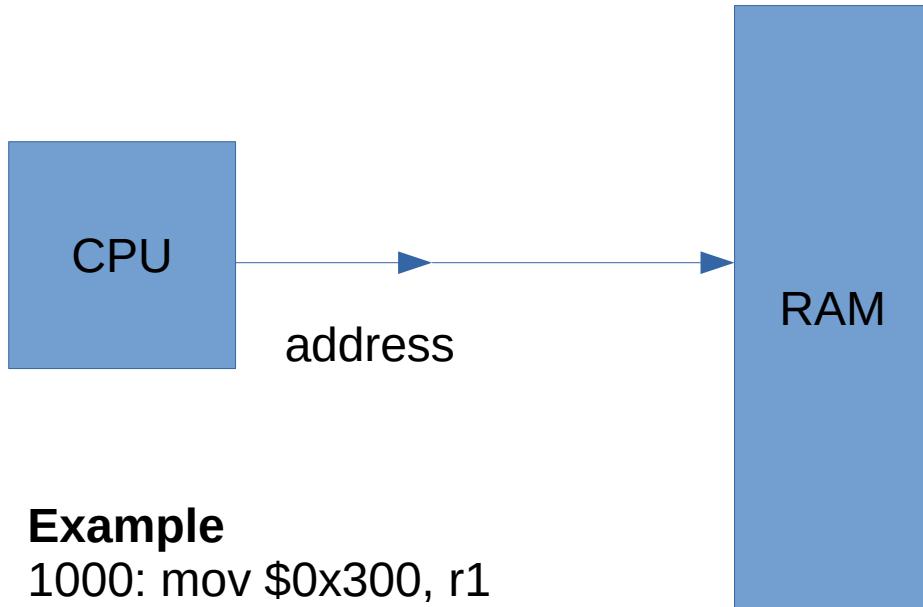


## Example

```
1000: mov $0x300, r1  
1004: add r1, -3  
1008: jnz 1000
```

- Solution: compiler assumes some fixed addresses for globals, code, etc.
- OS loads the program exactly at the same addresses specified in the executable file. **Non-relocatable code.**
- Now program can execute properly.

# Simplest case



## Example

```
1000: mov $0x300, r1  
1004: add r1, -3  
1008: jnz 1000
```

- Problem with this solution
  - Programs once loaded in RAM must stay there, can't be moved
  - What about 2 programs?
    - Compilers being “programs”, will make same assumptions and are likely to generate same/overlapping addresses for two different programs
    - Hence only one program can be in memory at a time !
    - No need to check for any memory boundary violations – all memory belongs to one process
- Example: DOS

# Base/Relocation + Limit scheme

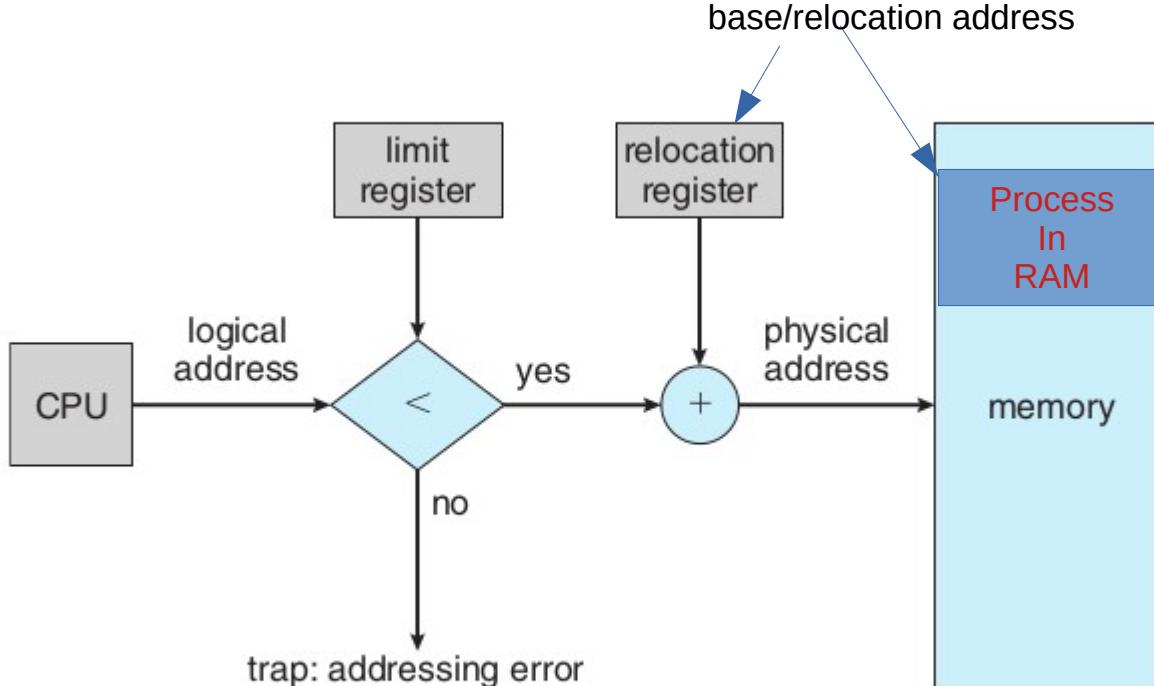
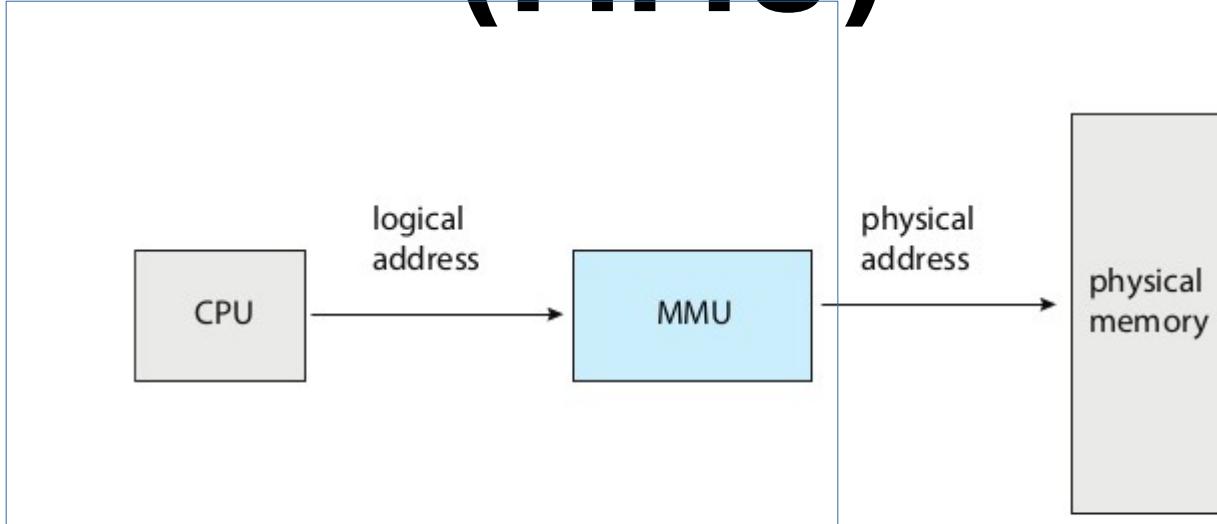


Figure 9.6 Hardware support for relocation and limit registers.

- Base and Limit are two registers inside CPU's Memory Management Unit
- 'base' is added to the address generated by CPU
- The result is compared with base+limit and if less passed to memory, else hardware interrupt is raised

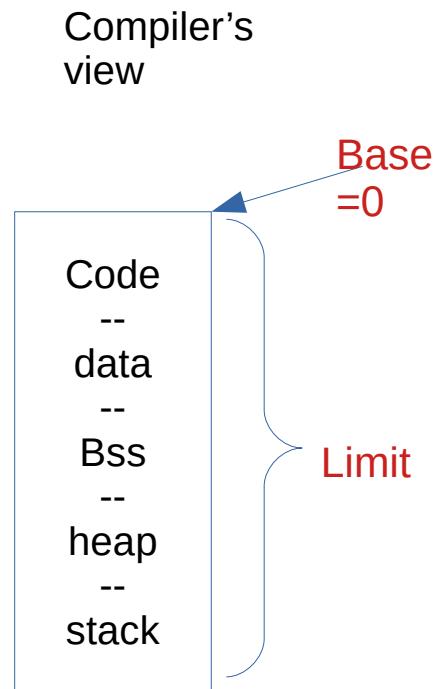
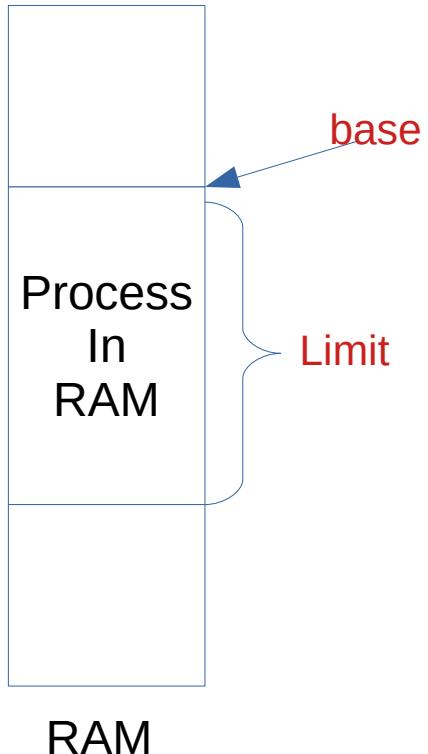
# Memory Management Unit (MMU)



**Figure 9.4** Memory management unit (MMU).

- Is part of the CPU chip, acts on every memory address issue by execution unit of the CPU
- In the scheme just discussed, the base, limit calculation parts are part of MMU

# Base/Relocation + Limit scheme



- Compiler's work
  - Assume that the process is one continuous chunk in memory, with a size limit
  - Assume that the process starts at address zero (!) and calculate addresses for globals, code, etc. And accordingly generate machine code

# Base/Relocation + Limit scheme

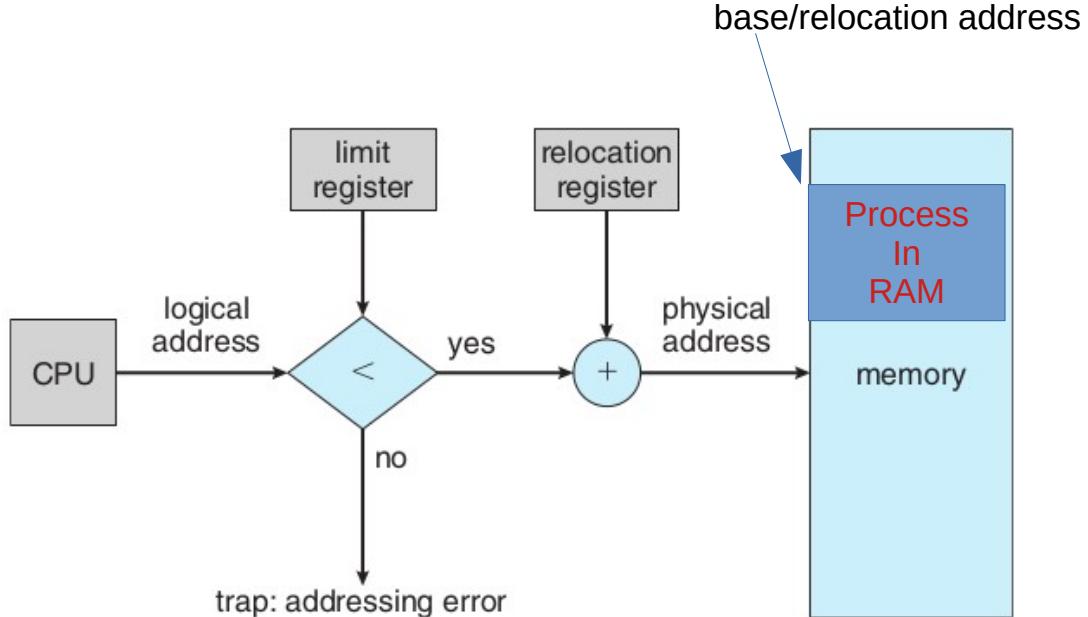


Figure 9.6 Hardware support for relocation and limit registers.

- Loader's job

- While loading the process in memory - must load as one continuous segment
  - Find an empty slot for this!
- Remember the 'base' and 'limit' values in the struct proc (memory management info in PCB)
  - Setup the limit to be the size of the process as set by compiler in the executable file.
- Call the scheduler()

- Scheduler()

- Will replace the base-limit registers with values obtained from PCB before scheduling the process

# Base/Relocation + Limit scheme

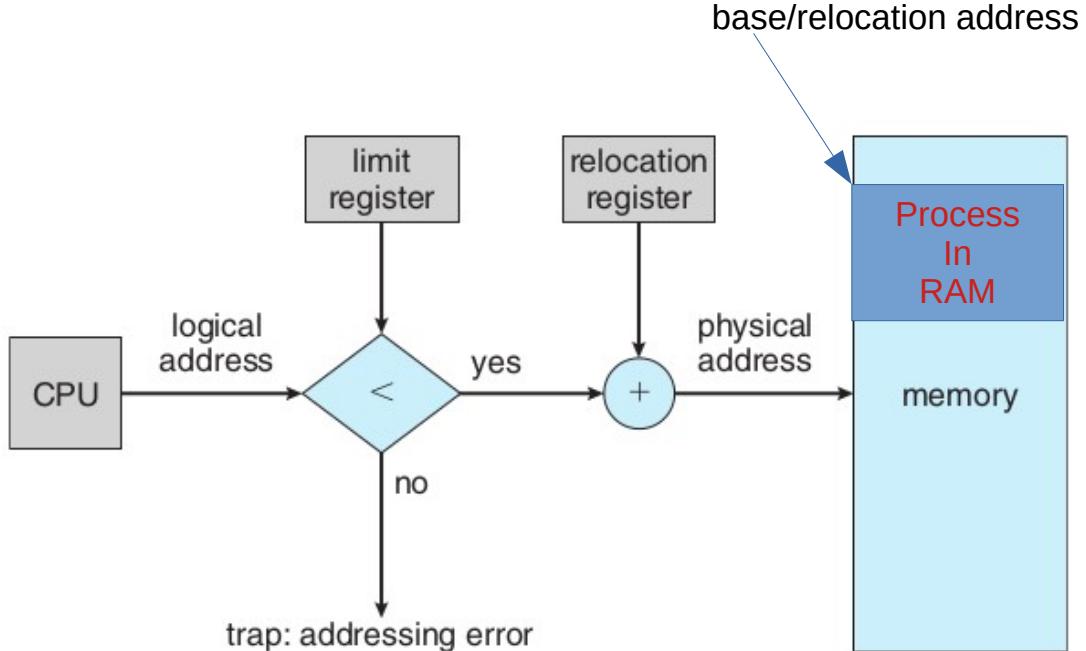
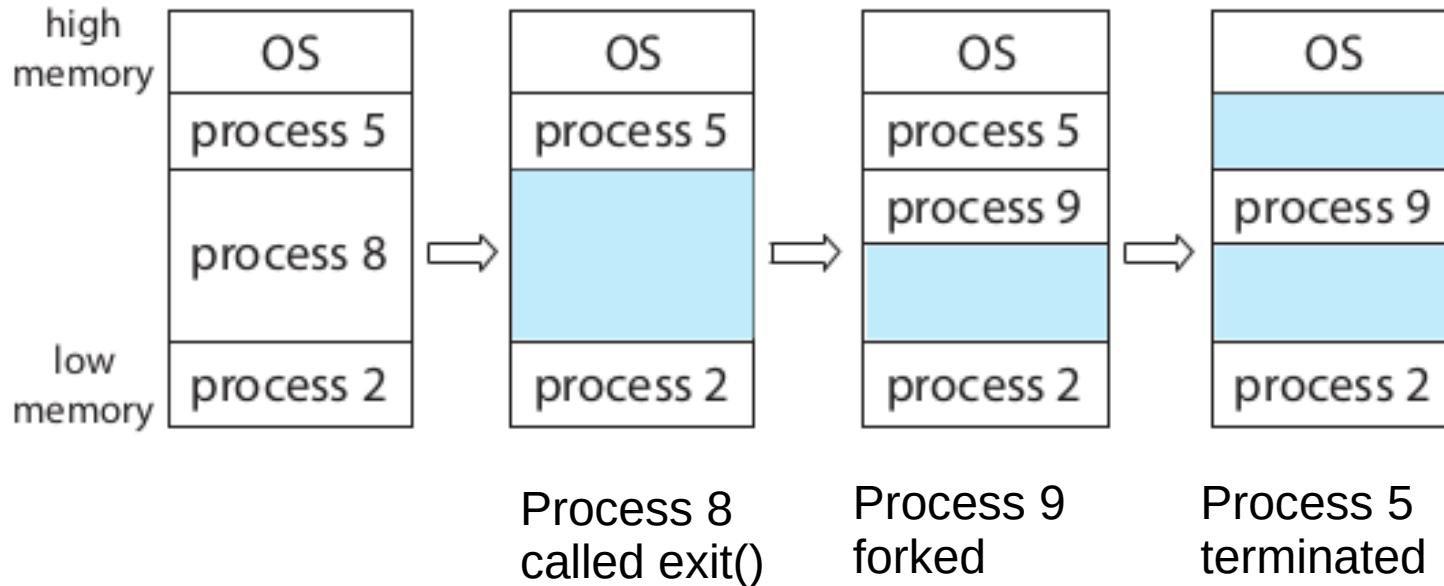


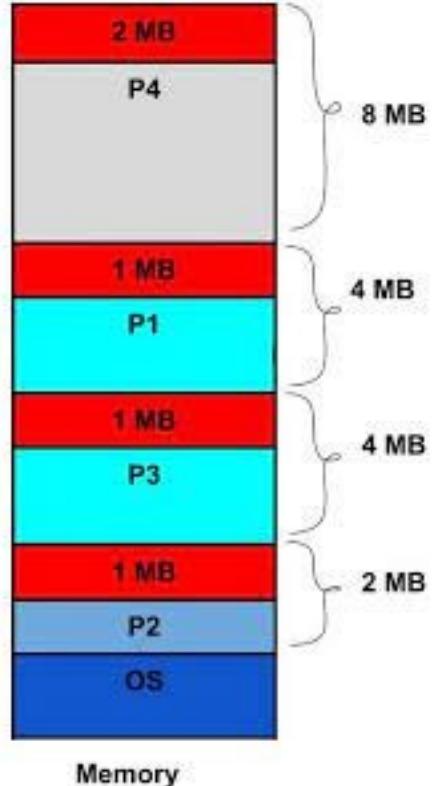
Figure 9.6 Hardware support for relocation and limit registers.

- Combined effect
  - “**Relocatable code**” - the process can go anywhere in RAM at the time of loading
  - Some memory violations can be detected - a memory access beyond base+limit will raise interrupt, thus running OS in turn, which may take action against the process

# Example scenario of memory in base+limit scheme



# Continuous memory management and external fragmentation problem



Free chunks: 2 MB, 1MB, 1MB, 1MB

Total 5 MB is available!

Can we create a process of size 3MB?

No! - 3 MB not continuous!

# External Fragmentation

- OS needs to find a continuous free chunk of memory that fits the size of the “segment”
  - If not available, your exec() can fail due to lack of memory
- Suppose 50k is needed
  - Possible that among 3 free chunks total 100K may be available, but no single chunk of 50k!
  - **External fragmentation**
- Solution to external fragmentation: **compaction** – move the chunks around and make a continuous big chunk available. Time consuming, tricky.

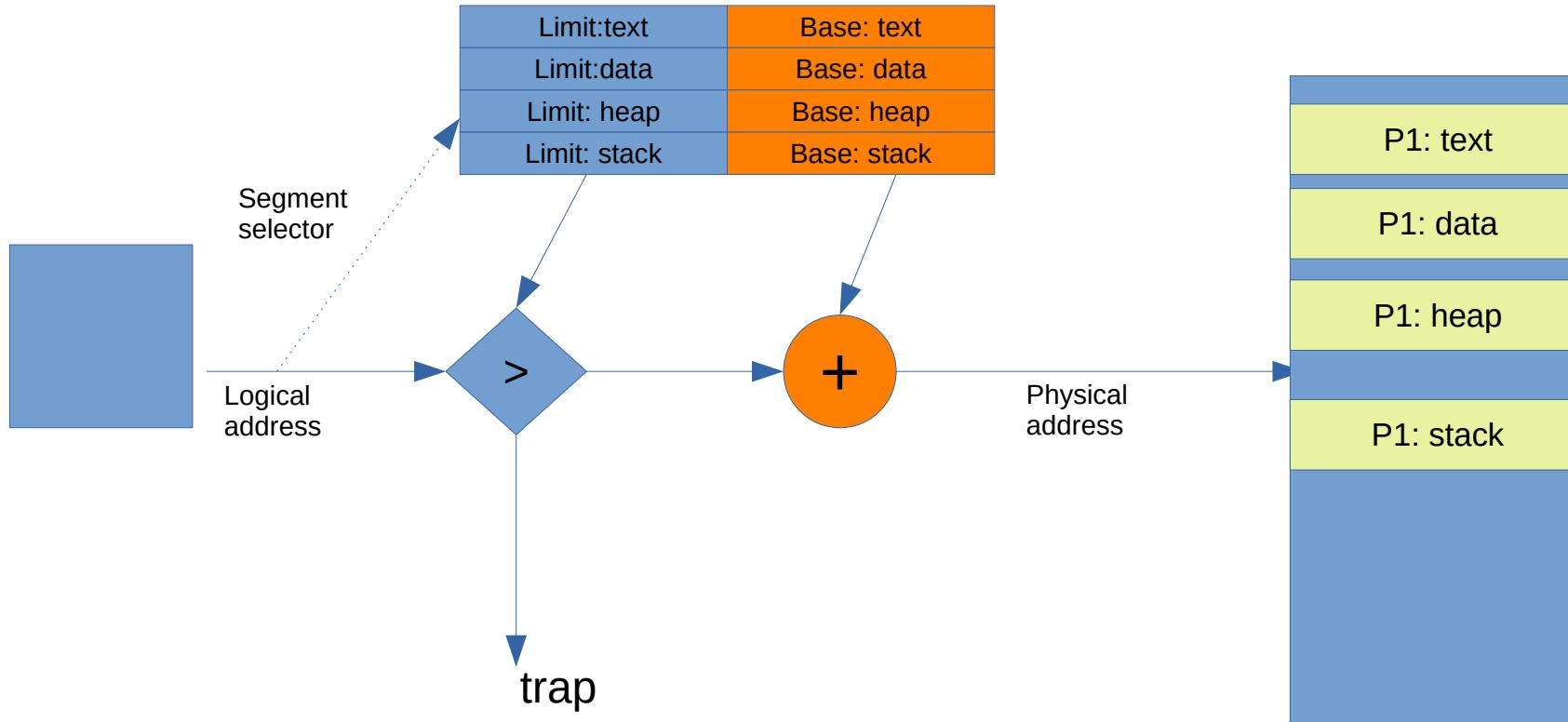
It should be possible to have relocatable code  
even with “simplest case”

By doing extra work during “loading”.

How?

(Ans: loader replaces all addresses in the code!  
That is quite a lot of work, and challenging too)

# Next scheme: Multiple base + limit pairs



# Specifying “selector”

- Explicitely
  - Mov ES: 300, \$30
  - Here selector is Extra Segment Register and Logical address (or offset) is 300
- Implicitely, like in x86
  - Mov 300, \$30
  - Here Data Segment register is “implicit” selector

# Next scheme: Segmentation

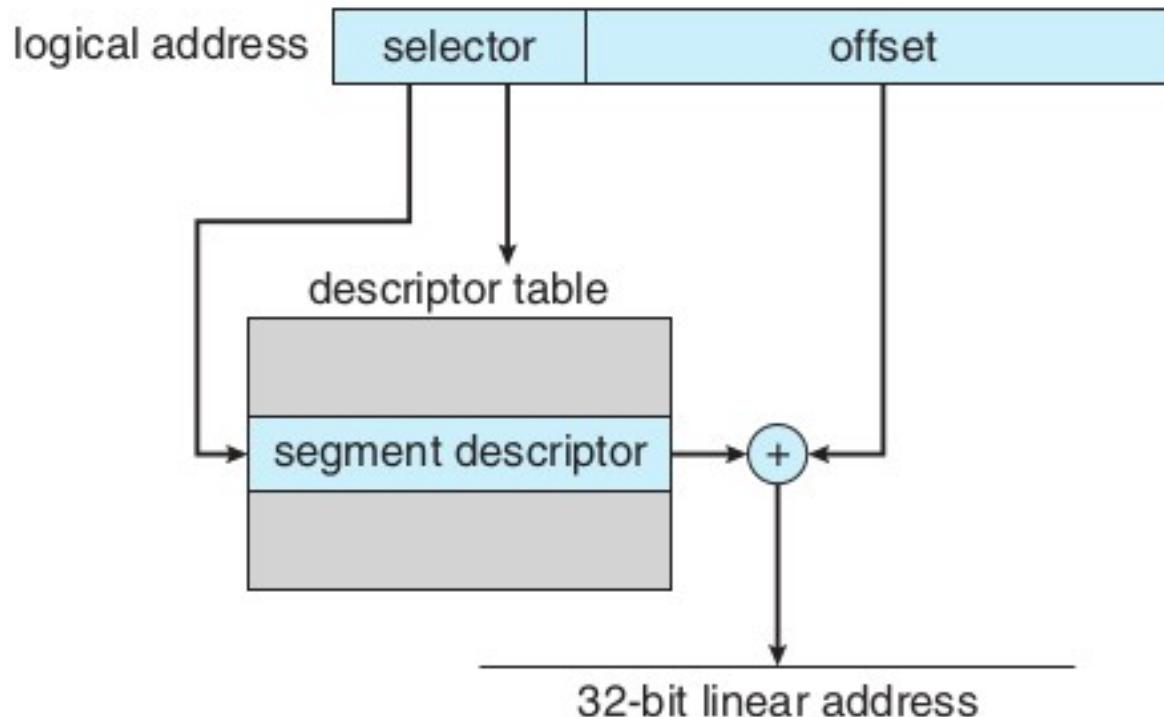
## Multiple base + limit pairs

- Multiple sets of base + limit registers
- Whenever an address is issued by execution unit of CPU, it will also include reference to some base register
  - And hence limit register paired to that base register will be used for error checking
- Compiler: can assume a separate chunk of memory for code, data, stack, heap, etc. And accordingly calculate addresses . Each “segment” starting at address 0.
- OS (Loader): will load the different ‘sections’ in different memory regions and accordingly set different ‘base’ registers

# Next scheme: Multiple base +limit pairs, with further indirection

- Base + limit pairs can also be stored in some memory location (not in registers). Then it's called **Segment Table**.
  - Question: how will the cpu know where it's in memory?
  - One CPU register to point to the location of table in memory . **Segment Table Base Register (STBR)**
    - X86 has two tables. Local Descriptor Table and Global Descriptor Table. Both are segment tables.
    - Accordingly it has LDTR and GDTR.
- Segment registers still in use, they give an index in this table
- This is x86 segmentation
  - Flexibility to have lot more “base+limits” in the array/table in memory

# Next scheme: Multiple base + limit pairs, with further indirection



Note:

"Selector" here is normally a segment register, like CS, DS

The machine code only has offset in it

The "Descriptor table" is a segmentation table

Figure 9.22 IA-32 segmentation.

# **Segmentation and External fragmentation**

- Does segmentation also suffer from external fragmentation?
  - Yes!

# How many segment tables?

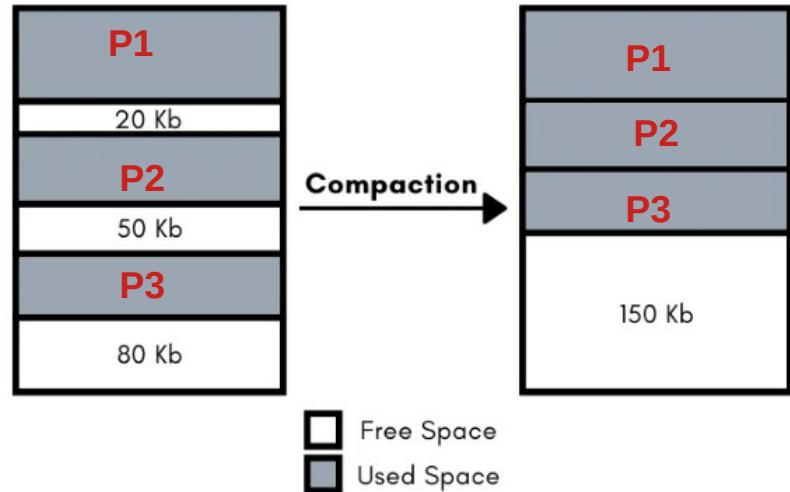
- One per process
- One for the kernel!
  - Remember: kernel code also undergoes translation in MMU!

# A point of confusion on “root” user

- “root” user has privileges in accessing files
- When “root” user runs an application, it runs in process-mode, not in kernel-mode!
  - That process can do “more” but , it’s not kernel mode code!
- Kernel is not run by “root” user!
  - Kernel is not run by any user

# Solution to external fragmentation

- Compaction !
- OS moves the process chunks in memory to make available continuous memory region
  - Then it must update the memory management information in PCB (e.g. base of the process) of each process
- Time consuming
- Possible only if the relocation+limit scheme of MMU is available



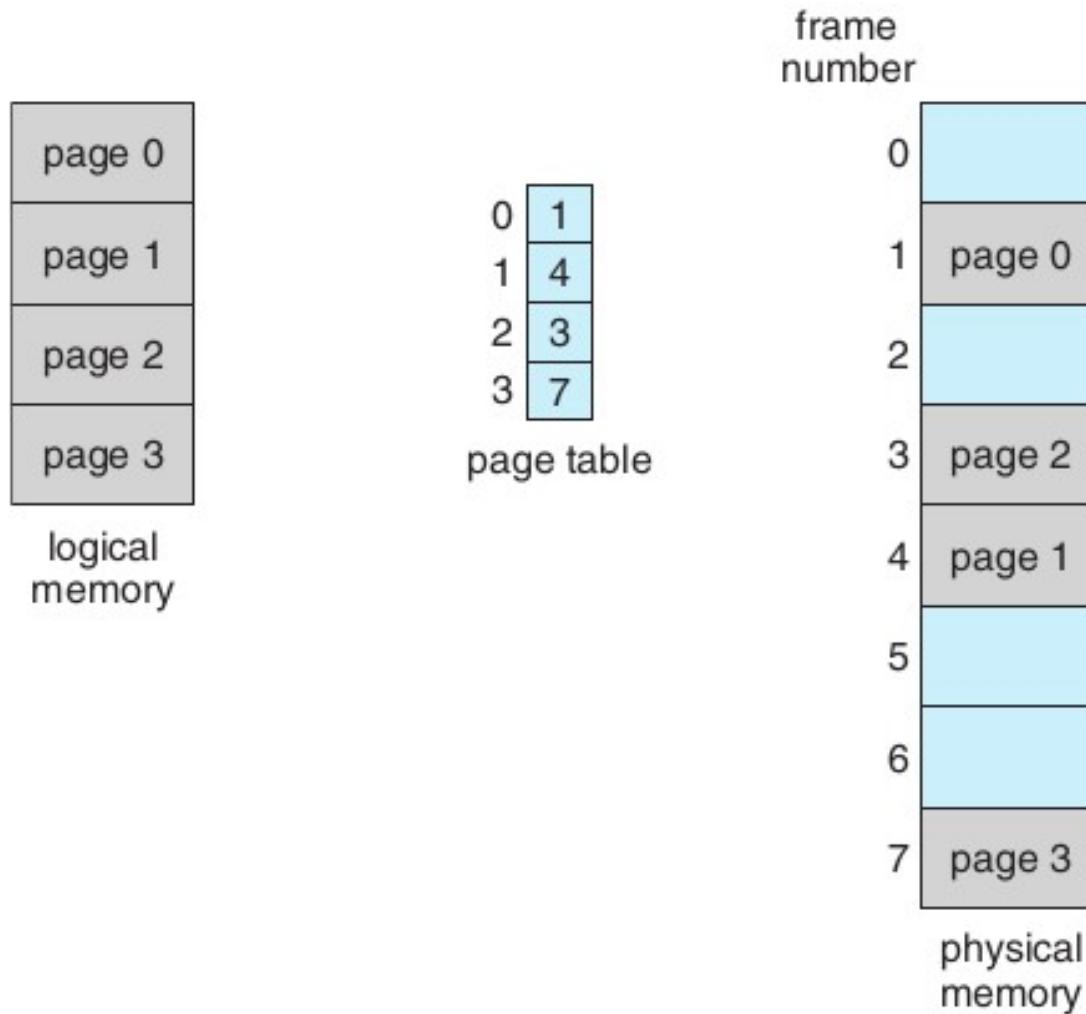
# Another solution to external fragmentation: Fixed size partitions

- Fixed partition scheme
- Memory is divided by OS into chunks of equal size: e.g., say, 50k
  - If total 1M memory, then 20 such chunks
- Allocate one or more chunks to a process, such that the total size is  $\geq$  the size of the process
  - E.g. if request is 50k, allocate 1 chunk
  - If request is 40k, still allocate 1 chunk
  - If request is 60k, then allocate 2 chunks
- Leads to internal fragmentation
  - space wasted in the case of 40k or 60k requests above



# Solving external fragmentation problem

- Process should not be continuous in memory!
  - The trouble is finding a big continuous chunk!
- Divide the continuous process image in smaller chunks (let's say 4k each) and locate the chunks anywhere in the physical memory
  - Need a way to map the *logical* memory addresses into *actual physical memory addresses*



**Figure 9.9** Paging model of logical and physical memory.

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

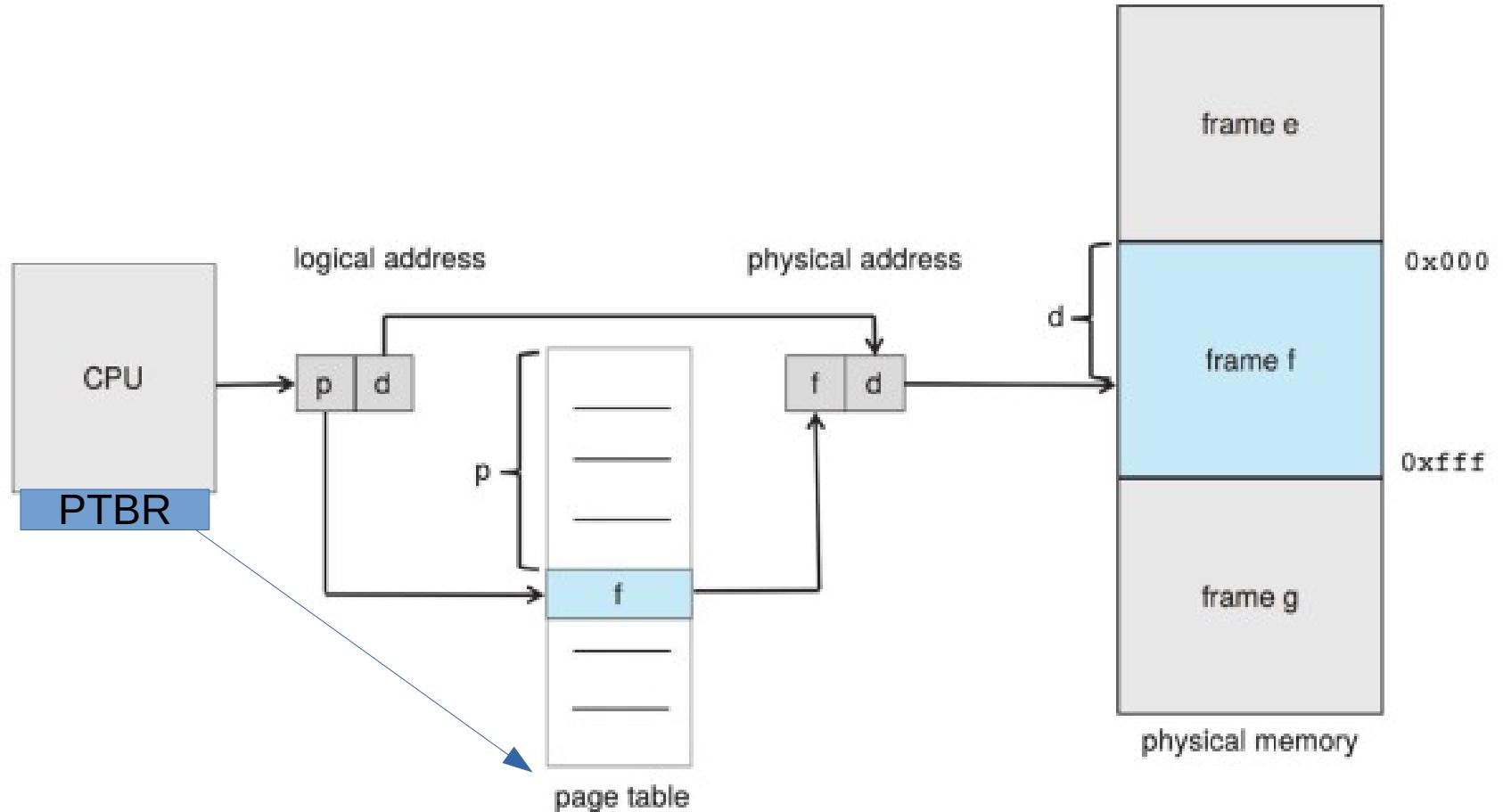
0	5
1	6
2	1
3	2

page table

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

physical memory

Figure 9.10 Paging example for a 32-byte memory with 4-byte pages.



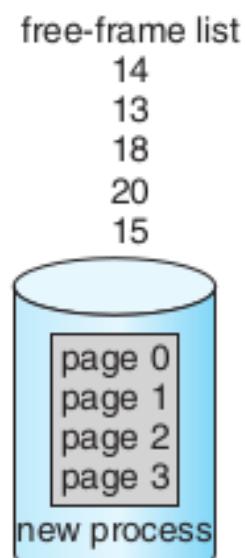
**Figure 9.8** Paging hardware.

# Paging

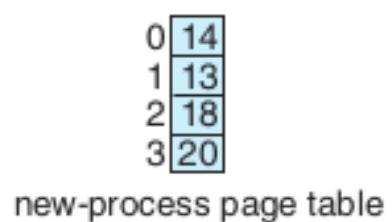
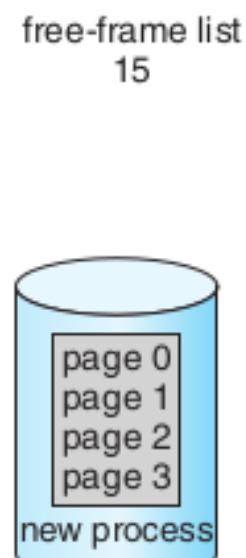
- Process is assumed to be composed of equally sized “pages” (e.g. 4k page)
- Actual memory is considered to be divided into page “frames”.
- CPU generated logical address is split into a page number and offset
- A Page Table Base Register (PTBR) inside CPU will give location of an in memory table called page table
- Page number used as offset in a table called page table, which gives the physical page frame number
- Frame number + offset are combined to get physical memory address

# Paging

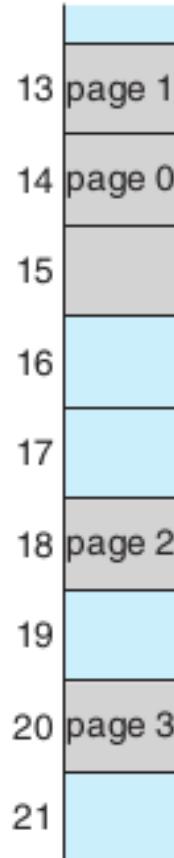
- Compiler: assume the process to be one continuous chunk of memory (!) . Generate addresses accordingly . Compiler still treats text,data,bss,... to be separate chunks, but in multiples of page-sizes.
- OS: at exec() time - allocate different frames to process, allocate a page table(!), setup the page table to map page numbers with frame numbers, setup the page table base register, start the process
- Now hardware will take care of all translations of logical addresses to physical addresses



(a)



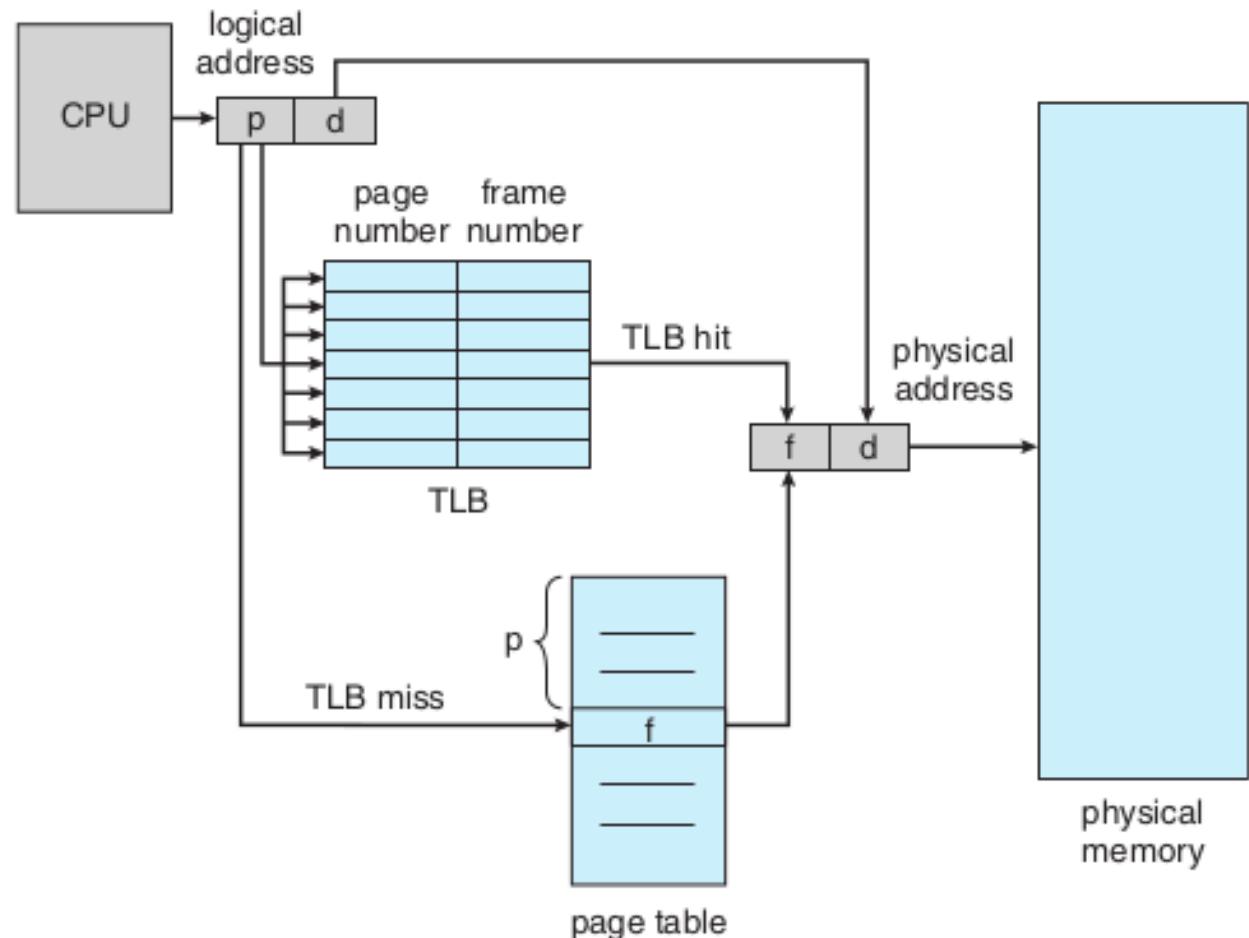
(b)



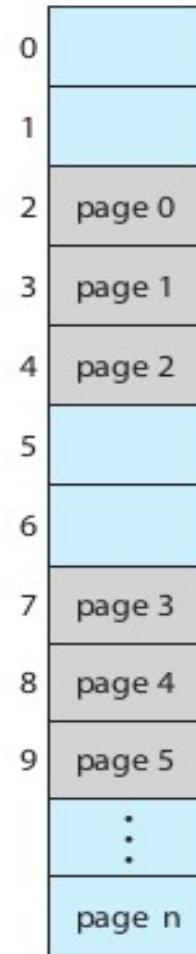
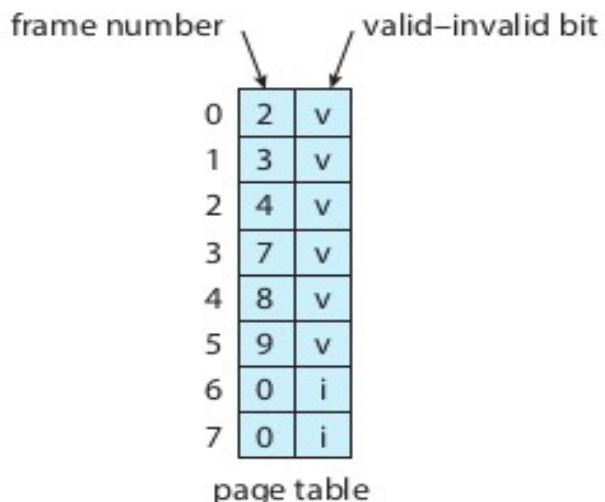
**Figure 9.11** Free frames (a) before allocation and (b) after allocation.

# Speeding up paging

- Translation Lookaside Buffer (TLB)
- Part of CPU hardware
- A cache of Page table entries
- Searched in parallel for a page number



12,287	page 5
10,468	page 4
	page 3
	page 2
	page 1
00000	page 0



# Memory protection with paging

Figure 9.13 Valid (v) or invalid (i) bit in a page table.

# Shared pages (e.g. library) with paging

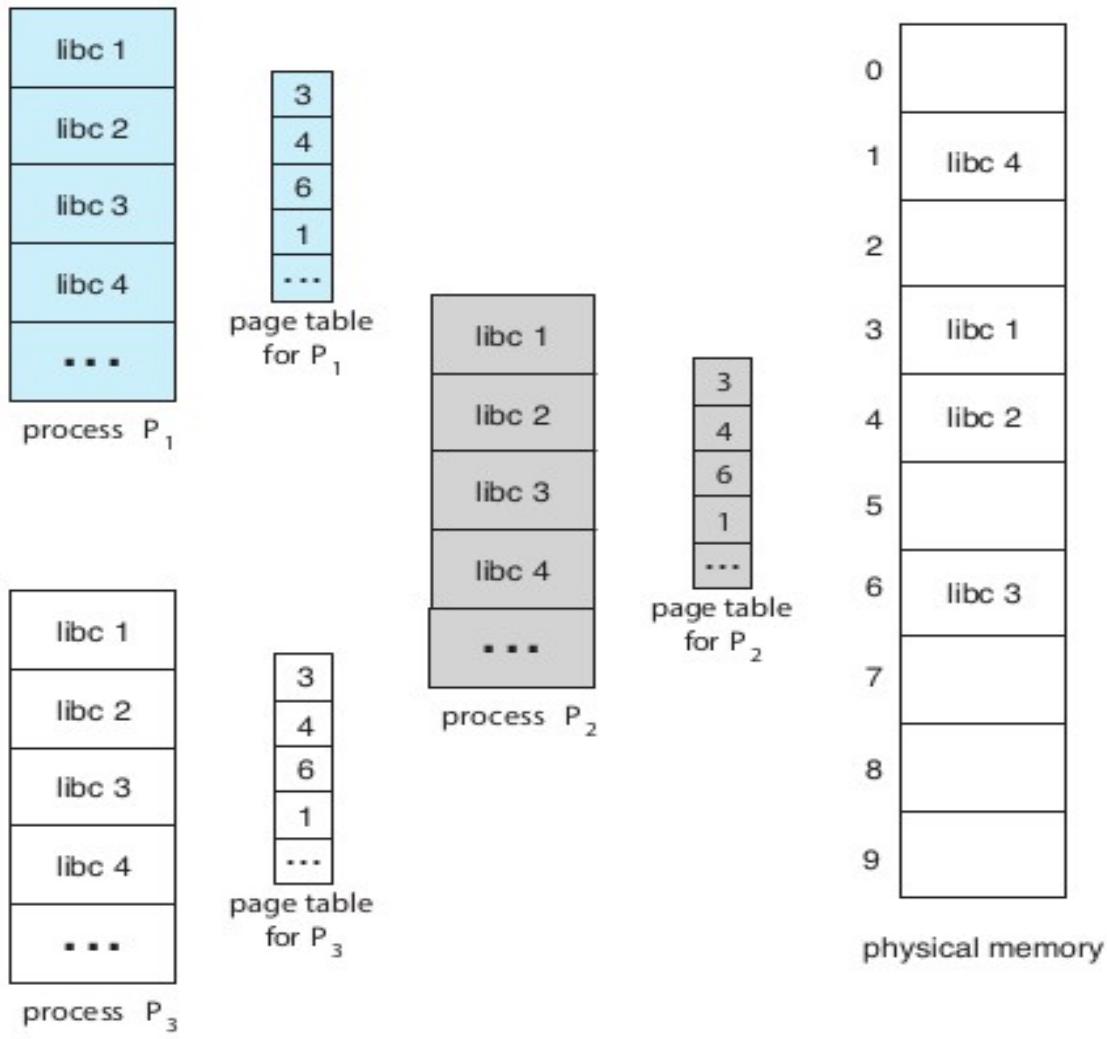


Figure 9.14 Sharing of standard C library in a paging environment.

# Paging: problem of large PT

- 64 bit address
- Suppose 20 bit offset
  - That means  $2^{20} = 1 \text{ MB}$  pages
  - 44 bit page number:  $2^{44}$  that is trillion sized page table!
  - Can't have that big continuous page table!

# Paging: problem of large PT

- 32 bit address
- Suppose 12 bit offset
  - That means  $2^{12} = 4$  KB pages
  - 20 bit page number:  $2^{20}$  that is a million entries
  - Can't always have that big continuous page table as well, for each process!

# Hierarchic al paging

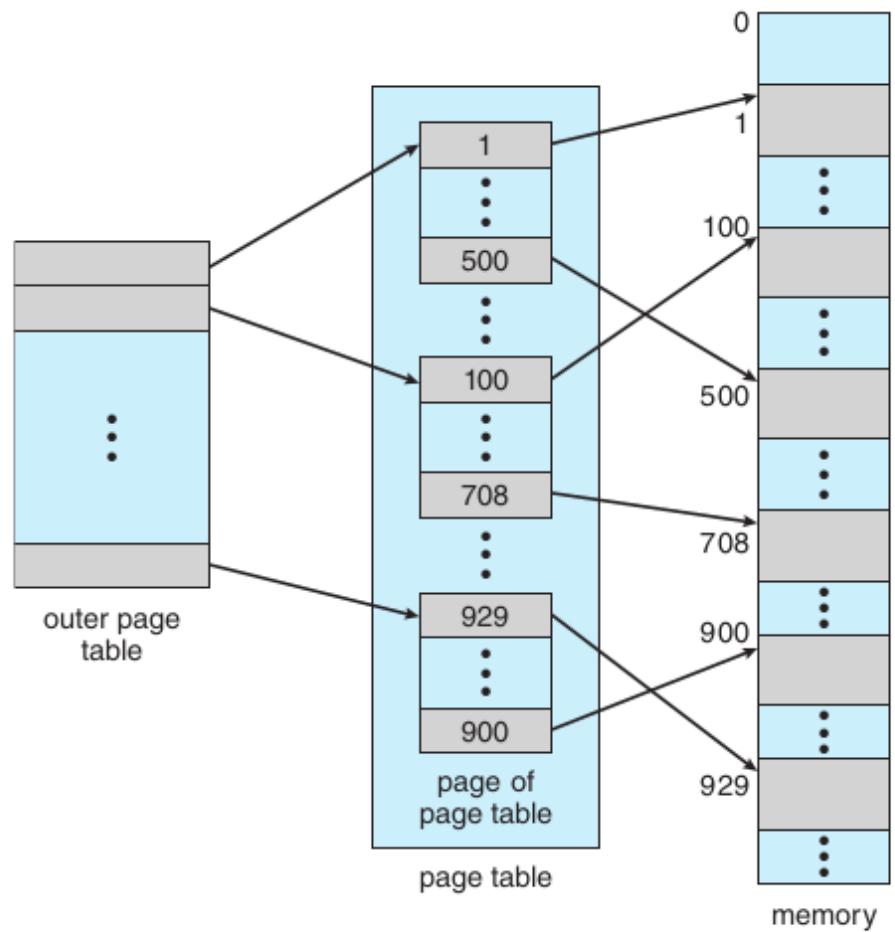
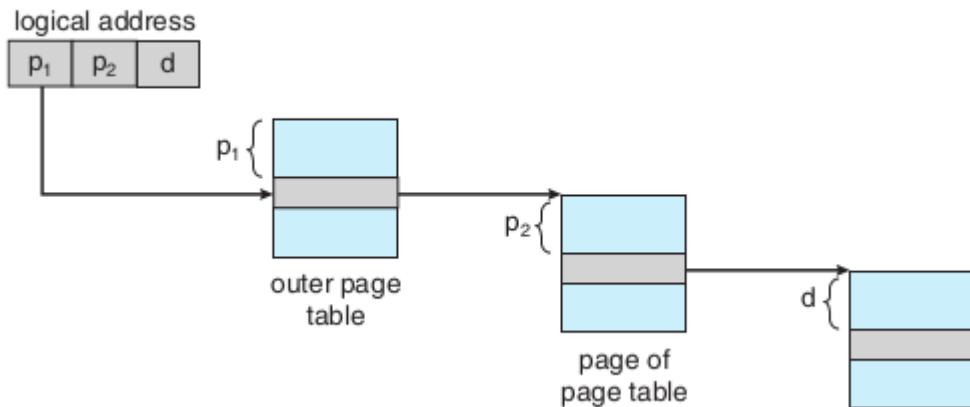


Figure 9.15 A two-level page-table scheme.

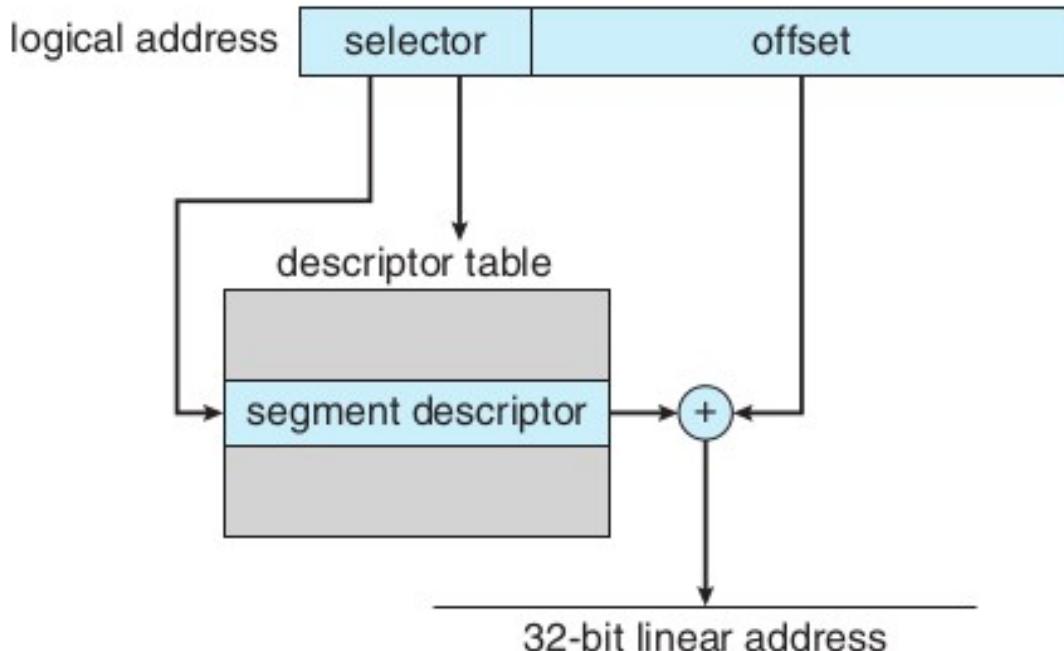
outer page	inner page	offset
$p_1$	10	12

# X86 memory management



**Figure 9.21** Logical to physical address translation in IA-32.

# Segmentation in x86



- The selector is automatically chosen using Code Segment (CS) register, or Data Segment (DS) register depending on which type of memory address is being fetched
- Descriptor table (that is a segmentation table) is in memory
- The location of Descriptor table (Global DT- GDT or Local DT – LDT) is given by a GDT-register i.e. GDTR or LDT-register i.e. LDTR

**Figure 9.22** IA-32 segmentation.

# Paging in x86

- Depending on a flag setup in CR3 register, either 4 MB or 4 KB pages can be enabled
- Page directory, page table are both in memory

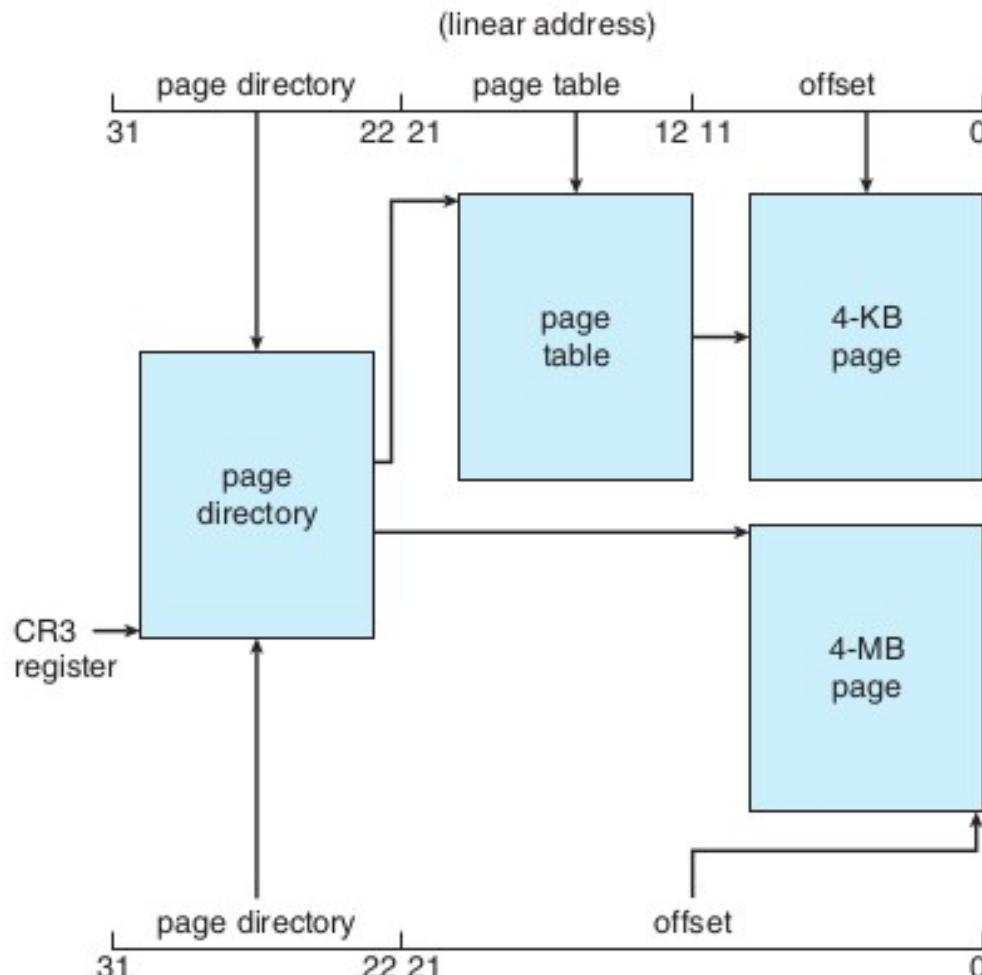
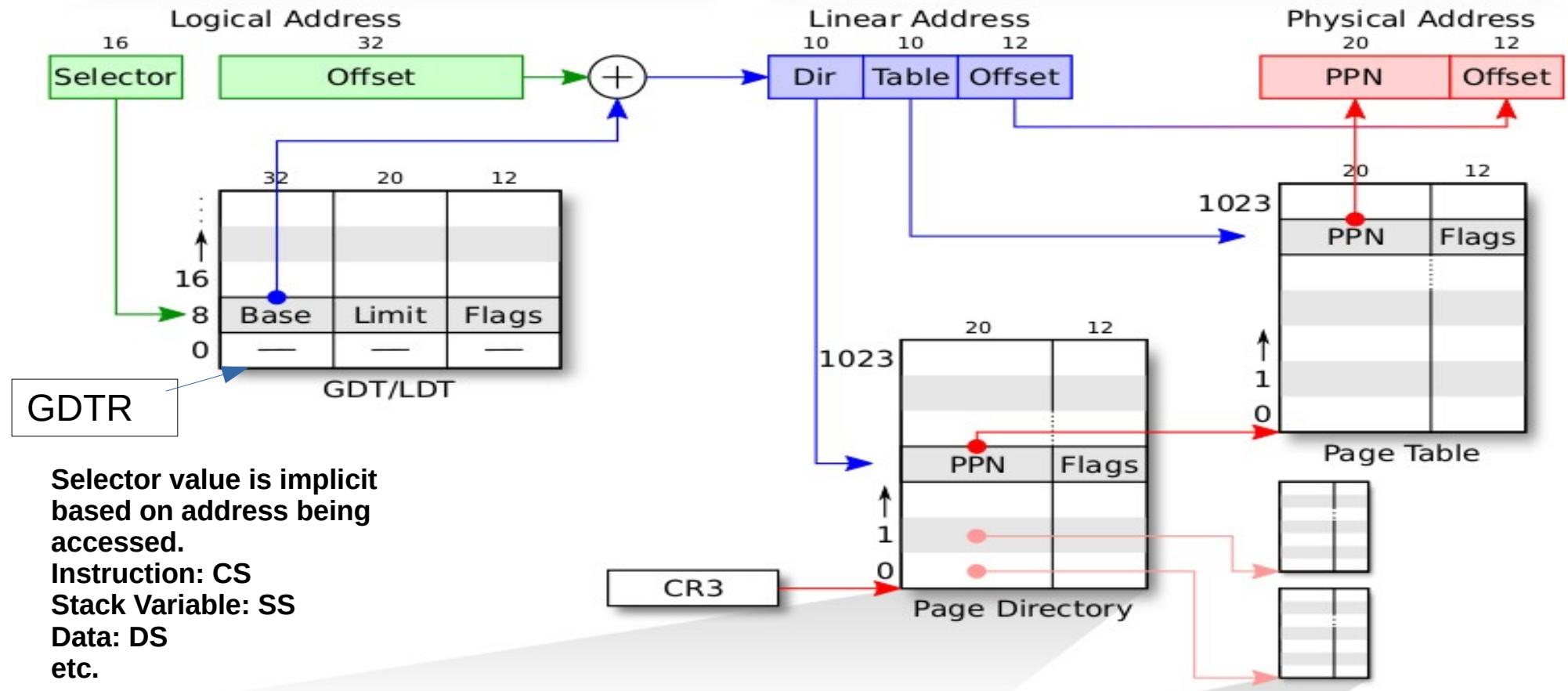


Figure 9.23 Paging in the IA-32 architecture.

# X86 Segmentation + Paging



# X86 Segmentation + Paging

- **Paging is optional, segmentation compulsory**
  - Setting flags in Control Registers (CR) enables this
- **Page Table, Page Directory, page - are all size=4k, if 2-level paging is used**
  - Makes life simpler for the kernel

# **Notes on reading xv6 code**

**Abhijit A. M.**  
**[abhijit.comp@coep.ac.in](mailto:abhijit.comp@coep.ac.in)**

**Credits:**  
**xv6 book by Cox, Kaashoek, Morris**  
**Notes by Prof. Sorav Bansal**

# **Introduction to xv6**

## **Structure of xv6 code**

## **Compiling and executing xv6 code**

# About xv6

- Unix Like OS
- Multi tasking, Single user
- On x86 processor
- Supports some system calls
- Small code, 7 to 10k
- Meant for learning OS concepts
- No : demand paging, no copy-on-write fork, no shared-memory, fixed size stack for user programs

# Use cscope and ctags with VIM

- Go to folder of xv6 code and run

```
cscope -q *. [chS]
```

- Also run

```
ctags *. [chS]
```

- Now download the file

[http://cscope.sourceforge.net/cscope\\_maps.vim](http://cscope.sourceforge.net/cscope_maps.vim)  
as `.cscope_maps.vim` in your `~` folder

- And add line “`source ~/.cscope_maps.vim`” in your  
`~/.vimrc` file

- Read this tutorial

[http://cscope.sourceforge.net/cscope\\_vim\\_tutorial.html](http://cscope.sourceforge.net/cscope_vim_tutorial.html)

# Use call graphs (using doxygen)

- Doxygen – a documentation generator.
- Can also be used to generate “call graphs” of functions
- Download xv6
- Install doxygen on your Ubuntu machine.
- cd to xv6 folder
- Run “doxygen -g doxyconfig”
  - This creates the file “doxyconfig”

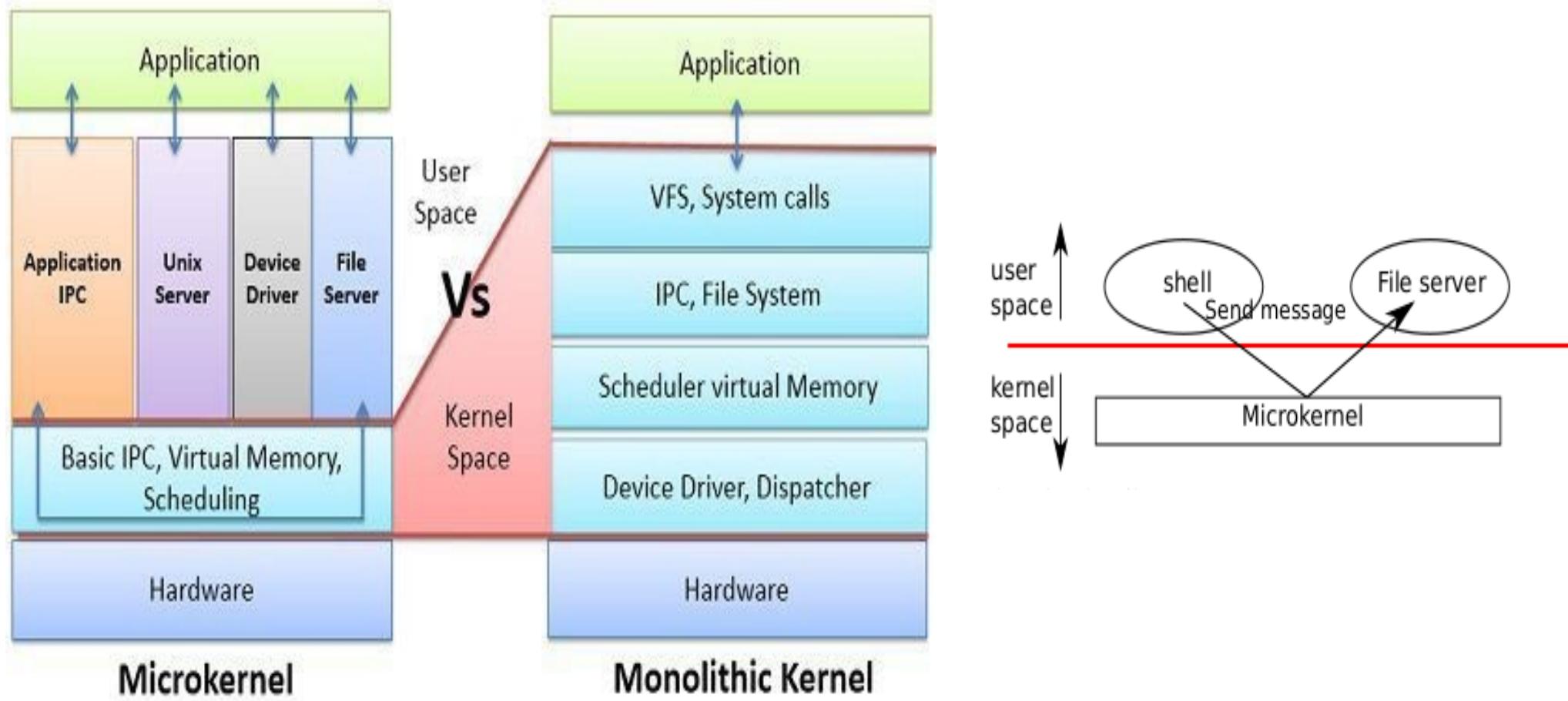
# Use call graphs (using doxygen)

- Create a folder “doxygen”
- Open “doxyconfig” file and make these changes.

```
PROJECT_NAME          = "XV6"
OUTPUT_DIRECTORY      = ./doxygen
CREATE_SUBDIRS        = YES
EXTRACT_ALL           = YES
EXCLUDE               = usertests.c cat.c yes.c echo.c
forktest.c grep.c init.c kill.c ln.c ls.c mkdir.c rm.c sh.c
stressfs.c wc.c zombie.c
CALL_GRAPH             = YES
CALLER_GRAPH           = YES
```

- Now run “doxygen doxyconfig”
- Go to “doxygen”/html and open “firefox index.html” --> See call graphs in files -> any file

# Xv6 follows monolithic kernel approach



# qemu

- A virtual machine manager, like Virtualbox
- Qemu provides us
  - BIOS
  - Virtual CPU, RAM, Disk controller, Keyboard controller
  - IOAPIC, LAPIC
- Qemu runs xv6 using this command

```
qemu -serial mon:stdio -drive  
file=fs.img,index=1,media=disk,format=raw -drive  
file=xv6.img,index=0,media=disk,format=raw -smp 2 -  
m 512
```

- Invoked when you run “make qemu”

# qemu

- **Understanding qemu command**
  - **-serial mon:stdio**
    - the window of xv6 is also multiplexed in your normal terminal.
    - Run “make qemu”, then Press “Ctrl-a” and “c” in terminal and you get qemu prompt
  - **-drive file=fs.img,index=1,media=disk,format=raw**
    - Specify the hard disk in “fs.img”, accessible at first slot in IDE(or SATA, etc), as a “disk” , with “raw” format
  - **-smp 2**
    - Two cores in SMP mode to be simulated
  - **-m 512**
    - Use 512 MB ram

# About files in XV6 code

- **cat.c echo.c forktest.c grep.c init.c kill.c ln.c ls.c mkdir.c rm.c sh.c stressfs.c usertests.c wc.c yes.c zombie.c**
  - User programs for testing xv6
- **Makefile**
  - To compile the code
- **dot-bochssrc**
  - For running with emulator bochs

# About files in XV6 code

- **bootasm.S entryother.S entry.S  
initcode.S swtch.S trapasm.S  
usys.S**
  - Kernel code written in Assembly. Total 373 lines
- **kernel.ld**
  - Instructions to Linker, for linking the kernel properly
- **README Notes LICENSE**
  - Misc files

# Using Makefile

- **make qemu**
  - Compile code and run using “qemu” emulator
- **make xv6.pdf**
  - Generate a PDF of xv6 code
- **make mkfs**
  - Create the mkfs program
- **make clean**
  - Remove all intermediary and final build files

# Files generated by Makefile

- **.o files**
  - Compiled from each .c file
  - No need of separate instruction in Makefile to create .o files
  - %: %.o \$(ULIB) line is sufficient to build each .o for a \_xyz file
-

# Files generated by Makefile

- **asm files**

- Each of them has an equivalent object code file or C file. For example  
**bootblock:** bootasm.S bootmain.c

```
$ (CC) $ (CFLAGS) -fno-pic -O -nostdinc -I. -c  
bootmain.c
```

```
$ (CC) $ (CFLAGS) -fno-pic -nostdinc -I. -c  
bootasm.S
```

```
$ (LD) $ (LDFLAGS) -N -e start -Ttext 0x7C00 -o  
bootblock.o bootasm.o bootmain.o
```

```
$ (OBJDUMP) -S bootblock.o > bootblock.asm
```

```
$ (OBJCOPY) -S -O binary -j .text bootblock.o  
bootblock
```

```
./sign.pl bootblock
```

# Files generated by Makefile

- **\_ln, \_ls, etc**
  - Executable user programs
  - Compilation process is explained after few slides

# Files generated by Makefile

- **xv6.img**

- Image of xv6 created

```
xv6.img: bootblock kernel
```

```
dd if=/dev/zero of=xv6.img  
count=10000
```

```
dd if=bootblock of=xv6.img  
conv=notrunc
```

```
dd if=kernel of=xv6.img seek=1  
conv=notrunc
```

# Files generated by Makefile

- **bootblock**

```
bootblock: bootasm.S bootmain.c  
          $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I.  
          -c bootmain.c  
          $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c  
bootasm.S  
          $(LD) $(LDFLAGS) -N -e start -Ttext  
0x7C00 -o bootblock.o bootasm.o bootmain.o  
          $(OBJDUMP) -S bootblock.o > bootblock.asm  
          $(OBJCOPY) -S -O binary -j .text  
bootblock.o bootblock  
          ./sign.pl bootblock
```

# Files generated by Makefile

## kernel

```
kernel: $(OBJS) entry.o entryother initcode  
kernel.ld  
          $(LD) $(LDFLAGS) -T kernel.ld -  
o kernel entry.o $(OBJS) -b binary  
initcode entryother  
          $(OBJDUMP) -S kernel >  
kernel.asm  
          $(OBJDUMP) -t kernel | sed  
'1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d'  
> kernel.sym
```

# Files generated by Makefile

- **fs.img**

- A disk image containing user programs and README

```
fs.img: mkfs README $(UPROGS)
```

```
    ./mkfs fs.img README $(UPROGS)
```

- **.sym files**

- Symbol tables of different programs

- E.g. for file “kernel”

```
$(OBJDUMP) -t kernel | sed '1,/SYMBOL  
TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

# Size of xv6 C code

- **wc \*[ch] | sort -n**
  - **10595 34249 278455 total**
  - Out of which
    - **738 4271 33514 dot-bochssrc**
- **wc cat.c echo.c forktest.c grep.c init.c kill.c ln.c ls.c mkdir.c rm.c sh.c stressfs.c usertests.c wc.c yes.c zombie.c**
  - **2849 6864 51993 total**
- So total code is  $10595 - 2849 - 738 = 7008$  lines

# List of commands to try (in given order)

**usertests** # Runs lot of tests and takes upto 10 minutes to run

**stressfs** # opens , reads and writes to files in parallel

**ls** # out put is filetyp, inode number, type

**cat README**

**ls;ls**

**cat README | grep BUILD**

**echo hi there**

**echo hi there | grep hi**

**echo "hi there**

# List of commands to try (in this order)

`echo README | grep Wa`

`ls ..` # works from inside test

`echo README | grep Wa |  
grep ty` # does not work

`cd #` fails

`cat README | grep Wa |  
grep bl` # works

`cd / #` works

`ls > out` # takes time!

`wc README`

`mkdir test`

`rm out`

`cd test`

`ls . test` # listing both  
directories

`ls #` fails

`In cat xyz; ls`

`rm xyz; ls`

# User Libraries: Used to link user land programs

- **Ulib.c**
  - Strcpy, strcmp, strlen, memset, strchr, stat, atoi, memmove
  - Stat uses open()
- **Usys.S -> compiles into usys.o**
  - Assembly code file. Basically converts all calls like open() (e.g. used in ulib.c) into assembly code using “int” instruction.

Run following command see the last 4 lines in the output

```
objdump -d usys.o
```

```
00000048 <open>:
```

48:	b8 0f 00 00 00	mov	\$0xf, %eax
4d:	cd 40	int	\$0x40
4f:	c3	ret	

# User Libraries: Used to link user land programs

- **printf.c**
  - Code for printf()!
  - Interesting to read this code.
    - Uses variable number of arguments. Normal technique in C is to use va\_args library, but here it uses pointer arithmetic.
  - Written using two more functions: printint() and putc() - both call write()
    - Where is code for write()?

# User Libraries: Used to link user land programs

- **umalloc.c**
  - This is an implementation of malloc() and free()
  - Almost same as the one done in “The C Programming Language” by Kernighan and Ritchie
  - Uses sbrk() to get more memory from xv6 kernel

# Understanding the build process in more details

- Run
  - make qemu | tee make-output.txt
- You will get all compilation commands in make-output.txt

# Compiling user land programs

Normally when you compile a program on Linux

You compile it for the same ‘target’ machine (= CPU + OS)

The compiler itself runs on the same OS

To compile a user land program for xv6, we don’t have a compiler on xv6,

So we compile the programs (using make, cc) on Linux , for xv6

Obviously they can’t link with the standard libraries on Linux

# Compiling user land programs

```
ULIB = ulib.o usys.o printf.o umalloc.o

_%: %.o $(ULIB)

        $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^
        $(OBJDUMP) -S $@ > $*.asm
        $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d;
s/ .* / /; /^$$/d' > $*.sym
```

**\$@ is the name of the file being generated**

**\$^ is dependencies . i.e. \$(ULIB) and %.o in this case**

# Compiling user land programs

```
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing  
-O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-  
pointer -fno-stack-protector -fno-pie -no-pie -c -o  
cat.o cat.c
```

```
ld -m elf_i386 -N -e main -Ttext 0 -o _cat cat.o  
ulib.o usys.o printf.o umalloc.o
```

```
objdump -S _cat > cat.asm
```

```
objdump -t _cat | sed '1,/SYMBOL TABLE/d;  
s/ .* / /; /^$/d' > cat.sym
```

# Compiling user land programs

Mkfs is compiled like a Linux program !

```
gcc -Werror -Wall -o mkfs mkfs.c
```

# How to read kernel code ?

- **Understand the data structures**
  - Know each global variable, typedefs, lists, arrays, etc.
  - Know the purpose of each of them
- **While reading a code path, e.g. exec()**
  - Try to ‘locate’ the key line of code that does major work
  - Initially (but not forever) ignore the ‘error checking’ code
- **Keep summarising what you have read**
  - Remembering is important !
- **To understand kernel code, you should be good with concepts in OS , C, assembly, hardware**

# Pre-requisites for reading the code

- **Understanding of core concepts of operating systems**
  - Memory Management, processes, fork-exec, file systems, synchronization, x86 architecture, calling convention , computer organization
- **2 approaches:**
  - 1) Read OS basics first, and then start reading xv6 code
    - Good approach, but takes more time !
  - 2) Read some basics, read xv6, repeat
    - Gives a headstart, but you will always have gaps in your understanding of the code, until you are done with everything
    - We normally follow this approach
- Good knowledge of C, pointers, **function pointers** particularly
  - Data structures: doubly linked lists, queues, structures and pointers

# XV6 bootloader

Abhijit A. M.  
[abhijit.comp@coep.ac.in](mailto:abhijit.comp@coep.ac.in)

Credits:  
**xv6 book by Cox, Kaashoek, Morris**  
**Notes by Prof. Sorav Bansal**

# A word of caution

- We begin reading xv6 code
- But it's not possible to read this code in a “linear fashion”
  - The dependency between knowing OS concepts and reading/writing a kernel that is written using all concepts

# **What we have seen ....**

- **Compilation process, calling conventions**
- **Basics of Memory Management by OS**
- **Basics of x86 architecture**
  - Registers, segments, memory management unit, addressing, some basic machine instructions,
- **ELF files**
  - Objdump, program headers
  - Symbol tables

# Boot-process

- **Bootloader itself**
  - Is loaded by the BIOS at a fixed location in memory and BIOS makes it run
  - Our job, as OS programmers, is to write the bootloader code
- **Bootloader does**
  - Pick up code of OS from a ‘known’ location and loads it in memory
  - Makes the OS run
- **Xv6 bootloader: bootasm.S bootmain.c (see Makefile)**

# bootloader

- BIOS Runs (automatically)
- Loads boot sector into RAM at 0x7c00
- Starts executing that code
  - Make sure that your bootloader is loaded at 0x7c00
  - Makefile has

```
bootblock: bootblock.S bootmain.c  
$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S .....  
...  
$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o  
bootasm.o bootmain.o
```

Results in:

00007c00 <start>: in bootblock.asm

# Processor starts in real mode

- Processor starts in real mode – works like 16 bit 8088
- eight 16-bit general-purpose registers,
- Segment registers %cs, %ds, %es, and %ss --> additional bits necessary to generate 20-bit memory addresses from 16-bit registers.

$\text{addr} = \text{seg} \ll 4 + \text{addr}$



**Effective memory translation in the beginning  
At \_start in bootasm.S:**

**%cs=0 %ip=7c00.**

**So effective address = 0\*16+ip = ip**

# bootloader

- **First instruction is ‘cli’**
  - disable interrupts
- **So that until your code loads all hardware interrupt handlers, no interrupt will occur**

# Zeroing registers

```
# Zero data segment registers DS, ES, and  
ss.  
  
xorw %ax,%ax      # Set %ax to zero  
  
movw %ax,%ds      # -> Data Segment  
  
movw %ax,%es      # -> Extra Segment  
  
movw %ax,%ss      # -> Stack Segment
```

- **zero ax and ds, es, ss**
- **BIOS did not put in anything perhaps**

# A not so necessary detail Enable 21 bit address

seta20.1:

```
inb    $0x64,%al          # Wait for not busy
testb   $0x2,%al
jnz    seta20.1
movb   $0xd1,%al          # 0xd1 -> port
0x64
```

```
outb   %al,$0x64
```

seta20.2:

```
inb    $0x64,%al          # Wait for not busy
testb   $0x2,%al
jnz    seta20.2
movb   $0xdf,%al          # 0xdf -> port 0x60
outb   %al,$0x60
```

- **Seg:off with 16 bit segments can actually address more than 20 bits of memory. After 0x100000 (=2<sup>20</sup>), 8086 wrapped addresses to 0.**
- **80286 introduced 21<sup>st</sup> bit of address. But older software required 20 bits only. BIOS disabled 21<sup>st</sup> bit. Some OS needed 21<sup>st</sup> Bit. So enable it.**
- **Write to Port 0x64 and 0x60 -> keyboard controller**
  - to enable 21<sup>st</sup> bit out of address translation
  - **Why?** Before the A20, i.e. 21<sup>st</sup> bit was introduced, it belonged to keyboard controller
  - **For more details see [https://en.wikipedia.org/wiki/A20\\_line](https://en.wikipedia.org/wiki/A20_line)**

[https://en.wikipedia.org/wiki/A20\\_line](https://en.wikipedia.org/wiki/A20_line)

**After this**

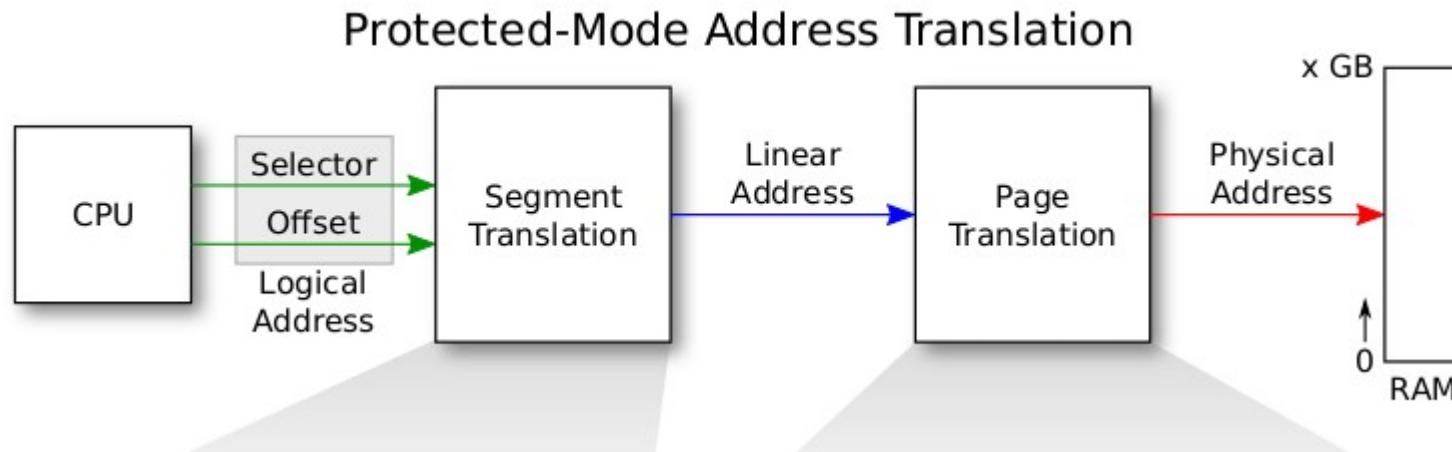
**Some instructions are run  
to enter protected mode**

**And further code runs in protected mode**

# Real mode Vs protected mode

- **Real mode 16 bit registers**
- **Protected mode**
  - Enables segmentation + Paging both
    - No longer seg\*16+offset calculations
    - Segment registers is index into segment descriptor table. But segment:offset pairs continue
      - `mov %esp, $32 # SS will be used with esp`
    - More in next few slides
    - Other segment registers need to be explicitly mentioned in instructions
      - `Mov FS:$200, 30`
  - **32 bit registers**
    - can address upto  $2^{32}$  memory
    - Can do arithmetic in 32 bits

# X86 address : protected mode address translation



**Both Segmentation and Paging are used in x86  
X86 allows optionally one-level or two-level paging**

**Segmentation is a must to setup, paging is optional (needs to be enabled)  
Hence different OS can use segmentation+paging in different ways**

# X86 segmentation

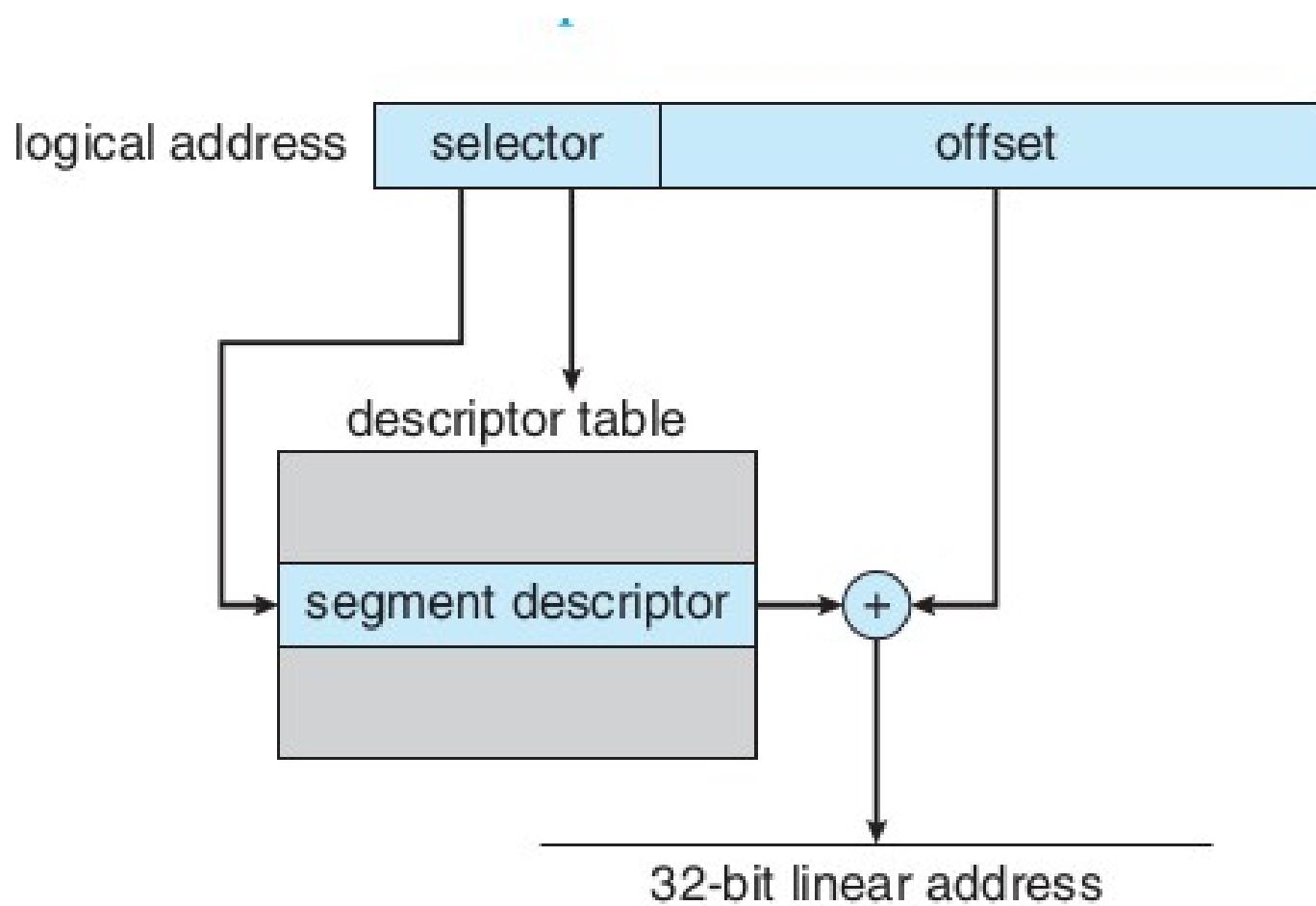


Figure 8.22 IA-32 segmentation.

# Paging concept, hierarchical paging

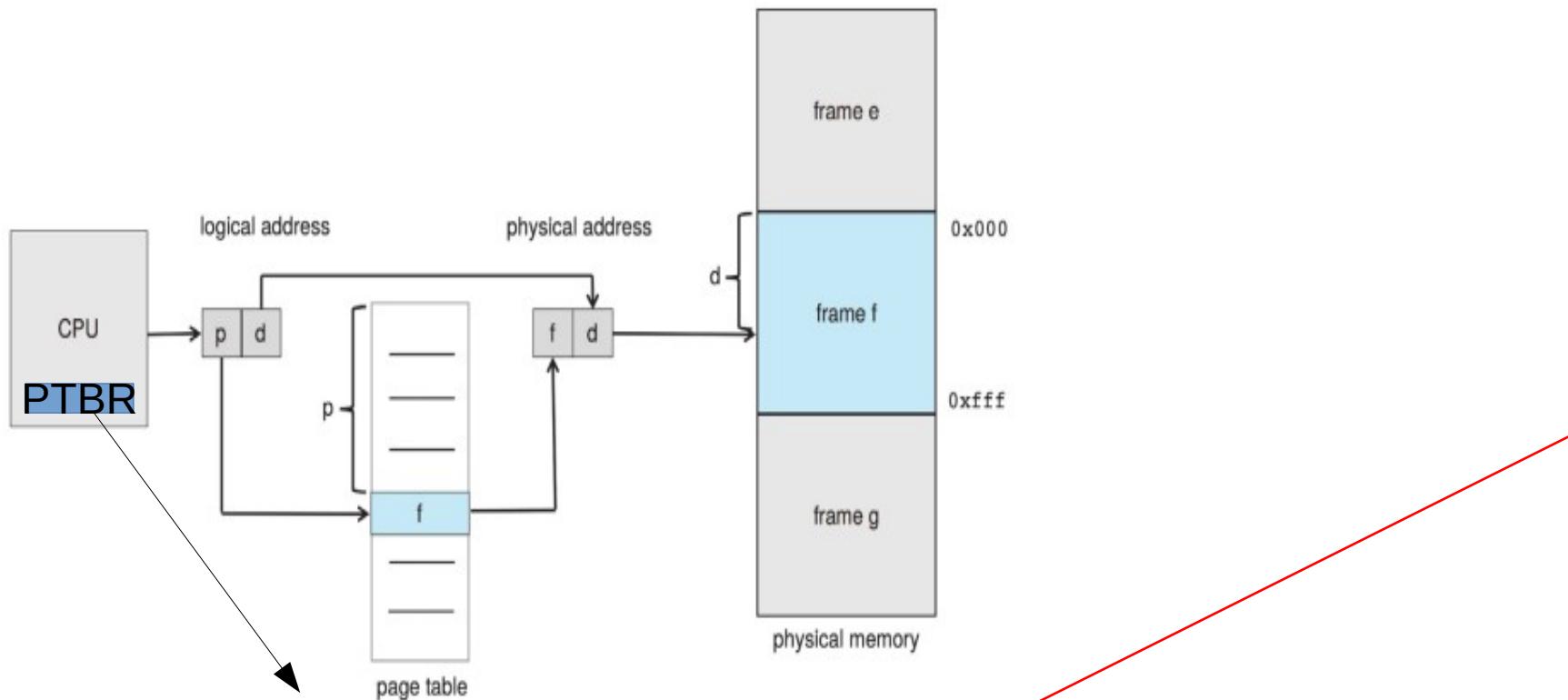


Figure 9.8 Paging hardware.

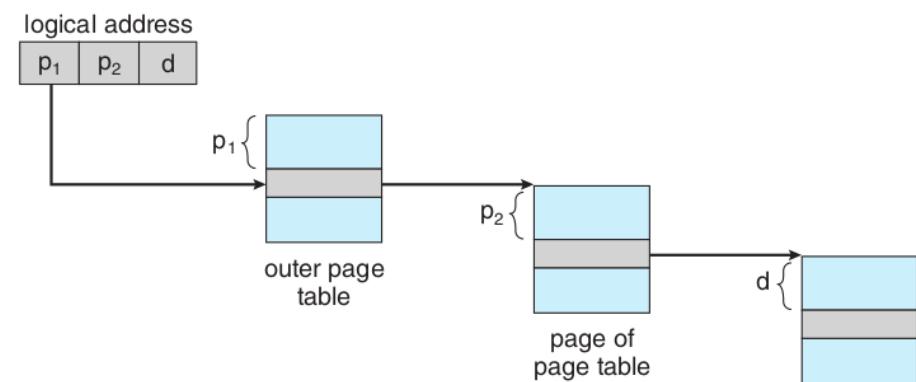
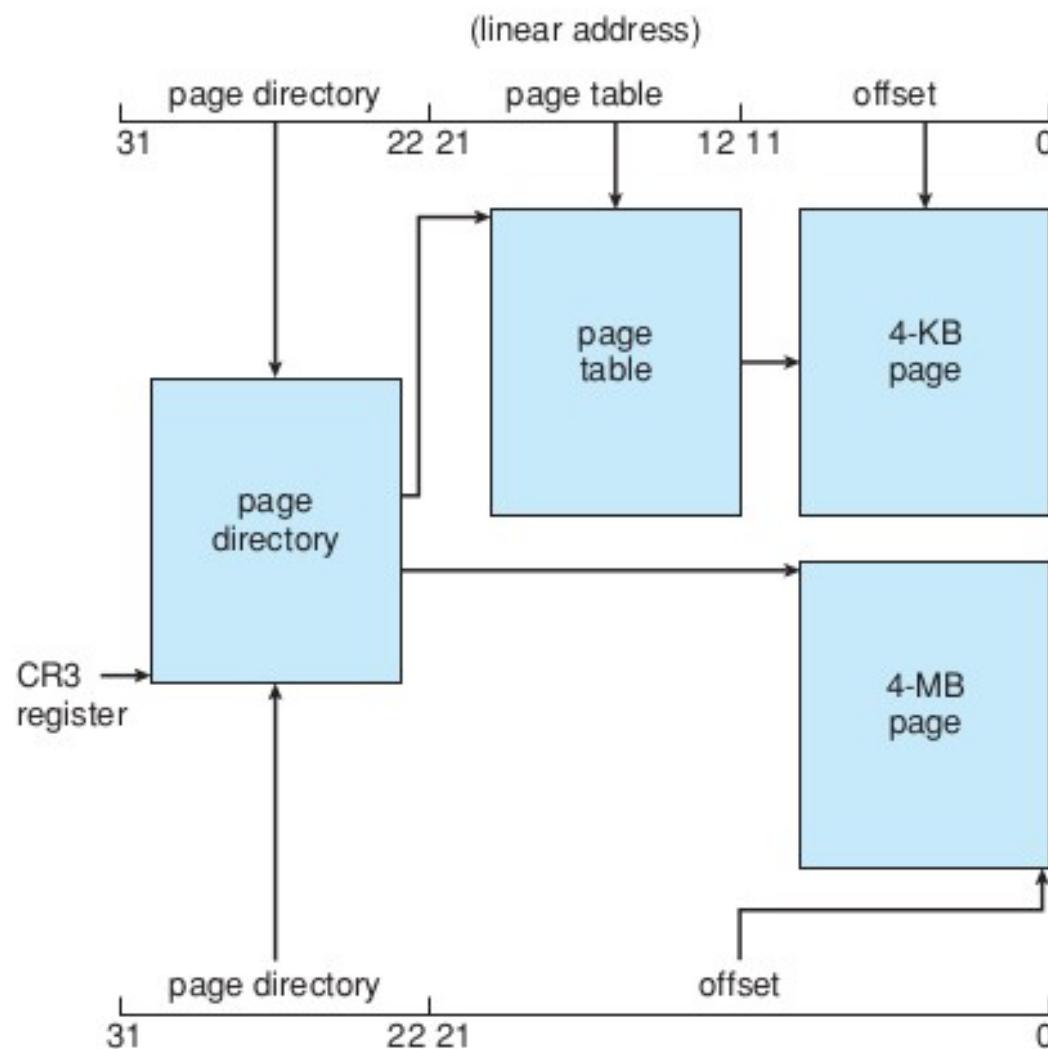


Figure 9.16 Address translation for a two-level 32-bit paging architecture.

# X86 paging



**Figure 8.23** Paging in the IA-32 architecture.

# Page Directory Entry (PDE) Page Table Entry (PTE)

31

Page table physical page number	12	11	10	9	8	7	6	5	4	3	2	1	0
	A V L	G S	P 0	A	C D	W T	U D	W T	U D	W T	U D	W T	P

PDE

P Present

W Writable

U User

WT 1=Write-through, 0=Write-back

CD Cache disabled

A Accessed

D Dirty

PS Page size (0=4KB, 1=4MB)

PAT Page table attribute index

G Global page

AVL Available for system use

31

Physical page number	12	11	10	9	8	7	6	5	4	3	2	1	0
	A V L	G T	P A	D A	A D	C D	W T	U D	W T	U D	W T	U D	P

PTE

# CR3

CR3



PWT Page-level writes transparent

PCT Page-level cache disable

# CR4

CR4



VME Virtual-8086 mode extensions

PVI Protected-mode virtual interrupts

TSD Time stamp disable

DE Debugging extensions

PSE Page size extensions

PAE Physical-address extension

MCE Machine check enable

PGE Page-global enable

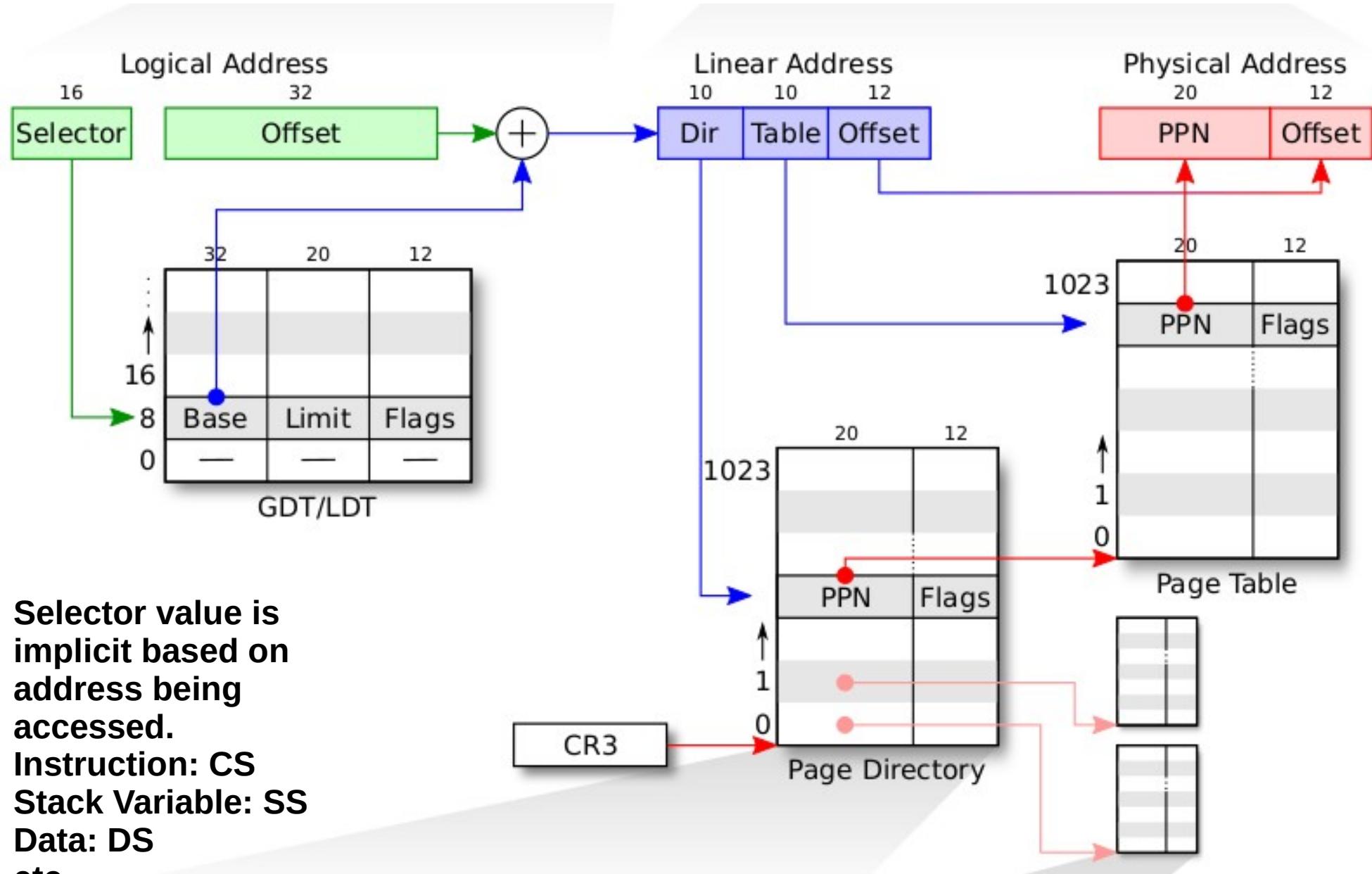
PCE Performance counter enable

OSFXSR OS FXSAVE/FXRSTOR support

OSXMM- OS unmasked exception support

EXCPT

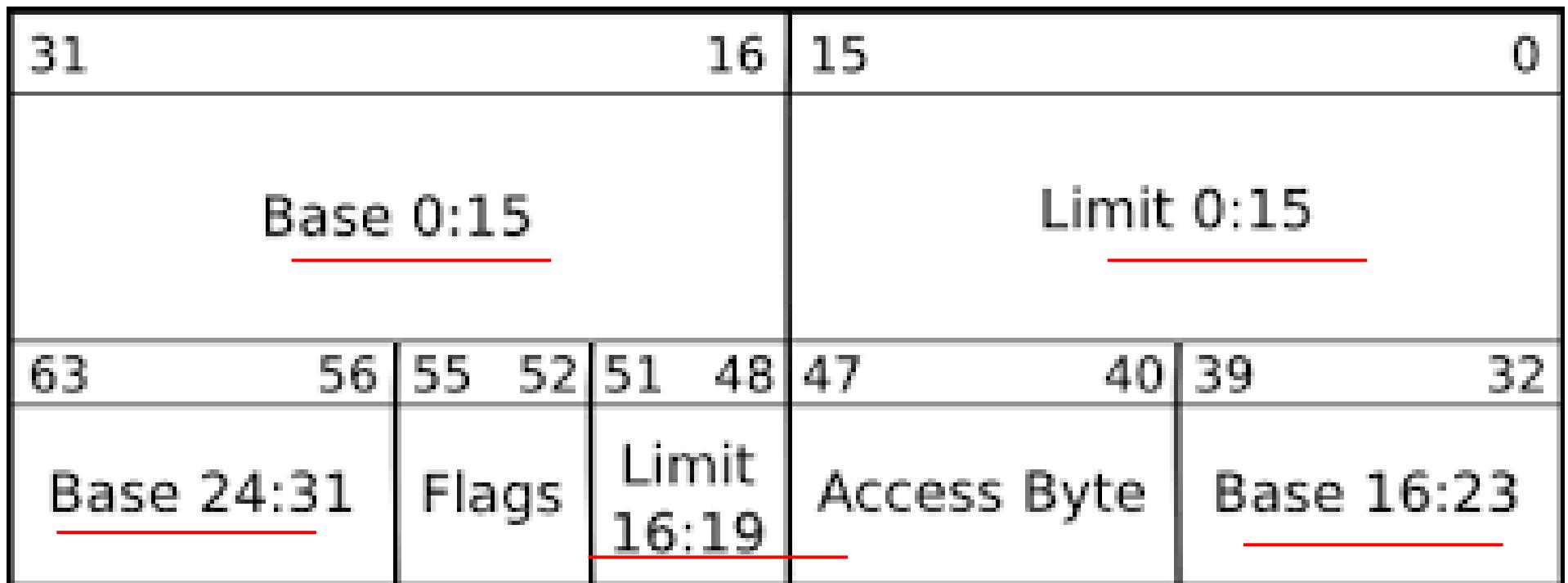
# Segmentation + Paging



# Segmentation + Paging setup of xv6

- xv6 configures the segmentation hardware by setting Base = 0, Limit = 4 GB
  - translate logical to linear addresses without change, so that they are always equal.
  - Segmentation is practically off
- Once paging is enabled, the only interesting address mapping in the system will be linear to physical.
  - In xv6 paging is NOT enabled while loading kernel
  - After kernel is loaded 4 MB pages are used for a while
  - Later the kernel switches to 4 kB pages!

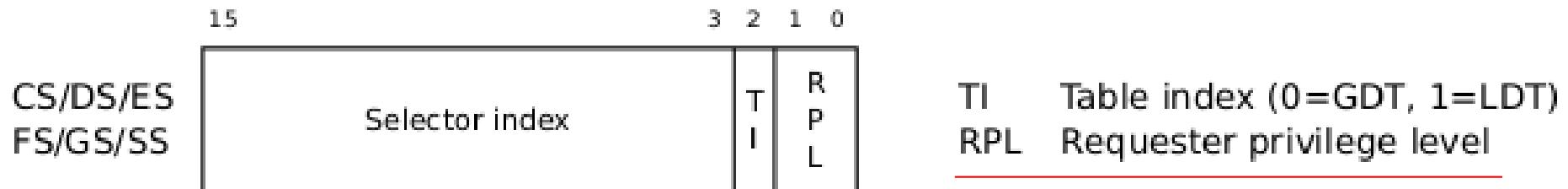
# GDT Entry



## asm.h

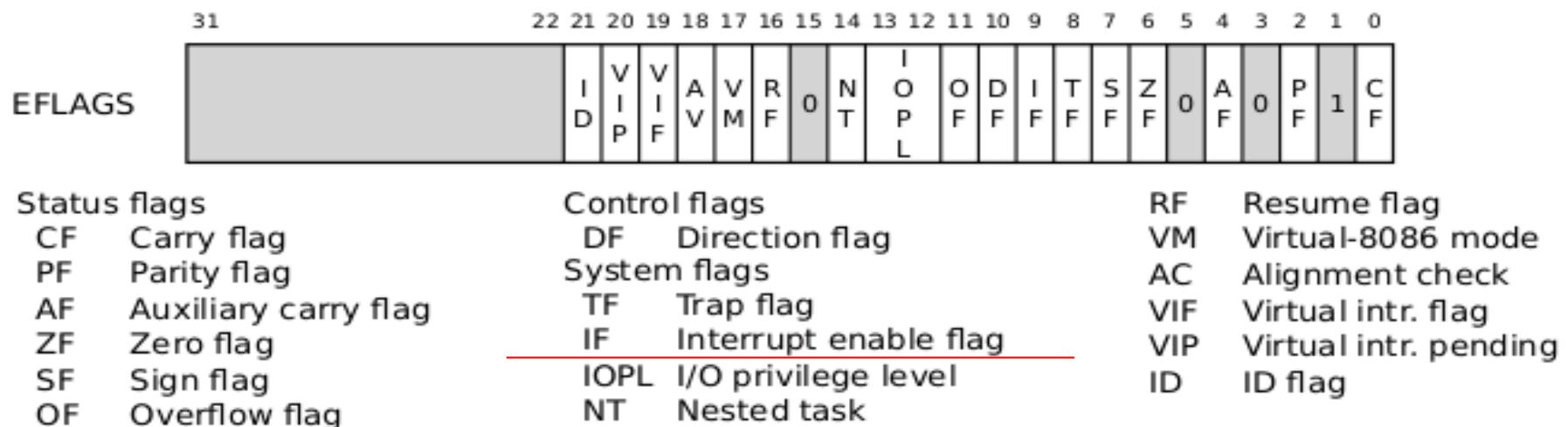
```
#define SEG_ASM(type,base,lim) \
    .word (((lim) >> 12) & 0xffff), ((base) & 0xffff); \
    .byte (((base) >> 16) & 0xff), (0x90 | (type)), \
          (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
```

# Segment selector



**Note in 16 bit mode, segment selector is 16 bit, here it's 13 bit + 3 bits**

# EFLAGS register



# Igdt

Igdt gdtdesc

...

# Bootstrap GDT

.p2align 2 # force 4 byte alignment

gdt:

SEG\_NULLASM # null seg

SEG\_ASM(STA\_X|STA\_R, 0x0,  
0xffffffff) # code seg

SEG\_ASM(STA\_W, 0x0, 0xffffffff)

# data seg

gdtdesc:

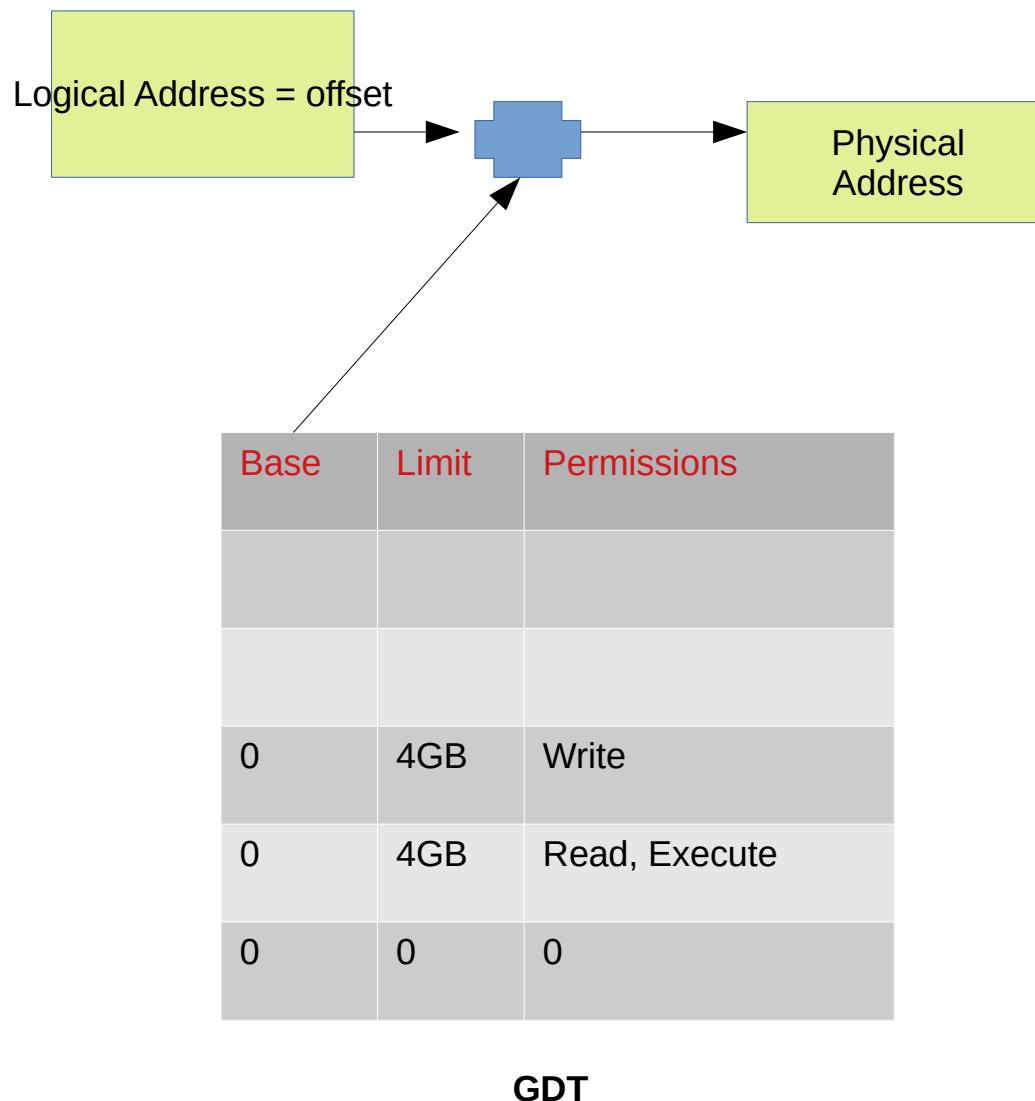
.word (gdtdesc - gdt - 1)

# sizeof(gdt) - 1

.long gdt

- load the processor's (GDT) register with the value gdtdesc which points to the table gdt.
- **table gdt :** The table has a null entry, one entry for executable code, and one entry to data.
- all segments have a base address of zero and the maximum possible limit
- The code segment descriptor has a flag set that indicates that the code should run in 32-bit mode
- With this setup, when the boot loader enters protected mode, logical addresses map one-to-one to physical addresses.
- At gdtdesc we have this data  
2, <4 byte addr of gdt>  
Total 6 bytes
- GDTR is : <address 4 byte>, <table limit 2 byte>
- So Igdt gdtdesc loads these two values in GDTR

**bootasm.S after “lgdt gdtdesc”  
till jump to “entry”**



Still  
Logical Address =  
Physical address!

But with GDT in picture  
and  
Protected Mode  
operation

**During this time,**

**Loading kernel from  
ELF into physical  
memory**

**Addresses in “kernel”  
file translate to same  
physical address!**

# Prepare to enable protected mode

- Prepare to enable protected mode by setting the 1 bit (CR0\_PE) in register %cr0

```
movl %cr0, %eax  
orl $CR0_PE, %eax  
movl %eax, %cr0
```

# CR0

CR0	31 30 29 28	19 18 17 16 15	6 5 4 3 2 1 0		
	P G   C D   N W	A M   W P	N E T S E M P P E		
PE	Protection enabled	ET	Extension type	NW	Not write-through
MP	Monitor coprocessor	NE	Numeric error	CD	Cache disable
EM	Emulation	WP	Write protect	PG	Paging
TS	Task switched	AM	Alignment mask		

**PG: Paging enabled or not**

**WP: Write protection on/off**

**PE: Protection Enabled --> protected mode.**

# Complete transition to 32 bit mode

`Ijmp $(SEG_KCODE<<3), $start32`

**Complete the transition to 32-bit protected mode by using a long jmp**

**to reload %cs (=1) and %eip (=start32).**

**Note that ‘start32’ is the address of next instruction after Ijmp.**

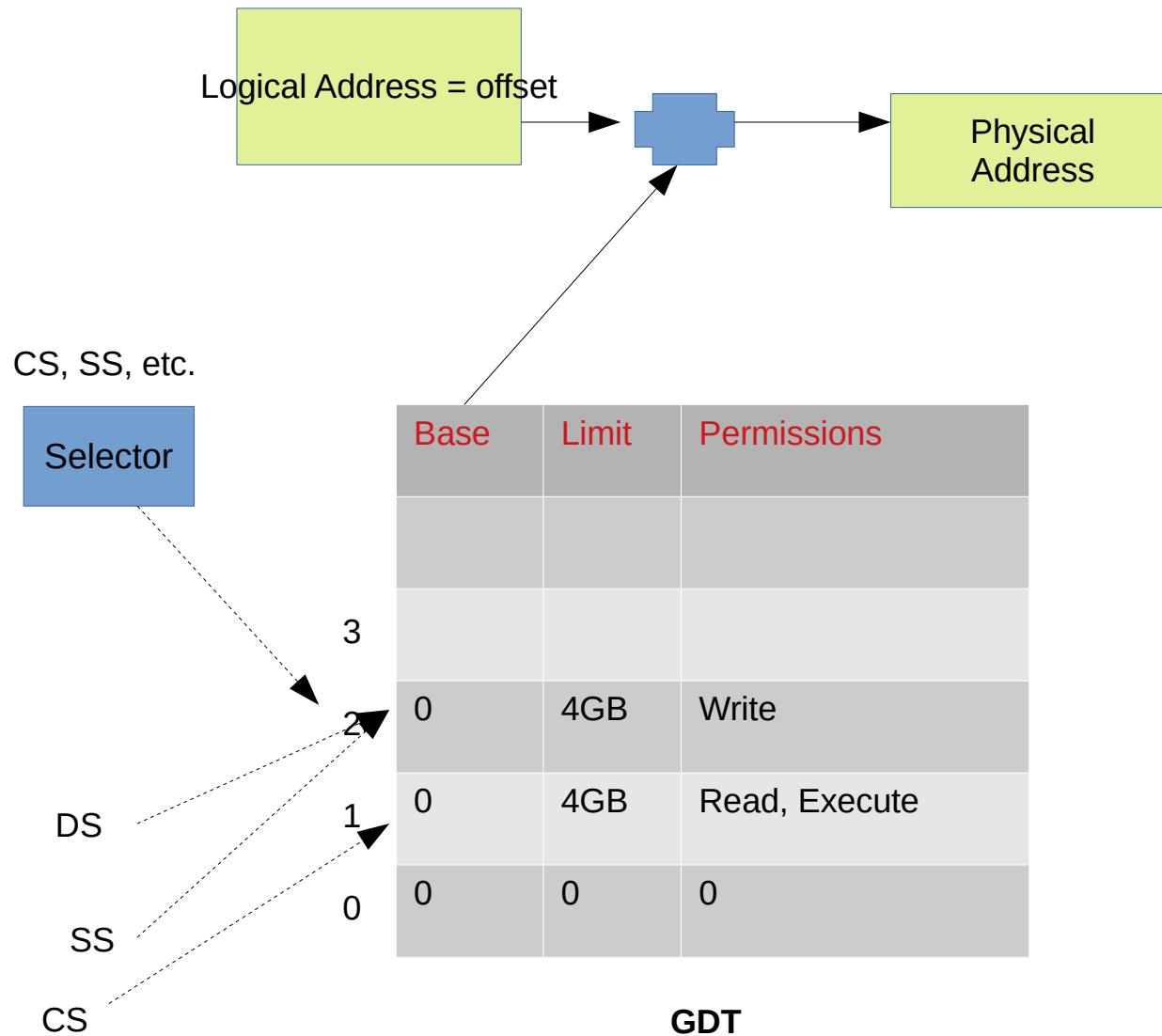
**Note: The segment descriptors are set up with no translation (that is 0-4GB setting), so that the mapping is still the identity mapping.**

# Jumping to “C” code

```
movw $(SEG_KDATA<<3), %ax # Our data  
segment selector  
  
    movw %ax, %ds      # -> DS: Data  
Segment  
  
    movw %ax, %es      # -> ES: Extra  
Segment  
  
    movw %ax, %ss      # -> SS: Stack  
Segment  
  
    movw $0, %ax        # Zero segments  
not ready for use  
  
    movw %ax, %fs      # -> FS  
  
    movw %ax, %gs      # -> GS  
  
  
# Set up the stack pointer and call into C.  
movl $start, %esp  
call bootmain
```

- **Setup Data, extra, stack segment with SEG\_KDATA (=2), FS & GS (=0)**
- **Copy “\$start” i.e. 7c00 to stack-ptr**
  - It will grow from 7c00 to 0000
- **Call bootmain() a C function**
  - In bootmain.c

## Setup now



# bootmain(): already in memory, as part of ‘bootblock’

- **bootmain.c , expects to find a copy of the kernel executable on the disk starting at the second sector (sector = 1).**
  - Why?
- **The kernel is an ELF format binary**
- **Bootmain loads the first 4096 bytes of the ELF binary. It places the in-memory copy at address 0x10000**
- **readseg() is a function that runs OUT instructions in particular IO ports, to issue commands to read from Disk**

```
void bootmain(void)
{
    struct elfhdr *elf;
    struct proghdr *ph, *eph;
    void (*entry)(void);
    uchar* pa;

    elf = (struct elfhdr*)0x10000; // scratch space

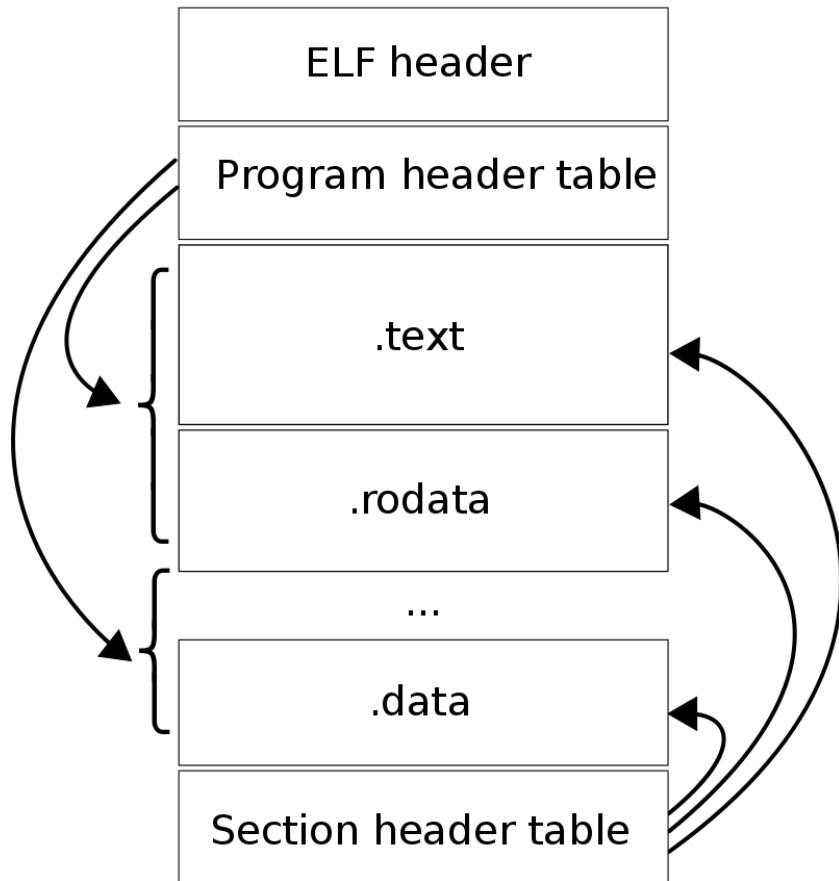
    // Read 1st page off disk
    readseg((uchar*)elf, 4096, 0);
```

# bootmain()

- Check if it's really ELF or not // Is this an ELF executable?
- Next load kernel code from ELF file “kernel” into memory  

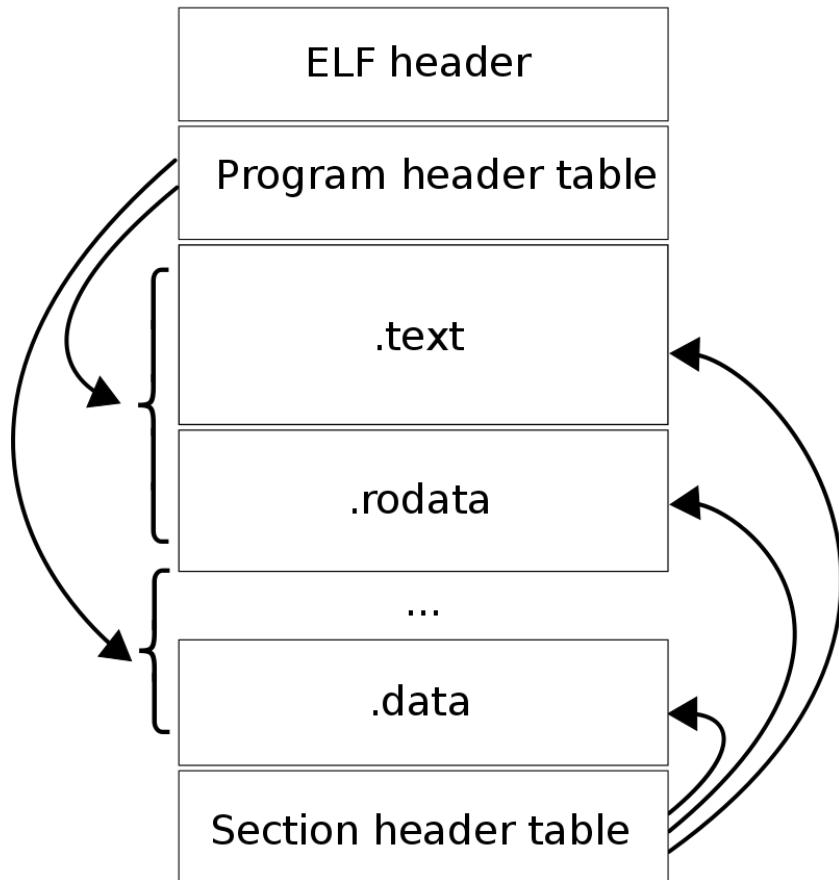
```
if(elf->magic != ELF_MAGIC)
    return; // let bootasm.S handle error
```

# ELF



```
struct elfhdr {  
    uint magic; // must equal  
ELF_MAGIC  
    uchar elf[12];  
    ushort type;  
    ushort machine;  
    uint version;  
    uint entry;  
    uint phoff; // where is program  
header table  
    uint shoff;  
    uint flags;  
    ushort ehsiz;  
    ushort phentsiz;  
    ushort phnum; // no. Of program  
header entries  
    ushort shentsiz;  
    ushort shnum;  
    ushort shstrndx;  
};
```

# ELF



// Program header

struct proghdr {

    uint type; // Loadable segment ,  
    Dynamic linking information ,  
    Interpreter information , Thread-  
    Local Storage template , etc.

    uint off; //Offset of the segment  
    in the file image.

    uint vaddr; //Virtual address of  
    the segment in memory.

    uint paddr; // physical address to  
    load this program, if PA is relevant

    uint filesz; //Size in bytes of the  
    segment in the file image.

    uint memsz; //Size in bytes of the  
    segment in memory. May be 0.

    uint flags;

    uint align;

};

# Run ‘objdump -x -a kernel | head -15’ & see this

```
kernel: file format elf32-i386  
kernel  
architecture: i386, flags 0x000000112:  
EXEC_P, HAS_SYMS, D_PAGED  
start address 0x0010000c
```

Program Header:

```
LOAD off 0x00001000 vaddr 0x80100000 paddr 0x00100000 align 2**12  
  filesz 0x0000a516 memsz 0x000154a8 flags rwx  
STACK off 0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4  
  filesz 0x00000000 memsz 0x00000000 flags rwx
```

Code to be  
loaded at  
KERNBASE +  
KERNLINK

Diff  
between  
memsz &  
filesz, will  
be filled  
with zeroes  
in memory

Stack :  
everything  
zeroes

# Load code from ELF to memory

```
// Load each program segment (ignores ph flags).
ph = (struct proghdr*)((uchar*)elf + elf->phoff);
eph = ph + elf->phnum;
// Abhijit: number of program headers
for(; ph < eph; ph++) {
    // Abhijit: iterate over each program header
    pa = (uchar*)ph->paddr;
    // Abhijit: the physical address to load program
    /* Abhijit: read ph->filesz bytes, into 'pa',
       from ph->off in kernel/disk */
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
        stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
// Zero the remainder section*/
}
```

# Jump to Entry

```
// Call the entry point from the ELF header.  
// Does not return!  
/* Abhijit:  
 * elf->entry was set by Linker using kernel.ld  
 * This is address 0x80100000 specified in  
kernel.ld  
 * See kernel.asm for kernel assembly code).  
 */  
entry = (void(*)(void))(elf->entry);  
entry();
```

To understand  
further  
code

Remember: 4  
MB pages  
are possible

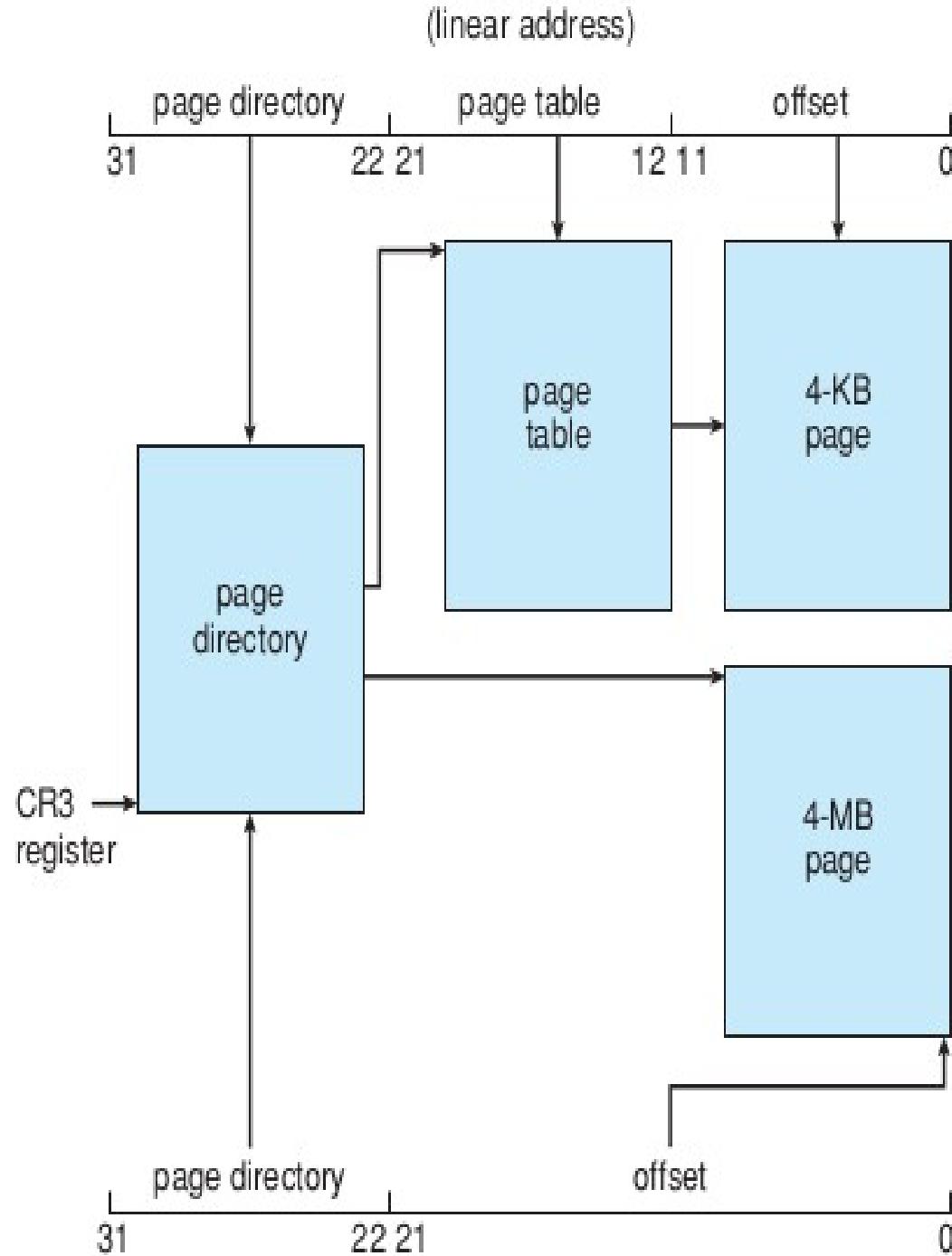
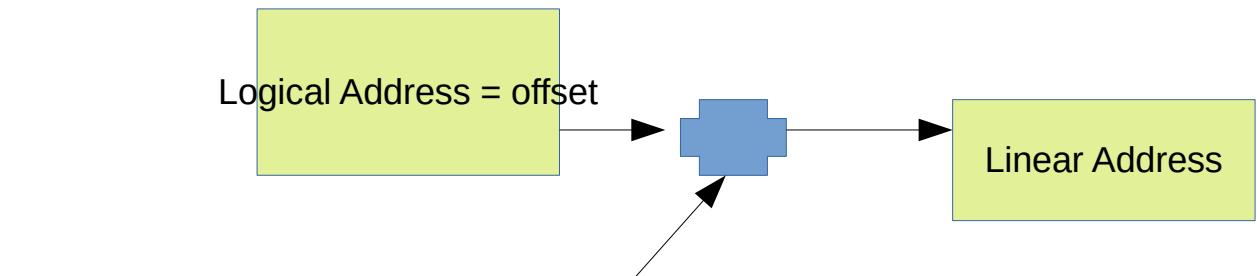


Figure 8.23 Paging in the IA-32 architecture.

**From entry:  
Till: inside main(), before kvmalloc()**



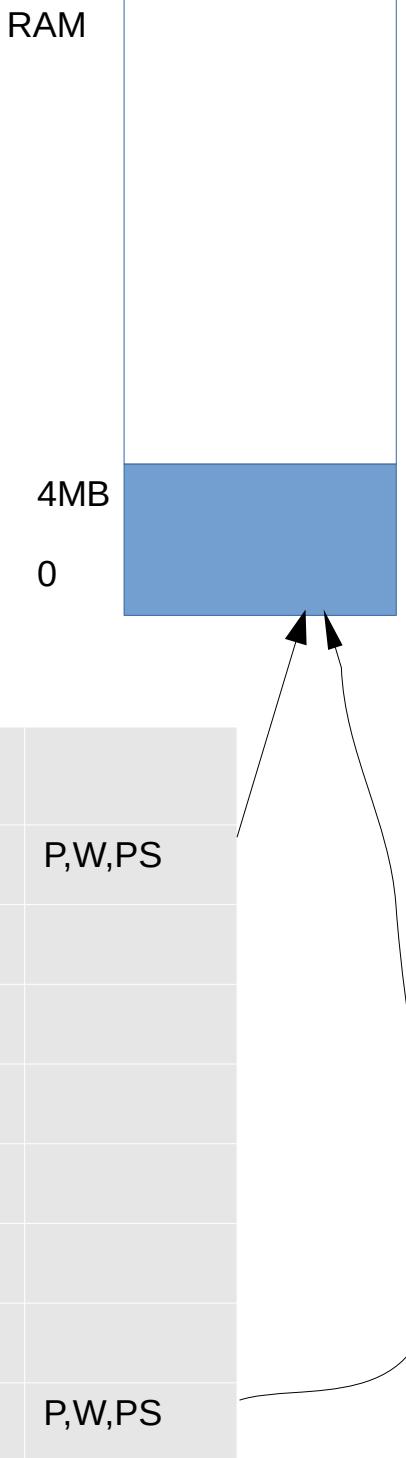
CS, SS, etc.

**GDT**

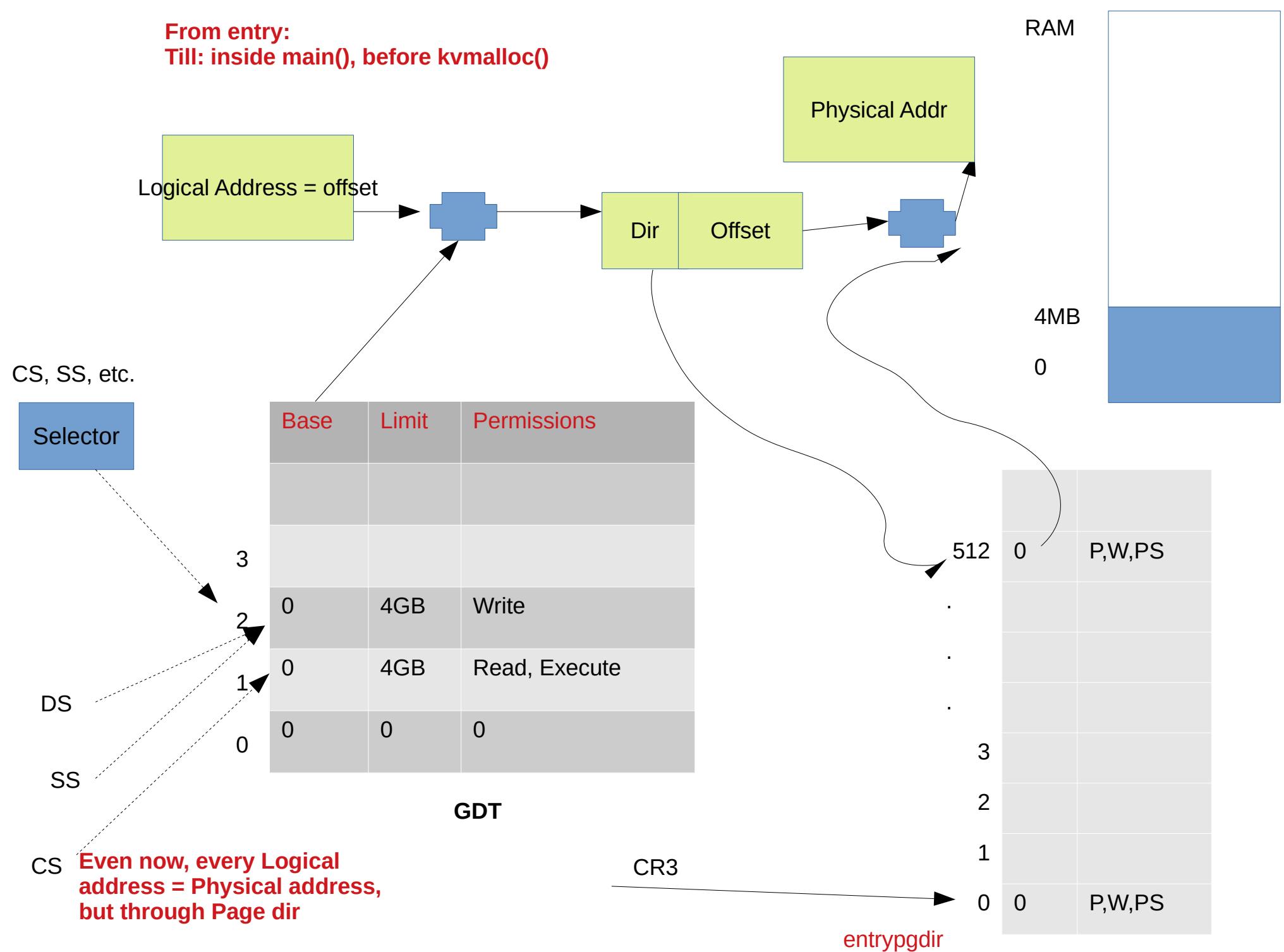
Selector	Base	Limit	Permissions
3	0	4GB	Write
2	0	4GB	Read, Execute
DS	0	0	0
SS			
CS			

CR3

entrypgdir



From entry:  
Till: inside main(), before kvmalloc()



# entrypgdir in main.c, is used by entry()

```
_attribute__((aligned_(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
    // Map VA's [0, 4MB) to PA's [0, 4MB)
    [0] = (0) | PTE_P | PTE_W | PTE_PS,
    // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB). This is entry 512
    [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

This is entry page directory during entry(), beginning of kernel  
Mapping 0:0x400000 (i.e. 0: 4MB) to physical addresses 0:0x400000. is required as long  
as entry is executing at low addresses, but will eventually be removed.  
This mapping restricts the kernel instructions and data to 4 Mbytes.

#define PTE_P	0x001 // Present
#define PTE_W	0x002 // Writeable
#define PTE_U	0x004 // User
#define PTE_PS	0x080 // Page Size
#define PDXSHIFT	22 // offset of PDX in a linear address

// Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB). This is entry 512

# entry() in entry.S

entry:

```
movl %cr4, %eax  
orl $(CR4_PSE), %eax  
movl %eax, %cr4  
movl $(V2P_WO(entrypgdir)),  
%eax  
movl %eax, %cr3  
movl %cr0, %eax  
orl $(CR0_PG|CR0_WP), %eax  
movl %eax, %cr0  
movl $(stack + KSTACKSIZE),  
%esp  
mov $main, %eax  
jmp *%eax
```

- # Turn on page size extension for 4Mbyte pages
- # Set page directory. 4 MB pages (temporarily only. More later)
- # Turn on paging.
- # Set up the stack pointer.
- # Jump to main(), and switch to executing at high addresses. The indirect call is needed because the assembler produces a PC-relative instruction for a direct jump.
-

# More about entry()

```
movl $(V2P_WO(entrypgdir)), %eax  
movl %eax, %cr3
```

-> Here we use physical address using V2P\_WO because paging is not turned on yet

- **V2P is simple: subtract 0x80000000 i.e. KERNBASE from address**

# More about entry()

```
movl %cr0, %eax  
orl $(CR0_PG|  
CR0_WP), %eax  
movl %eax, %cr0
```

- But we have already set 0'th entry in pgdir to address 0
- So it still works!

This turns on paging

After this also, entry() is running and processor is executing code at lower addresses

# entry()

```
movl $(stack +  
KSTACKSIZE), %esp
```

```
mov $main, %eax  
jmp *%eax
```

```
.comm stack,  
KSTACKSIZE
```

```
# Abhijit: allocate here 'stack' of size =  
KSTACKSIZE
```

- # Set up the stack pointer.
- # Abhijit: +KSTACKSIZE is done as stack grows downwards
- # Jump to main(), and switch to executing at high addresses. The indirect call is needed because the assembler produces a PC-relative instruction for a direct jump.

# **bootmasm.S bootmain.c: Steps**

- 1) Starts in “real” mode, 16 bit mode. Does some misc legacy work.**
- 2) Runs instructions to do MMU set-up for protected-mode & only segmentation (0-4GB, identity mapping), changes to protected mode.**
- 3) Reads kernel ELF file and loads it in RAM, as per instructions in ELF file**
- 4) Sets up paging (4 MB pages)**
- 5) Runs main() of kernel**

Code from bootasm.S bootmain.c is over!  
Kernel is loaded.  
Now kernel is going to prepare itself

# Processes in xv6 code

# Process Table

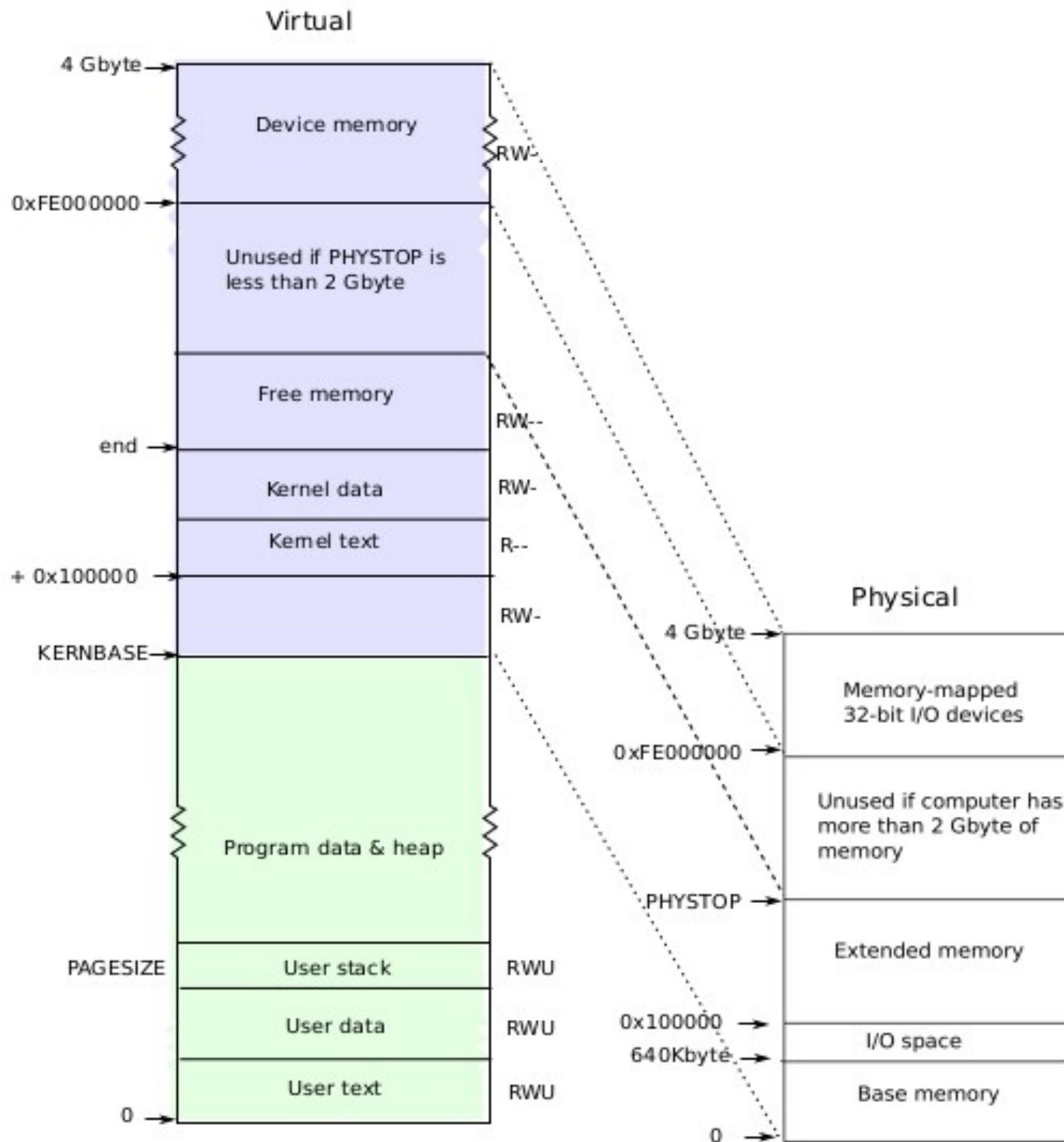
```
struct {  
    struct spinlock lock;  
    struct proc proc[NPROC];  
} ptable;
```

- One single global array of processes
- Protected by `ptable.lock`

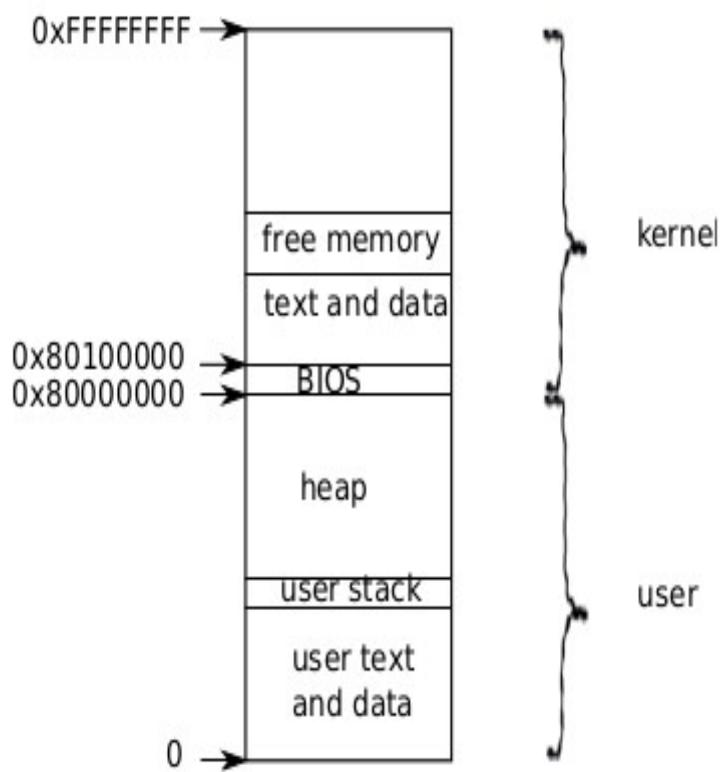
## Layout of process's VA space

xv6  
schema!

different  
from Linux

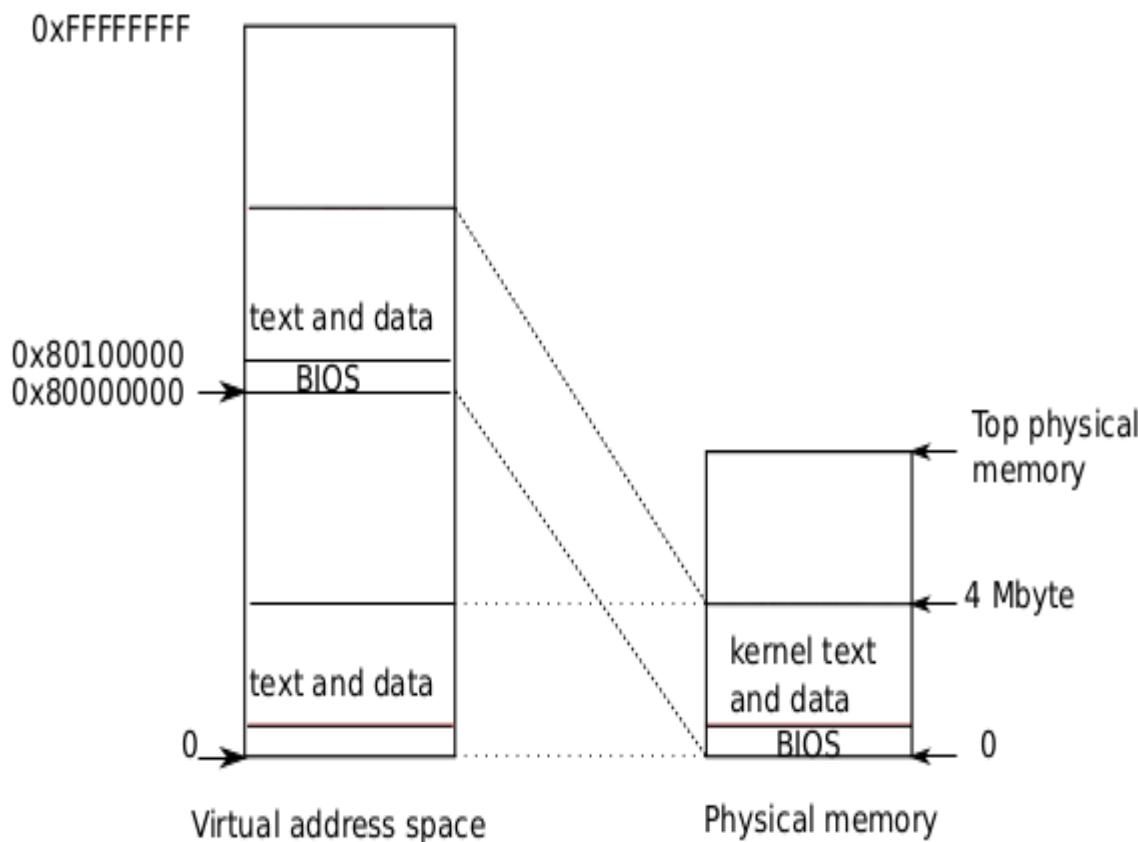


# Logical layout of memory for a process



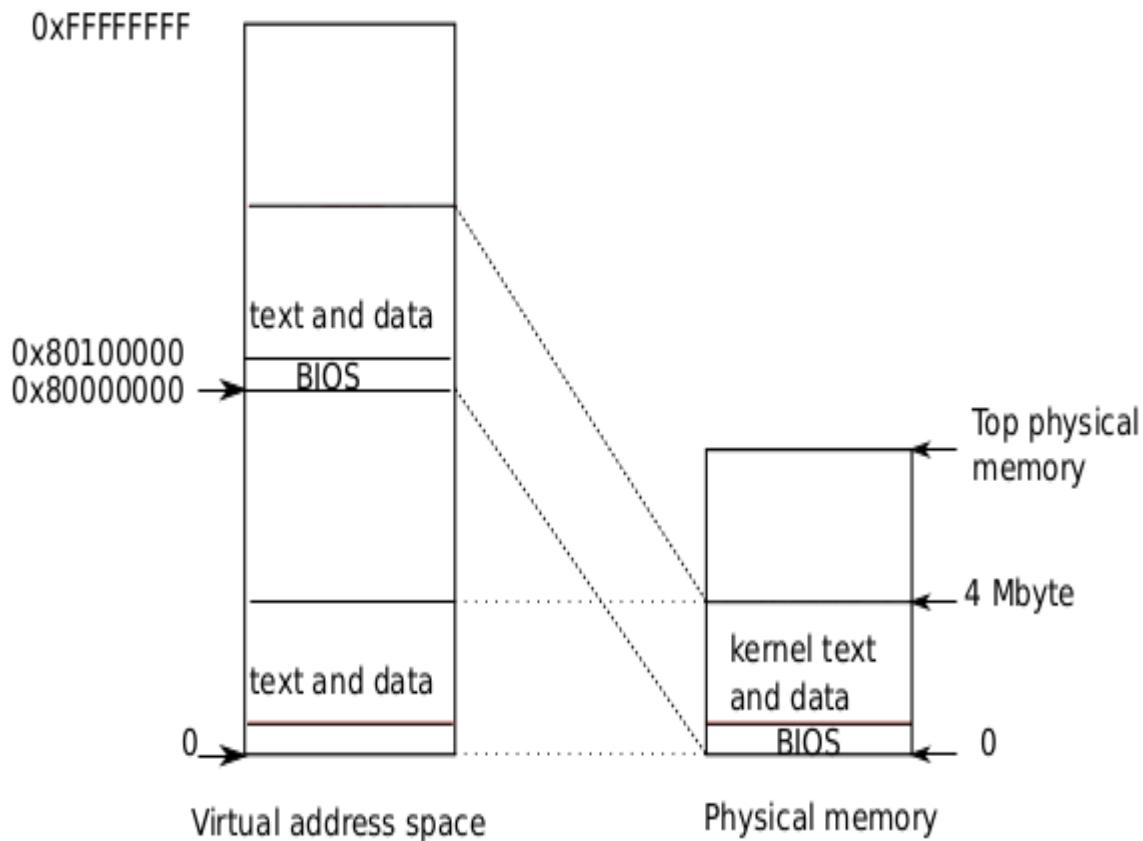
- **Address 0: code**
- **Then globals**
- **Then stack**
- **Then heap**
- **Each process's address space maps kernel's text, data also --> so that system calls run with these mappings**
- **Kernel code can directly access user memory now**

# Kernel mappings in user address space actual location of kernel



- Kernel is loaded at **0x100000** physical address
- PA 0 to 0x100000 is **BIOS and devices**
- Process's page table will map **VA 0x80000000** to **PA 0x00000** and **VA 0x80100000** to **0x100000**

# Kernel mappings in user address space actual location of kernel



- Kernel is not loaded at the PA 0x80000000 because some systems may not have that much memory
- 0x80000000 is called **KERNBASE** in xv6

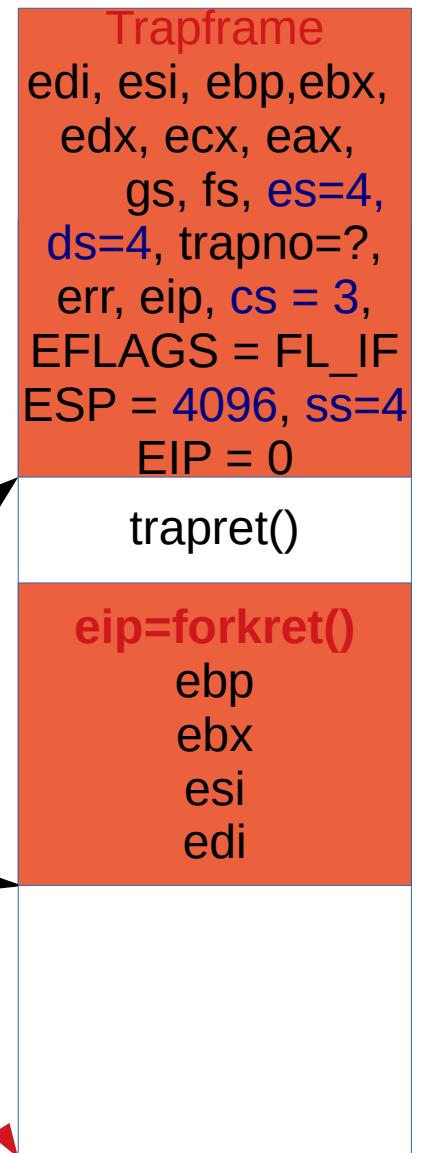
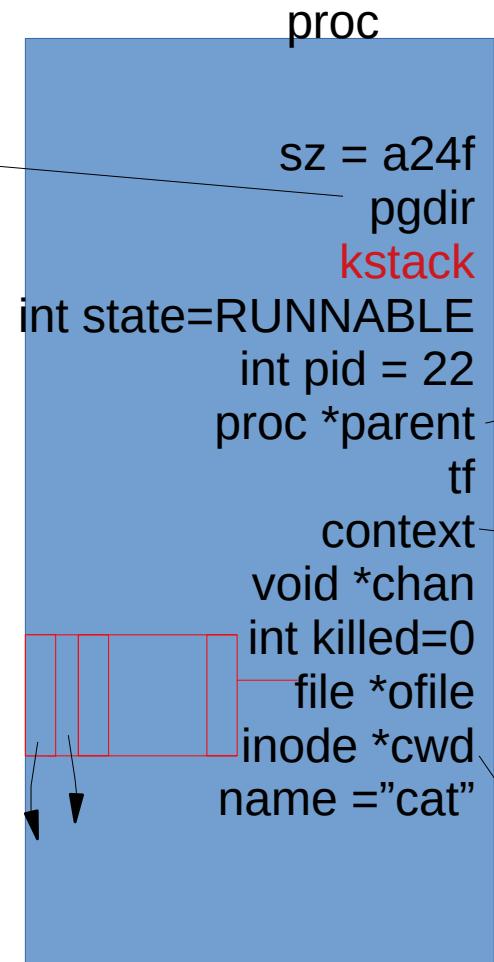
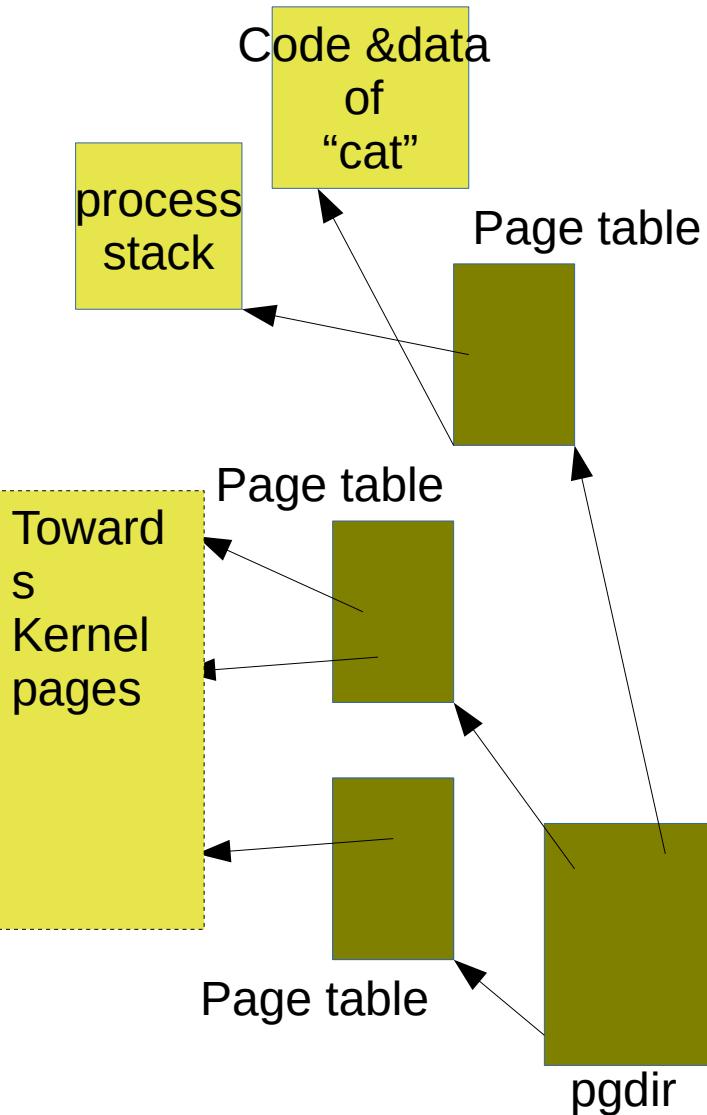
# Imp Concepts

- **A process has two stacks**
  - user stack: used when user code is running
  - kernel stack: used when kernel is running on behalf of a process
- **Note: there is a third stack also!**
  - The kernel stack used by the scheduler itself
  - Not a per process stack

# Struct proc

```
// Per-process state
struct proc {
    uint sz;                      // Size of process memory (bytes)
    pde_t* pgdir;                 // Page table
    char *kstack;                 // Bottom of kernel stack for this process
    enum procstate state;         // Process state. allocated, ready to run, running, waiting for I/O, or exiting.
    int pid;                      // Process ID
    struct proc *parent;          // Parent process
    struct trapframe *tf;         // Trap frame for current syscall
    struct context *context;      // swtch() here to run process. Process's context
    void *chan;                   // If non-zero, sleeping on chan. More when we discuss sleep, wakeup
    int killed;                   // If non-zero, have been killed
    struct file *ofile[NOFILE];   // Open files, used by open(), read(),...
    struct inode *cwd;            // Current directory, changed with "chdir()"
    char name[16];                // Process name (for debugging)
};
```

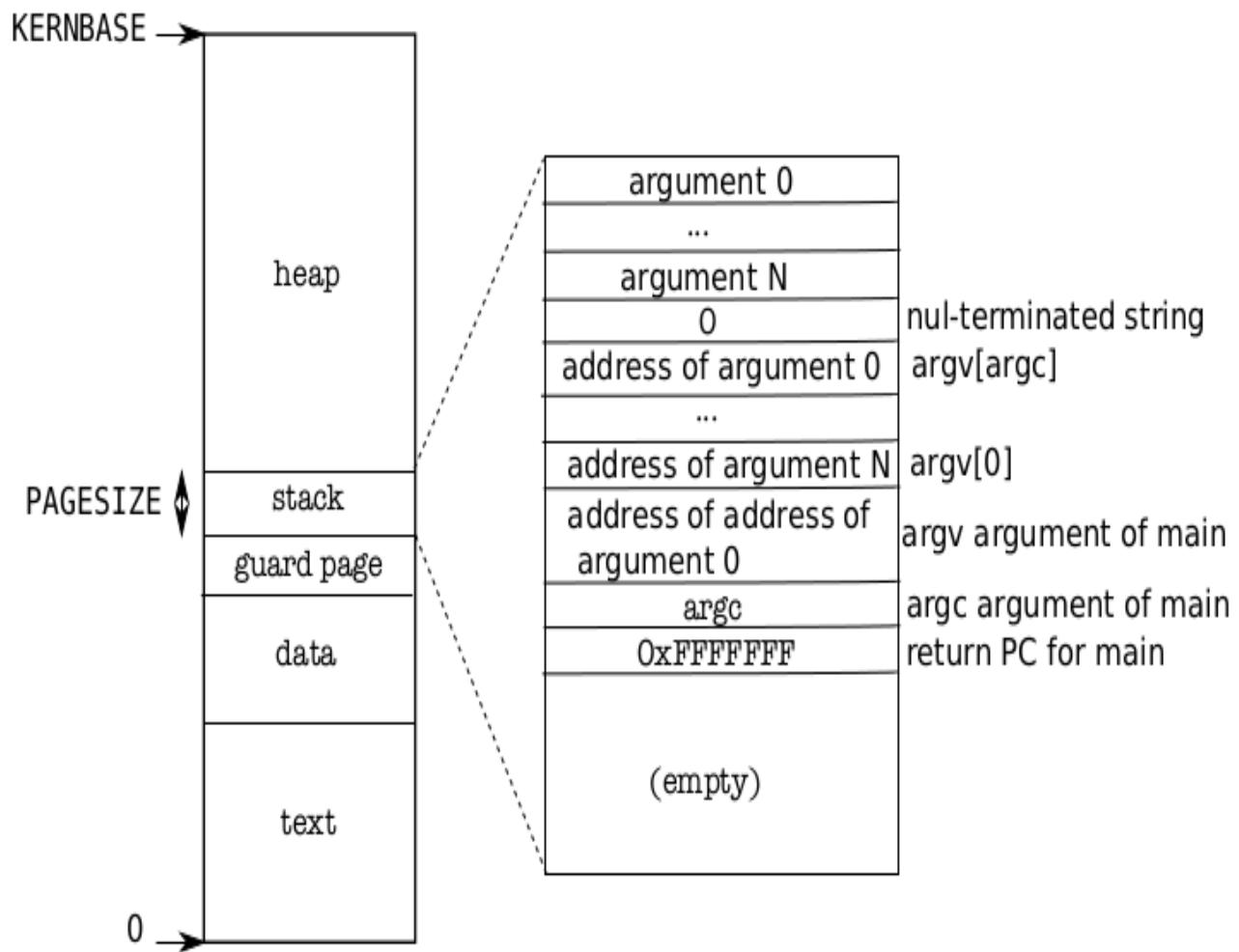
# struct proc diagram: Very imp!



$sz = \text{ELF-code-} \rightarrow \text{memsz}$  (includes data, check "ld -N"  
+  $2 \times 4096$  (for stack)

In use only when you are in kernel on a "trap" = interrupt/syscall. "tf" always used. trapret,forkret used during fork()

# Memory Layout of a user process



## Memory Layout of a user process

**After exec()**

**Note the argc, argv on stack**

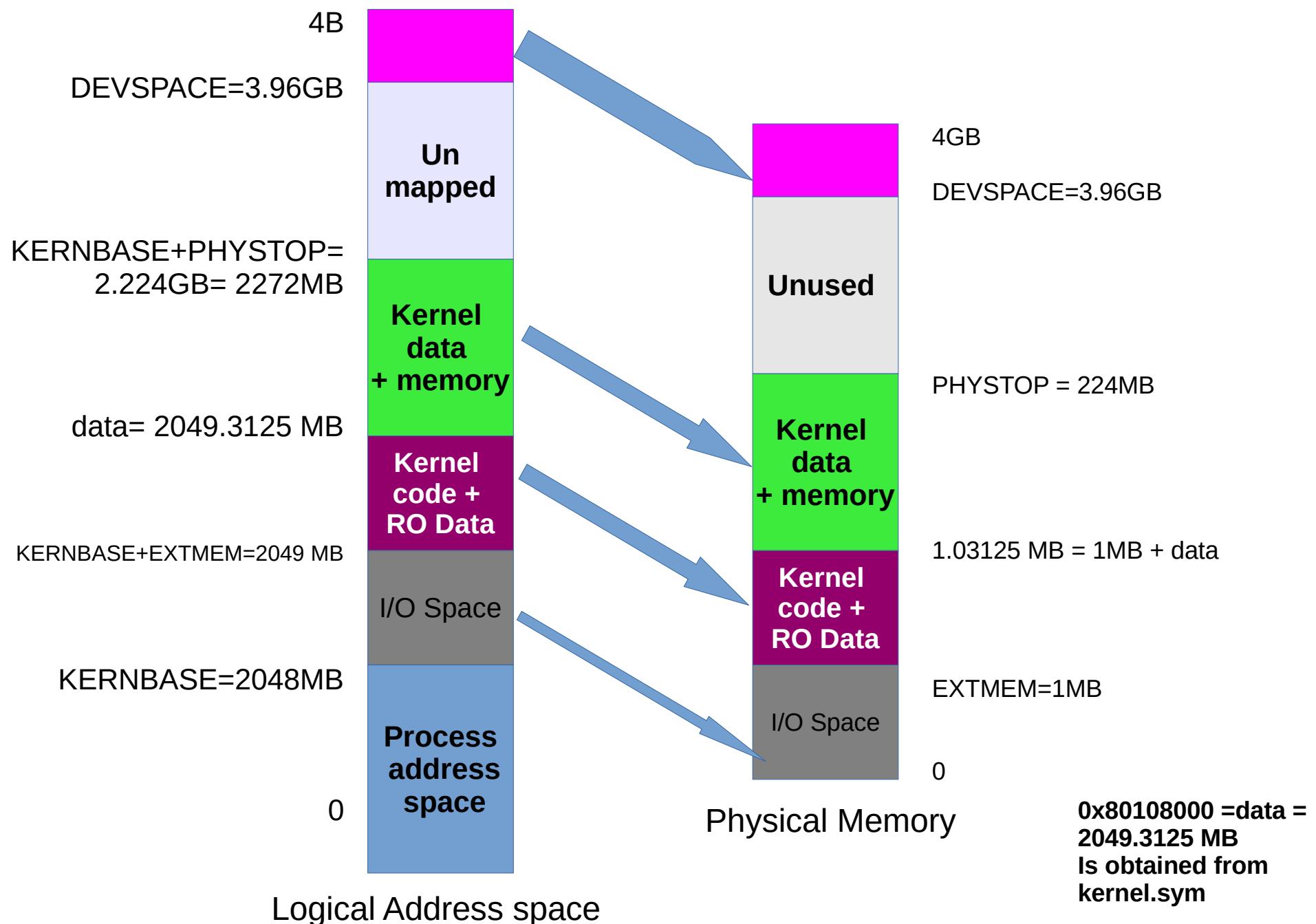
**The “guard page” is just a mapping in page table. No frame allocated. It’s marked as invalid. So if stack grows (due to many function calls), then OS will detect it with an exception**

# Handling Traps

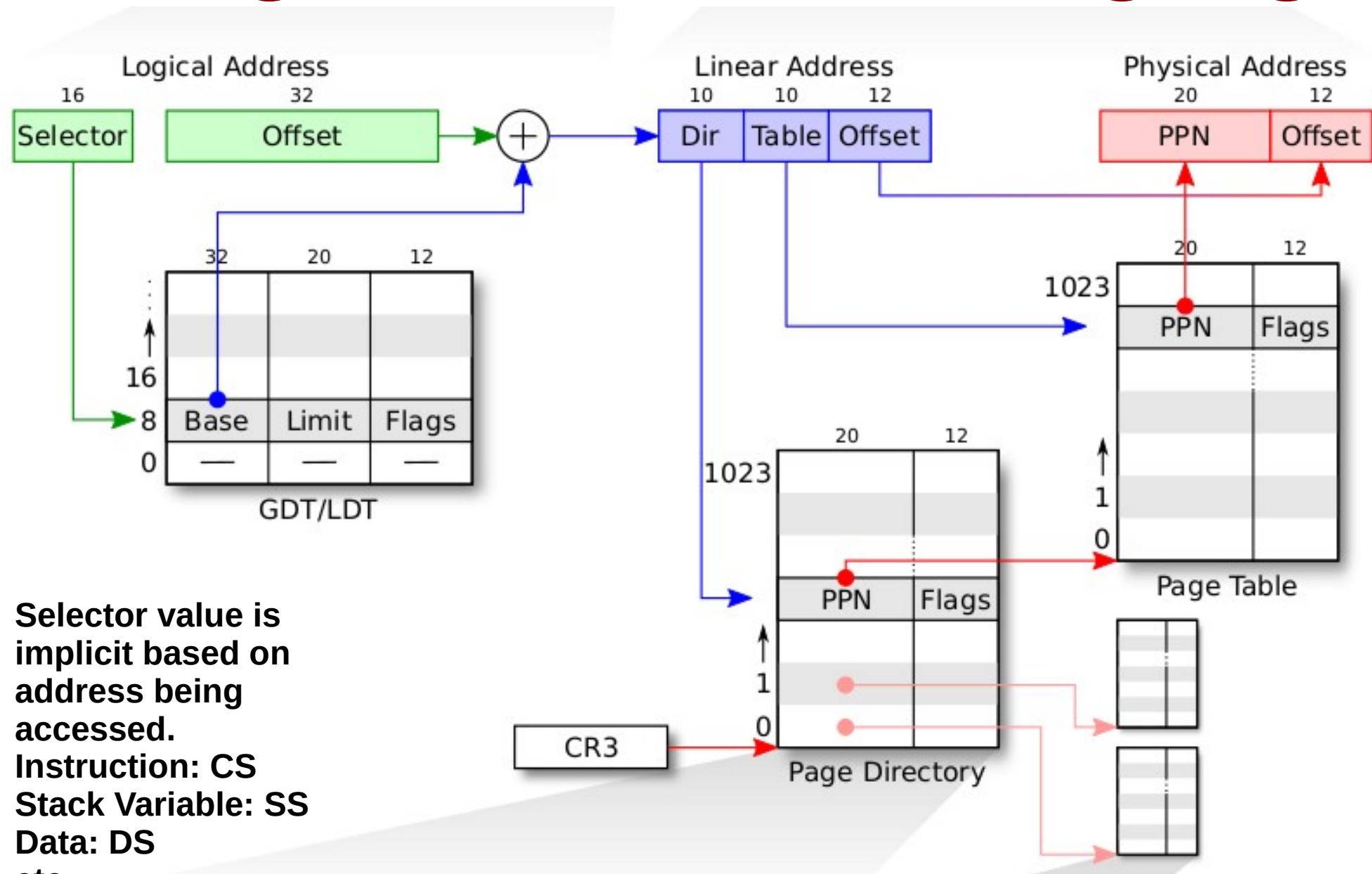
# Some basic steps

- **Xv6.img is created by “make”**
  - Contains bootsector, kernel code, kernel data
- **QEMU boots using xv6.img**
  - First it runs bootloader
  - Bootloader loads kernel of xv6 (from xv6.img)
  - Kernel starts running
- **Kernel running..**
  - Kernel calls main() of kernel (NOT a C application!) & Initializes:
    - memory management data structures
    - process data structures
    - file handling data structures
    - Multi-processors
    - Multi-processor data structures
    - Interrupt Descriptor Table
    - ...
  - Then creates init()
  - Init() fork-execs shell

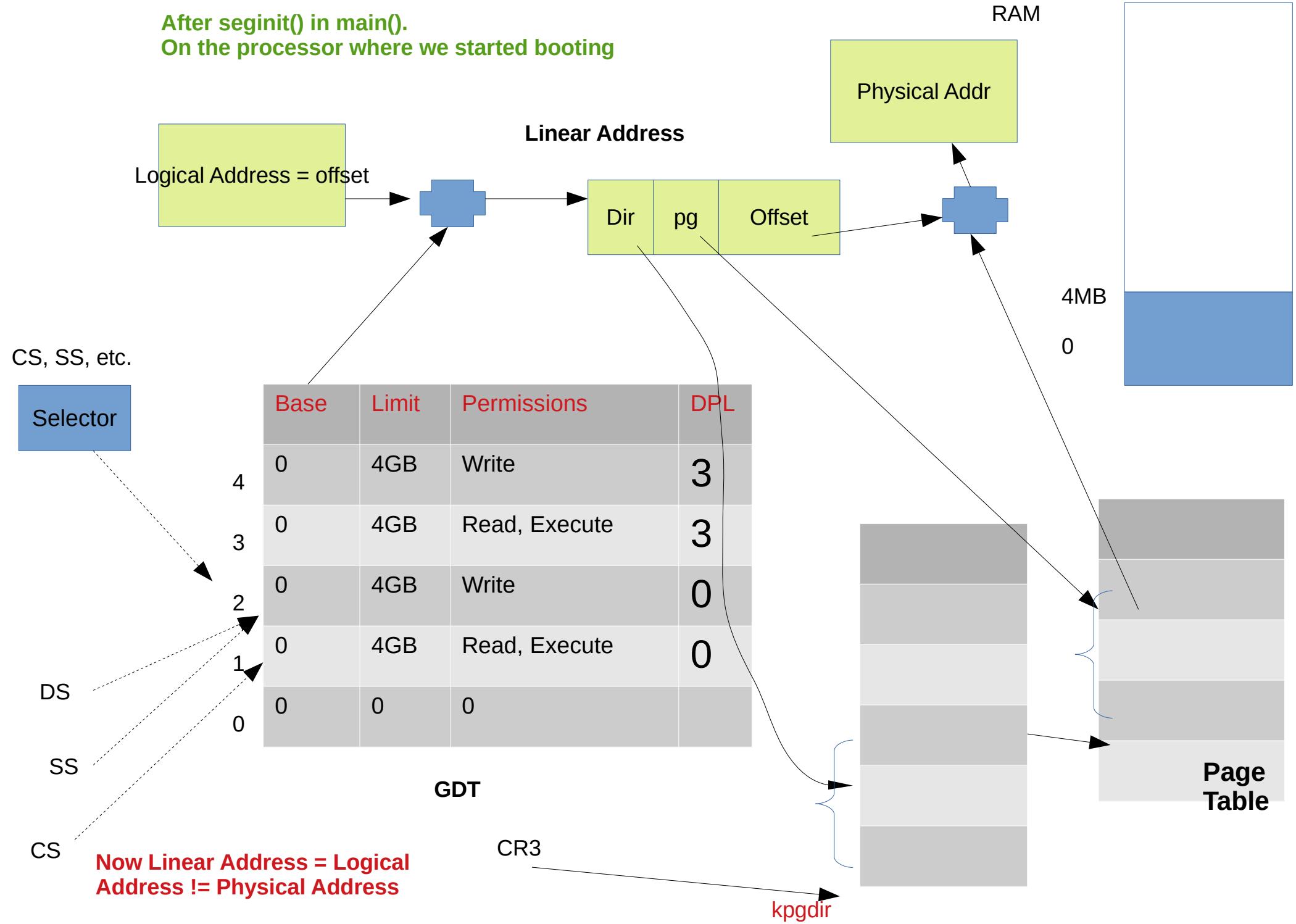
kmap[] mappings done in kvmalloc(). This shows segmentwise, entries are done in page directory and page table for corresponding VA->PA mappings



# Segmentation + Paging



After seginit() in main().  
On the processor where we started booting



# Handling traps

- **Transition from user mode to kernel mode**
  - On a system call
  - On a hardware interrupt
  - User program doing illegal work (exception)
- **Actions needed, particularly w.r.t. to hardware interrupts**
  - Change to kernel mode & switch to kernel stack
  - Kernel to work with devices, if needed
  - Kernel to understand interface of device

# Handling traps

- **Actions needed on a trap**
  - Save the processor's registers (**context**) for future use
  - Set up the system to run kernel code (**kernel context**) on kernel stack
  - Start kernel in appropriate place (**sys call, intr handler, etc**)
  - Kernel to get all info related to event (**which block I/O done?, which sys call called, which process did exception and what type, get arguments to system call, etc**)

# Privilege level

- The x86 has 4 protection levels, numbered 0 (most privilege) to 3 (least privilege).
- In practice, most operating systems use only 2 levels: 0 and 3, which are then called kernel mode and user mode, respectively.
- The current privilege level with which the x86 executes instructions is stored in %cs register, in the field CPL.



TI      Table index (0=GDT, 1=LDT)  
RPL    Requester privilege level

# Privilege level

- Changes automatically on
  - “int” instruction
  - hardware interrupt
  - exception
- Changes back on
  - iret
- “int” 10 --> makes 10<sup>th</sup> hardware interrupt. S/w interrupt can be used to ‘create hardware interrupt’
- Xv6 uses “int 64” for actual system calls

# Interrupt Descriptor Table (IDT)

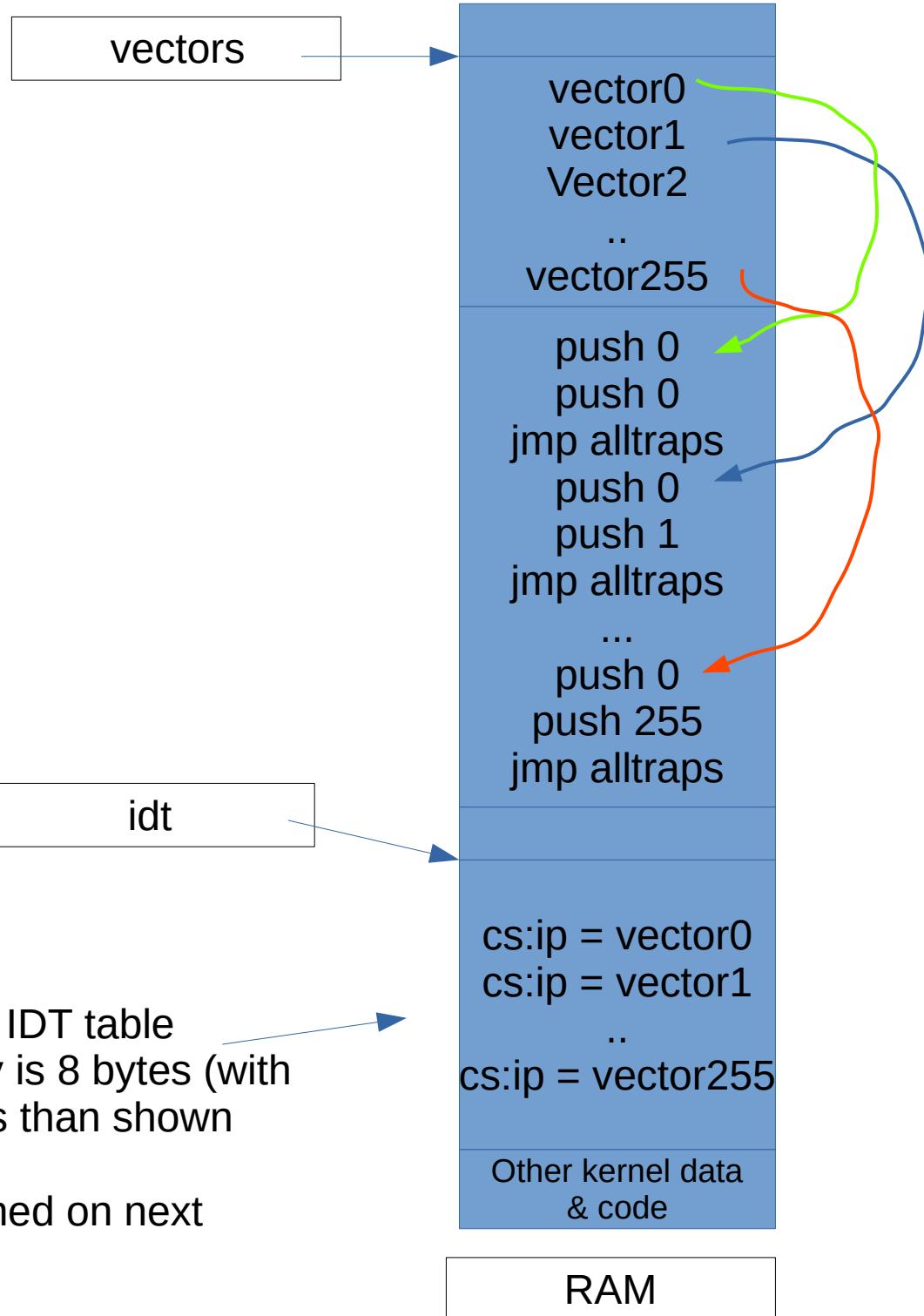
- **IDT defines interrupt handlers**
- **Has 256 entries**
  - each giving the %cs and %eip to be used when handling the corresponding interrupt.
- **Mapping**
  - Interrupts 0-31 are defined for software exceptions, like divide errors or attempts to access invalid memory addresses.
  - Xv6 maps the 32 hardware interrupts to the range 32-63
  - and uses interrupt 64 as the system call interrupt

# IDT setup done by tvinit() function

The array of “vectors”  
And the code of  
“push, ..jmp”  
Is part of kernel image  
(xv6.img)

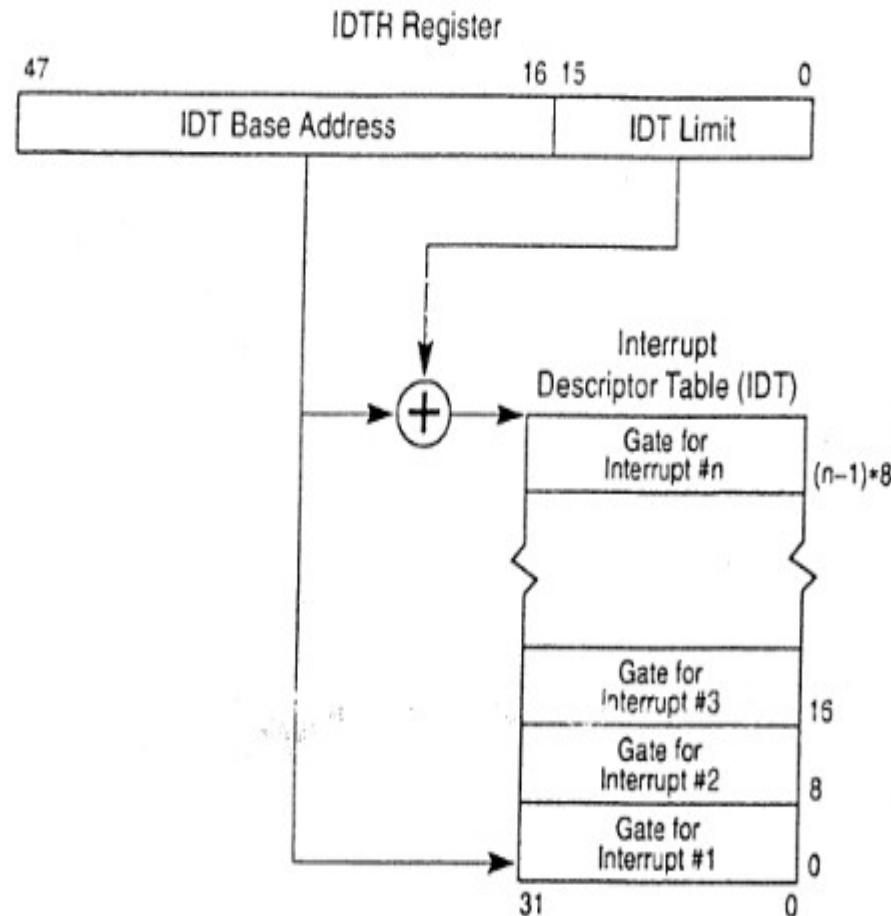
The tvinit() is called during  
kernel initialization

This is the IDT table  
Each entry is 8 bytes (with  
more fields than shown  
here)  
as mentioned on next  
slides



RAM

# IDTR and IDT



IDT is in RAM

IDTR is in CPU

# Interrupt Descriptor Table (IDT) entries (in RAM)

```
// Gate descriptors for interrupts and traps

struct gatedesc {

    uint off_15_0 : 16; // low 16 bits of offset in segment
    uint cs : 16; // code segment selector
    uint args : 5; // # args, 0 for interrupt/trap gates
    uint rsv1 : 3; // reserved(should be zero I guess)
    uint type : 4; // type(STS_{IG32,TG32})
    uint s : 1; // must be 0 (system)
    uint dpl : 2; //descriptor(new) privilege level
    uint p : 1; // Present
    uint off_31_16 : 16; // high bits of offset in segment
};
```

# Setting IDT entries

```
void
tvinit(void)
{
    int i;
    for(i = 0; i < 256; i++)
        SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
             vectors[T_SYSCALL], DPL_USER);
    /* value 1 in second argument --> don't disable
    interrupts
       * DPL_USER means that processes can raise
    this interrupt. */
    initlock(&tickslock, "time");
}
```

# Setting IDT entries

```
#define SETGATE(gate, istrap, sel, off, d)
{
    (gate).off_15_0 = (uint)(off) & 0xffff;
    (gate).cs = (sel);
    (gate).args = 0;
    (gate).rsv1 = 0;
    (gate).type = (istrap) ? STS_TG32 : STS_IG32;
    (gate).s = 0;
    (gate).dpl = (d);
    (gate).p = 1;
    (gate).off_31_16 = (uint)(off) >> 16;
}
```

# Setting IDT entries

**Vectors.S**

```
# generated by vectors.pl -  
do not edit  
  
# handlers  
  
.globl alltraps  
  
.globl vector0  
  
vector0:  
    pushl $0  
    pushl $0  
    jmp alltraps  
  
.globl vector1  
  
vector1:  
    pushl $0  
    pushl $1  
  
    jmp alltraps
```

**trapasm.S**

```
#include "mmu.h"  
  
# vectors.S sends all traps  
here.  
  
.globl alltraps  
  
alltraps:  
    # Build trap frame.  
    pushl %ds  
    pushl %es  
    pushl %fs  
    pushl %gs  
    Pushal  
  
    ....
```

**How will interrupts be handled?**

# On int instruction/interrupt the CPU does this:

- Fetch the n'th descriptor from the IDT, where n is the argument of int. (IDTR->idt[n])
- Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
  - Temporarily save %esp and %ss in CPU-internal registers, but only if the target segment selector's PL < CPL.
    - Switching from user mode to kernel mode. Hence save user code's SS and ESP
- Load %ss and %esp from a task segment descriptor.
  - Stack changes to kernel stack now. TS descriptor is on GDT, index given by TR register. See switchuvm()
- Push %ss. // optional
- Push %esp. // optional (also changes ss,esp using TSS)
- Push %eflags.
- Push %cs.
- Push %eip.
- Clear the IF bit in %eflags, but only on an interrupt.
- Set %cs and %eip to the values in the descriptor.

# After “int” ‘s job is done

- **IDT was already set**
  - Remember vectors.S
- **So jump to 64<sup>th</sup> entry in vector’s vector64:**

```
pushl $0
pushl $64
jmp alltraps
```
- **So now stack has ss, esp,eflags, cs, eip, 0 (for error code), 64**
- **Next run alltraps from trapasm.S**

```
# Build trap frame.  
pushl %ds  
pushl %es  
pushl %fs  
pushl %gs  
pushal // push all gen purpose  
regs  
# Set up data segments.  
movw $(SEG_KDATA<<3), %ax  
movw %ax, %ds  
movw %ax, %es  
# Call trap(tf), where tf=%esp  
pushl %esp # first arg to trap()  
call trap  
addl $4, %esp
```

## alltraps:

- Now stack contains
- **ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi**
  - This is the struct trapframe !
  - So the kernel stack now contains the trapframe
  - Trapframe is a part of kernel stack

```
void  
trap(struct trapframe *tf)  
{  
    if(tf->trapno == T_SYSCALL){  
        if(myproc()->killed)  
            exit();  
        myproc()->tf = tf;  
        syscall();  
        if(myproc()->killed)  
            exit();  
        return;  
    }  
    switch(tf->trapno){  
        ....
```

## trap()

- Argument is trapframe
- In alltraps
  - Before “call trap”, there was “push %esp” and stack had the trapframe
  - Remember calling convention --> when a function is called, the stack contains the arguments in reverse order (here only 1 arg)

# trap()

- **Has a switch**
  - `switch(tf->trapno)`
  - Q: who set this trapno?
- **Depending on the type of trap**
  - Call interrupt handler
- Timer
  - `wakeup(&ticks)`
- IDE: disk interrupt
  - `Ideintr()`
- KBD
  - `KbdINTR()`
- COM1
  - `UatINTR()`
- If Timer
  - Call `yield()` -- calls `sched()`
- If process was killed (how is that done?)
  - Call `exit()`!

# when trap() returns

- #Back in alltraps

```
call trap
```

```
addl $4, %esp
```

```
# Return falls through to trapret...
```

```
.globl trapret
```

```
trapret:
```

```
popal
```

```
popl %gs
```

```
popl %fs
```

```
popl %es
```

```
popl %ds
```

```
addl $0x8, %esp # trapno and errcode
```

```
iret
```

```
.
```

- Stack had (trapframe)
  - ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi, esp
- add \$4 %esp
  - esp
- popal
  - eax, ecx, edx, ebx, oesp, ebp, esi, edi
- Then gs, fs, es, ds
- add \$0x8, %esp
  - 0 (for error code), 64
- iret
  - ss, esp,eflags, cs, eip,

# Basics of X86 architecture

Abhijit A M  
[abhijit.comp@coep.ac.in](mailto:abhijit.comp@coep.ac.in)

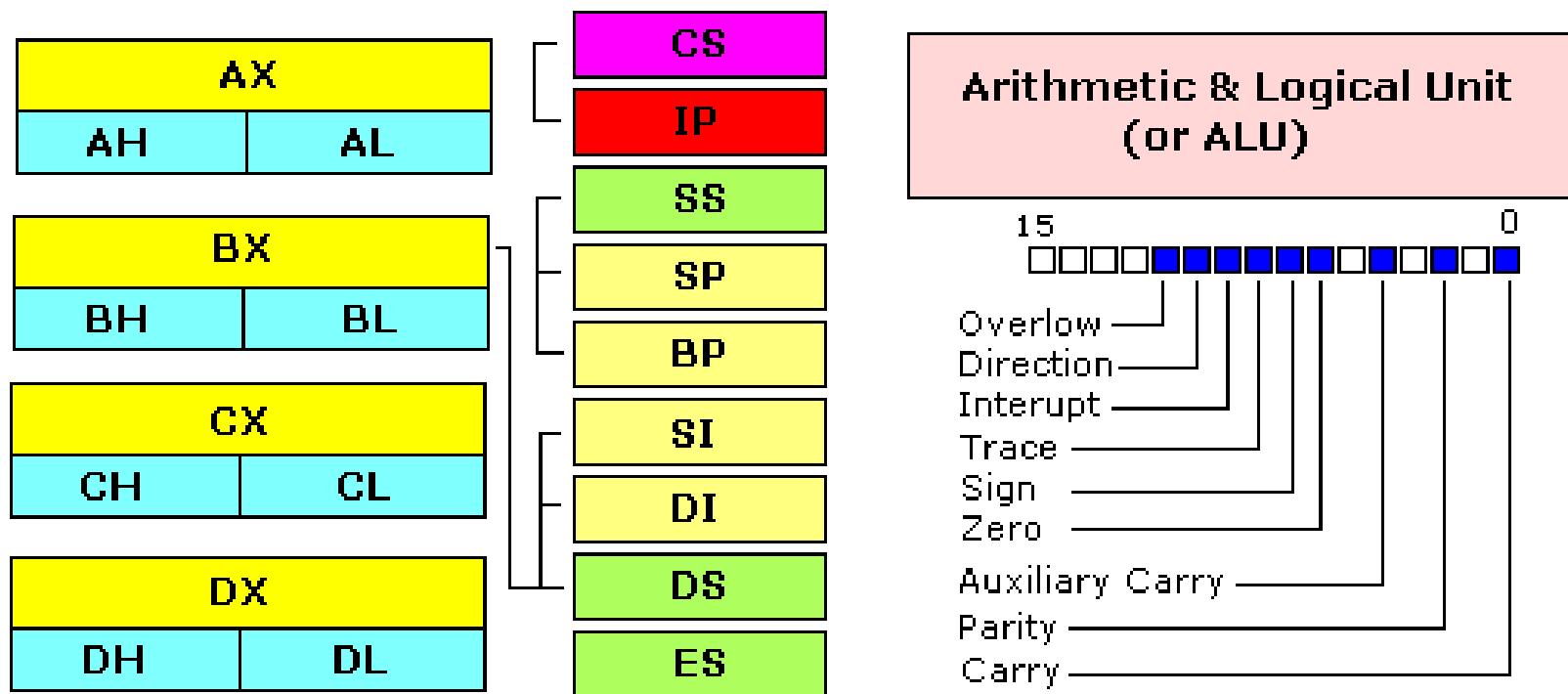
Credits: Notes by Prof. Sorav Bansal, <https://www.felixcloutier.com/x86/>, Intel Documentation

# A processor typically has

- **integer registers and their execution unit**
- **floating-point/vector registers and execution unit(s)**
- **memory management unit (MMU)**
- **multiprocessor/multicore: local interrupt controller (APIC)**
- **etc**

# 8086: 16 bit CPU (precursor to x86 family)

## Central Processing Unit (or CPU)



# 8086 Registers

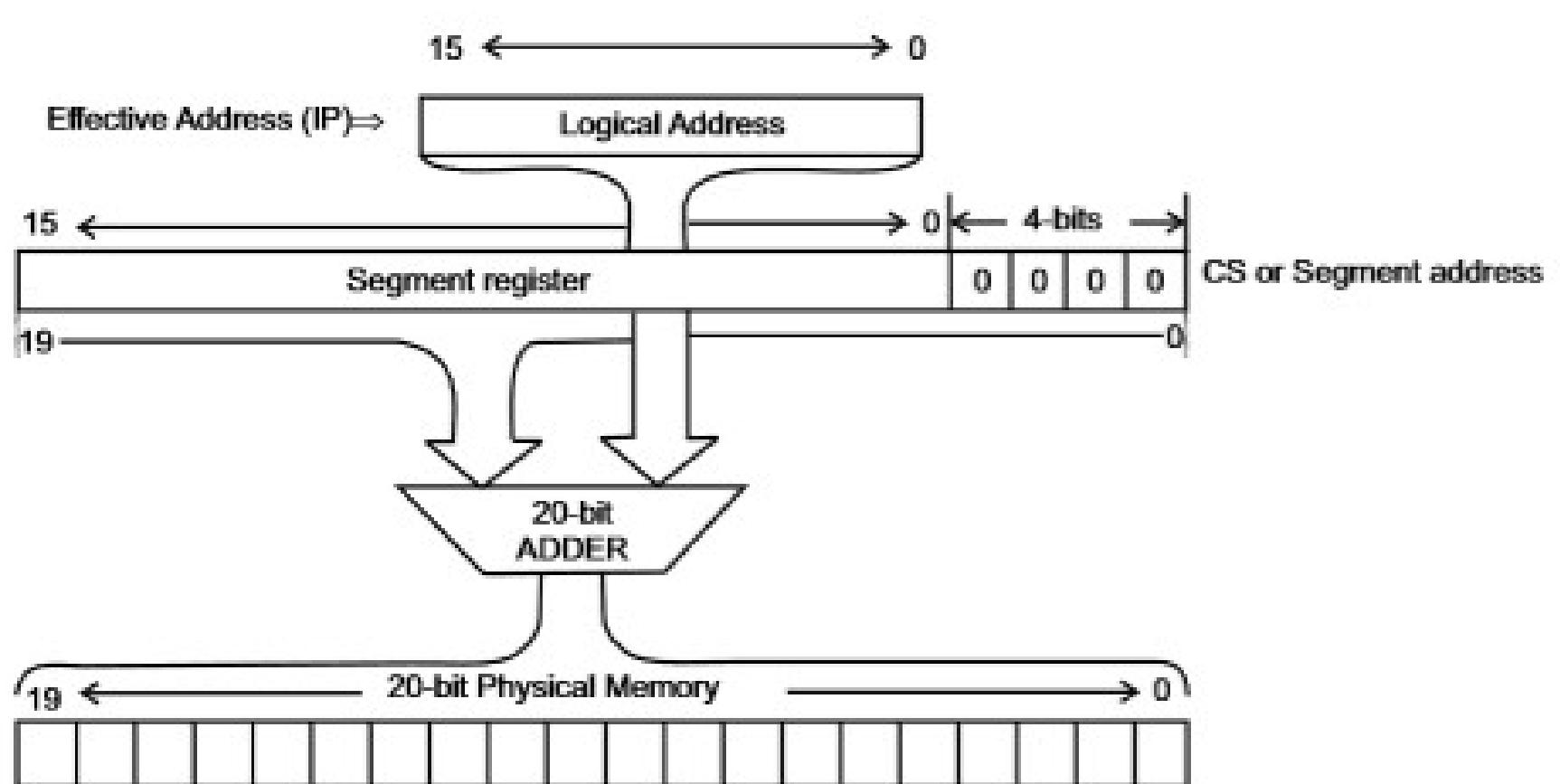
- **16 bit ought to be enough ! (?)**
- **General purpose registers**
  - four 16-bit data registers: AX, BX, CX, DX
  - each in two 8-bit halves, e.g. AH and AL
- **Registers for memory addressing**
  - SP, BP, SI, DI : 16 bit
  - SP stack pointer, BP base pointer, SI Source Index, DI Destination Index
  - IP instruction Pointer
  - Addressable memory:  $2^{16} = 64 \text{ kb}$

# 8086 address extension

- 8086 has 20-bit physical addresses ==> 1 MB AS
- the extra four bits usually come from a 16-bit "segment register":
  - CS - code segment, for fetches via IP
  - SS - stack segment, for load/store via SP and BP
  - DS - data segment, for load/store via other registers
  - ES - another data segment, destination for string operations
- %cs:%ip full address: %cs \* 16 + %ip is actual address that goes on bus. virtual to physical translation is
  - $pa = va + seg * 16$
- e.g. set CS = 4096 to execute starting at 65536

# 8086 address extension

- Extending ‘address space’ with the help of MMU



# FLAGS register

- **FLAGS - various condition codes: whether last arithmetic operation**
  - overflowed
  - was positive/negative
  - was [not] zero
  - carry/borrow on add/subtract
  - etc.
- **whether interrupts are enabled**
- **direction of data copy instructions**
- **Uses in JP, JN, J[N]Z, J[N]C, J[N]O ...**

# 32 bit 80386

- boots in 16-bit mode, then on running particular instructions switches to 32-bit mode (protected mode)
  - For backward compatibility, INTEL continued this. The CPU starts in 16 bit mode, called real mode
  - If particular instructions are not executed (may be in boot loader) all processors will continue in real mode
- registers are 32 bits wide, called EAX rather than AX
  - EAX EBX ECX EDX
  - ESP EBP ESI EDI
- operands and addresses 32-bit in 32-bit mode,
  - e.g. ADD does 32-bit arithmetic
- Segment registers are 16 bit: CS, SS, DS, etc.

# 32 bit 80386

- Still possible to access 16 bit registers using AX or BX
  - Specifix **coding of machine instructions** to tell whether operands are 16 or 32 bit
  - prefixes **0x66/0x67** toggle between 16-bit and 32-bit **operands/addresses** respectively
  - in 32-bit mode, MOVW is expressed as 0x66 MOVW
  - the .code32 in bootasm.S tells assembler to generate 0x66 for e.g. MOVW
- **80386 also changed segments and added paged memory...**

# **Summary of registers in 80386 (32 bit)**

- **General registers**
  - 32 bits : EAX EBX ECX EDX
  - 16 bits : AX BX CX DX
  - 8 bits : AH AL BH BL CH CL DH DL

# Summary of registers in 80386

- **Expected usage of Segment Registers**
  - **CS:** Holds the Code segment in which your program runs.
  - **DS :** Holds the Data segment that your program accesses.
  - **ES,FS,GS :** These are extra segment registers
  - **SS :** Holds the Stack segment your program uses.
- **All 16 bit**

# Summary of registers in 80386

- For a typical register, the corresponding segment is used. Pairs of Indexes & pointers (Segment & Registers)
  - CS:EIP
    - Code Segment: Index Pointer
    - E.g. mov \$32, %eax ==> The code of move instruction uses CS: EIP
  - SS:ESP
    - Stack Segment: ESP
    - E.g. push \$32 ==> The \$32 will be pushed on stack. Using SS: ESP address
- SS:EBP
  - Stack Segment: EBP ==> mov (%ebp), %eax ==> for accessing (%ebp) SS: EBP address will be used
- DS:ESI , ES: EDI .
  - ESI: Extended Source Index, EDI: Extended Destination Index
- The EFLAGS register for flags

# X86 Assembly Code

- **Syntax**
  - Intel syntax: op dst, src (Intel manuals!)
  - AT&T (gcc/gas) syntax: op src, dst (xv6)
  - uses b, w, l suffix on instructions to specify size of operands
- **Operands are registers, constant, memory via register, memory via constant**

# Examples of X86 instructions

AT&T syntax	"C"-ish equivalent	Operands
movl %eax, %edx	edx = eax;	register mode
movl \$0x123, %edx	edx = 0x123;	immediate
movl 0x123, %edx	edx = *(int32_t*)0x123;	direct
movl (%ebx), %edx	edx = *(int32_t*)ebx;	indirect
movl 4(%ebx), %edx	edx = *(int32_t*)(ebx+4)	displaced

# Instructions suffix/prefix

`mov %eax, %ebx # 32 bit data`

`movw %ax, %bx # move 16 bit data`

`mov %ax, %bx # ax is 16 bit, so equivalent  
to movw`

`mov $123, 0x123 # Ambigious`

`movw $123, 0x123 # correct, move 16 bit data`

# Types of Instructions

- **data movement**
  - MOV, PUSH, POP, ...
- **Arithmetic**
  - TEST, SHL, ADD, AND, ...
- **i/o**
  - IN, OUT, ...
- **Control**
  - JMP, JZ, JNZ, CALL, RET
- **String**
  - REP MOVSB, ...
- **System**
  - IRET, INT

# Interrupt handling

# Privilege levels

- The x86 has 4 protection levels, numbered 0 (most privilege) to 3 (least privilege).
- In practice, most operating systems use only 2 levels: 0 and 3, which are then called kernel mode and user mode, respectively.
- The current privilege level with which the x86 executes instructions is stored in CPL field inside %cs register

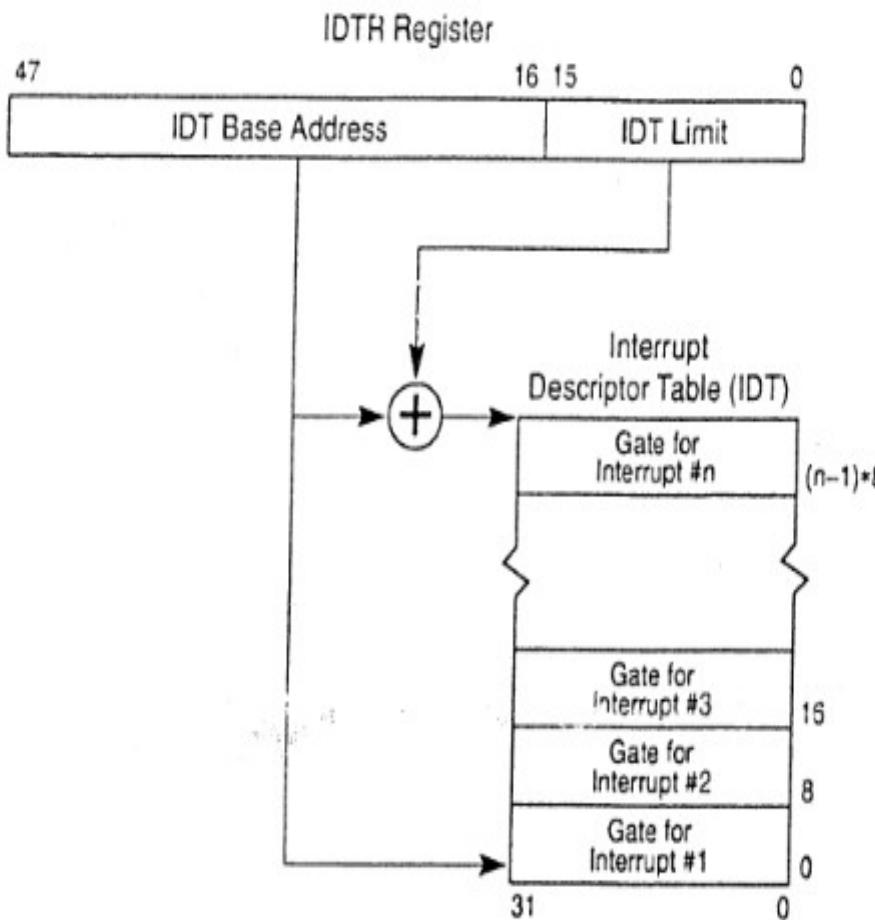
# Privilege levels

- Changes automatically on
  - “int” instruction
  - hardware interrupt
  - exception
- Changes back on
  - iret
- “int” 10 --> makes 10<sup>th</sup> hardware interrupt. S/w interrupt can be used to ‘create hardware interrupt’
- Xv6 uses “int 64” for actual system calls

# Interrupt Descriptor Table (IDT)

- **IDT is an in memory table. IDT defines interrupt handlers**
- **Has 256 entries**
  - each giving the %cs and %eip to be used when handling the corresponding interrupt.
- **Interrupts 0-31 are defined for software exceptions, like divide errors or attempts to access invalid memory addresses.**
  - Xv6 maps the 32 hardware interrupts to the range 32-63 and uses interrupt 64 as the system call interrupt

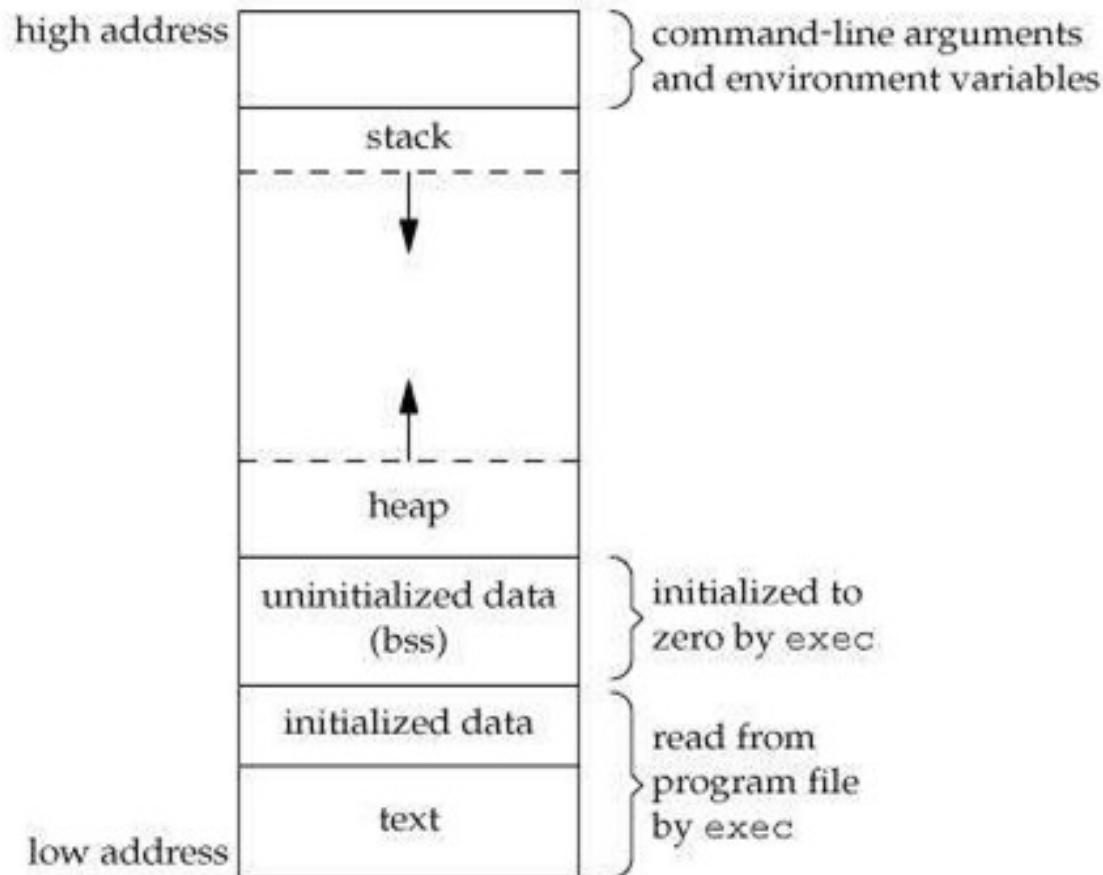
# IDTR and IDT



- An entry in IDT is called a “gate”
- IDTR is a CPU register, IDT is in memory table

# Memory Management in x86

# Memory Layout of a C Program



text, data, bss are also present in the ELF file  
stack and heap are not present in ELF file  
WHY?

**Text: object code of the program**

**Data: Global variables + (local/global) Static variables**

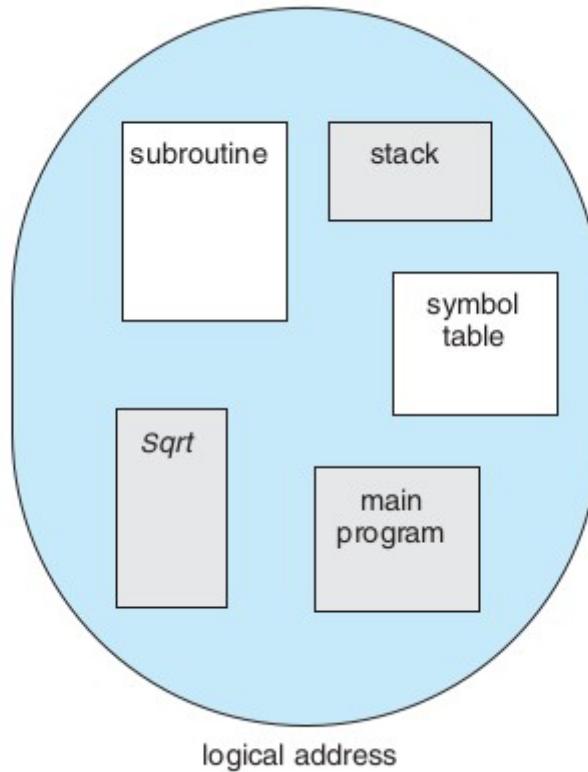
**BSS: Uninitialized data, globals that were not initialized**

**Stack: Region for allocating local variables, function parameters, return values**

**Heap: Region for use by malloc(), free()**

**Arguments, Environment variables: Initialized by kernel (during exec)**

# Why the Segment:Offset Pairs In x86 ?



Segmentation  
Compiler's view of the program

Compiler generates object code **assuming** that different memory regions corresponding to the program are different “segments” in memory

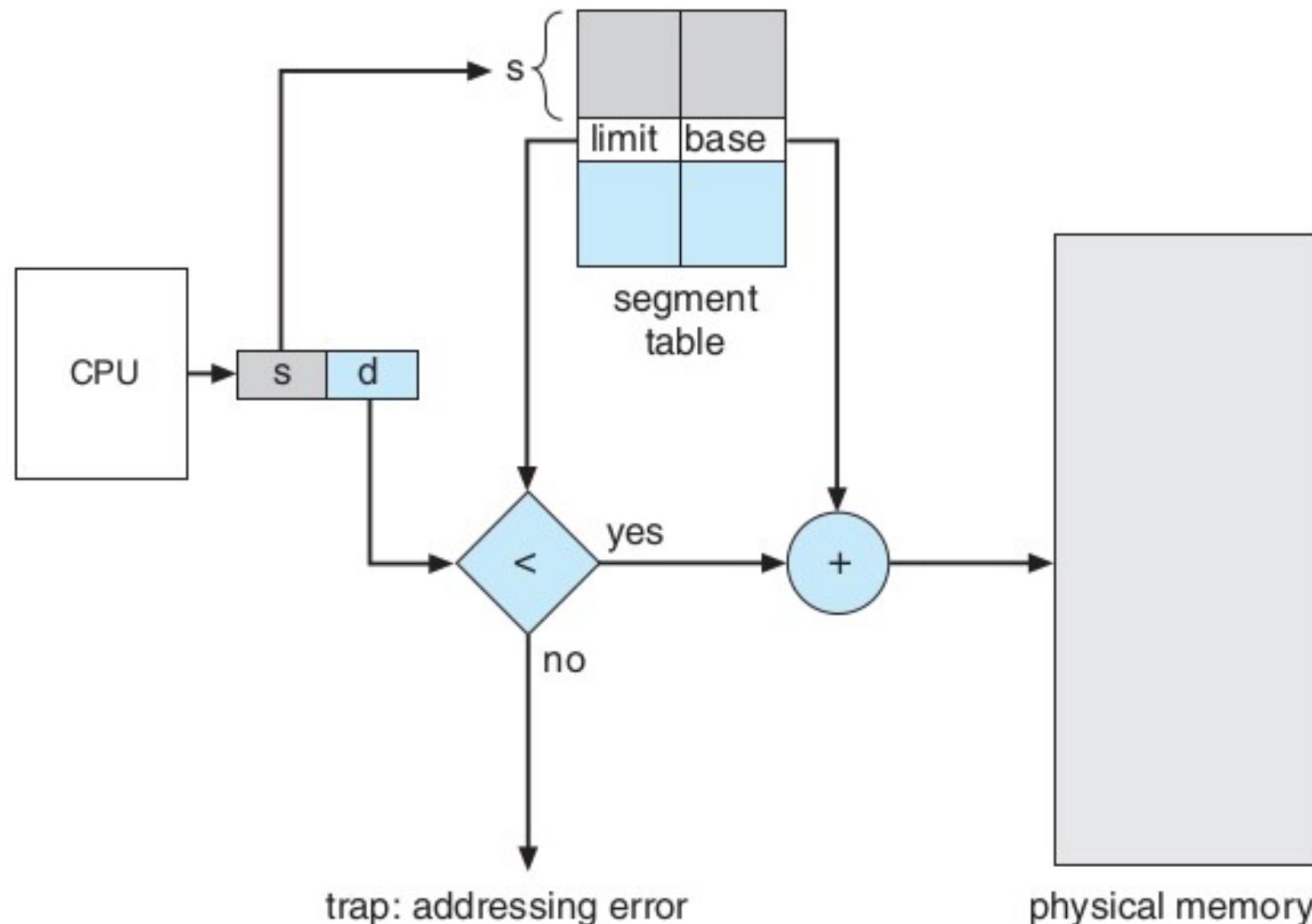
Segment: a continuous chunk

Segment register gives the location of the chunk,  
index/offset register gives the offset

E.g. in CS:IP CS gives the location of the segment, IP gives the index in it

# Segmentation

## Address Translation, general idea (not x86)



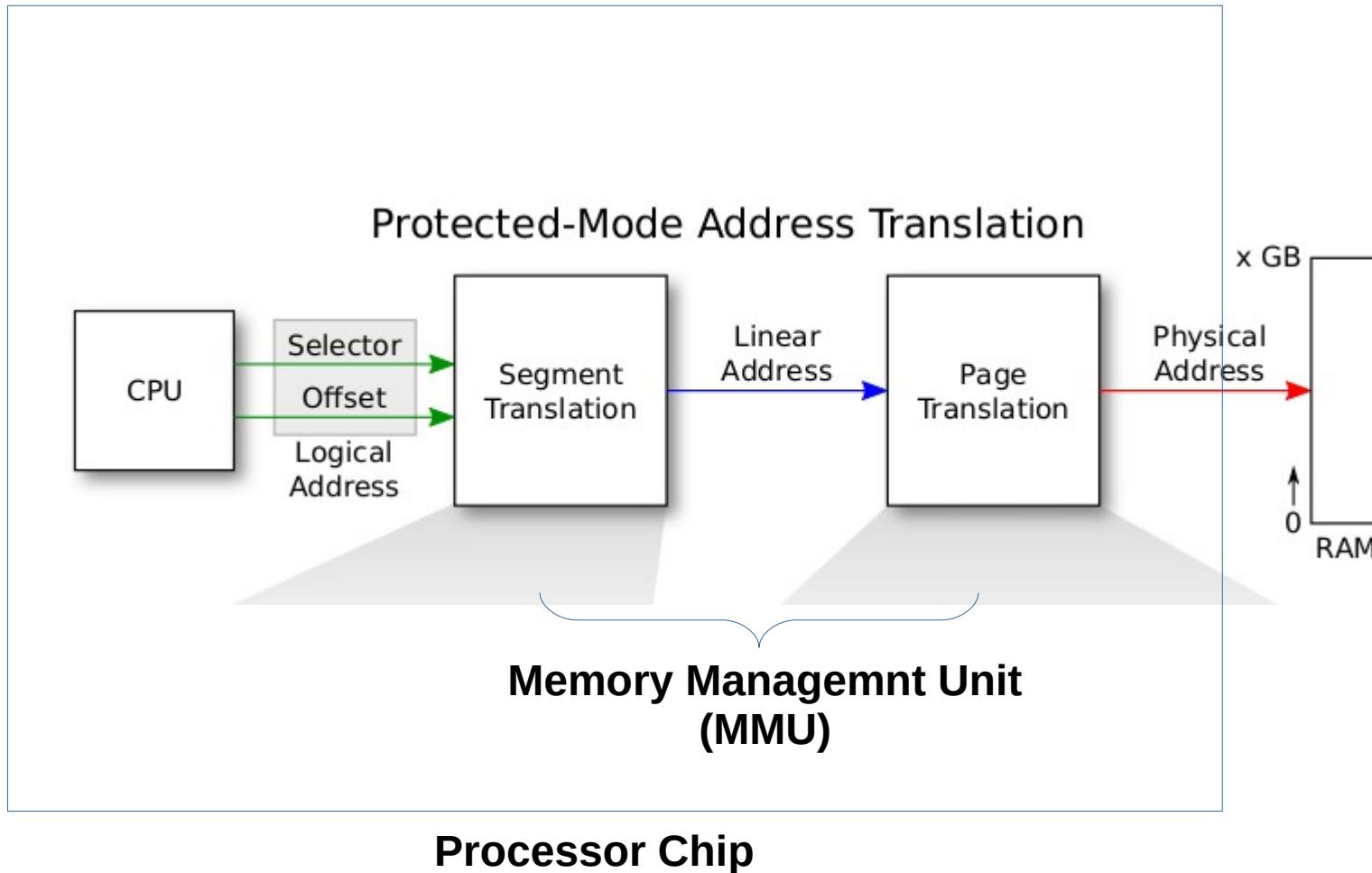
# Real mode and protected mode

- **Beware: note the same as user mode and kernel mode!**
- **Backward compatibility was desired by Intel**
  - 80286 was 32 bit, 8086 was 16 bit
  - Binary encoding for 80286 was different, registers were 32 bit, etc; also more speed, different memory management hardware, etc.
  - But still they wanted their customers to keep running earlier object code on new processor
  - So they ensured that the processor boots up as if it was 16 bit 8086/8088. **REAL MODE!**
  - On running particular machine instruction sequence, the CPU will change to 32 bit . **PROTECTED MODE!**

# More on real mode

- **addresses in real mode always correspond to real locations in memory.**
  - Memory address calculation done by MMU in real mode:  
 $\text{segment} \times 16 + \text{offset}$
- **no support for memory protection, multitasking, or code privilege levels (user/kernel mode!)**
- **May use the BIOS routines or OS routines**
- **DOS like OS were written when PCs were in the era of 8088/real-mode.**
  - Single tasking systems

# X86 address : protected mode address translation



# X86 segmentation

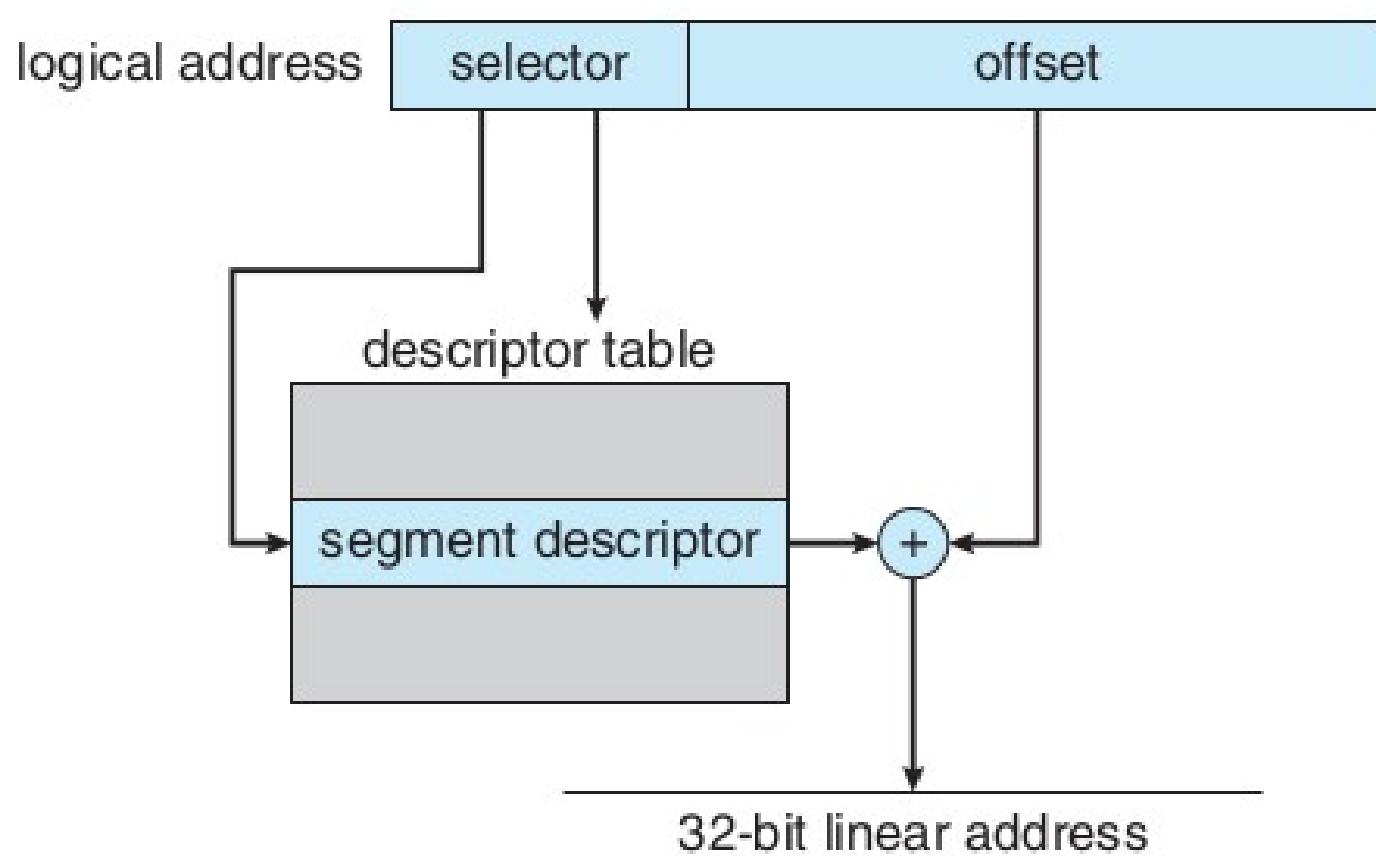
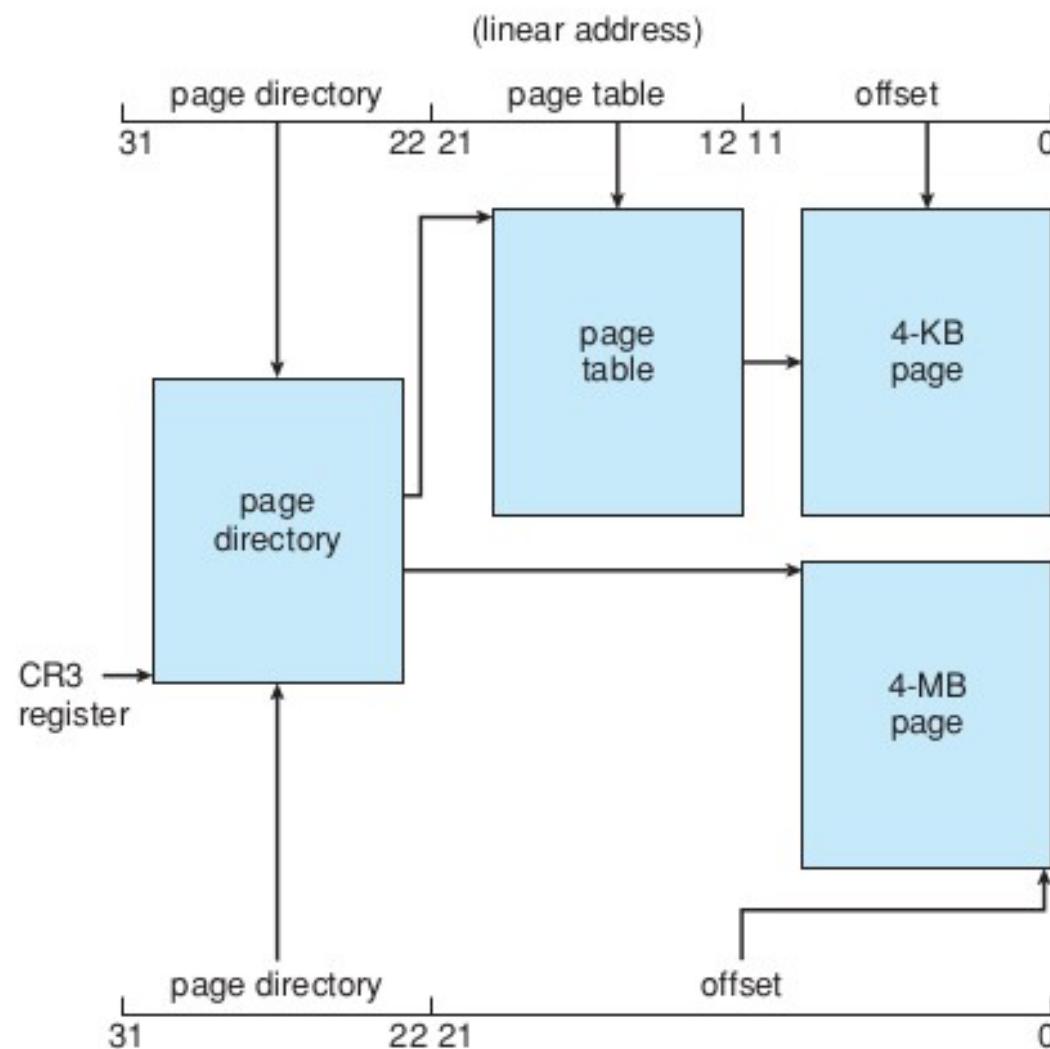


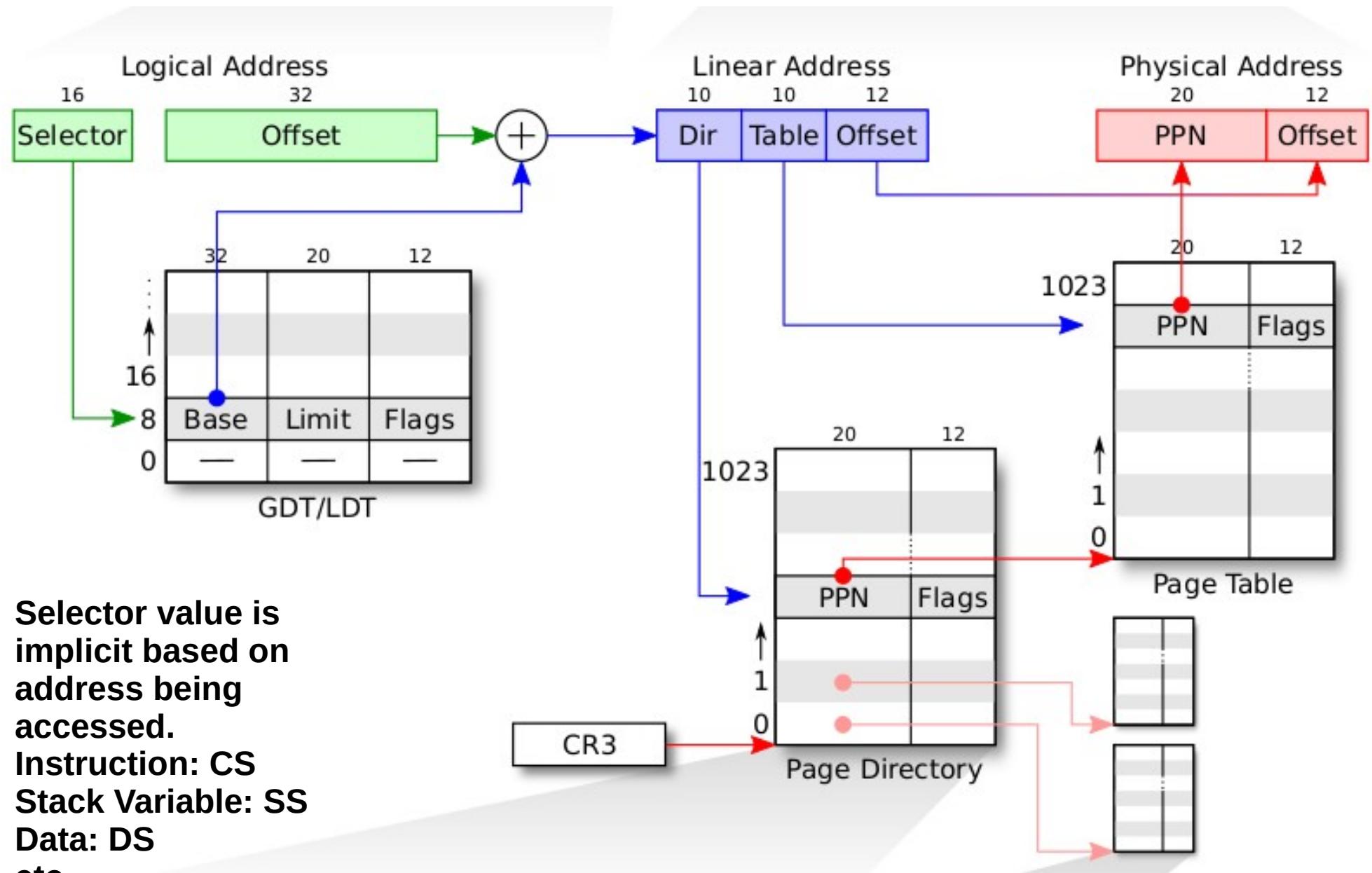
Figure 8.22 IA-32 segmentation.

# X86 paging

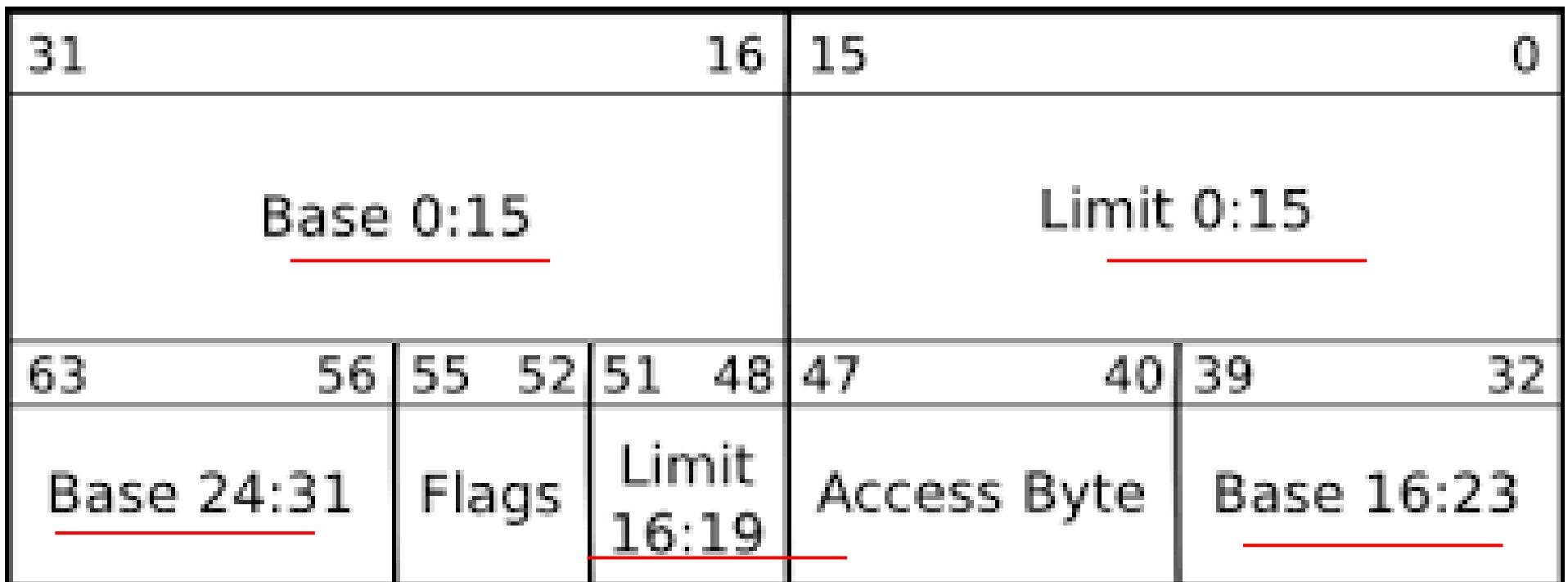


**Figure 8.23** Paging in the IA-32 architecture.

# Segmentation + Paging



# GDT Entry



This is an in Memory entry. The diagram shows the format.

# Page Directory Entry (PDE) Page Table Entry (PTE)

31

Page table physical page number	12	11	10	9	8	7	6	5	4	3	2	1	0
	A V L	G S	P 0	A	C D	W T	U D	W T	U D	W T	U D	W T	P

PDE

P Present

W Writable

U User

WT 1=Write-through, 0=Write-back

CD Cache disabled

A Accessed

D Dirty

PS Page size (0=4KB, 1=4MB)

PAT Page table attribute index

G Global page

AVL Available for system use

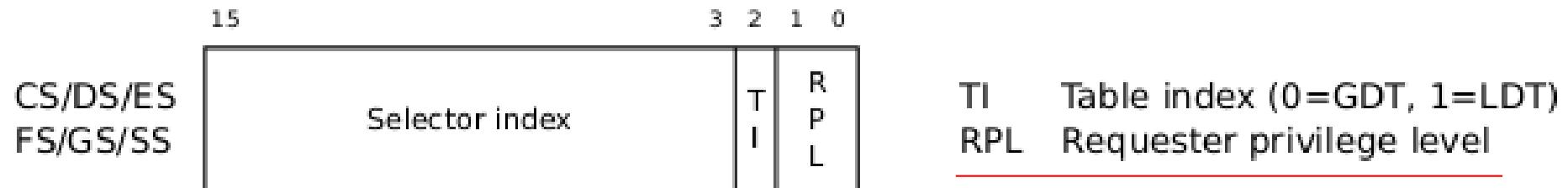
31

Physical page number	12	11	10	9	8	7	6	5	4	3	2	1	0
	A V L	G T	P A	D A	A D	C D	W T	U D	W T	U D	W T	U D	P

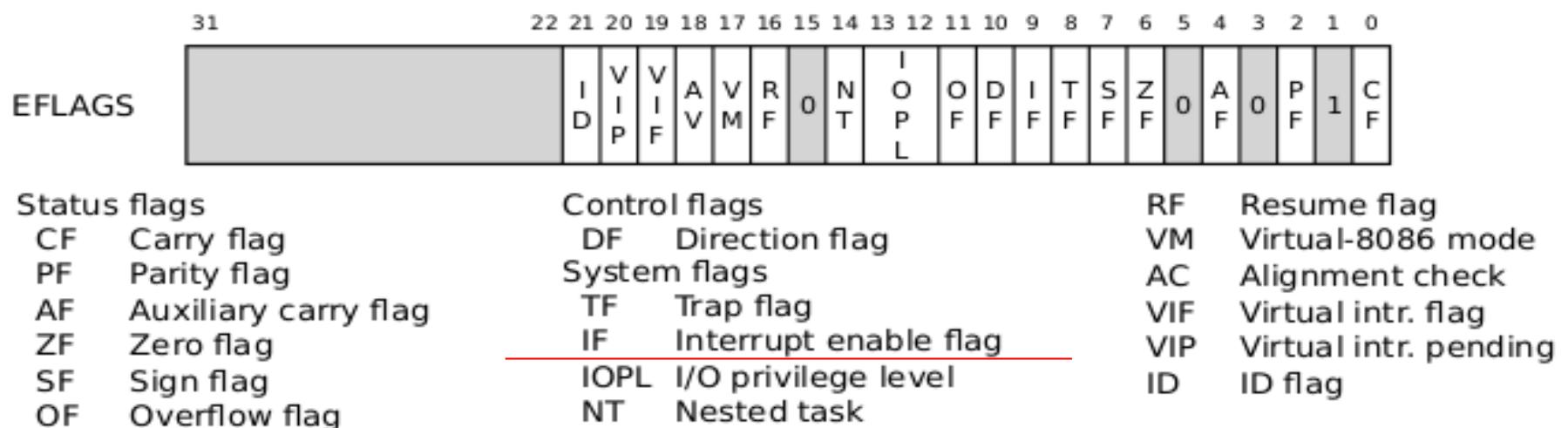
PTE

These are in memory entries. The diagrams show the format of each entry.

# Segment selector



# EFLAGS register



# CR0

CR0	31 30 29 28	19 18 17 16 15	6 5 4 3 2 1 0		
	P G C D N W	A M W P	N E T S E M P P E		
PE	Protection enabled	ET	Extension type	NW	Not write-through
MP	Monitor coprocessor	NE	Numeric error	CD	Cache disable
EM	Emulation	WP	Write protect	PG	Paging
TS	Task switched	AM	Alignment mask		

PG: Paging enabled or not

WP: Write protection on/off

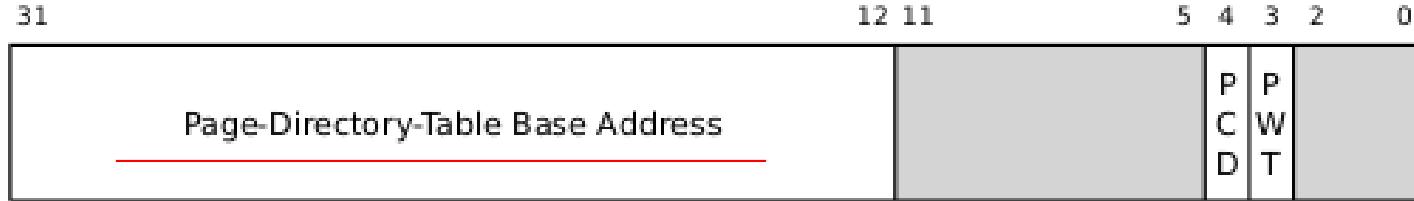
PE: Protection Enabled --> protected mode.

# CR2

CR2	31	0
		Page fault virtual address

# CR3

CR3



PWT Page-level writes transparent

PCT Page-level cache disable

# CR4

CR4



VME Virtual-8086 mode extensions

PVI Protected-mode virtual interrupts

TSD Time stamp disable

DE Debugging extensions

PSE Page size extensions

PAE Physical-address extension

MCE Machine check enable

PGE Page-global enable

PCE Performance counter enable

OSFXSR OS FXSAVE/FXRSTOR support

OSXMM- OS unmasked exception support

EXCPT