

Module 1

1. Introduction

A computer is made up of hardware and software. With its software, a computer can store, process, and retrieve information; play music and videos; send e-mail, search the Internet; and engage in many other valuable activities to earn its keep. Computer software can be divided roughly into two kinds: system programs, which manage the operation of the computer itself, and application programs, which perform the actual work the user wants. The most fundamental system program is the **operating system**, whose job is to control all the computer's resources and provide a base upon which the application programs can be written. OS acts as an interface between user and computer.

Many years ago it became abundantly clear that some way had to be found to shield programmers from the complexity of the hardware. The way that has evolved gradually is to put a layer of software on top of the bare hardware, to manage all parts of the system, and present the user with an interface or **virtual machine** that is easier to understand and program. This layer of software is the operating system.

The placement of the operating system is shown in [Fig. 1-1](#). At the bottom is the hardware, which, in many cases, is itself composed of two or more levels (or layers). The lowest level contains physical devices, consisting of integrated circuit chips, wires, power supplies, cathode ray tubes, and similar physical devices.

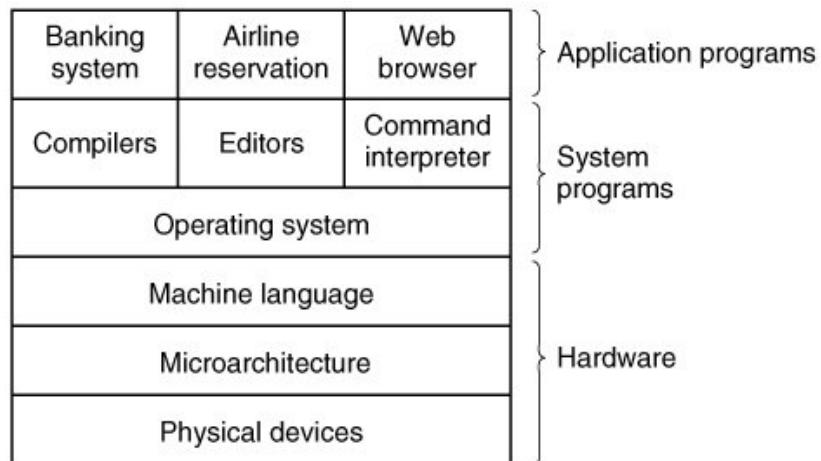


Figure 1-1. A computer system consists of hardware, system programs, and application programs.

Next comes the **microarchitecture level**, in which the physical devices are grouped together to form functional units. Typically this level contains some registers internal to the CPU (Central Processing Unit) and a data path containing an arithmetic logic unit.

To make these units to work we give instructions in 1's and 0's. this is called machine language understandable by computer.

A major function of the operating system is to hide all this complexity and give the programmer a more convenient set of instructions to work with. On top of the operating system is the rest of the system software. Here we find the command interpreter (shell), window systems, compilers, editors, and similar application-independent programs.

The operating system is (usually) that portion of the software that runs in **kernel mode** or **supervisor mode**. It is protected from user tampering by the hardware. Compilers and editors run in **user mode**

That said, in many systems there are programs that run in user mode but which help the operating system or perform privileged functions. For example, there is often a program that allows users to change their passwords. This program is not part of the operating system and does not run in kernel mode, but it clearly carries out a sensitive function and has to be protected in a special way.

Finally, above the system programs come the application programs. These programs are purchased (or written by) the users to solve their particular problems, such as word processing, spreadsheets, engineering calculations, or storing information in a database.

1.1 The Operating System as an Extended Machine

As mentioned earlier, the **architecture** (instruction set, memory organization, I/O, and bus structure) of most computers at the machine language level is primitive and awkward to program, especially for input/output. To make this point more concrete, let us briefly look at how floppy disk I/O is done using the NEC PD765 compatible controller chips used on many Intel-based personal computers. The PD765 has 16 commands, each specified by loading between 1 and 9 bytes into a device register. These commands are for reading and writing data, moving the disk arm, and formatting tracks, as well as initializing, sensing, resetting, and recalibrating the controller and the drives.

The most basic commands are **read** and **write**, each of which requires 13 parameters, packed into 9 bytes. These parameters specify such items as the address of the disk block to be read, the number of sectors per track, the recording mode used on the physical medium, the intersector gap spacing, and what to do with a deleted-data-address-mark. When the operation is completed, the controller chip returns 23 status and error fields packed into 7 bytes. As if this were not enough, the floppy disk programmer must also be constantly aware of whether the motor is on or off. If the motor is off, it must be turned on (with a long startup delay) before data can be read or written. The motor cannot be left on too long, however, or the floppy disk will wear out. The programmer is thus forced to deal with the trade-off between long startup delays versus wearing out floppy disks (and losing the data on them).

Without going into the *real* details, it should be clear that the average programmer

probably does not want to get too intimately involved with the programming of floppy disks (or hard disks, which are just as complex and quite different). Instead, what the programmer wants is a simple, high-level abstraction to deal with. In the case of disks, a typical abstraction would be that the disk contains a collection of named files. Each file can be opened for reading or writing, then read or written, and finally closed. Details such as whether or not recording should use modified frequency modulation and what the current state of the motor is should not appear in the abstraction presented to the user.

The program that hides the truth about the hardware from the programmer and presents a nice, simple view of named files that can be read and written is, of course, the operating system. Just as the operating system shields the programmer from the disk hardware and presents a simple file-oriented interface, it also conceals a lot of unpleasant business concerning interrupts, timers,

memory management, and other low-level features. In each case, the abstraction offered by the operating system is simpler and easier to use than that offered by the underlying hardware.

In this view, the function of the operating system is to present the user with the equivalent of an **extended machine** or **virtual machine** that is easier to program than the underlying hardware. The operating system provides a variety of services that programs can obtain using special instructions called system calls.

1.2 The Operating System as a Resource Manager

The concept of the operating system as primarily providing its users with a convenient interface is a top-down view. An alternative, bottom-up, view holds that the operating system is there to manage all the pieces of a complex system. Modern computers consist of processors, memories, timers, disks, mice, network interfaces, printers, and a wide variety of other devices. In the

alternative view, the job of the operating system is to provide for an orderly and controlled allocation of the processors, memories, and I/O devices among the various programs competing for them.

Imagine what would happen if three programs running on some computer all tried to print their output simultaneously on the same printer. The first few lines of printout might be from program 1, the next few from program 2, then some from program 3, and so forth. The operating system can do buffering all the output destined for the printer on the disk. When one program is finished, the operating system can then copy its output from the disk file where it has been stored to the printer, while at the same time the other program can continue generating more output, oblivious to the fact that the output is not really going to the printer.

When a computer has multiple users, the need for managing and protecting the memory, I/O devices, and other resources is even greater, since the users might otherwise interfere with one another. In addition, users often need to share not only hardware, but information such as files, databases, etc. as well. In short, this view of the operating system holds that its primary task is to keep track of who is

using which resource, to grant resource requests, to account for usage, and to mediate conflicting requests from different programs and users.

Resource management includes sharing resources in two ways: in time and in space. When a resource is time multiplexed, different programs or users take turns using it. First one of them gets to use the resource, then another, and so on. For example, with only one CPU and multiple programs that want to run on it, the operating system first allocates the CPU to one program, then after it has run long enough, another one gets to use the CPU, then another, and then eventually the first one again. Determining how the resource is time multiplexed, who goes next and for how long is the task of the operating system. Another example of time multiplexing is sharing the printer. When multiple print jobs are queued up for printing on a single printer, a decision has to be made about which one is to be printed next.

The other kind of multiplexing is space multiplexing. Instead of the customers taking turns, each one gets part of the resource. For example, main memory is normally divided up among several running programs, so each one can be resident at the same time, for example, in order to take turns using the CPU. Assuming there is enough memory to hold multiple programs, it is more efficient to hold several programs in memory at once rather than give one of them all of it, especially if it only needs a small fraction of the total. Another resource that is space multiplexed is the hard disk. In many systems a single disk can hold files from many users at the same time. Allocating disk space and keeping track of who is using which disk blocks is a typical operating system resource management task.

1.3 History of Operating Systems

Operating systems have been evolving through the years. The first true digital computer was designed by the English mathematician Charles Babbage (1792-1871). Although Babbage spent most of his life and fortune trying to build his "analytical engine," he never got it working properly because it was purely mechanical, and the technology of his day could not produce the required wheels, gears, and cogs to the high precision that he needed. Needless to say, the analytical engine did not have an operating system.

As an interesting historical aside, Babbage realized that he would need software for his analytical engine, so he hired a young woman named Ada Lovelace, who was the daughter of the famed British poet Lord Byron, as the world's first programmer. The programming language Ada was named after her.

1.3.1 The First Generation (1945-55) Vacuum Tubes and Plug boards

After Babbage's unsuccessful efforts, little progress was made in constructing digital computers until World War II. Around the mid-1940s, John von Neumann and others succeeded in building calculating engines. The first ones used mechanical relays but were very slow, with cycle times measured in seconds. Relays were later replaced by vacuum tubes. These machines were enormous,

filling up entire rooms with tens of thousands of vacuum tubes, but they were still millions of times slower than even the cheapest personal computers available today.

In these early days, a single group of people designed, built, programmed, operated, and maintained each machine. All programming was done in absolute machine language, often by wiring up plug boards to control the machine's basic functions. Programming languages were unknown. Operating systems were unheard of. The usual mode of operation was for the programmer to sign up for a block of time on the signup sheet on the wall, then come down to the machine room, insert his or her plug board into the computer, and spend the next few hours hoping that none of the 20,000 or so vacuum tubes would burn out during the run. Virtually all the problems were straightforward numerical calculations, such as sines, cosines, and logarithms.

By the early 1950s, the routine had improved somewhat with the introduction of punched cards. It was now possible to write programs on cards and read them in instead of using plugboards; otherwise, the procedure was the same.

1.3.2 The Second Generation (1955-65) Transistors and Batch Systems

The introduction of the transistor in the mid-1950s changed the scenario. Computers became reliable enough that they could be manufactured and sold to paying customers with the expectation that they would continue to function long enough to get some useful work done. For the first time, there was a clear separation between designers, builders, operators, programmers, and maintenance personnel.

These machines, now called **mainframes**, were locked away in specially air-conditioned computer rooms, with staffs of specially-trained professional operators to run them. Only big corporations or major government agencies or universities could afford their multimillion dollar price tags. To run a **job** i.e., a program or set of programs, a programmer would first write the program on paper in FORTRAN or possibly even in assembly language, then punch it on cards. He would then bring the card deck down to the input room and hand it to one of the operators and wait until the output was ready.

When the computer finished whatever job it was currently running, an operator would go over to the printer and tear off the output and carry it over to the output-room, so that the programmer could collect it later. Then he would take one of the card decks that had been brought from the input room and read it in. If the FORTRAN compiler was needed, the operator would have to get it from a file cabinet and read it in. Much computer time was wasted while operators were walking around the machine room.

Given the high cost of the equipment, it is not surprising that people quickly looked for ways to reduce the wasted time. The solution generally adopted was the **batch system**. The idea behind it was to collect a tray full of jobs in the input room and then read them onto a magnetic tape using a small (relatively) inexpensive computer, such as the IBM 1401, which was very good at reading cards, copying tapes, and printing output, but not at all good at numerical calculations. Other, much

more expensive machines, such as the IBM 7094, were used for the real computing. This situation is shown in [Fig. 1-2](#).

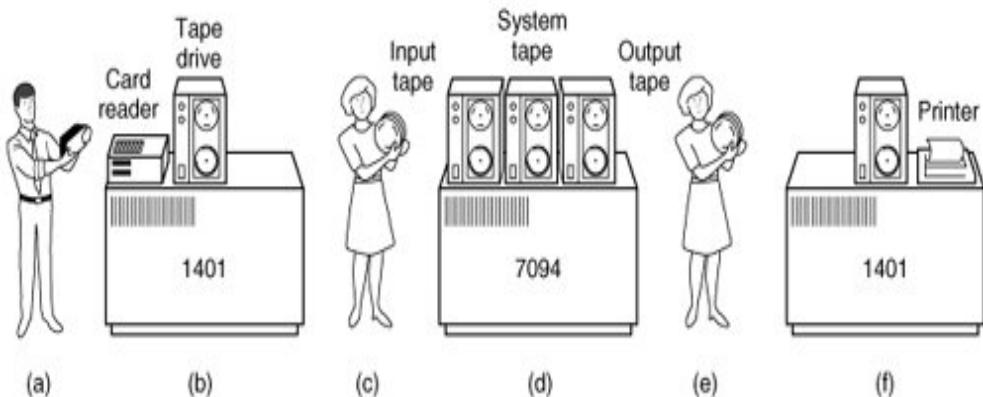


Figure 1-2. An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape. (c) Operator carries input tape to 7094. (d) 7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.

After about an hour of collecting a batch of jobs, the tape was rewound and brought into the machine room, where it was mounted on a tape drive. The operator then loaded a special program (the ancestor of today's operating system), which read the first job from tape and ran it. The output was written onto a second tape, instead of being printed. After each job finished, the operating system automatically read the next job from the tape and began running it. When the whole batch was done, the operator removed the input and output tapes, replaced the input tape with the next batch, and brought the output tape to a 1401 for printing **off line** (i.e., not connected to the main computer).

The structure of a typical input job is shown in [Fig. 1-3](#). It started out with a \$JOB card, specifying the maximum run time in minutes, the account number to be charged, and the programmer's name. Then came a \$FORTRAN card, telling the operating system to load the FORTRAN compiler from the system tape. It was followed by the program to be compiled, and then a \$LOAD card, directing the operating system to load the object program just compiled. (Compiled programs were often written on scratch tapes and had to be loaded explicitly.) Next came the \$RUN card, telling the operating system to run the program with the data following it. Finally, the \$END card marked the end of the job. These primitive control cards were the forerunners of modern job control languages and command interpreters.

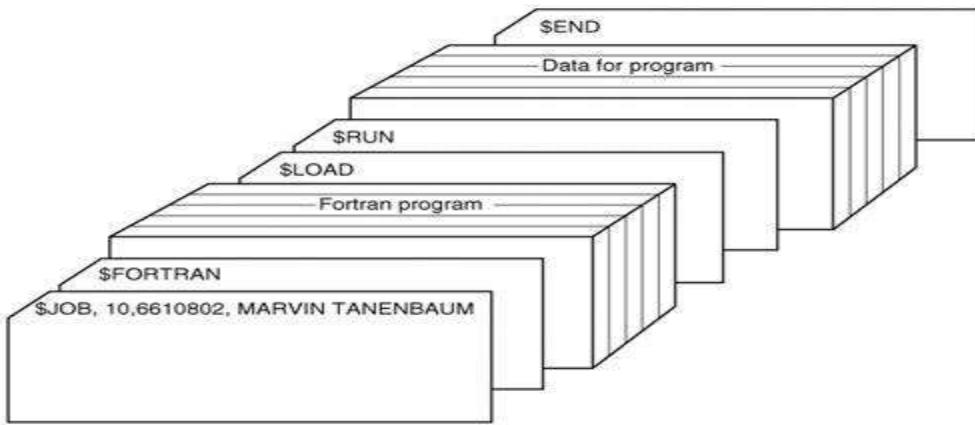


Figure 1-3. Structure of a typical FMS job.

Large second-generation computers were used mostly for scientific and engineering calculations, such as solving the partial differential equations that often occur in physics and engineering. They were largely programmed in FORTRAN and assembly language. Typical operating systems were FMS (the Fortran Monitor System) and IBSYS, IBM's operating system for the 7094.

1.3.3 The Third Generation (1965-1980) ICs and Multiprogramming

By the early 1960s, most computer manufacturers had two distinct, and totally incompatible, product lines. On the one hand there were the word-oriented, large-scale scientific computers, such as the 7094, which were used for numerical calculations in science and engineering. On the other hand, there were the character-oriented, commercial computers, such as the 1401, which were widely used for tape sorting and printing by banks and insurance companies.

Developing, maintaining, and marketing two completely different product lines was an expensive proposition for the computer manufacturers. In addition, many new computer customers initially needed a small machine but later outgrew it and wanted a bigger machine that had the same architectures as their current one so it could run all their old programs, but faster.

IBM attempted to solve both of these problems at a single stroke by introducing the System/360. The 360 was a series of software-compatible machines ranging from 1401-sized to much more powerful than the 7094. The machines differed only in price and performance (maximum memory, processor speed, number of I/O devices permitted, and so forth). Since all the machines had the same architecture and instruction set, programs written for one machine could run on all the others, at least in theory.

Furthermore, the 360 was designed to handle both scientific (i.e., numerical) and commercial computing. Thus a single family of machines could satisfy the needs of all customers. In subsequent years, IBM has come out with compatible successors to the 360 line, using more modern technology, known as the 370, 4300, 3080, 3090, and Z series.

The 360 was the first major computer line to use (small-scale) Integrated Circuits (ICs), thus providing a major price/performance advantage over the second-generation machines, which were built up from individual transistors. It was an immediate success, and the idea of a family of compatible computers was soon adopted by all the other major manufacturers. The descendants of these machines are still in use at computer centers today. Nowadays they are often used for managing huge databases (e.g., for airline reservation systems) or as servers for World Wide Web sites that must process thousands of requests per second.

The greatest strength of the "one family" idea was simultaneously its greatest weakness. The intention was that all software, including the operating system, **OS/360**, had to work on all models. It had to run on small systems, which often just replaced 1401s for copying cards to tape, and on very large systems, which often replaced 7094s for doing weather forecasting and other heavy computing. It had to be good on systems with few peripherals and on systems with many peripherals. It had to work in commercial environments and in scientific environments.

Above all, it had to be efficient for all of these different uses.

There was no way that IBM (or anybody else) could write a piece of software to meet all those conflicting requirements. The result was an enormous and extraordinarily complex operating system, probably two to three orders of magnitude larger than FMS. It consisted of millions of lines of assembly language written by thousands of programmers, and contained thousands upon thousands of bugs, which necessitated a continuous stream of new releases in an attempt to correct them. Each new release fixed some bugs and introduced new ones, so the number of bugs probably remained constant in time.

One of the designers of OS/360, Fred Brooks, subsequently wrote a witty and incisive book describing his experiences with OS/360. While it would be impossible to summarize the book here, suffice it to say that the cover shows a herd of prehistoric beasts stuck in a tar pit.

Despite its enormous size and problems, OS/360 and the similar third-generation operating systems produced by other computer manufacturers actually satisfied most of their customers reasonably well. They also popularized several key techniques absent in second-generation operating systems. Probably the most important of these was **multiprogramming**. On the 7094, when the current job paused to wait for a tape or other I/O operation to complete, the CPU simply sat idle until the I/O finished. With heavily CPU-bound scientific calculations, I/O is infrequent, so this wasted time is not significant. With commercial data processing, the I/O wait time can often be 80 or 90 percent of the total time, so something had to be done to avoid having the (expensive) CPU be idle so much.

The solution that evolved was to partition memory into several pieces, with a different job in each partition, as shown in [Fig. 1-4](#). While one job was waiting for I/O to complete, another job could be using the CPU. If enough jobs could be held in main memory at once, the CPU could be kept busy nearly 100 percent of the time. Having multiple jobs safely in memory at once requires special hardware to

protect each job against snooping and mischief by the other ones, but the 360 and other third-generation systems were equipped with this hardware.

Another major feature present in third-generation operating systems was the ability to read jobs from cards onto the disk as soon as they were brought to the computer room. Then, whenever a running job finished, the operating system could load a new job from the disk into the now-empty partition and run it. This technique is called **spooling** (from Simultaneous Peripheral Operation On Line) and was also used for output. With spooling, the 1401s were no longer needed, and much carrying of tapes disappeared.

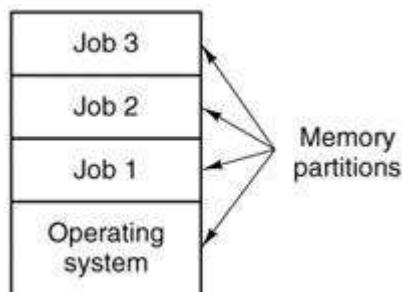


Figure 1-4. A multiprogramming system with three jobs in memory.

Although third-generation operating systems were well suited for big scientific calculations and massive commercial data processing runs, they were still basically batch systems. Many programmers pined for the first-generation days when they had the machine all to themselves for a few hours, so they could debug their programs quickly. With third-generation systems, the time between submitting a job and getting back the output was often hours, so a single misplaced comma could cause a compilation to fail, and the programmer to waste half a day.

This desire for quick response time paved the way for **timesharing**, a variant of multiprogramming, in which each user has an online terminal. In a timesharing system, if 20 users are logged in and 17 of them are thinking or talking or drinking coffee, the CPU can be allocated in turn to the three jobs that want service. Since people debugging programs usually issue short commands (e.g., compile a five-page procedure) rather than long ones (e.g., sort a million-record file), the computer can provide fast, interactive service to a number of users and perhaps also work on big batch jobs in the background when the CPU is otherwise idle. The first serious timesharing system, **CTSS** (Compatible Time Sharing System), was developed at M.I.T. on a specially modified 7094. However, timesharing did not really become popular until the necessary protection hardware became widespread during the third generation.

After the success of the CTSS system, MIT, Bell Labs, and General Electric (then a major computer manufacturer) decided to embark on the development of a "computer utility," a machine that would support hundreds of simultaneous timesharing users. Their model was the electricity distribution system when you need electric power, you just stick a plug in the wall, and within reason, as much

power as you need will be there. The designers of this system, known as **MULTICS** (Multiplexed Information and Computing Service), envisioned one huge machine providing computing power for everyone in the Boston area. The idea that machines far more powerful than their GE-645 mainframe would be sold for under a thousand dollars by the millions only 30 years later was pure science fiction.

MULTICS was a mixed success. It was designed to support hundreds of users on a machine only slightly more powerful than an Intel 80386-based PC, although it had much more I/O capacity. This is not quite as crazy as it sounds, since people knew how to write small, efficient programs in those days, a skill that has subsequently been lost. There were many reasons that MULTICS did not take over the world, not the least of which is that it was written in PL/I, and the PL/I compiler was years late and barely worked at all when it finally arrived. In addition, MULTICS was enormously ambitious for its time, much like Charles Babbage's analytical engine in the nineteenth century.

MULTICS introduced many seminal ideas into the computer literature, but turning it into a serious product and a commercial success was a lot harder than anyone had expected. Bell Labs dropped out of the project, and General Electric quit the computer business altogether. However, M.I.T. persisted and eventually got MULTICS working. It was ultimately sold as a commercial product by the company that bought GE's computer business (Honeywell) and installed by about 80 major companies and universities worldwide. While their numbers were small, MULTICS users were fiercely loyal. General Motors, Ford, and the U.S. National Security Agency, for example, only shut down their MULTICS systems in the late 1990s. The last MULTICS running, at the Canadian Department of National Defence, shut down in October 2000.

Despite its lack of commercial success, MULTICS had a huge influence on subsequent operating systems PCs or **workstations** (high-end PCs) in a business or a classroom may be connected via a **LAN (Local Area Network)** to a **file server** on which all programs and data are stored. An administrator then has to install and protect only one set of programs and data, and can easily reinstall local software on a malfunctioning PC or workstation without worrying about retrieving or preserving local data. In more heterogeneous environments, a class of software called **middleware** has evolved to bridge the gap between local users and the files, programs, and databases they use on remote servers.

Middleware makes networked computers look local to individual users' PCs or workstations and presents a consistent user interface even though there may be a wide variety of different servers, PCs, and workstations in use. The World Wide Web is an example. A web browser presents documents to a user in a uniform way, and a document as seen on a user's browser can consist of text from one server and graphics from another server, presented in a format determined by a style sheet on yet another server. Businesses and universities commonly use a web interface to access databases and run programs on a computer in another building or even another city. Middleware appears to be the operating system of a **distributed system**.

Another major development during the third generation was the phenomenal

growth of minicomputers, starting with the Digital Equipment Company (DEC) PDP-1 in 1961. The PDP-1 had only 4K of 18-bit words, but at \$120,000 per machine (less than 5 percent of the price of a 7094), it sold like hotcakes. For certain kinds of nonnumerical work, it was almost as fast as the 7094 and gave birth to a whole new industry. It was quickly followed by a series of other PDPS (unlike IBM's family, all incompatible) culminating in the PDP-11.

One of the computer scientists at Bell Labs who had worked on the MULTICS project, Ken Thompson, subsequently found a small PDP-7 minicomputer that no one was using and set out to write a stripped-down, one-user version of MULTICS. This work later developed into the **UNIX** operating system, which became popular in the academic world, with government agencies, and with many companies.

In Unix two major versions developed, **System V**, from AT&T, and **BSD**, Berkeley Software Distribution from the University of California at Berkeley. These had minor variants as well, now including FreeBSD, OpenBSD, and NetBSD. To make it possible to write programs that could run on any UNIX system, IEEE developed a standard for UNIX, called **POSIX**, that most versions of UNIX now support. POSIX defines a minimal system call interface that conformant UNIX systems must support. In fact, some other operating systems now also support the POSIX interface.

1.4 The Fourth Generation (1980 - Present) Personal Computers

With the development of LSI (Large Scale Integration) circuits, chips containing thousands of transistors on a square centimeter of silicon, the age of the **microprocessor**-based personal computer dawned. The minicomputer made it possible for a department in a company or university to have its own computer. The microcomputer made it possible for an individual to have his or her own computer.

There were several families of microcomputers. Intel came out with the 8080, the first general-purpose 8-bit microprocessor, in 1974. A number of companies produced complete systems using the 8080 (or the compatible Zilog Z80) and the **CP/M** (Control Program for Microcomputers) operating system from a company called Digital Research was widely used with these. Many application programs were written to run on CP/M, and it dominated the personal computing world for about 5 years.

Motorola also produced an 8-bit microprocessor, the 6800. A group of Motorola engineers left to form MOS Technology and manufacture the 6502 CPU after Motorola rejected their suggested improvements to the 6800. The 6502 was the CPU of several early systems. One of these, the

Apple II, became a major competitor for CP/M systems in the home and educational markets. But CP/M was so popular that many owners of Apple II computers purchased Z-80 coprocessor add-on cards to run CP/M, since the 6502 CPU was not compatible with CP/M. The CP/M cards were sold by a little company called Microsoft, which also had a market niche supplying BASIC interpreters used by a number of microcomputers running CP/M.

The next generation of microprocessors were 16-bit systems. Intel came out with the 8086, and in the early 1980s, IBM designed the IBM PC around Intel's 8088 (an 8086 on the inside, with an 8 bit external data path). Microsoft offered IBM a package which included Microsoft's BASIC and an operating system, **DOS** (Disk Operating System) originally developed by another company Microsoft bought the product and hired the original author to improve it. The revised system was renamed **MS-DOS** (MicroSoft Disk Operating System) and quickly came to dominate the IBM PC market.

CP/M, MS-DOS, and the Apple DOS were all command-line systems: users typed commands at the keyboard. Years earlier, Doug Engelbart at Stanford Research Institute had invented the **GUI (Graphical User Interface)**, pronounced "gooey," complete with windows, icons, menus, and mouse. Apple's Steve Jobs saw the possibility of a truly **user-friendly** personal computer (for users who knew nothing about computers and did not want to learn), and the Apple Macintosh was announced in early 1984. It used Motorola's 16-bit 68000 CPU, and had 64 KB of **ROM (Read Only Memory)**, to support the GUI. The Macintosh has evolved over the years. Subsequent Motorola CPUs were true 32-bit systems, and later still Apple moved to IBM PowerPC CPUs, with RISC 32-bit (and later, 64-bit) architecture. In 2001 Apple made a major operating system change, releasing **Mac OS X**, with a new version of the Macintosh GUI on top of Berkeley UNIX. And in 2005 Apple announced that it would be switching to Intel processors.

To compete with the Macintosh, Microsoft invented Windows. Originally Windows was just a graphical environment on top of 16-bit MS-DOS (i.e., it was more like a shell than a true operating system). However, current versions of Windows are descendants of Windows NT, a full 32-bit system, rewritten from scratch.

The other major contender in the personal computer world is UNIX (and its various derivatives). UNIX is strongest on workstations and other high-end computers, such as network servers. It is especially popular on machines powered by high-performance RISC chips. On Pentium-based computers, Linux is becoming a popular alternative to Windows for students and increasingly many corporate users. (Throughout this book we will use the term "Pentium" to mean the entire Pentium family, including the low-end Celeron, the high end Xeon, and compatible AMD microprocessors).

Although many UNIX users, especially experienced programmers, prefer a command-based interface to a GUI, nearly all UNIX systems support a windowing system called the **X Window** system developed at M.I.T. This system handles the basic window management, allowing users to create, delete, move, and resize windows using a mouse. Often a complete GUI, such as **Motif**, is available to run on top of the X Window system giving UNIX a look and feel something like the Macintosh or Microsoft Windows for those UNIX users who want such a thing.

An interesting development that began taking place during the mid-1980s is the growth of networks of personal computers running **network operating systems** and **distributed operating systems**. In a network operating system, the

users are aware of the existence of multiple computers and can log in to remote machines and copy files from one machine to another. Each machine runs its own local operating system and has its own local user (or users). Basically, the machines are independent of one another.

Network operating systems are not fundamentally different from single-processor operating systems. They obviously need a network interface controller and some low-level software to drive it, as well as programs to achieve remote login and remote file access, but these additions do not change the essential structure of the operating system.

A distributed operating system, in contrast, is one that appears to its users as a traditional uniprocessor system, even though it is actually composed of multiple processors. The users should not be aware of where their programs are being run or where their files are located; that should all be handled automatically and efficiently by the operating system.

True distributed operating systems require more than just adding a little code to a uniprocessor operating system, because distributed and centralized systems differ in critical ways. Distributed systems, for example, often allow applications to run on several processors at the same time, thus requiring more complex processor scheduling algorithms in order to optimize the amount of parallelism.

Communication delays within the network often mean that these (and other) algorithms must run with incomplete, outdated, or even incorrect information. This situation is radically different from a single-processor system in which the operating system has complete information about the system state.

1.4 Operating System Concepts

The interface between the operating system and the user programs is defined by the set of "extended instructions" that the operating system provides. These extended instructions have been traditionally known as **system calls**, although they can be implemented in several ways. To really understand what operating systems do, we must examine this interface closely. The calls available in the interface vary from operating system to operating system (although the underlying concepts tend to be similar).

1.4.1 Processes

A process is basically a program in execution. Associated with each process is its **address space**, a list of memory locations from some minimum (usually 0) to some maximum, which the process can read and write. The address space contains the executable program, the program's data, and its stack. Also associated with each process is some set of registers, including the program counter, stack pointer, and other hardware registers, and all the other information needed to run the program.

For the time being, the easiest way to get a good intuitive feel for a process is to think about multiprogramming systems. Periodically, the operating system decides

to stop running one process and start running another, for example, because the first one has had more than its share of CPU time in the past second.

When a process is suspended temporarily like this, it must later be restarted in exactly the same state it had when it was stopped. This means that all information about the process must be explicitly saved somewhere during the suspension. For example, the process may have several files open for reading at once. Associated with each of these files is a pointer giving the current position (i.e., the number of the byte or record to be read next). When a process is temporarily suspended, all these pointers must be saved so that a **read** call executed after the process is restarted will read the proper data. In many operating systems, all the information about each process, other than the contents of its own address space, is stored in an operating system table called the **process table**, which is an array (or linked list) of structures, one for each process currently in existence.

Thus, a (suspended) process consists of its address space, usually called the **core image** (in honor of the magnetic core memories used in days of yore), and its process table entry, which contains its registers, among otherthings.

The key process management system calls are those dealing with the creation and termination of processes. Consider a typical example. A process called the **command interpreter** or **shell** reads commands from a terminal. The user has just typed a command requesting that a program be compiled. The shell must now create a new process that will run the compiler. When that process has finished the compilation, it executes a system call to terminate itself.

On Windows and other operating systems that have a GUI, (double) clicking on a desktop icon launches a program in much the same way as typing its name at the command prompt. Although we will not discuss GUIs much, they are really simple command interpreters.

If a process can create one or more other processes (usually referred to as **child processes**) and these processes in turn can create child processes, we quickly arrive at the process tree structure of [Fig. 1-5](#). Related processes that are cooperating to get some job done often need to communicate with one another and synchronize their activities.

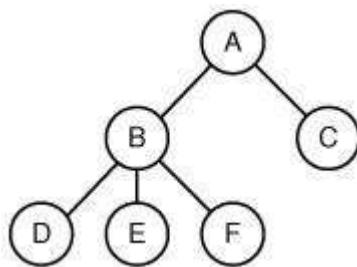


Figure 1-5. A process tree. Process **A** created two child processes, **B** and **C**. Process **B** created three child processes, **D**, **E**, and **F**.

Other process system calls are available to request more memory (or release

unused memory), wait for a child process to terminate, and overlay its program with a different one.

Occasionally, there is a need to convey information to a running process that is not sitting around waiting for it. For example, a process that is communicating with another process on a different computer does so by sending messages to the remote process over a network. To guard against the possibility that a message or its reply is lost, the sender may request that its own operating system notify it after a specified number of seconds, so that it can retransmit the message if no acknowledgement has been received yet. After setting this timer, the program may continue doing other work.

When the specified number of seconds has elapsed, the operating system sends an alarm Signal to the process. The signal causes the process to temporarily suspend whatever it was doing, save its registers on the stack, and start running a special signal handling procedure, for example, to retransmit a presumably lost message. When the signal handler is done, the running process is restarted in the state it was in just before the signal. Signals are the software analog of hardware interrupts. They are generated by a variety of causes in addition to timers expiring. Many traps detected by hardware, such as executing an illegal instruction or using an invalid address, are also converted into signals to the guilty process.

Every process started has the UID of the person who started it. A child process has the same UID as its parent. Users can be members of groups, each of which has a **GID** (Group IDentification).

One UID, called the **superuser** (in UNIX), has special power and may violate many of the protection rules. In large installations, only the system administrator knows the password needed to become superuser, but many of the ordinary users (especially students) devote considerable effort to trying to find flaws in the system that allow them to become superuser without the password.

1.4.2. Files

The other broad category of system calls relates to the file system. As noted before, a major function of the operating system is to hide the peculiarities of the disks and other I/O devices and present the programmer with a nice, clean abstract model of device-independent files. System calls are obviously needed to create files, remove files, read files, and write files. Before a file can be read, it must be opened, and after it has been read it should be closed, so calls are provided to do these things.

To provide a place to keep files the concept of a **directory** as a way of grouping files together. A student, for example, might have one directory for each course he is taking (for the programs needed for that course), another directory for his electronic mail, and still another directory for his World Wide Web home page. System calls are then needed to create and remove directories. Calls are also provided to put an

existing file into a directory, and to remove a file from a directory. Directory entries may be either files or other directories. This model also gives rise to a hierarchy the file system as shown in [Fig. 1-6](#).

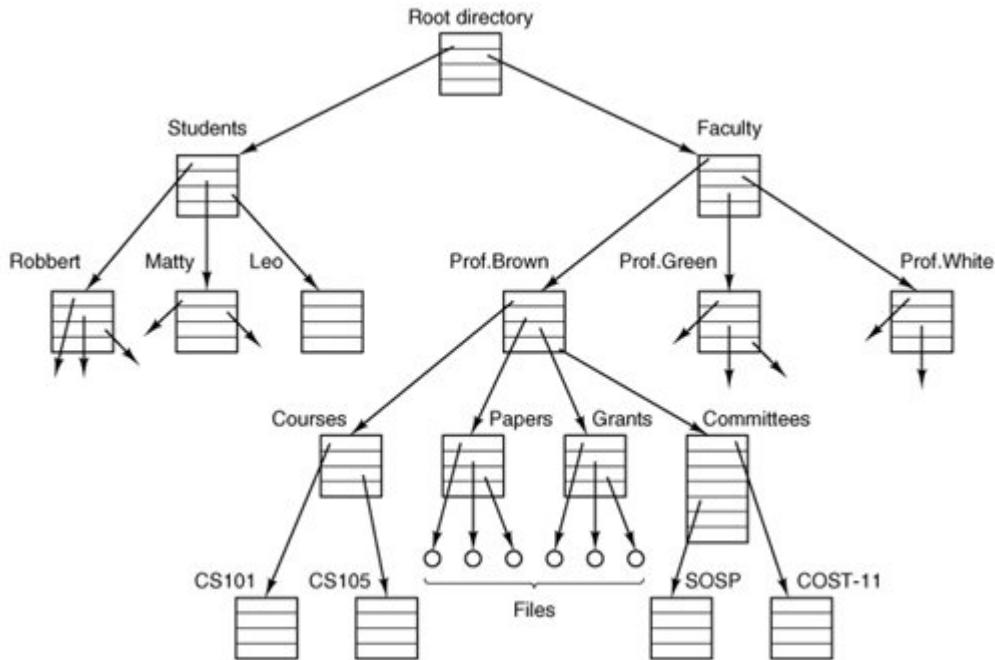


Figure 1-6. A file system for a university department.

The process and file hierarchies both are organized as trees, but the similarity stops there. Process hierarchies usually are not very deep, more than three levels is unusual, whereas file hierarchies are commonly four, five, or even more levels deep. Process hierarchies are typically short-lived, generally a few minutes at most, whereas the directory hierarchy may exist for years. Ownership and protection also differ for processes and files. Typically, only a parent process may control or even access a child process, but mechanisms nearly always exist to allow files and directories to be read by a wider group than just the owner.

Every file within the directory hierarchy can be specified by giving its **path name** from the top of the directory hierarchy, the **root directory**. Such absolute path names consist of the list of directories that must be traversed from the root directory to get to the file, with slashes separating the components.

In [Fig. 1-6](#), the path for file *CS101* is */Faculty/Prof.Brown/Courses/CS101*. The leading slash indicates that the path is absolute, that is, starting at the root directory. As an aside, in Windows, the backslash (\) character is used as the separator instead of the slash (/) character, so the file path given above would be written as *\Faculty\Prof.Brown\Courses\CS101*. Throughout this book we will use the UNIX convention for paths.

At every instant, each process has a current **working directory**, in which path names not beginning with a slash are looked for. As an example, in [Fig. 1-6](#), if */Faculty/Prof.Brown* were the working directory, then use of the path name

Courses/CS101 would yield the same file as the absolute path name given above. Processes can change their working directory by issuing a system call specifying the new working directory.

Files and directories in MINIX 3 are protected by assigning each one an 11-bit binary protection code. The protection code consists of three 3-bit fields: one for the owner, one for other members of the owner's group (users are divided into groups by the system administrator), one for everyone else, and 2 bits we will discuss later. Each field has a bit for read access, a bit for write access, and a bit for execute access. These 3 bits are known as the rwx bits. For example, the protection code `rwxr-x--x` means that the owner can read, write, or execute the file, other group

members can read or execute (but not write) the file, and everyone else can execute (but not read or write) the file. For a directory (as opposed to a file), x indicates search permission. A dash means that the corresponding permission is absent (the bit is zero).

Before a file can be read or written, it must be opened, at which time the permissions are checked. If access is permitted, the system returns a small integer called a file descriptor to use in subsequent operations. If the access is prohibited, an error code (1) is returned.

Another important concept in MINIX 3 is the mounted file system. Nearly all personal computers have one or more CD-ROM drives into which CD-ROMs can be inserted and removed. To provide a clean way to deal with removable media (CD-ROMs, DVDs, floppies, Zip drives, etc.), MINIX 3 allows the file system on a CD-ROM to be attached to the main tree. Consider the situation of Fig. 1-7(a). Before the mount call, the root file system, on the hard disk, and a second file system, on a CD-ROM, are separate and unrelated.

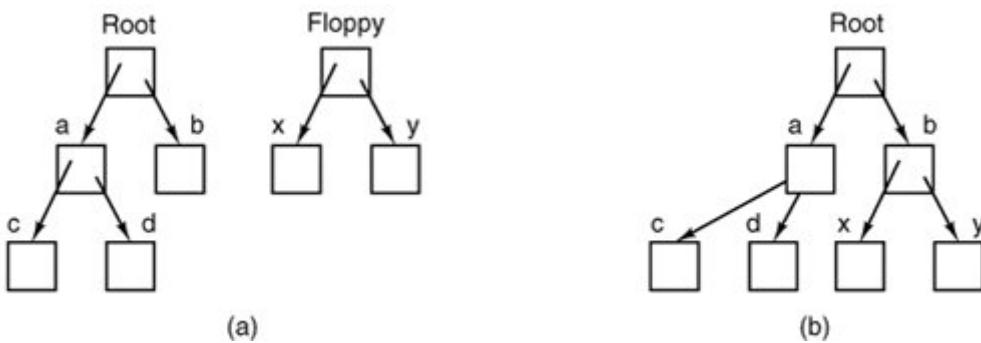


Figure 1-7. (a) Before mounting, the files on drive 0 are not accessible. (b) After mounting, they are part of the file hierarchy.

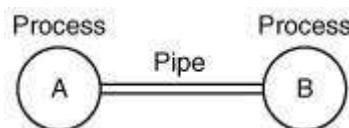
However, the file system on the CD-ROM cannot be used, because there is no way to specify path names on it. MINIX 3 does not allow path names to be prefixed by a drive name or number; that is precisely the kind of device dependence that operating systems ought to eliminate. Instead, the mount system call allows the file system on the CD-ROM to be attached to the root file system wherever the program wants it to

be. In Fig. 1-7(b) the file system on drive 0 has been mounted on directory b, thus allowing access to files /b/x and /b/y. If directory b had originally contained any files they would not be accessible while the CD-ROM was mounted, since /b would refer to the root directory of drive 0. (Not being able to access these files is not as serious as it at first seems: file systems are nearly always mounted on empty directories.) If a system contains multiple hard disks, they can all be mounted into a single tree as well.

Another important concept in MINIX 3 is the special file. Special files are provided in order to make I/O devices look like files. That way, they can be read and written using the same system calls as are used for reading and writing files. Two kinds of special files exist: block special files and character special files. Block special files are normally used to model devices that consist of a collection of randomly addressable blocks, such as disks. By opening a block special file and reading, say, block 4, a program can directly access the fourth block on the device, without regard to the structure of the file system contained on it. Similarly, character special files are used to model printers, modems, and other devices that accept or output a character stream. By convention, the special files are kept in the `/dev` directory. For example, `/dev/lp` might be the line printer.

The last feature we will discuss in this overview is one that relates to both processes and files: pipes. A **pipe** is a sort of pseudofile that can be used to connect two processes, as shown in [Fig. 1-8](#). If processes A and B wish to talk using a pipe, they must set it up in advance. When process A wants to send data to process B, it writes on the pipe as though it were an output file. Process B can read the data by reading from the pipe as though it were an input file. Thus, communication between processes in MINIX 3 looks very much like ordinary file reads and writes. Stronger yet, the only way a process can discover that the output file it is writing on is not really a file, but a pipe, is by making a special system call.

Figure 1-8. Two processes connected by a pipe.



1.4.3. The Shell

The operating system is the code that carries out the system calls. Editors, compilers, assemblers, linkers, and command interpreters definitely are not part of the operating system, even though they are important and useful. At the risk of confusing things somewhat, in this section we will look briefly at the MINIX 3 command interpreter, called the **shell**. Although it is not part of the operating system, it makes heavy use of many operating system features and thus serves as a good example of how the system calls can be used. It is also the primary interface between a user sitting at his terminal and the operating system, unless the user is using a graphical user interface. Many shells exist, including `csh`, `ksh`, `zsh`, and `bash`.

All of them support the functionality described below, which derives from the original shell (*sh*).

When any user logs in, a shell is started up. The shell has the terminal as standard input and standard output. It starts out by typing the **prompt**, a character such as a dollar sign, which tells the user that the shell is waiting to accept a command. If the user now types

`date`

for example, the shell creates a child process and runs the *date* program as the child. While the child process is running, the shell waits for it to terminate. When the child finishes, the shell types the prompt again and tries to read the next input line.

The user can specify that standard output be redirected to a file, for example,

`date >file`

Similarly, standard input can be redirected, as in

`sort <file1 >file2`

which invokes the *sort* program with input taken from *file1* and output sent to *file2*.

The output of one program can be used as the input for another program by connecting them with a pipe. Thus

`cat file1 file2 file3 | sort >/dev/lp`

invokes the *cat* program to concatenate three files and send the output to *sort* to arrange all the lines in alphabetical order. The output of *sort* is redirected to the file */dev/lp*, typically the printer.

If a user puts an ampersand after a command, the shell does not wait for it to complete. Instead it just gives a prompt immediately. Consequently,

`cat file1 file2 file3 | sort >/dev/lp &`

starts up the *sort* as a background job, allowing the user to continue working normally while the *sort* is going on.

2. Operating System Structures

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs. The specific services provided, of course, differ from one operating system to another, but we can identify common classes. These operating-system services are provided for the convenience of the programmer, to make programming task easier.

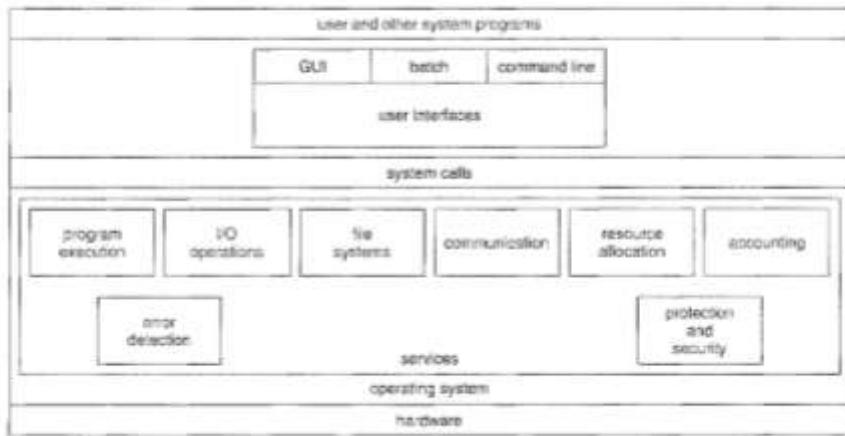


Figure 2.1 A view of operating system services.

task easier. Figure 2.1 shows one view of the various operating-system services and how they interrelate.

2.1 Operating System Services

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

User interface. Almost all operating systems have a User interface. This interface can take several forms. One is a command line interface (CLI) which uses text commands and a method for entering them (say, a program to allow entering and editing of commands). Another is a batch interface in which commands and directives to control those commands are entered into files, and those files are executed. Most commonly, a Graphical User Interface(GUI) is used. Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. 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 recording to a CD or DVD drive or blanking a display screen). 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 programs 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 computer network. Communications may be implemented via *shared memory* or through *message passing*, in which packets of information are moved between processes by the operating system.

Error detection. The operating system needs to be constantly aware of possible errors. 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 tape, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing. Of course, there is variation in how operating systems react to and correct errors. Debugging facilities can greatly enhance the user's and programmer's abilities to use the system efficiently.

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 users can gain efficiency by sharing the computer resources among the users.

Resource allocation. When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. Many different types of resources are managed by the operating system. 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 jobs that must be executed, the number of registers available, and other factors. There may also be routines to allocate printers, modems, USB storage drives, and other peripheral devices.

Accounting. We want to keep track of which users 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 researchers 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 could 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 modems and network adapters, from invalid access attempts and to 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 Operating System Interfaces

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

2.2.1 Command Interpreter

Some operating systems include the command interpreter in the kernel. Others, such as Windows XP and UNIX, treat the command interpreter as a special program that is running when a job 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 *Bourne shell*, *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 shell command interpreter being used on Solaris 10.

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 MS-DOS and UNIX shells 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 function associated with the rm command would be 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 names. The command-interpreter program, which can be small, does not have to be changed for new commands to be added.

```

File Edit View Terminal Tabs Help
fd0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0 0
sd0 0.0 0.2 0.0 0.2 0.0 0.0 0.4 0 0
sd1 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0 0
extended device statistics
device r/s w/s kr/s ks/s wait activ svc_t %w %b
fd0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0 0
sd0 0.5 0.0 38.4 0.0 0.0 0.0 8.2 0 0
sd1 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0 0
(continued from previous line)
-(root@log-in:~) swap -sh
total: 1.1G allocated + 190M reserved = 1.3G used, 1.6G available
(continued from previous line)
-(root@log-in:~) uptime
12:53am up 9 min(s), 3 users, load average: 33.29, 67.68, 36.81
(continued from previous line)
-(root@log-in:~) w
4:07pm up 17 day(s), 15:24, 3 users, load average: 0.09, 0.11, 0.66
User     tty      login@   idle   JCPU   FCPU  what
root    console  15Jun07 18days   1      /usr/bin/ssh-agent -- /usr/bi
n/d
root    pts/3    15Jun07      18      4  w
root    pts/4    15Jun07 18days      w
(continued from previous line)
-(root@log-in:~) w

```

Figure 2.2 The Bourne shell command interpreter in Solaris I 0.

2.2.2 Graphical User Interfaces

A second strategy for interfacing with the operating system is through a user-friendly graphical user interface, or CUI. 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 CUI 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 (Mac OS) has undergone various changes over the years, the most significant being the adoption of the *Aqua* interface that appeared with Mac OS X. Microsoft's first version of Windows- Version 1.0-was based on the addition of a CUI interface to the MS-DOS operating system. Later versions of Windows have made cosmetic changes in the appearance of the CUI along with several enhancements in its functionality, including Windows Explorer.

Traditionally, UNIX systems have been dominated by command-line interfaces. Various GUI interfaces are available, however, including the Common Desktop Environment (CDE) and X-Windows systems, which are common on commercial versions of UNIX, such as Solaris and IBM's AIX system. In addition, there has been significant development in GUI designs from various projects, such as *I*< *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.

The choice of whether to use a command-line or GUI interface is mostly one of personal preference. As a very general rule, many UNIX users prefer command-line interfaces, as they often provide powerful shell interfaces. In contrast, most Windows users are pleased to use the Windows GUI environment and almost never use the MS-DOS shell interface. The various changes undergone by the Macintosh operating systems 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 Mac OS X (which is in part implemented using a UNIX kernel), the operating system now provides both a new Aqua interface and a command-line interface. Figure 2.3 is a screenshot of the Mac OS X GUI.

The user interface can vary from system to system and even from user to user within a system. It typically is substantially removed from the actual system structure. The design of a useful and friendly user interface is therefore not a direct function of the operating system.



Figure 2.3 The Mac OS X GUI

2.3 System Calls

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

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 for the program to ask the user for the names of the two files. 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 are obtained, the program must open the input file and create the output file. Each of these operations requires another system call. There are also possible error conditions for each operation. 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 print a message on the console (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 (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.

Now that 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 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 (no more disk space, printer out of paper, and so on).

Finally, after the entire file is copied, the program may close both files (another system call), write a message to the console or window (more system calls), and finally terminate normally (the final system call). As we can see, even simple programs may make heavy use of the operating system. Frequently, systems execute thousands of system calls per second. This system-call sequence is shown in Figure 2.4

Most programmers never see this level of detail however. Typically application developers design programs according to an Application Programmer 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 API is available to application programmers are the Win32 API for Windows systems, the POSIX API for POSIX-based systems (which include virtually all versions of UNIX, Linux/ and Mac OS X), and the Java API for designing programs that run on the Java virtual machine. 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 Win32 function CreateProcess () (which unsurprisingly is used to create a new process) actually calls 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 of programming according to an API 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. Regardless/ 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 Win32 API is are similar to the native system calls provided by the UNIX, Linux, and Windows operating systems.

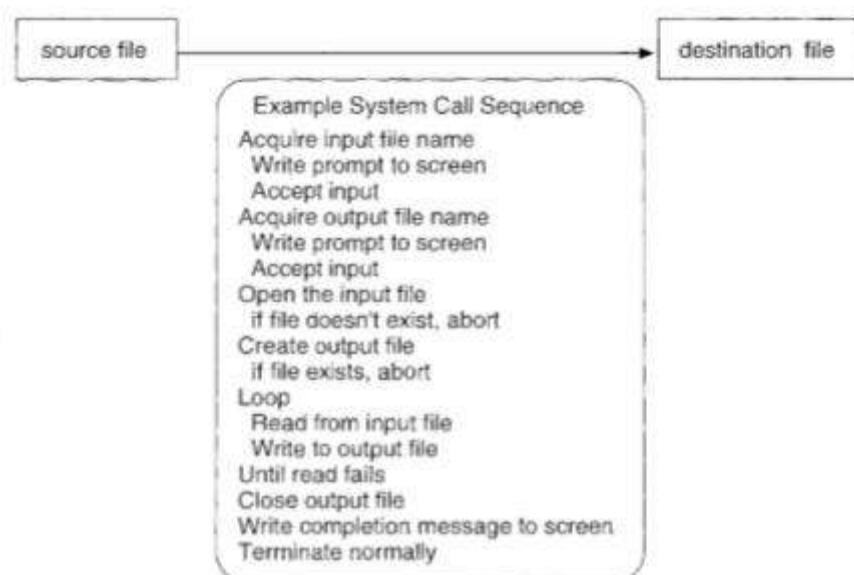


Figure 2.4 Example of how system calls are used.

EXAMPLE OF STANDARD API

As an example of a standard APT, consider the ReadFile 0 function in the Win32 API-a function for reading £rom a file. The API for this function appears in Figure 2.5 .

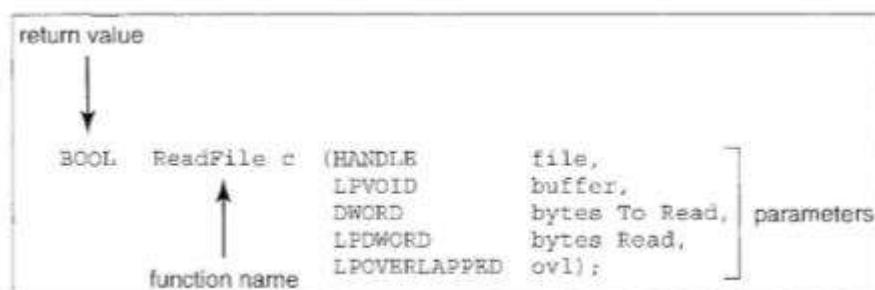


Figure 2.5 The API for the ReadFile () function.

A description of the parameters passed to ReadFile 0 is as follows:

HANDLE file-the file to be read

LPVOID buffer-a buffer where the data will be read into and written from

DWORD bytesToRead-the number of bytes to be read into the buffer

LPDWORD bytesRead -the number of bytes read during the last read

LPOVERLAPPED ovlp-indicates if overlapped I/O is being used

In fact, many of the POSIX and Win32 API is are similar to the native system calls provided by the UNIX, Linux, and Windows operating systems.

The run-time support system (a set of functions built into libraries included with a compiler) for most programming languages provides a 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 and any return values.

The caller need know nothing about how the system call is implemented or what it does during execution. Rather it 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 run-time support library. The relationship between 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.

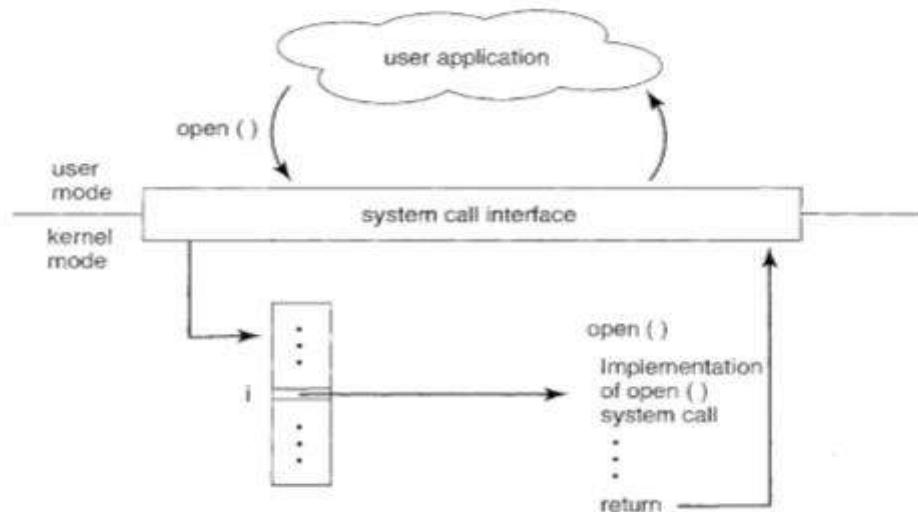


Figure 2.6 The handling of a user application invoking the open() system call.
System

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).

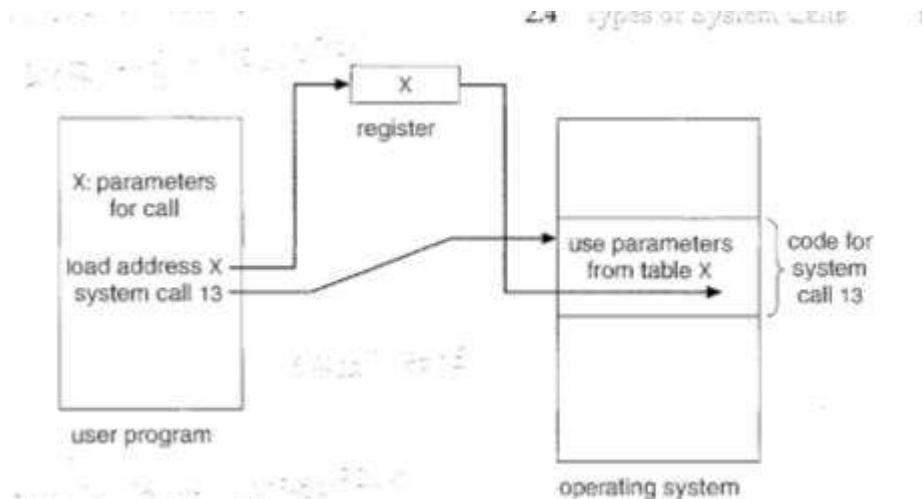


Figure 2.7 Passing of parameters as a table.

This is the approach taken by Linux and Solaris. Parameters also can be placed, or *pushed*, onto 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.

2.4 Types of System Calls

System calls can be grouped roughly into six major categories: process control, file manipulation, device manipulation, information maintenance, communications and protection.

2.4.1 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 disk and may be examined by a debugger system program designed to aid the programmer in finding and correcting 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 any error. In a GUI system, a pop-up window might alert the user to the error and ask for guidance. In a batch system, the command interpreter usually terminates the entire job and continues with the next job. Some systems allow control cards to indicate special recovery actions in case an error occurs.

A is a batch-system concept. It is a command to manage the execution of a process. 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. Process or job executing one program allows the command interpreter to execute a program as directed by, for example, a user command, the click of a mouse, or a batch command. An interesting question is where to return control when the loaded program terminates. This question is related to the problem of

Process control

- end, abort
- load, execute
- create process, terminate process
- get process attributes, set process attributes
- wait for time
- 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
- o 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

Figure 2.8 Types of system calls.

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 effectively created a mechanism for one program to call another program. If both programs continue concurrently, we have created a new job or process to

	Windows	Unix
Process Control	CreateProcessO ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimerO Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmapO
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

be multiprogrammed. Often, there is a system call specifically for this purpose (create process or submit job).

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

Having created new jobs or 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 jobs or 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 a process *to* lock shared data, thus preventing another process from accessing the data while it is locked. Typically such system calls include acquire lock and release lock.

EXAMPLE OF 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(s) 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. This is shown in Figure 2.9.

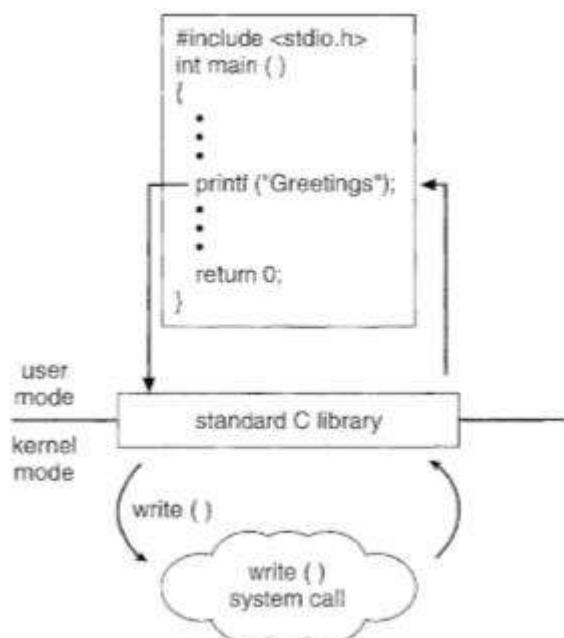


Figure 2.9 Standard C library handling of `write()`.

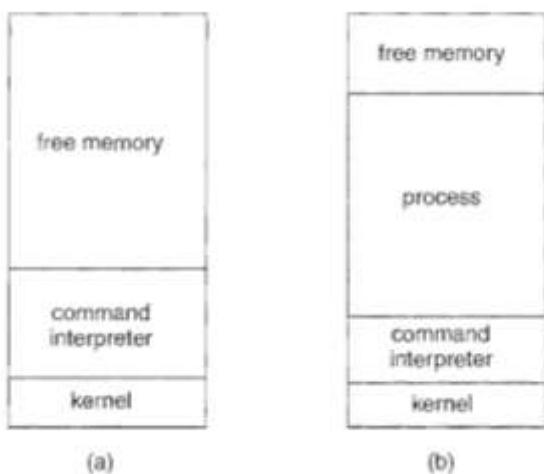


Figure 2.10 MS-DOS execution. (a) At system startup. (b) Running a program

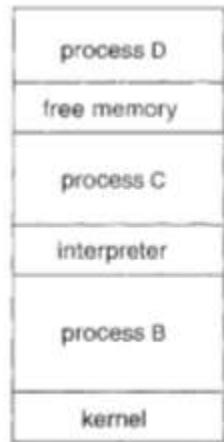


Figure 2.11 FreeBSD running multiple programs

There are so many facets of and variations in process and job control that we next use two examples-one involving a single-tasking system and the other a multitasking system - to clarify these concepts. The MS-DOS operating system is an example of a single-tasking system. It has a command interpreter that is invoked when the computer is started (Figure 2.10(a)). Because MS-DOS is single-tasking, it uses a simple method to run a program and does not create a new process. It loads the program into memory, writing over most of itself to give the program as much memory as possible (Figure 2.10(b)). Next, it sets the instruction pointer to the first instruction of the program. The program then runs, and either an error causes a trap, or the program executes a system call to terminate. In either case, the error code is saved in the system memory for later use. Following this action, the small portion of the command interpreter that was not overwritten resumes execution. Its first task is to reload the rest of the command interpreter from disk. Then the command interpreter makes the previous error code available to the user or to the next program.

FreeBSD is an example of a multitasking system. When a user logs on to the system, the shell of the user's choice is run. This shell is similar to the MS-DOS shell in that it accepts commands and executes programs that the user requests. However, since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed (Figure 2.11). 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 the way 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 requests another command. 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 CUI 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()` 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.

2.4.2 File Management

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 (rewinding or skipping 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 reset 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 attribute and set file attribute, 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 just provide system programs to perform those tasks. If the system programs are callable by other programs, then each can be considered an API by other system programs.

2.4.3 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 the 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.

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 similar₁ 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.

2.4.4 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 number of current users, 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. A program trace 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 reset the process information (get process attributes and set process attributes). In Section 3.1.3, we discuss what information is normally kept.

2.4.5 Communication

There are two common mode of Inter Process communication : message passing model and the shared-memory model. In message passing model, the communicating processes exchange their messages with one another to transfer the 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 hosted 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 permission for communication to take place with an accept connection call. Most processes that will be receiving connections are special-purpose *daemons*, which are systems 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 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 are 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 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 inter computer 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.

2.4.6 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 PDAs, 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.

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.

2.5 Operating System Design and Implementation

Here, we discuss problems we face in designing and implementing an operating system.

2.5.1 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: batch, time shared, single user, multiuser, distributed, real time, or general purpose.

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 desire 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 those people 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 VxWorks, a real-time operating system for embedded systems, must have been substantially different from those for MVS, a large multiuser, multiaccess operating system for IBM mainframes.

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.

2.5.2 Mechanisms and Principles

One important principle is the separation of **policy** from mechanisms. Mechanisms determine how to do something. Policies determine what will be done. For example, the timer construct is a mechanism for ensuring the 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 over time. In the worst case, each change in policy would require a change in the underlying mechanism. A general mechanism insensitive to changes in policy would be more desirable. 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 take the separation of mechanisms and policies 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 via user programs themselves. As an example, consider the history of UNIX. At first, it had a time-sharing scheduler. In the latest version of Solaris, scheduling is controlled by loadable tables. Depending on the table currently loaded, the system can be time shared, batch processing, real time, fair share, or any combination. Making the scheduling mechanism general purpose allows vast policy changes to be made with a single load-new-table command.

At the other extreme is a system such as Windows in which both mechanisms and policies are encoded in the system to enforce global look and feel. All applications have similar interfaces, because the interface itself is built into the kernel and system libraries. The Mac OS X operating system has similar functionality.

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.

2.5.3 Implementation

Once an operating system is designed, it must be implemented. Traditionally, operating systems have been written in assembly language. Now, however, they are most commonly written in higher-level languages such as C or C++.

The first system that was not written in assembly language was probably the Master Control Program (MCP) for Burroughs computers. MCP was written in a variant of ALGOL. MULTICS, developed at MIT, was written mainly in PL/1. The Linux and Windows XP operating systems are written mostly in C, although there are some small sections of assembly code for device drivers and for saving and restoring the state of registers.

The advantages of using a higher-level language, or at least a systems-implementation language, for implementing operating systems are the same as those accrued 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* move to some other hardware-if it is written in a higher-level language. For example, MS-DOS was written in Intel 8088 assembly language. Consequently, it runs natively only on the Intel X86 family of CPUs. (Although MS-DOS runs natively only on Intel X86, emulators of the X86 instruction set allow the operating system to run non-natively- slower, with more resource use-on other CPUs. are programs that duplicate the functionality of one system with another system.) The Linux operating system, in contrast, is written mostly in C and is available natively on a number of different CPUs, including Intel X86, Sun SPARC, and IBM Power PC. The only possible disadvantages of implementing an operating system in a higher-level language are reduced speed and increased

storage requirements. This, however is no longer 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 memory manager and the CPU scheduler are probably the most critical routines. After the system is written and is working correctly, bottleneck routines can be identified and can be replaced with assembly-language equivalents. The layered approach described in Section 2.7.2 is taken to its logical conclusion in the concept of a virtual machine.

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 execution environment is running its own private computer.

2.6 Virtual Machines

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 execution environment is running its own private computer. By using CPU scheduling and virtual-memory techniques, an operating system can create the illusion that a process has its own processor with its own (virtual) memory. The virtual machine provides an interface that is identical to the underlying bare hardware. Each process is provided with a (virtual) copy of the underlying computer (Figure 2.17). Usually, the guest process is in fact an operating system, and that is how a single physical machine can run multiple operating systems concurrently, each in its own virtual machine.

2.6.1 History

Virtual machines first appeared commercially on IBM mainframes via the VM operating system in 1972. VM has evolved and is still available, and many of the original concepts are found in other systems, making this facility worth exploring.

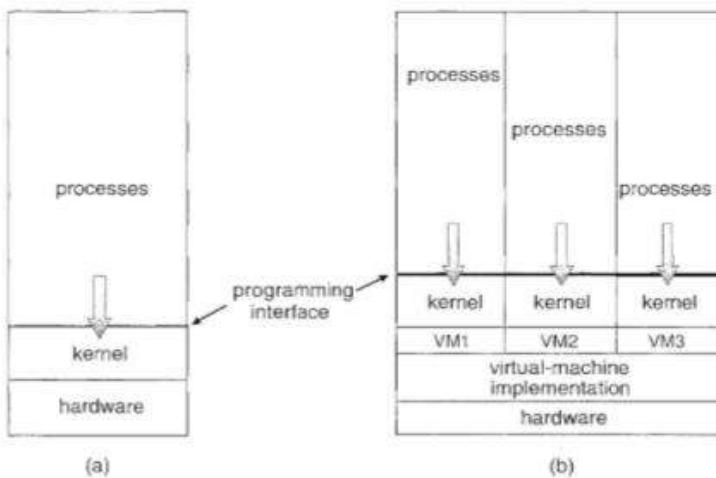


Figure 2.17 System models. (a) Non virtual machine. (b) Virtual machine.

IBM VM370 divided a mainframe into multiple virtual machines, each running its own operating system. A major difficulty with the VM virtual-machine 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, because the virtual-machine software itself needed substantial disk space to provide virtual memory and spooling. The solution was to provide virtual disks-termed *minidisks* in IBM's VM operating system -that are identical in all respects except size. The system implemented each minidisk by allocating as many tracks on the physical disks as the minidisk needed.

Once these 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.

2.6.2 Benefits

There are several reasons for creating a virtual machine. Most of them are fundamentally related to being able to share the same hardware yet run several different execution environments (that is, different operating systems) concurrently.

One important advantage 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 completely isolated from all other virtual machines, there are no protection problems. At the same time, however, there is no direct sharing of resources. Two approaches to provide 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.

A virtual-machine system is a perfect vehicle for operating-systems research and development. Normally, changing an operating system is a difficult task. Operating systems are large and complex programs, and it is difficult to be sure that a change in one part will not 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.

The operating system, however, runs on and controls the entire machine. Therefore, the current 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 is often scheduled late at night or on weekends, when system load is low.

A virtual-machine system can eliminate much of this 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 seldom needs to be disrupted for system development.

Another advantage of virtual machines for developers is that multiple operating systems can be running on the developer's workstation concurrently. This virtualized workstation allows for rapid porting and testing of programs in varying environments. 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 a system 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, as many lightly used systems can be combined to create one more heavily used system.

If the use of virtual machines continues to spread, application deployment will evolve accordingly. If a system can easily add, remove, and move a virtual machine, then why install applications on that system directly? Instead, application developers would pre-install the application on a tuned and customized operating system in a virtual machine. That virtual environment would be the release mechanism for the application. This method would be an improvement 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 do just that, and it could succeed in unifying virtual-machine formats.

2.6.3 Simulation

System virtualization as discussed so far is just one of many system-emulation methodologies. Virtualization is the most common because it makes guest operating systems and applications "believe" they are running on native hardware. Because only the system's resources need to be virtualized, these guests run at almost full speed.

Another methodology is in which 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, but its major challenge is performance. Instruction-set emulation can run an order of magnitude slower than native instructions. Thus, unless the new machine is ten times faster than the old, the program running on the new machine will run slower than it did on its native hardware. Another challenge is that it is difficult to create a correct emulator because, in essence, this involves writing an entire CPU in software.

2.6.4 Para-virtualization

Para Virtualization is another variation on this theme. Rather than try to trick a guest operating system into believing it has a system to itself, para-virtualization 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 Para virtualized hardware. The gain for this extra work is more efficient use of resources and a smaller virtualization layer.

Solaris 10 includes containers or nodes 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 container 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 resources can be divided up among the containers and the system wide processes. Figure 2.18 shows a Solaris 10 system with two containers and the standard "global" user space. Broadly, is the activity of finding and fixing errors, or in a system. Debugging seeks to find and fix errors in both hardware and software. Performance problems are considered bugs, so debugging can also include which seeks to improve performance by removing -"---" in the processing taking place within a system. A discussion of hardware debugging is outside of the scope of this text. In this section, we explore debugging kernel and process errors and performance problems.

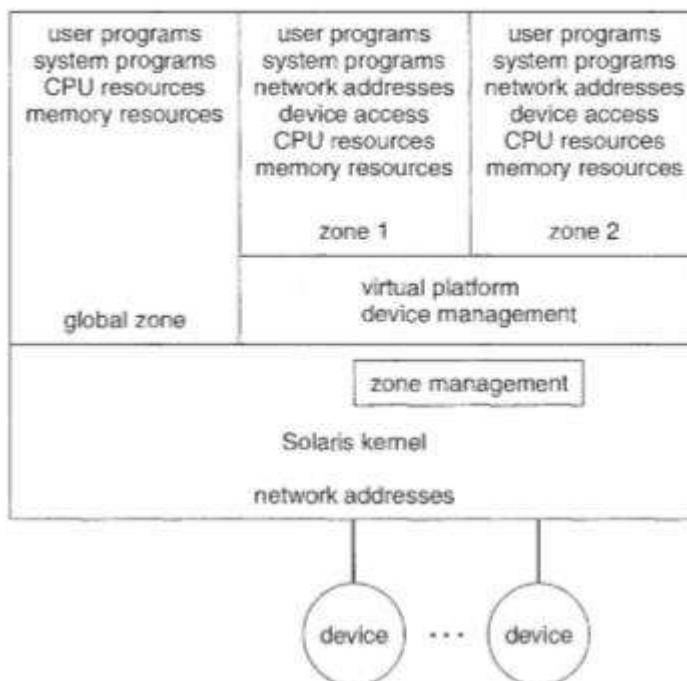


Figure 2.18 Solaris I 0 with two containers.

2.6.5 Implementation

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. Remember that the underlying machine typically has two modes: user mode and kernel mode. The virtual-machine software can run in kernel mode, since it is the operating system. The virtual machine itself can execute in only user mode. Just as the physical machine has two modes, however, so must the virtual machine. Consequently, we must have a virtual user mode and a virtual kernel mode, both of which run in a physical user mode. Those actions that cause a transfer from user mode to kernel mode on a real machine (such as a system call or an attempt to execute a privileged instruction) must also cause a transfer from virtual user mode to virtual kernel mode on a virtual machine.

Such a transfer can be accomplished as follows. When a system call for example, is made by a program running on a virtual machine in virtual user mode, it will cause a transfer to the virtual-machine monitor in the real machine. When the virtual-machine monitor gains control it can change the register contents and program counter for the virtual machine to simulate the effect of the system call. It can then restart the virtual machine, noting that it is now in virtual kernel mode.

The major difference, of course, is time. Whereas the real I/O might have taken 100 milliseconds, the virtual I/O might take less time (because it is spooled) or more time (because it is interpreted). In addition, the CPU is being multiprogrammed among many virtual machines, further slowing down the virtual machines in unpredictable ways. In the extreme case, it may be necessary to simulate all instructions to provide a true virtual machine. VM, discussed earlier, works for IBM machines because normal instructions for the virtual machines can execute directly on the hardware. Only the privileged instructions (needed mainly for I/O) must be simulated and hence execute more slowly.

Without some level of hardware support, virtualization would be impossible. The more hardware support available within a system, the more features rich, stable, and well performing the virtual machines can be. All major general-purpose CPUs provide some amount of hardware support for virtualization. For example, AMD virtualization technology is found in several AMD processors. It defines two new modes of operation—host and guest. Virtual machine software 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 the 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 certain devices (those included in the host's definition of the guest). If the guest tries to access a virtualized resource, then control is passed to the host to manage that interaction.

2.6.6 Examples

Despite the advantages of virtual machines, they received little attention for a number of years after they were first developed. Today, however, virtual machines are coming into fashion as a means of solving system compatibility problems. In this section, we explore two popular contemporary virtual machines: the VMware Workstation, the Java virtual machine

VMware Workstation

VMware Workstation is a popular commercial application that abstracts Intel X86 and compatible hardware into isolated virtual machines. VMware Workstation runs as an application on a host operating system such as Windows or Linux allows the host system to concurrently run several different guest operating systems as independent virtual machines

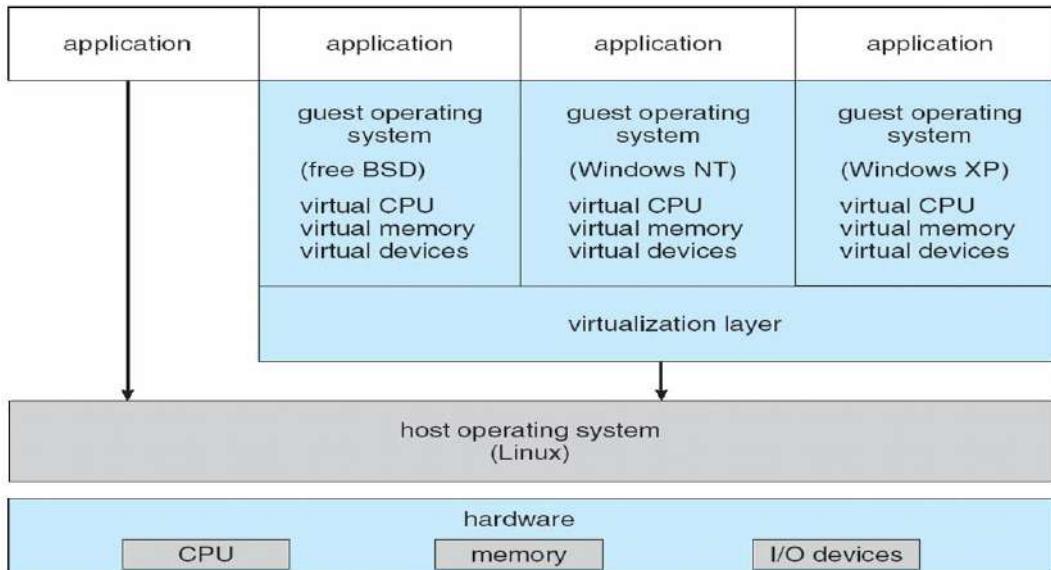


Figure 2.19 : VMWare Architecture

Linux is running as the host operating system; and FreeBSD, Windows NT, and Windows XP are running as guest operating systems. The virtualization layer is the heart of VMware, as it abstracts the physical hardware into isolated virtual machines running as guest operating systems. Each virtual machine has its own virtual CPU, memory, disk drives, network interfaces, and so forth.

The Java Virtual Machine(JVM)

Java is a popular object-oriented programming language introduced by Sun Microsystems in 1995.

Java provides a specification for a Java virtual machine-or JVM. Java objects are specified with the class construct having one or more classes. or each Java class, the compiler produces an architecture-neutral bytecode output (.class) file that will run on any implementation of the JVM.

The JVM is a specification for an abstract computer. It consists of a class loader and a Java interpreter that executes the architecture-neutral bytecodes. The class loader loads the compiled. class files from both the Java program and the Java API for execution by the Java interpreter

If the class passes verification, it is run by the Java interpreter. The JVM also automatically manages memory by performing **garbage collection** -the practice of reclaiming memory from objects no longer in use and returning it to the system. The JVM may be implemented in software on top of a host operating system, such as Windows, Linux, or Mac OS X, or as part of a Web browser. Alternatively, the JVM may be implemented in hardware on a chip specifically designed to run Java programs

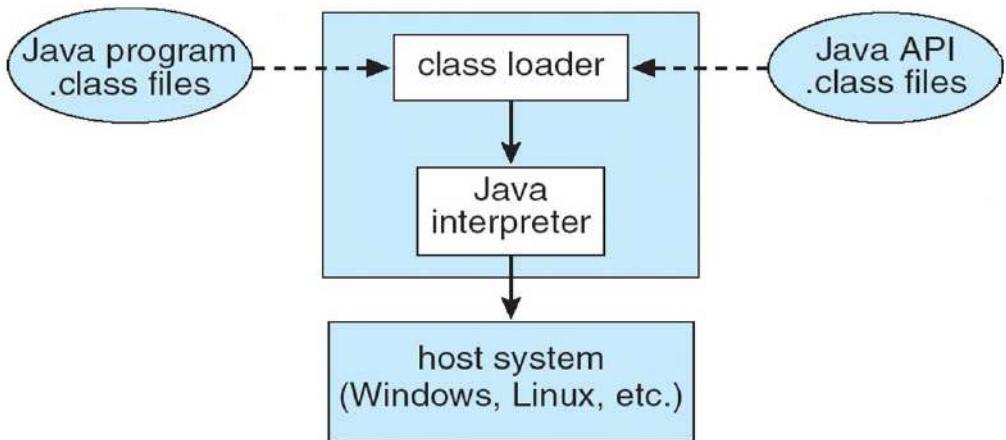


Figure 2.20 : JVM Architecture

2.7 Operating System Debugging

Broadly, debugging is the activity of finding and fixing errors, or bugs in a system. Debugging seeks to find and fix errors in both hardware and software. Performance problems are considered bugs, so debugging can also include performance tuning which seeks to improve performance by removing bottlenecks in the processing taking place within a system. Here we explore debugging kernel and process errors and performance problems.

2.7.1 Failure Analysis

If a process fails, most operating systems write the error information to a core dump to alert system operators or users that the problem occurred. The operating system can also take a capture of the memory (referred to as the "core" in the early days of computing) of the process. This core image is stored in a file for later analysis. Running programs and core dumps can be probed by a tool designed to allow a programmer to explore the code and memory a process.

Debugging user-level process code is a challenge. Operating system kernel debugging 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 kernel failure is called a *panic*. As with a process failure, error information is saved to a log file, and the memory state is saved to a

Operating system debugging frequently uses different tools and techniques than process debugging 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.

2.7.2 Performance Tuning

To identify bottlenecks, we must be able to monitor system performance. Code must be added to compute and display measures of system behavior. In a number of systems, the operating system does this task by producing trace listings of system behavior. All interesting events are logged with their time and important parameters and are written to a file. Later, an analysis program can process the log file to determine system performance and to identify bottlenecks and inefficiencies. These same traces can be run as input for a simulation of a suggested improved system. Traces also can help people to find errors in operating-system behavior.

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."

Another approach to performance tuning is to include interactive tools with the system that allow users and administrators to question the state of various components of the system to look for bottlenecks. The UNIX command top displays resources used on the system, as well as a sorted list of the "top" resource-using processes. Other tools display the state of disk I/O, memory allocation, and network traffic. The authors of these single-purpose tools try to guess what a user would want to see while analyzing a system and to provide that information.

Making running operating systems easier to understand, debug, and tune is an active area of operating system research and implementation. The cycle of enabling tracing as system problems occur and analyzing the traces later is being broken by a new generation of kernel-enabled performance analysis tools. Further, these tools are not single-purpose or merely for sections of code that were written to emit debugging data. The Solaris 10 DTrace dynamic tracing facility is a leading example of such a tool.

2.7.3 DTrace

is a facility that dynamically adds probes to a running system, both in user processes and in the kernel. These probes can be queried via the D programming language to determine an astonishing amount about the kernel, the system state, and process activities. For example, Figure 2.21 follows an application as it executes a system call (ioctl) and further shows the functional calls within the kernel as they execute to perform the system call. Lines ending with "U" are executed in user mode, and lines ending in "K" in kernel mode.

Debugging the interactions between user-level and kernel code is nearly impossible without a toolset that understands both sets of code and can instrument the 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 tool must also have a minimum performance impact-ideally it should have no impact when not in use and a proportional impact during use. The DTrace tool meets these requirements and provides a dynamic, safe, low-impact debugging environment. Until the DTrace framework and tools became available with Solaris 10, kernel debugging was usually shrouded in mystery and accomplished via happenstance and archaic code and tools. For example, CPUs have a breakpoint feature that will halt execution and allow a debugger to examine the state of the system. Then execution can continue until the next breakpoint or termination. This method cannot be used in a multiuser operating-system kernel without negatively affecting all of the users on the system. Code can be included in the kernel to emit specific data under specific circumstances, but that code slows down the kernel and tends not

to be included in the part of the kernel where the specific problem being debugged is occurring. In contrast, DTrace runs on production systems-systems that are running important or critical applications-and causes no harm to the system. It slows activities while enabled, but after execution it resets the system to its pre-debugging state. It is also a broad and deep tool. It can broadly debug everything happening in the system (both at the user and kernel levels and between the user and kernel layers). DTrace can also delve deeply into code, showing individual CPU instructions or kernel subroutine activities.

DTrace is composed of a compiler, a framework, provides of probes written within that framework, and of those probes.

DTrace providers create probes. Kernel structures exist to keep track of all probes that the providers have created. The probes are stored in a hash table data structure that is hashed by name and indexed according to unique probe identifiers. When a probe is enabled, a bit of code in the area to be probed is rewritten to call dtrace_probe (probe identifier) and then continue with the code's original operation. Different providers create different kinds of probes. For example, a kernel system-call probe works differently from a user-process probe, and that is different from an I/O probe. DTrace features a compiler that generates a byte code that is run in the kernel. This code is assured to be "safe" by the compiler. For example, no loops are allowed, and only specific kernel state modifications are allowed when specifically requested. Only users with the DTrace "privileges" (or "root" users) are allowed to use DT!·ace, as it can retrieve private kernel data (and modify data if requested). The generated code runs in the kernel and enables probes. It also enables consumers in user mode and enables communications between the two.

```
# ./all.d `pgrep xclock` XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued          U
  0 -> _XEventsQueued         U
  0 -> _X11TransBytesReadable U
  0 <- _X11TransBytesReadable U
  0 -> _X11TransSocketBytesReadable U
  0 <- _X11TransSocketBytesreadable U
  0 -> ioctl                  U
  0 -> ioctl                  K
  0 -> getf                  K
  0 -> set_active_fd          K
  0 <- set_active_fd          K
  0 -> getf                  K
  0 -> get_udatamodel         K
  0 <- get_udatamodel         K
  ...
  0 -> releasef              K
  0 -> clear_active_fd        K
  0 <- clear_active_fd        K
  0 -> cv_broadcast           K
  0 <- cv_broadcast           K
  0 -> releasef              K
  0 <- ioctl                 U
  0 -> _XEventsQueued         U
  0 <- _XEventsQueued         U
```

Figure 2.21 Solaris 10 dtrace follows a system call within the kernel.

A DTrace consumer is code that is interested in a probe and its results. A consumer

requests that the provider create one or more probes. When a probe fires, it emits data that are managed by the kernel.

Within the kernel,

actions called or are performed when probes fire. One probe can cause multiple ECBs to execute if more than one consumer is interested in that probe. Each ECB contains a predicate ("if statement") that can filter out that ECB. Otherwise, the list of actions in the ECB is executed. The most usual action is to capture some bit of data, such as a variable's value at that point of the probe execution. By gathering such data, a complete picture of a user or kernel action can be built. Further, probes firing from both user space and the kernel can show how a user-level action caused kernel-level reactions. Such data are invaluable for performance monitoring and code optimization.

Once the probe consumer terminates, its ECBs are removed. If there are no ECBs consuming a probe, the probe is removed. That involves rewriting the code to remove the `dtrace_probe` call and put back the original code. Thus, before a probe is created and after it is destroyed, the system is exactly the same, as if no probing occurred.

DTrace takes care to assure that probes do not use too much memory or CPU capacity, which could harm the running system. The buffers used to hold the probe results are monitored for exceeding default and maximum limits. CPU time for probe execution is monitored as well. If limits are exceeded, the consumer is terminated, along with the offending probes. Buffers are allocated per CPU to avoid contention and data loss.

Because DTrace is part of the open-source Solaris 10 operating system, it is being added to other operating systems when those systems do not have conflicting license agreements. For example, DTrace has been added to Mac OS X 10.5 and FreeBSD and will likely spread further due to its unique capabilities. Other operating systems, especially the Linux derivatives, are adding kernel-tracing functionality as well.

2.8 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 procedure of starting a computer by loading the kernel is known as *booting* the system. On most computer systems, a small piece of code known as the bootstrap program or bootstrap loader locates the kernel, loads it into main memory, and starts its execution. Some computer systems, such as PCs, use a two-step process in which a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel. When a CPU receives a reset event—for instance, when it is powered up or rebooted—the instruction register is loaded with a predefined memory location, and execution starts there. At that location is the initial bootstrap program. This program is in the form of read-only memory (ROM), because the RAM is in an unknown state at system startup. ROM is convenient because it needs no initialization and cannot easily be infected by a computer virus.

The bootstrap program can perform a variety of tasks. Usually, one task is to run diagnostics to determine the state of the machine. If the diagnostics pass, the program can continue with the booting steps. It 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.

Some systems—such as cellular phones, PDAs, and game consoles—store the entire operating system in ROM. Storing the operating system in ROM is suitable for small operating systems, simple supporting hardware, and rugged operation. A problem with this approach is that changing the bootstrap code requires changing the ROM hardware chips.

Some systems resolve this problem by using erasable programmable read-only memory (EPROM), which is read-only except when explicitly given a command to become writable. All forms of ROM are also known as firmware, since their characteristics fall somewhere between those of hardware and those of software. A problem with firmware in general is that executing code there is slower than executing code in RAM. Some systems store the operating system in firmware and copy it to RAM for fast execution. A final issue with firmware is that it is relatively expensive, so usually only small amounts are available.

For large operating systems (including most general-purpose operating systems like Windows, Mac OS X, and UNIX) or for systems that change frequently, the bootstrap loader is stored in firmware, and the operating system is on disk. In this case, the bootstrap runs diagnostics and has a bit of code that can read a single block at a fixed location (say block zero) from disk into memory and execute the code from that 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 fits in a single disk block) and knows only the address on disk and length of the remainder of the bootstrap program. is an example of an open-source bootstrap program for Linux systems. All of the disk-bound bootstrap, and the operating system itself, can be easily changed by writing new versions to disk.

Module 2

Introduction to Processes

All modern computers can do several things at the same time. While running a user program, a computer can also be reading from a disk and outputting text to a screen or printer. In a multiprogramming system, the CPU also switches from program to program, running each for tens or hundreds of milliseconds. While, strictly speaking, at any instant of time, the CPU is running only one program, in the course of 1 second, it may work on several programs, thus giving the users the illusion of parallelism. Sometimes people speak of **pseudoparallelism** in this context, to contrast it with the true hardware parallelism of **multiprocessor** systems (which have two or more CPUs sharing the same physical memory). Keeping track of multiple, parallel activities is hard for people to do. Therefore, operating system designers over the years have evolved a conceptual model (sequential processes) that makes parallelism easier to deal with.

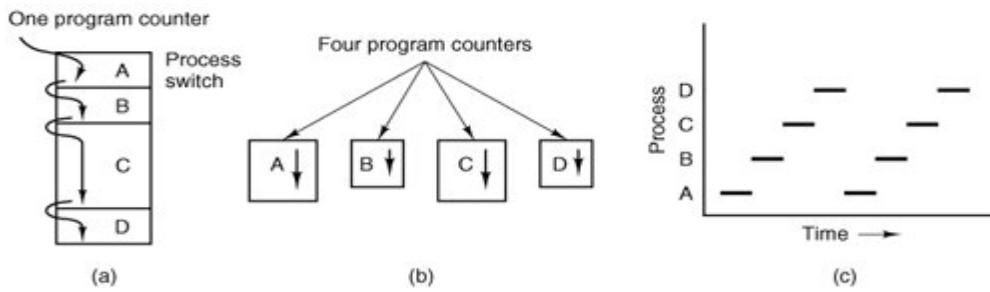
That model, its uses, and some of its consequences form the subject of this chapter.

2.1.1. The Process Model

In this model, all the runnable software on the computer, sometimes including the operating system, is organized into a number of **sequential processes**, or just **processes** for short. A process is just an executing program, including the current values of the program counter, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process to process, but to understand the system, it is much easier to think about a collection of processes running in (pseudo) parallel, than to try to keep track of how the CPU switches from program to program.

In [Fig. 2-1\(a\)](#) we see a computer multiprogramming four programs in memory. In [Fig. 2-1\(b\)](#) we see four processes, each with its own flow of control (i.e., its own program counter), and each one running independently of the other ones. Of course, there is only one physical program counter, so when each process runs, its logical program counter is loaded into the real program counter. When it is finished for the time being, the physical program counter is saved in the process' logical program counter in memory. In [Fig. 2-1\(c\)](#) we see that viewed over a long enough time interval, all the processes have made progress, but at any given instant only one process is actually running.

Figure 2-1. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at any instant.



With the CPU switching back and forth among the processes, the rate at which a process performs its computation will not be uniform, and probably not even reproducible if the same processes are run again. Thus, processes must not be programmed with built-in assumptions about timing. Consider, for example, an I/O process that starts a streamer tape to restore backed up files, executes an idle loop 10,000 times to let it get up to speed, and then issues a command to read the first record. If the CPU decides to switch to another process during the idle loop, the tape process might not run again until after the first record was already past the read head. When a process has critical real-time requirements like this, that is, particular events *must* occur within a specified number of milliseconds, special measures must be taken to ensure that they do occur. Normally, however, most processes are not affected by the underlying multiprogramming of the CPU or the relative speeds of different processes.

The difference between a process and a program is subtle, but crucial. An analogy may help make this point clearer. Consider a culinary-minded computer scientist who is baking a birthday cake for his daughter. He has a birthday cake recipe and a kitchen well stocked with the necessary input: flour, eggs, sugar, extract of vanilla, and so on. In this analogy, the recipe is the program (i.e., an algorithm expressed in some suitable notation), the computer scientist is the processor (CPU), and the cake ingredients are the input data. The process is the activity consisting of our baker reading the recipe, fetching the ingredients, and baking the cake.

Now imagine that the computer scientist's son comes running in crying, saying that he has been stung by a bee. The computer scientist records where he was in the recipe (the state of the current process is saved), gets out a first aid book, and begins following the directions in it. Here we see the processor being switched from one process (baking) to a higher priority process (administering medical care), each having a different program (recipe vs. first aid book). When the bee sting has been taken care of, the computer scientist goes back to his cake, continuing at the point where he left off.

The key idea here is that a process is an activity of some kind. It has a program, input, output, and a state. A single processor may be shared among several processes, with some scheduling algorithm being used to determine when to stop work on one process and service a different one.

2.1.2. Process Creation

Operating systems need some way to make sure all the necessary processes exist. In very simple systems, or in systems designed for running only a single application (e.g., controlling a device in real time), it may be possible to have all the processes that will ever be needed be present when the system comes up. In general-purpose systems, however, some way is needed to create and terminate processes as needed during operation.

There are four principal events that cause processes to be created:

1. System initialization.
2. Execution of a process creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

When an operating system is booted, often several processes are created. Some of these are foreground processes, that is, processes that interact with (human) users and perform work for them. Others are background processes, which are not associated with particular users, but instead have some specific function. For example, a background process may be designed to accept incoming requests for web pages hosted on that machine, waking up when a request arrives to service the request. Processes that stay in the background to handle some activity such as web pages, printing, and so on are called **daemons**. Large systems commonly have dozens of them. In MINIX 3, the *ps* program can be used to list the running processes.

In addition to the processes created at boot time, new processes can be created afterward as well. Often a running process will issue system calls to create one or more new processes to help it do its job. Creating new processes is particularly useful when the work to be done can easily be formulated in terms of several related, but otherwise independent interacting processes. For example, when compiling a large program, the *make* program invokes the C compiler to convert source files to object code, and then it invokes the *install* program to copy the program to its destination, set ownership and permissions, etc. In MINIX 3, the C compiler itself is actually several different programs, which work together. These include a preprocessor, a C language parser, an assembly language code generator, an assembler, and a linker.

In interactive systems, users can start a program by typing a command. In MINIX 3, virtual consoles allow a user to start a program, say a compiler, and then switch to an alternate console and start another program, perhaps to edit documentation while the compiler is running.

The last situation in which processes are created applies only to the batch systems found on large mainframes. Here users can submit batch jobs to the system (possibly remotely). When the operating system decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.

Technically, in all these cases, a new process is created by having an existing process execute a process creation system call. That process may be a running user process, a system process invoked from the keyboard or mouse, or a batch manager process. What that process does is execute a system call to create the new process. This system call tells the operating system to create a new process and indicates, directly or indirectly, which program to run in it.

In MINIX 3, there is only one system call to create a new process: `fork`. This call creates an exact clone of the calling process. After the `fork`, the two processes, the parent and the child, have the same memory image, the same environment strings, and the same open files. That is all there is. Usually, the child process then executes `execve` or a similar system call to change its memory image and run a new program. For example, when a user types a command, say, *sort*, to the shell, the shell forks off a child process and the child executes *sort*. The reason for this two-step process is to allow the child to manipulate its file descriptors after the `fork` but before the `execve` to accomplish redirection of standard input, standard output, and standard error.

In both MINIX 3 and UNIX, after a process is created both the parent and child have their own distinct address spaces. If either process changes a word in its address space, the change is not visible to the other process. The child's initial address space is a *copy* of the parent's, but there are two distinct address spaces

involved; no writable memory is shared (like some UNIX implementations, MINIX 3 can share the program text between the two since that cannot be modified). It is, however, possible for a newly created process to share some of its creator's other resources, such as open files.

2.1.3. Process Termination

After a process has been created, it starts running and does whatever its job is. However, nothing lasts forever, not even processes. Sooner or later the new process will terminate, usually due to one of the following conditions:

1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).

Most processes terminate because they have done their work. When a compiler has compiled the program given to it, the compiler executes a system call to tell the operating system that it is finished. This call is `exit` in MINIX 3. Screen-oriented programs also support voluntary termination. For instance, editors always have a key combination that the user can invoke to tell the process to save the working file, remove any temporary files that are open and terminate.

The second reason for termination is that the process discovers a fatal error. For example, if a user types the command

```
cc foo.c
```

to compile the program `foo.c` and no such file exists, the compiler simply exits.

The third reason for termination is an error caused by the process, perhaps due to a program bug. Examples include executing an illegal instruction, referencing nonexistent memory, or dividing by zero. In MINIX 3, a process can tell the operating system that it wishes to handle certain errors itself, in which case the process is signaled (interrupted) instead of terminated when one of the errors occurs.

The fourth reason a process might terminate is that one process executes a system call telling the operating system to kill some other process. In MINIX 3, this call is `kill`. Of course, the killer must have the necessary authorization to do in the killee. In some systems, when a process terminates, either voluntarily or otherwise, all processes it created are immediately killed as well.

2.1.4. Process Hierarchies

In some systems, when a process creates another process, the parent and child continue to be associated in certain ways. The child can itself create more

processes, forming a process hierarchy. Unlike plants and animals that use sexual reproduction, a process has only one parent (but zero, one, two, or more children).

In MINIX 3, a process, its children, and further descendants together may form a process group. When a user sends a signal from the keyboard, the signal may be delivered to all members of the process group currently associated with the keyboard (usually all processes that were created in the current window). This is signal-dependent. If a signal is sent to a group, each process can catch the signal, ignore the signal, or take the default action, which is to be killed by the signal.

As a simple example of how process trees are used, let us look at how MINIX 3 initializes itself. Two special processes, the **reincarnation server** and **init** are present in the boot image. The reincarnation server's job is to (re)start drivers and servers. It begins by blocking, waiting for a message telling it what to create.

In contrast, *init* executes the */etc/rc* script that causes it to issue commands to the reincarnation server to start the drivers and servers not present in the boot image. This procedure makes the drivers and servers so started children of the reincarnation server, so if any of them ever terminate, the reincarnation server will be informed and can restart (i.e., reincarnate) them again. This mechanism is intended to allow MINIX 3 to tolerate a driver or server crash because a new one will be started automatically. In practice, replacing a driver is much easier than replacing a server, however, since there fewer repercussions elsewhere in the system. (And, we do not say this always works perfectly; it is still work in progress.)

When *init* has finished this, it reads a configuration file */etc/ttymtab* to see which terminals and virtual terminals exist. *Init* forks a *getty* process for each one, displays a login prompt on it, and then waits for input. When a name is typed, *getty* execs a *login* process with the name as its argument. If the user succeeds in logging in, *login* will exec the user's shell. So the shell is a child of *init*. User commands create children of the shell, which are grandchildren of *init*. This sequence of events is an example of how process trees are used. As an aside, the code for the reincarnation server and *init* is not listed in this book; neither is the shell. The line had to be drawn somewhere. But now you have the basic idea.

2.1.5. Process States

Although each process is an independent entity, with its own program counter registers, stack, open files, alarms, and other internal state, processes often need to interact, communicate, and synchronize with other processes. One process may generate some output that another process uses as input, for example. In that case, the data needs to be moved between processes. In the shell command

```
cat chapter1 chapter2 chapter3 | grep tree
```

the first process, running *cat*, concatenates and outputs three files. The second process, running *grep*, selects all lines containing the word "tree." Depending on the relative speeds of the two processes (which depends on both the relative complexity of the programs and how much CPU time each one has had), it may happen that *grep* is ready to run, but there is no input waiting for it. It must then **block** until some input is available.

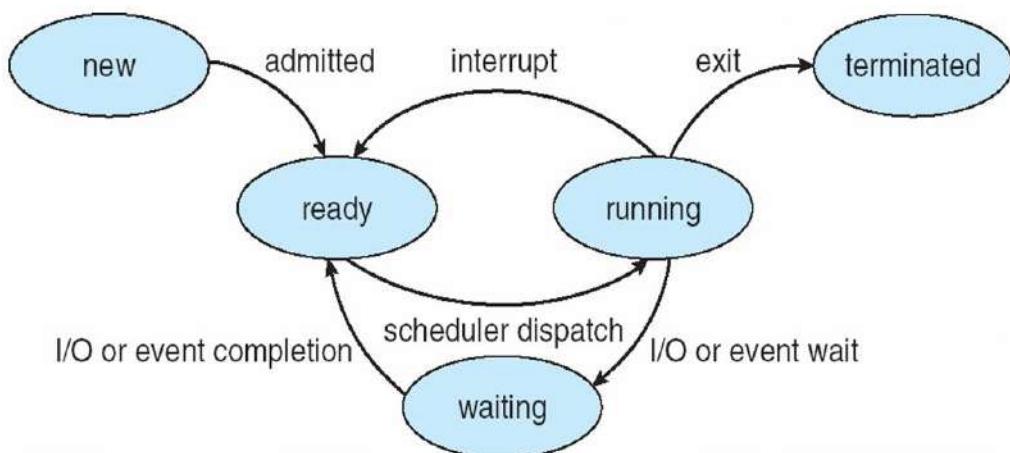
When a process blocks, it does so because logically it cannot continue, typically because it is waiting for input that is not yet available. It is also possible for a process that is conceptually ready and able to run to be stopped because the operating system has decided to allocate the CPU to another process for a while. These two conditions are completely different. In the first case, the suspension is inherent in the problem (you cannot process the user's command line until it has

been typed). In the second case, it is a technicality of the system (not enough CPUs to give each process its own private processor). In [Fig. 2-2](#) we see a state diagram showing the five states a process may be in:

- As a process executes, it changes *state*.
- Each process may be in one of the following states:
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **Waiting or blocked**: The process is waiting for some event to occur ,such as I/O operation
 - **ready**: The process is waiting to be assigned to a processor
 - **terminated**: The process has finished execution

Figure 2-2. A process can be in new, terminated, running, blocked, or ready state.

Transitions between these states are as shown below.



Logically, the first two states are similar. In both cases the process is willing to run, only in the second one, there is temporarily no CPU available for it. The third state is different from the first two in that the process cannot run, even if the CPU has nothing else to do.

Four transitions are possible among these three states, as shown. Transition 1 occurs when a process discovers that it cannot continue. In some systems the process must execute a system call, **block** or **pause** to get into blocked state. In other systems, including MINIX 3, when a process reads from a pipe or special file (e.g., a terminal) and there is no input available, the process is automatically moved from the running state to the blocked state.

Transitions 2 and 3 are caused by the process scheduler, a part of the operating-system, without the process even knowing about them. Transition 2 occurs when the scheduler decides that the running process has run long enough, and it is time to let another process have some CPU time. Transition 3 occurs when all the other processes have had their fair share and it is time for the first process to get the CPU

to run again. The subject of scheduling deciding which process should run when and for how long is an important one. Many algorithms have been devised to try to balance the competing demands of efficiency for the system as a whole and fairness to individual processes. We will look at scheduling and study some of these algorithms later in this chapter.

Transition 4 occurs when the external event for which a process was waiting (e.g., the arrival of some input) happens. If no other process is running then, transition 3 will be triggered immediately, and the process will start running. Otherwise it may have to wait in *ready* state for a little while until the CPU is available.

Using the process model, it becomes much easier to think about what is going on inside the system. Some of the processes run programs that carry out commands typed in by a user. Other processes are part of the system and handle tasks such as carrying out requests for file services or managing the details of running a disk or a tape drive. When a disk interrupt occurs, the system may make a decision to stop running the current process and run the disk process, which was blocked waiting for that interrupt. We say "may" because it depends upon relative priorities of the running process and the disk driver process. But the point is that instead of thinking about interrupts, we can think about user processes, disk processes, terminal processes, and so on, which block when they are waiting for something to happen. When the disk block has been read or the character typed, the process waiting for it is unblocked and is eligible to run again.

2.1.6. Implementation of Processes

To implement the process model, the operating system maintains a table (an array of structures), called the **process table**, with one entry per process. (Some authors call these entries **process control blocks**.) This entry contains information about the process' state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, alarms and other signals, and everything else about the process that must be saved

when the process is switched from *running* to *ready* state so that it can be restarted later as if it had never been stopped.

To implement the process model, the operating system maintains a table (an array of structures), called the **process table or process control blocks**, with one entry per process.

This entry contains information about the

- process' state,
- its program counter,
- stack pointer,
- memory allocation,
- the status of its open files,
- its accounting and scheduling information,
- alarms and other signals, etc
- the process that must be saved when the process is switched from *running* to *ready* state so that it can be restarted later as if it had never been stopped

Process Control Block (PCB)



Figure 2.3 : Process Control Block

CPU Switch From Process to Process

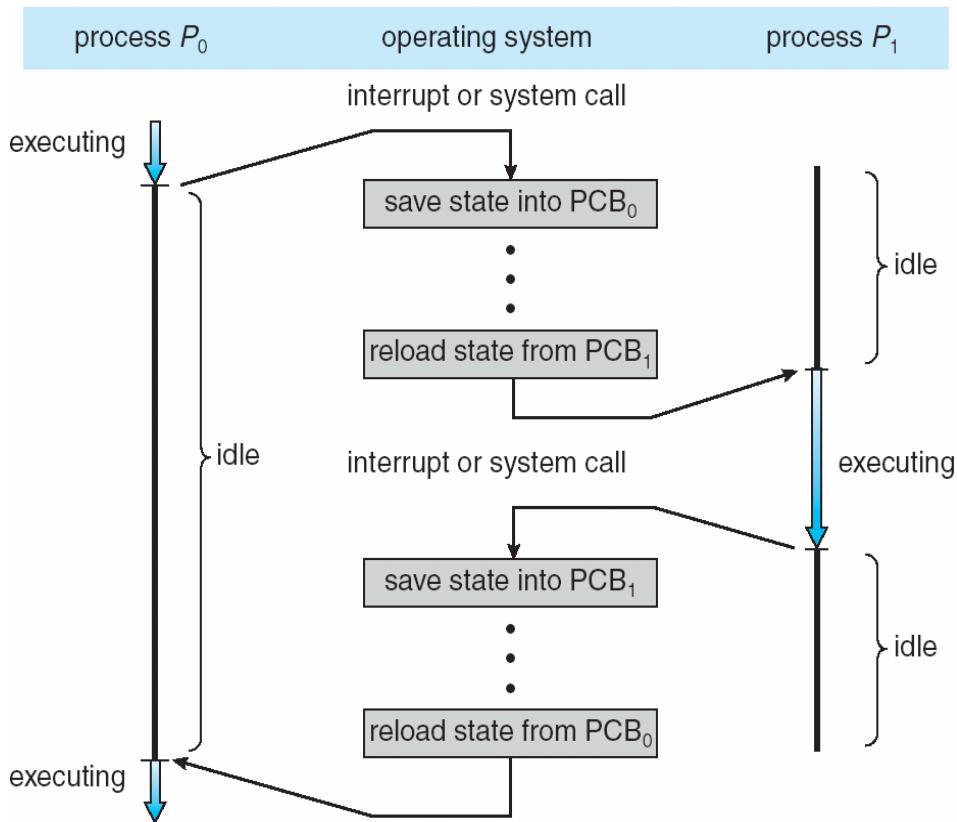


Figure 2.4 : CPU Switch From Process to Process

In MINIX 3, interprocess communication, memory management, and file management are each handled by separate modules within the system, so the process table is partitioned, with each module maintaining the fields that it needs. [Figure 2-4](#) shows some of the more important fields. The fields in the first column

are the only ones relevant to this chapter. The other two columns are provided just to give an idea of what information is needed elsewhere in the system.

Figure 2-4. Some of the fields of the MINIX 3 process table.

The fields are distributed over the kernel, the process manager, and the file system.

Kernel	Process management	File management
Registers	Pointer to text segment	UMASK mask
Program counter	Pointer to data segment	Root directory
Program status word	Pointer to bss segment	Working directory
Stack pointer	Exit status	File descriptors
Process state	Signal status	Real id
Current scheduling priority	Process ID	Effective UID
Maximum scheduling priority	Parent process	Real GID
Scheduling ticks left	Process group	Effective GID
Quantum size	Children's CPU time	Controlling tty
CPU time used	Real UID	Save area for read/write
Message queue pointers	Effective UID	System call parameters
Pending signal bits	Real GID	Various flag bits
Various flag bits	Effective GID	
Process name	File info for sharing text Bitmaps for signals Various flag bits Process name	

Atomic Transaction

- The mutual exclusion of critical sections ensure that the critical sections are executed atomically
- That is, as one uninterruptible unit.
- If two critical sections are instead executed concurrently, the result is equivalent to their sequential execution in some unknown order.

Atomic Transaction

- **Consistency of data, along with storage and retrieval of data, is a concern often associated with database system.**
- Recently, there has been an increase of interest in using database-systems techniques in operating systems.
- Operating systems can be viewed as manipulators of data
- They can benefit from the advanced techniques and models available from database research.

System Model

- A collection of instructions (or operations) that performs a single logical function is called a Transaction
- A major issue in processing transactions is the preservation of atomicity despite the possibility of failures within the computer system.

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- A transaction must see a consistent data.
- During transaction execution the data may be temporarily inconsistent.
- When the transaction completes successfully (is committed), the data must be consistent.

Transaction Concept

- After a transaction commits, the changes it has made to the data persist, even if there are system failures.
- Multiple transactions can execute in parallel.
- Two main issues to deal with:
 - Failures of various kinds, such as hardware failures and system crashes
 - Concurrent execution of multiple transactions

Transaction

- **Atomicity.** Either all operations of the transaction are properly reflected in the data or none are.
- From our point of view, such a transaction is simply a sequence of read and write operations terminated by either a commit operation or an abort operation.
- A **commit operation** signifies that the transaction has terminated its execution successfully
- An **abort operation** signifies that the transaction has ended its normal execution due to some logical error or a system failure

Example of Fund Transfer

- Transaction to transfer Rs. 50 from account A to account B:
 1. **read(A)**
 2. $A := A - 50$
 3. **write(A)**
 4. **read(B)**
 5. $B := B + 50$
 6. **write(B)**
- **Atomicity requirement**
 - If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
 - Failure could be due to software or hardware
 - **The system should ensure that updates of a partially executed transaction are not reflected in the database**

Example

- Suppose we have a function update () that modifies shared data.
- Traditionally, this function would be written using locks such as the following:

```
update ()  
{  
    acquire();  
}  
/*modify shared data */  
release();
```

Alternative Approach

- In our example, suppose we add the construct `atomic{s}`, which ensures that the operations in `s` execute as a transaction.
- This allows us to rewrite the `update ()` method as follows:

```
update ()  
{  
atomic  
{  
/* modify shared data */  
}  
}
```

Storage Structure

- To ensure the atomicity and durability properties of a transaction,
- Storage structure specifies how the various data items in the database may be stored and accessed.
- Three classes of storage
- Volatile Storage
- Nonvolatile Storage
- Stable Storage

Storage Structure

- **Volatile storage**
- **Information residing in volatile storage does not usually survive system crashes.**
- Examples : **main memory and cache memory.**
- Access to volatile storage is extremely fast
- It is possible to access any data item in volatile storage directly.

Storage Structure

- **Nonvolatile storage.**
- Information residing in nonvolatile storage survives system crashes.
- Examples : secondary storage devices such as magnetic disk and flash storage and tertiary storage devices such as optical media, and magnetic tapes.
- nonvolatile storage is slower than volatile storage.
- Both secondary and tertiary storage devices, are susceptible to failure which may result in loss of information.

Storage Structure

- **Stable storage.** Information residing in stable storage is *never* lost
- Although stable storage is theoretically impossible to obtain, it can be closely approximated by techniques that make data loss extremely unlikely.
- To implement stable storage, we replicate the information in several nonvolatile storage media (usually disk) with independent failure modes.
- Updates must be done with care to ensure that a failure during an update to stable storage does not cause a loss of information.

Stable Storage

- **For a transaction to be durable**, its changes need to be written to stable storage.
- **For a transaction to be atomic**, log records need to be written to stable storage before any changes are made to the database on disk.
- The degree to which a system ensures durability and atomicity depends on how stable its implementation of stable storage really is.
- In some cases, a single copy on disk is considered sufficient,
- **but applications whose data are highly valuable and whose transactions are highly important require multiple copies as the stable storage.**

Log-Based Recovery

- The most widely used structure for recording database modifications is the **log**.
- The log is a sequence of **log records**, recording all the update activities in the database.
- There are several types of log records.
- An **update log record** describes a single database write.
- It has these fields:
 1. **Transaction identifier**, which is the unique identifier of the transaction that performed the write operation.

Log-Based Recovery

2. Data-item identifier, which is the unique identifier of the data item written.

- Typically, it is the location on disk of the data item,
- Consisting of the block identifier of the block on which the data item resides, and
- An offset within the block.

3. Old value, which is the value of the data item prior to the write.

4. New value, which is the value that the data item will have after the write.

Log-Based Recovery

- We represent an update log record as $\langle T_i, X_j, V_1, V_2 \rangle$,
- Indicating that transaction T_i has performed a write on data item X_j .
- X_j had value V_1 before the write, and has value V_2 after the write.

Log-Based Recovery

- Other special log records exist to record significant events during transaction processing,
- Such as the start of a transaction and the commit or abort of a transaction.
- **Among the types of log records are:**
 - **$\langle T_i, \text{start} \rangle$. Transaction T_i has started.**
 - **$\langle T_i, \text{commit} \rangle$. Transaction T_i has committed.**
 - **$\langle T_i, \text{abort} \rangle$. Transaction T_i has aborted.**

Database Modification

- A transaction creates a log record prior to modifying the database.
- The log records allow the system to undo changes made by a transaction in the event that the transaction must be aborted
- They allow the system also to redo changes made by a transaction if the transaction has committed
- But the system crashed before those changes could be stored in the database on disk.

Concurrent Executions

- **Multiple transactions are allowed to run concurrently in the system.**
- Advantages are:
 - **increased processor and disk utilization**, leading to better transaction *throughput*:
 - one transaction can be using the CPU while another is reading from or writing to the disk
 - **reduced average response time** for transactions: **short transactions need not wait behind long ones.**

Concurrent Execution - Example

Consider again the simplified banking system, which has several accounts, and a set of transactions that access and update those accounts.

Let T_1 and T_2 be two transactions that transfer funds from one account to another. Transaction T_1 transfers \$50 from account A to account B . It is defined as:

```
 $T_1$ : read( $A$ );  
 $A := A - 50$ ;  
write( $A$ );  
read( $B$ );  
 $B := B + 50$ ;  
write( $B$ ).
```

Transaction T_2 transfers 10 percent of the balance from account A to account B . It is defined as:

```
 $T_2$ : read( $A$ );  
 $temp := A * 0.1$ ;  
 $A := A - temp$ ;  
write( $A$ );  
read( $B$ );  
 $B := B + temp$ ;  
write( $B$ ).
```

Schedule 1

- Suppose the current values of accounts A and B are \$1000 and \$2000, respectively.
- Suppose also that the two transactions are executed one at a time in the order T_1 followed by T_2 .
- Let T_1 transfer \$50 from A to B , and T_2 transfer 10% of the balance from A to B .
- A **serial** schedule in which T_1 is followed by T_2 :

The final values of accounts A and B , after the execution takes place, are \$855 and \$2145, respectively.

Thus, the total amount of money in accounts A and B —that is, the sum $A + B$ —is preserved after the execution of both transactions.

T_1	T_2
read(A) $A := A - 50$ write (A) read(B) $B := B + 50$ write(B)	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B) $B := B + temp$ write(B)

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- **Serial execution of a set of transactions preserves database consistency.**
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.

Type : conflict serializability

- We ignore operations other than **read** and **write** instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Schedule 2

T_1	T_2
read(A) $A := A - 50$	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B)
write(A) read(B) $B := B + 50$ write(B)	$B := B + temp$ write(B)

Checkpoints

- Some earlier part of the log may be needed for undo operations
 - Continue scanning backwards till a record $\langle T_i, \text{start} \rangle$ is found for every transaction T_i in L .
 - Parts of log prior to earliest $\langle T_i, \text{start} \rangle$ record above are not needed for recovery, and can be erased whenever desired.

Checkpoints

- Streamline recovery procedure by periodically performing **check pointing**
 1. Output all log records currently residing in main memory onto stable storage.
 2. Output all modified buffer blocks to the disk.
 3. Write a log record < checkpoint L > onto stable storage where L is a list of all transactions active at the time of checkpoint.
 4. All updates are stopped while doing checkpointing

Checkpoints

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i .
 - Scan backwards from end of log to find the most recent <checkpoint L > record
 - Only transactions that are in L or started after the checkpoint need to be redone or undone
 - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.

Lock-Based Protocols

- One way to ensure isolation is to require that data items be accessed in a mutually exclusive manner
- that is, while one transaction is accessing a data item, no other transaction can modify that data item
- The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item.

Lock-Based Protocols

- Data items can be locked in two modes :
 1. *exclusive (X) mode*. Data item can be both read as well as written.
X-lock is requested using **lock-X** instruction.
 2. *shared (S) mode*. Data item can only be read.
S-lock is requested using **lock-S** instruction.
- **Lock requests are made to the concurrency-control manager by the programmer.**
- **Transaction can proceed only after request is granted.**

Locking Protocol

- Each transaction in the system follow a set of rules, called a **locking protocol**,
- **Indicating when a transaction may lock and unlock each of the data items.**
- **Locking protocols restrict the number of possible schedules.**
- We shall present several locking protocols that allow only **conflict-serializable schedules**, and thereby ensure isolation.

Two-Phase Locking Protocol

- One protocol that ensures serializability is the **two-phase locking protocol**.
- This protocol requires that each transaction issue lock and unlock requests in two phases:
- **Growing phase.** A transaction may obtain locks, but may not release any lock.
- **Shrinking phase.** A transaction may release locks, but may not obtain any new locks.
- Initially, a transaction is in the growing phase in which the transaction acquires locks as needed.
- Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system.
- If an old transaction T_i has time-stamp $\text{TS}(T_i)$, a new transaction T_j is assigned time-stamp $\text{TS}(T_j)$ such that $\text{TS}(T_i) < \text{TS}(T_j)$.
- There are two simple methods for implementing this scheme:
 - Use the value of the **system clock** as the timestamp;
 - That is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.
 - Use a **logical counter** that is incremented after a new timestamp has been assigned;
 - That is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

Timestamp-Based Protocols

- The timestamps of the transactions determine the serializability order.
- Thus, if $TS(T_i) < TS(T_j)$, then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction T_i appears before transaction T_j .
- To implement this scheme, we associate with each data item Q two timestamp values:
- **W-timestamp(Q)** denotes the largest timestamp of any transaction that executed $\text{write}(Q)$ successfully.
- **R-timestamp(Q)** denotes the largest timestamp of any transaction that executed $\text{read}(Q)$ successfully.
- These timestamps are updated whenever a new $\text{read}(Q)$ or $\text{write}(Q)$ instruction is executed.

The Timestamp-Ordering Protocol

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction T_i issues a **read**(Q)
 1. If $TS(T_i) \leq W\text{-timestamp}(Q)$, then T_i needs to read a value of Q that was already overwritten.
Hence, the **read** operation is rejected, and T_i is rolled back.
 2. If $TS(T_i) \geq W\text{-timestamp}(Q)$, then the **read** operation is executed, and $R\text{-timestamp}(Q)$ is set to $\max(R\text{-timestamp}(Q), TS(T_i))$.

The Timestamp-Ordering Protocol

- Suppose that transaction T_i issues **write**(Q).
 1. If $\text{TS}(T_i) < \text{R-timestamp}(Q)$, then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.

Hence, the **write** operation is rejected, and T_i is rolled back.
 2. If $\text{TS}(T_i) < \text{W-timestamp}(Q)$, then T_i is attempting to write an obsolete value of Q .

Hence, this **write** operation is rejected, and T_i is rolled back.
 3. Otherwise, the **write** operation is executed, and $\text{W-timestamp}(Q)$ is set to $\text{TS}(T_i)$.

Example

- To illustrate this protocol, we consider transactions T_{25} and T_{26} .
Transaction
- T_{25} displays the contents of accounts A and B :

T_{25} : $\text{read}(B);$
 $\text{read}(A);$
 $\text{display}(A + B).$

- Transaction T_{26} transfers \$50 from account B to account A , and then displays the contents of both:

T_{26} : $\text{read}(B);$
 $B := B - 50;$
 $\text{write}(B);$
 $\text{read}(A);$
 $A := A + 50;$
 $\text{write}(A);$
 $\text{display}(A + B).$

Example

- In presenting schedules under the timestamp protocol,
- we shall assume that a transaction is assigned a timestamp immediately before its first instruction.
- Thus, in schedule 3 of Figure , $TS(T_{25}) < TS(T_{26})$, and the schedule is possible under the timestamp protocol.
- The timestamp-ordering protocol ensures conflict serializability.
- This is because conflicting operations are processed in timestamp order.

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
	display($A + B$)
	$A := A + 50$
	write(A)
	display($A + B$)

INTERPROCESS COMMUNICATION

- Processes executing concurrently in the operating system may be either independent or cooperating processes.
- Reasons for providing an environment that allows process cooperation.

1) Information Sharing

Several users may be interested in the same piece of information.

2) Computational Speed up

Process can be divided into sub tasks to run faster, speed up can be achieved if the computer has multiple processing elements.

3) Modularity

Dividing the system functions into separate processes or threads.

4) Convenience

Even an individual user may work on many tasks at the same time.

COMMUNICATION MODELS

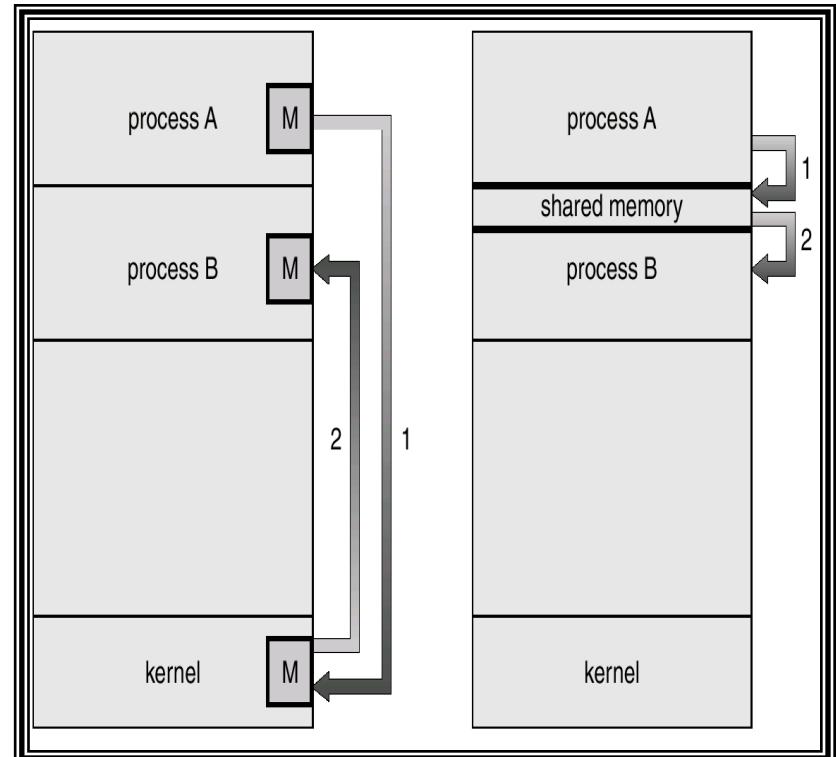
Cooperating processes require IPC mechanism that allow them to exchange data and information. Communication can take place either by Shared memory or Message passing Mechanisms.

Shared Memory:

- 1) Processes can exchange information by reading and writing data to the shared region.
- 2) Faster than message passing as it can be done at memory speeds when within a computer.
- 3) System calls are responsible only to establish shared memory regions.

Message Passing:

Mechanism to allow processes to communicate and synchronize their actions without sharing the same address space and is particularly useful in distributed environment.



Message Passing Communication

- Messages are collection of data objects and their structures
- Messages have a header containing system dependent control information and a message body that can be fixed or variable size.
- When a process interacts with another, two requirements have to be satisfied.

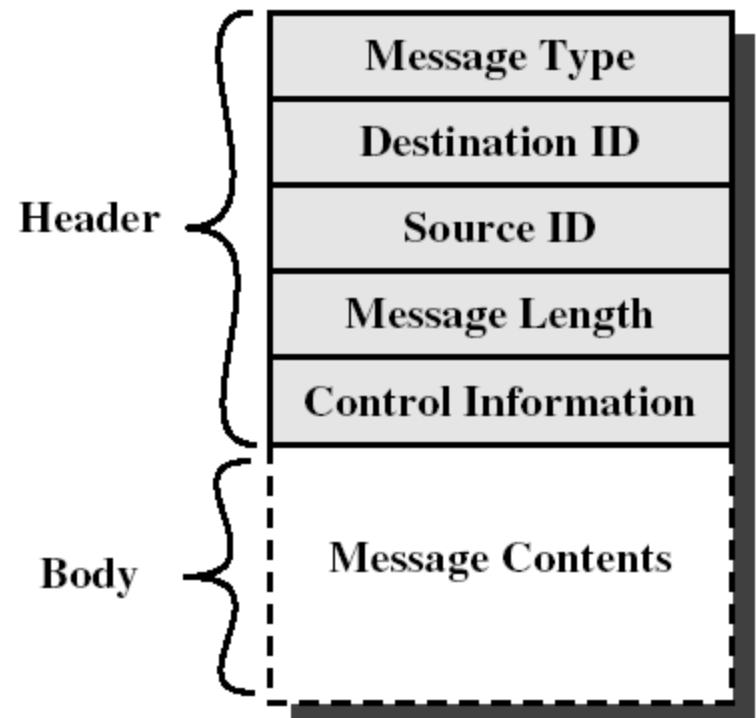
Synchronization and Communication.

Fixed Length

- Easy to implement
- Minimizes processing and storage overhead.

Variable Length

- Requires dynamic memory allocation, so fragmentation could occur.



Basic Communication Primitives

- Two generic message passing primitives for sending and receiving messages.
send (destination, message)
receive (source, message) source or dest={ process name, link, mailbox, port}

Addressing - Direct and Indirect

1) Direct Send/ Receive communication primitives

Communication entities can be addressed by process names (global process identifiers)

Global Process Identifier can be made unique by concatenating the network host address with the locally generated process id. This scheme implies that only one direct logical communication path exists between any pair of sending and receiving processes.

Symmetric Addressing : Both the processes have to explicitly name in the communication primitives.

Asymmetric Addressing : Only sender needs to indicate the recipient.

Message Passing

- If processes P and Q wish to communicate, they need to:
 - Establish a ***communication link*** between them
 - Exchange messages via send/receive
- Implementation issues:
 - How are links established?
 - Can a link be associated with more than two processes?
 - How many links can there be between every pair of communicating processes?
 - What is the capacity of a link?
 - Is the size of a message that the link can accommodate fixed or variable?
 - Is a link unidirectional or bi-directional?

Message Passing

- Implementation of communication link
 - Physical:
 - Shared memory
 - Hardware bus
 - Network
 - Logical:
 - Direct or indirect
 - Synchronous or asynchronous
 - Automatic or explicit buffering

Pipe & Socket API's

- More convenient to the users and to the system if the communication is achieved through a well defined set of standard API's.

Pipe

- Pipes are implemented with finite size, FIFO byte stream buffer maintained by the kernel.
- Used by 2 communicating processes, a pipe serves as unidirectional communication link so that one process can write data into tail end of pipe while another process may read from head end of the pipe.
- Pipe is created by a system call which returns 2 file descriptors, one for reading and another for writing.
- Pipe concept can be extended to include messages.
- For unrelated processes, there is need to uniquely identify a pipe since pipe descriptors cannot be shared. So concept of Named pipes.
- With a unique path name, named pipes can be shared among disjoint processes across different machines with a common file system.

SOCKETS

- A Socket is a communication end point of a communication link managed by the transport services.
It is not feasible to name a communication channel across different domains.
A Communication channel can be visualized as a pair of 2 communication endpoints.
- Sockets have become most popular message passing API.
Most recent version of the Windows Socket which is developed by WinSock Standard Group which has 32 companies (including Microsoft) also includes a SSL (Secure Socket Layer) in the specification.

The goal of SSL is to provide:

Privacy in socket communication by using symmetric cryptographic data encryption.
Integrity in socket data by using message integrity check.
Authenticity of servers and clients by using asymmetric public key cryptography.

Module 2 - Process Synchronization

- Race conditions
- Critical Sections
- Mutual exclusion
- Peterson's Solution
- Synchronization Hardware
- Semaphores

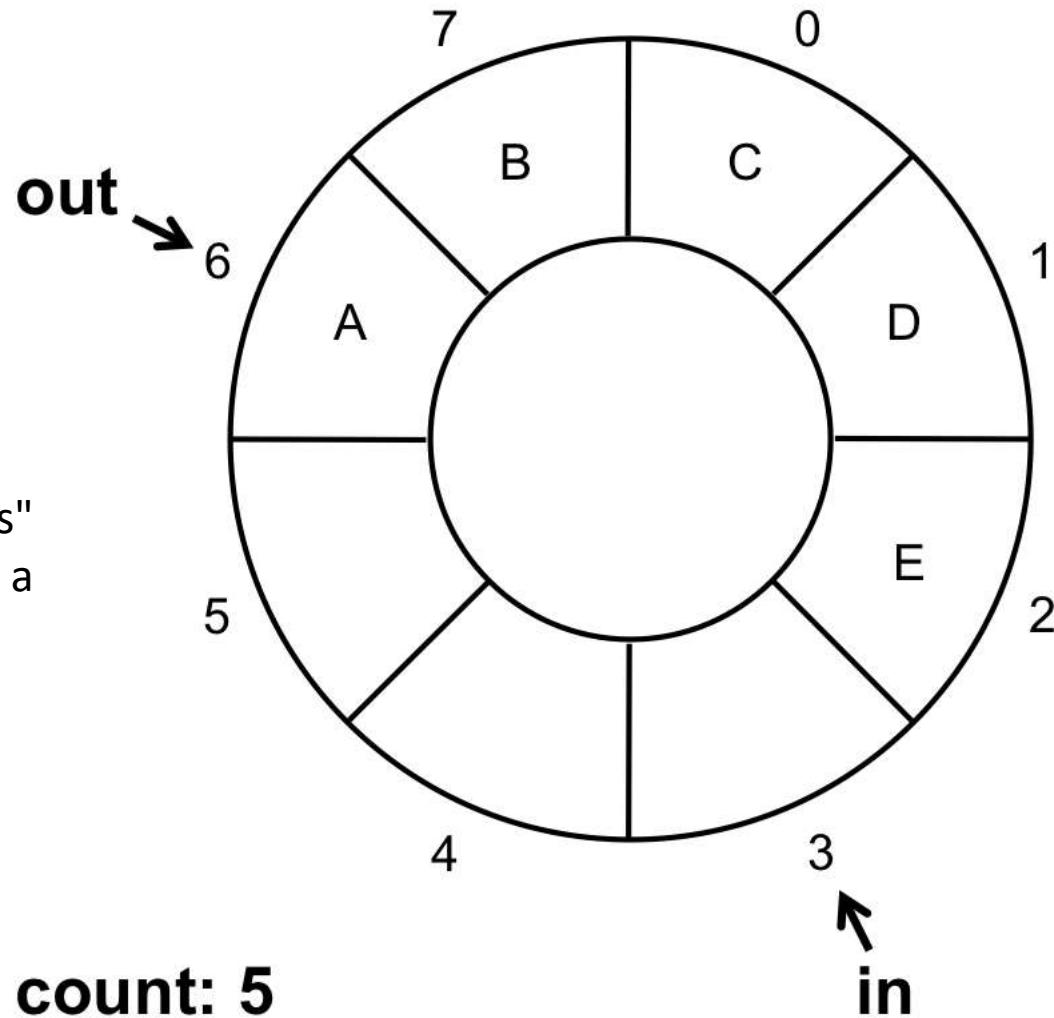
Process Synchronization

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the **producer-consumer** problem that fills **all** the buffers.
- We can do so by having an integer **count** that keeps track of the number of full buffers.
- Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Circular Buffer (Queue)

A **producer** process "produces" information "consumed" by a **consumer** process.

Count = 8



Producer

```
while (true) {  
  
    /* produce an item and put in nextProduced */  
  
    while (counter == BUFFER_SIZE)  
        ; // do nothing  
  
    buffer [in] = nextProduced;  
  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

Consumer

```
while (true) {  
  
    while (counter == 0)  
        ; // do nothing  
  
    nextConsumed = buffer[out];  
  
    out = (out + 1) % BUFFER_SIZE;  
  
    counter--;  
  
    /* consume the item in nextConsumed */  
}
```

Bounded Buffer

- Although both the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently.
- As an illustration, suppose that the value of the variable counter is currently 5 and that the producer and consumer processes execute the statements "counter++" and "counter--" concurrently.
- Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6!
- The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately

Bounded Buffer

- The statements

counter++;

counter--;

must be performed *atomically*.

- Atomic operation means an operation that completes in its entirety without interruption.

Bounded Buffer

- The statement “**count++**” may be implemented in machine language as:

register1 = counter

register1 = register1 + 1

counter = register1

- The statement “**count—**” may be implemented as:

register2 = counter

register2 = register2 – 1

counter = register2

Bounded Buffer

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.
- Interleaving depends upon how the producer and consumer processes are scheduled.

Bounded Buffer

- Assume **counter** is initially 5. One interleaving of statements is:

```
S0: producer execute register1 = counter {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = counter {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute counter = register1 {count = 6 }
S5: consumer execute counter = register2 {count = 4}
```

- Notice that we have arrived at the incorrect state "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full.
- If we reversed the order of the statements at S4 and S5 , we would arrive at the incorrect state "counter== 6".

Classic problem of synchronization – Producer Consumer or bounded buffer Problem

- We assume that the pool consists of n buffers, each capable of holding one item.
- The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.
- The empty and full semaphores count the number of empty and full buffers.
- The semaphore empty is initialized to the value n ; the semaphore full is initialized to the value 0

Classic problem of synchronization – Producer Consumer or bounded buffer Problem

- The code for the producer process and consumer process is shown in Figure .
- Note the symmetry between the producer and the consumer.
- We can interpret this code as the producer producing full buffers for the consumer or
- As the consumer producing empty buffers for the producer

Classic problem of synchronization – Producer Consumer Problem

```
do {  
    //produce an item in nextp  
    wait(empty);  
    wait(mutex);  
    // add nextp to buffer  
    signal(mutex);  
    signal(full);  
} while (TRUE);
```

Figure : The structure of the producer process.

```
do {  
    wait (full);  
    wait (mutex) ;  
    //remove an item from buffer  
    //to nextc  
    signal(mutex);  
    signal(empty);  
    //consume the item in nextc  
} while (TRUE);
```

Figure :The structure of the consumer process

Race Condition

- The incorrect state is because both processes allowed to manipulate the variable counter concurrently.
- A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.
- To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter.
- I.e.: to prevent race conditions, concurrent processes must be **synchronized**.

Critical Section

- Consider system of n processes $\{p_0, p_1, \dots p_{n-1}\}$
- Each process has **critical section** segment of code
 - Process may be changing common variables, updating table, writing file, etc
 - When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

General Structure

```
while (true) {  
    entry section  
    critical section  
    exit section  
    remainder section  
}
```

- Protocol:
 - Request entry to critical section
 - Execute critical section
 - Exit critical section
 - Execute remainder section

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
 - Assume that each process executes at a nonzero speed
 - No assumption concerning relative speed of the N processes

Peterson's Solution

- Software-based solution to the critical-section problem known as Peterson's solution.
- This solution provides a good algorithmic description of solving the critical-section problem
- **Also addresses the requirements of mutual exclusion, progress, and bounded waiting.**

Peterson's Solution

- Two process solution
- The two processes share two variables:
 - int **turn**;
 - Boolean **flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i]** = true implies that process **P_i** is ready!

Peterson's Solution

```
while ( true ) {  
    flag[i] = true;  
    turn = j;  
    while ( flag[j] && turn == j )  
        ; // busy wait  
    // critical section  
    // ...  
    flag[i] = false;  
    // remainder section  
    // ...  
}
```

entry section

exit section

Provable that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Peterson's Solution

P₀

```
i = 0, j = 1  
while (true) {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] &&  
           turn == j)  
        ; // busy wait  
    // critical section  
    // ...  
    flag[i] = false;  
    // remainder section  
    // ...  
}
```

P₁

```
i = 1, j = 0  
while (true) {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] &&  
           turn == j)  
        ; // busy wait  
    // critical section  
    // ...  
    flag[i] = false;  
    // remainder section  
    // ...  
}
```

Synchronization Hardware

- We have just described one software-based solution to the critical-section problem.
- However, as mentioned, software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures.
- Instead, we can generally state that any solution to the critical-section problem requires a simple tool-a **lock**.
- Race conditions are prevented by requiring that critical regions be protected by locks.
- That is, a process must acquire a lock before entering a critical section;
- it releases the lock when it exits the critical section.

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems
 - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
 - Atomic = non-interruptable
 - Either test memory word and set value
 - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);
```

Test And Set Instruction

- Definition:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

Solution using TestAndSet

- Shared boolean variable lock, initialized to FALSE
- Solution:

```
do {  
    while ( TestAndSet (&lock ) )  
        ; // do nothing  
  
        // critical section  
  
    lock = FALSE;  
  
        // remainder section  
  
} while (TRUE);
```

Swap Instruction

- Definition:

```
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key

- Solution:

```
do {  
    key = TRUE;  
    while ( key == TRUE)  
        Swap (&lock, &key );
```

// critical section

```
lock = FALSE;
```

// remainder section

```
} while (TRUE);
```

Bounded-waiting Mutual Exclusion with TestandSet()

```
do {  
    waiting[i] = TRUE;  
    key = TRUE;  
    while (waiting[i] && key)  
        key = TestAndSet(&lock);  
    waiting[i] = FALSE;  
        // critical section  
    j = (i + 1) % n;  
    while ((j != i) && !waiting[j])  
        j = (j + 1) % n;  
    if (j == i)  
        lock = FALSE;  
    else  
        waiting[j] = FALSE;  
        // remainder section  
} while (TRUE);
```

Semaphore

- The hardware-based solutions to the critical-section problem are complicated for application programmers to use.
- To overcome this difficulty, we can use a synchronization tool called a semaphore

Semaphore

- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () and signal ().
- The wait () operation was originally termed P (from the Dutch proberen, "to test");
- signal() was originally called V (from verhogen, "to increment"). The definition of wait () is as follows:

Semaphores

Definition of wait and signal

wait (S):

while $S \leq 0$ do no-op;

$S--;$

signal (S):

$S++;$

Critical Section of n Processes

- **Operating systems often distinguish between counting and binary semaphores.**
- The value of a counting semaphore can range over an unrestricted domain.
- We can use binary semaphores to deal with the critical-section problem for multiple processes
- The value of a binary semaphore can range only between 0 and 1
- Shared data:
semaphore mutex; //initially mutex = 1
- Process P_i :
do {
 wait(mutex);
 critical section
 signal(mutex);
 remainder section
} while (1);

Counting Semaphores

- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a `wait()` operation on the semaphore (thereby decrementing the count).
- When a process releases a resource, it performs a `signal()` operation (incrementing the count).
- When the count for the semaphore goes to 0, all resources are being used.
- After that, processes that wish to use a resource will block until the count becomes greater than 0.
- We can also use semaphores to solve various synchronization problems.

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section
 - Could now have **busy waiting** in critical section implementation
 - But implementation code is short
 - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
 - value (of type integer)
 - pointer to next record in the list
- Two operations:
 - **block** – place the process invoking the operation on the appropriate waiting queue
 - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Semaphore Implementation

- Define a semaphore as a record

```
typedef struct {  
    int value;  
    struct process *L;  
} semaphore;
```

- Assume two simple operations:
 - **block** suspends the process that invokes it.
 - **wakeup(*P*)** resumes the execution of a blocked process **P**.

Implementation

- Semaphore operations now defined as

wait(S):

```
S.value--;
if (S.value < 0) {
    add this process to S.L;
    block;
}
```

signal(S):

```
S.value++;
if (S.value <= 0) {
    remove a process P from S.L;
    wakeup(P);
}
```

Problems with Semaphores

- Incorrect use of semaphore operations:
 - signal (mutex) wait (mutex)
 - wait (mutex) ... wait (mutex)
 - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation

Module 3 – Part 2: Scheduling

- Batch systems,
- interactive systems,
- real time systems,
- threads
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Operating Systems Examples
- Algorithm Evaluation

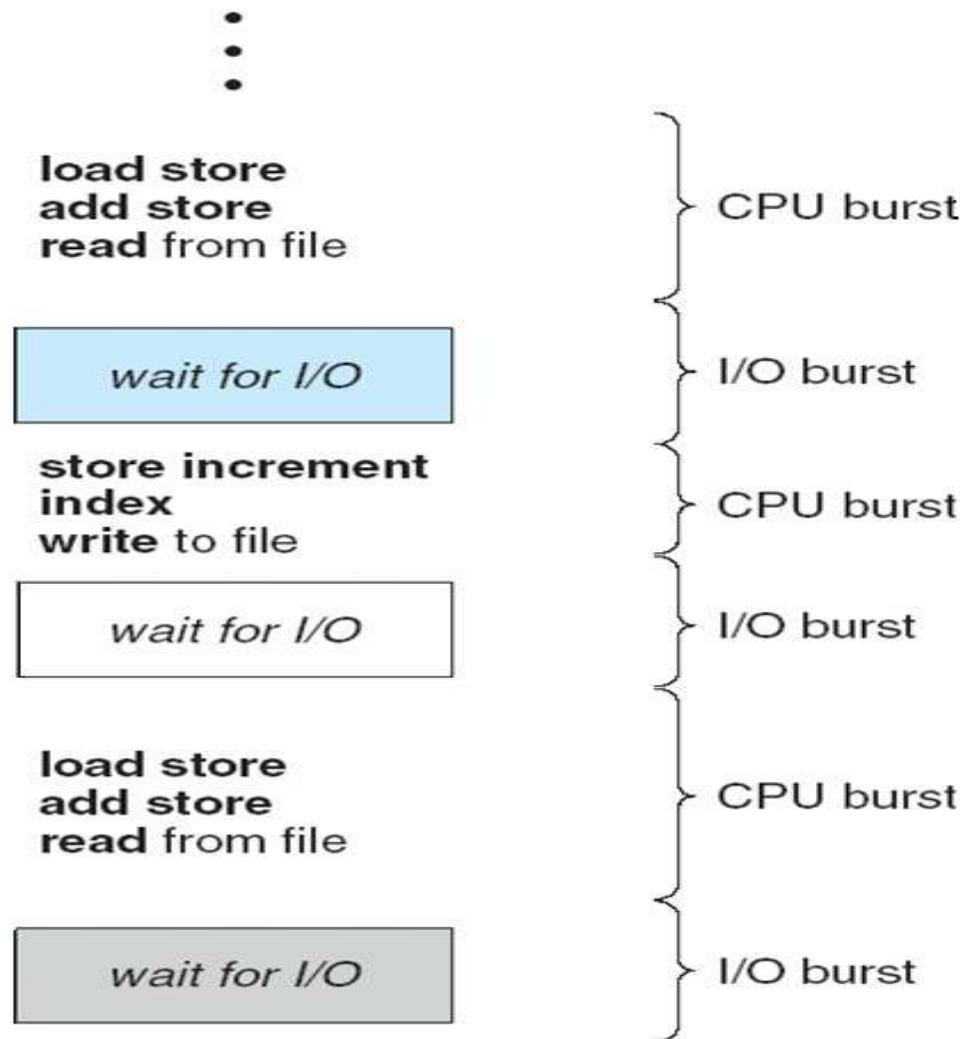
Scheduling

- When a computer is multiprogrammed, it frequently has multiple processes competing for the CPU at the same time.
- When more than one process is in the ready state and there is only one CPU available, **the operating system must decide which process to run first.**
- The part of the operating system that makes the choice is called the **scheduler or short term scheduler or CPU scheduler**
- The algorithm it uses is called the **scheduling algorithm or CPU scheduling.**

Scheduling

- Maximum CPU utilization obtained with multiprogramming
- **CPU-I/O Burst Cycle –**
- Process execution consists of a *cycle* of CPU execution and I/O wait

Alternating Sequence of CPU and I/O Bursts



When to Schedule ?

- There are a variety of situations in which scheduling may occur.
- First, scheduling is absolutely required on two occasions
 - When a process exits.
 - When a process blocks on I/O
- In each of these cases the process that had most recently been running becomes unready,
- **so another must be chosen to run next.**

When to Schedule ?

- There are three other occasions when scheduling is usually done, although logically it is not absolutely necessary at these times:
 - **When a new process is created.**
 - **When an I/O interrupt occurs.**
 - **When a clock interrupt occurs.**
- In the case of a new process, it makes sense to reevaluate priorities at this time.
- In some cases the parent may be able to request a different priority for its child.

When to Schedule ?

- In the case of an I/O interrupt, this usually means that an I/O device has now completed its work.
- So some process that was blocked waiting for I/O may now be ready to run.
- In the case of a clock interrupt, this is an opportunity to decide whether the currently running process has run too long.

Non-preemptive scheduling

- Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts.
- A **non-preemptive** scheduling algorithm picks a process to run and then just lets it run until it blocks
 - (either on I/O or waiting for another process)
 - or until it voluntarily releases the CPU.

Preemptive scheduling

- A **preemptive** scheduling algorithm picks a process and lets it run for a maximum of some fixed time.
- If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available).
- Doing preemptive scheduling requires having a clock interrupt occur at the end of the time interval to give control of the CPU back to the scheduler.
- If no clock is available, nonpreemptive scheduling is the only option.

Categories of Scheduling Algorithms

1. Batch
2. Interactive
3. Real time
4. Thread

1. Batch Scheduling

- In batch systems, there are no users impatiently waiting at their terminals for a quick response.
- Consequently, nonpreemptive algorithms, or preemptive algorithms with long time periods for each process are often acceptable.
- This approach reduces process switches and thus improves performance.

2. Interactive Scheduling

- In an environment with interactive users, preemption is essential to keep one process from hogging the CPU and denying service to the others.
- Even if a process intentionally ran forever, due to a program bug, then process might make all the others wait indefinitely.
- Preemption is needed to prevent this behavior.

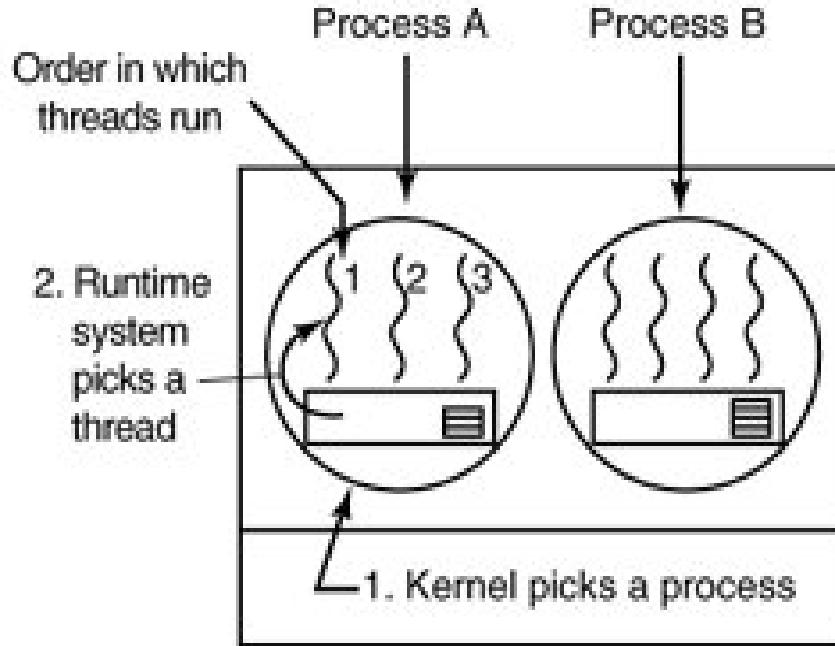
3. Real Time Scheduling

- In systems with real-time constraints, preemption is, enough
- sometimes not needed because the processes know that they may not run for long periods of time and usually do their work and block quickly.
- The difference with interactive systems is that real-time systems run only with cooperating programs
- Interactive systems are general purpose and may run arbitrary programs that are not cooperative or even malicious.

4. Thread Scheduling

- When several processes each have multiple threads, we have two levels of parallelism present: processes and threads.
- Scheduling has two types : **user-level threads or kernel-level threads (or both)**.
- In user level threads, the kernel is not aware of the existence of threads, it picks a process, say, *A*, and giving *A* control for its quantum.
- The thread scheduler inside *A* decides which thread to run, say *A1*.
- Since there are no clock interrupts to multiprogram threads, this thread may continue running as long as it wants to.
- If it uses up the process' entire quantum, the kernel will select another process to run.

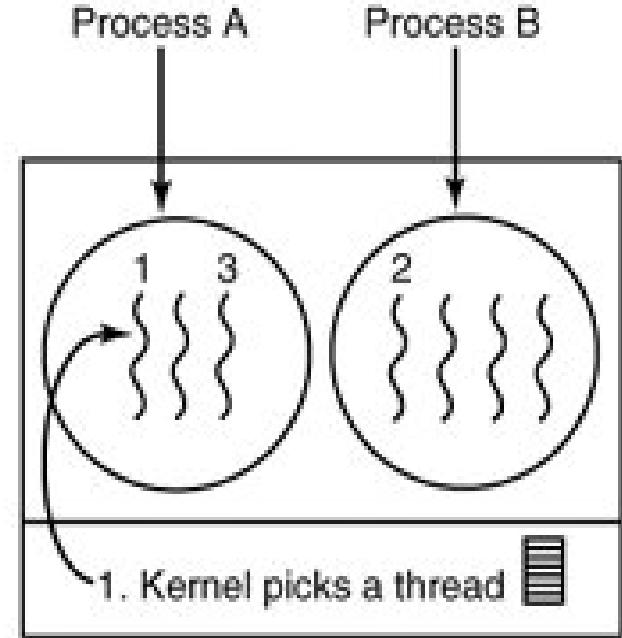
Thread Scheduling



Possible: A1, A2, A3, A1, A2, A3

Not possible: A1, B1, A2, B2, A3, B3

(a)



Possible: A1, A2, A3, A1, A2, A3

Also possible: A1, B1, A2, B2, A3, B3

(b)

Figure :(a) Possible scheduling of user-level threads with a 50- msec process quantum and threads that run 5 msec per CPU burst. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

Thread Scheduling

- In kernel-level threads, the kernel picks a particular thread to run.
- The thread is given a quantum and is forcibly suspended if it exceeds the quantum.
- With a 50-msec quantum but threads that block after 5 msec,
- the thread order for some period of 30 msec might be *A1, B1, A2, B2, A3, B3*,

- **Scheduling in Batch Systems**
 1. **First-Come First-Served**
 2. **Shortest Job First**
 3. **Shortest Remaining Time Next**
- **Scheduling in Interactive Systems**
 1. **Round-Robin Scheduling**
 2. **Priority Scheduling**

- **Scheduling in Real-Time Systems**
- Real-time systems are generally categorized as **hard real time**, meaning there are absolute deadlines that must be met
- **soft real time**, meaning that missing an occasional deadline is undesirable, but nevertheless tolerable
- **Thread scheduling**
- Any one of scheduling type supported for batch systems and interactive systems
- But, **Round-robin scheduling and priority scheduling** are most common

Scheduling Criteria

- Different CPU-scheduling algorithms have different properties
- The choice of a particular algorithm may favor one class of processes over another.
- In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.
- Many criteria have been suggested for comparing CPU-scheduling algorithms

Scheduling Criteria

- **CPU utilization – keep the CPU as busy as possible.**
- Conceptually, CPU utilization can range from 0 to 100 percent.
- In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- **Throughput – number of processes that complete their execution per time unit.**
- For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

Scheduling Criteria

- **Turnaround time** – amount of time to execute a particular process
- The **interval from the time of submission of a process to the time of completion** is the turnaround time.
- Turnaround time is the **sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O**

Scheduling Criteria

- **Waiting time** – Waiting time is the amount of time that a process spends waiting in the ready queue.
-
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output
- The turnaround time is generally limited by the speed of the output device.
- Useful for interactive systems in time-sharing environment

Scheduling Algorithm Optimization Criteria

- **Max CPU utilization**
- **Max throughput**
- **Min turnaround time**
- **Min waiting time**
- **Min response time**

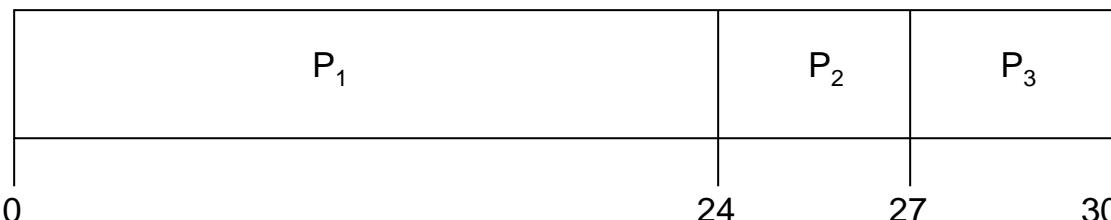
First-Come, First-Served Scheduling

- **Simplest CPU-scheduling** algorithm is the first-come, first-served (FCFS) scheduling algorithm.
- With this scheme, **the process that requests the CPU first is allocated the CPU first.**
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- When the CPU is free, it is allocated to the process at the head of the queue.
- The running process is then removed from the queue.
- The code for FCFS scheduling is **simple to write and understand.**
- On the negative side, the **average waiting time under the FCFS policy is often quite long.**

First-Come, First-Served (FCFS) Scheduling

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1, P_2, P_3
The Gantt Chart for the schedule is:



- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Turn around time $P_1 = 24$; $P_2 = 27$; $P_3 = 30$
- Average waiting time: $(0 + 24 + 27)/3 = 17\text{msec}$
- Average Turnaround Time is $(24+27+30)/3=27\text{msec}$

FCFS Scheduling

Suppose that the processes arrive in the order:

$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Turn around time for $P_1 = 30$; $P_2 = 3$; $P_3 = 6$
- Average waiting time: $(6 + 0 + 3)/3 = 3$ msec
- Average turn around time : $(30+ 3+6)/3 = 13$ msec
- Much better than previous case**
- Convoy effect - short process waiting behind long process for CPU**

Dynamic Situation

- Assume we have one CPU-bound process and many I/O-bound processes.
- As the processes flow around the system, the following scenario may result.
 - The CPU-bound process will get and hold the CPU.
 - During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU.
 - While the processes wait in the ready queue, the I/O devices are idle.
- Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device.

Dynamic Situation

- All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues.
- At this point, the CPU sits idle.
- The CPU-bound process will then move back to the ready queue and be allocated the CPU.
- Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done.
- There is a convoy effect as all the other processes wait for the one big process to get off the CPU.
- This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

Drawbacks of FCFS

- **The FCFS scheduling algorithm is nonpreemptive.**
- Once the CPU has been allocated to a process, that **process keeps the CPU until it releases the CPU,**
- **either by terminating or by requesting I/O.**
- The FCFS algorithm is thus particularly troublesome for time-sharing systems,
- where it is important that each user get a share of the CPU at regular intervals.
- It would be disastrous to allow one process to keep the CPU for an extended period.

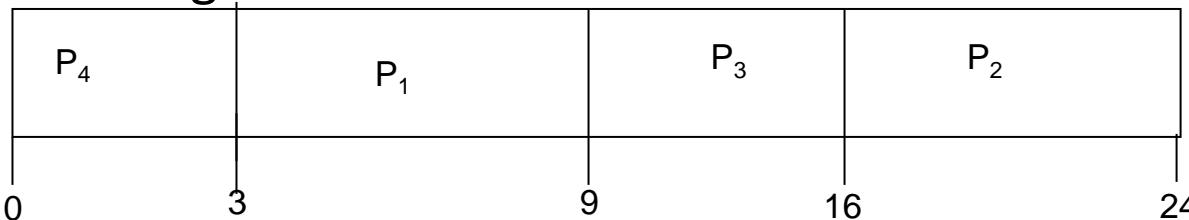
Shortest-Job-First (SJF) Scheduling

- This algorithm associates with each process the length of the process's next CPU burst(execution Time).
- When the CPU is available, **it is assigned to the process that has the smallest next CPU burst.**
- If the next CPU bursts of **two processes are the same, FCFS scheduling is used to break the tie.**

Example of SJF

<u>Process</u>	<u>Burst Time</u>
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Waiting Time for $P_1 = 3$ msec, $P_2 = 16$ msec, $P_3 = 9$ msec, $P_4 = 0$ msec
- Turn Around Time for $P_1 = 9$ msec, $P_2 = 24$ msec, $P_3 = 16$ msec, $P_4 = 3$ msec
- Average waiting time = $(9 + 24 + 16 + 3) / 4 = 13$ msec
- Average turn around time = $(3 + 16 + 9 + 0) / 4 = 7$ msec

Advantages

- The SJF scheduling algorithm is provably optimal,
- It gives the **minimum average waiting time for a given set of processes.**
- Moving a short process before a long one decreases the waiting time of the short process
- It increases the waiting time of the long process.
- Consequently, the average waiting time decreases.

SJF

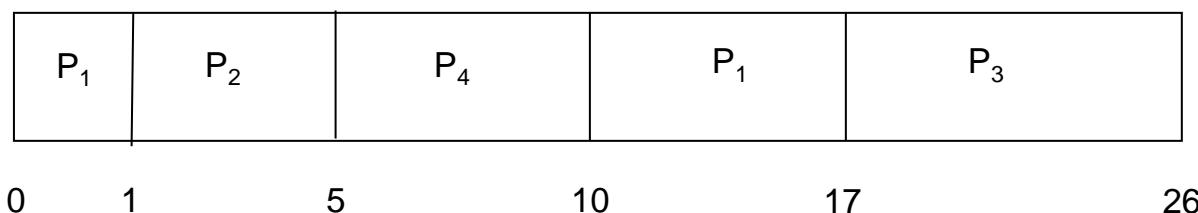
- The SJF algorithm can be either preemptive or non preemptive
- The choice arises when a new process arrives at the ready queue while a previous process is still executing
- The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process
- A preemptive SJF algorithm will preempt the currently executing process
- Whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst.
- Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Preemptive SJF Gantt Chart*

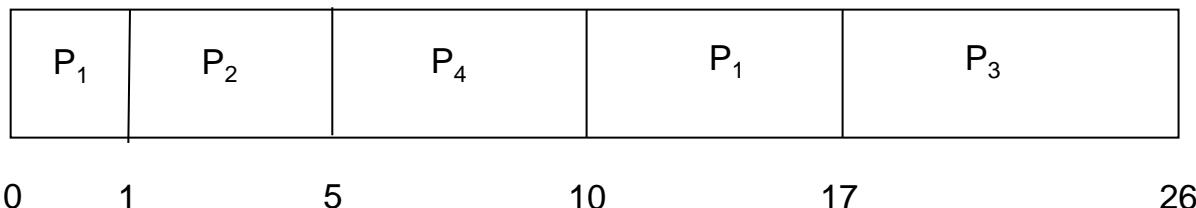


Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Preemptive SJF Gantt Chart*



- Waiting Time for $P_1 = 9m\ sec$, $P_2 = 0m\ sec$, $P_3 = 15m\ sec$, $P_4 = 2m\ sec$
- Turn Around Time for $P_1 = 17m\ sec$, $P_2 = 5\ msec$, $P_3 = 26m\ sec$, $P_4 = 10m\ sec$
- Average waiting time = $[(9)+(0)+(15)+(2)]/4 = 26/4 = 6.5\ msec$
- Average turn around time = $(17+5+26+10) / 4 = 14.5msec$
- Compare this with SJF algorithm**

EXERCISE of SRTF

- Now we add the concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0	5
P_2	1	3
P_3	2	2
P_4	3	4

- Draw Preemptive SJF Gantt Chart and Calculate Waiting Time , Turnaround Time, Average Waiting Time and Average Turnaround Time*

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority
- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095.
- Here , we assume that low numbers represent high priority.

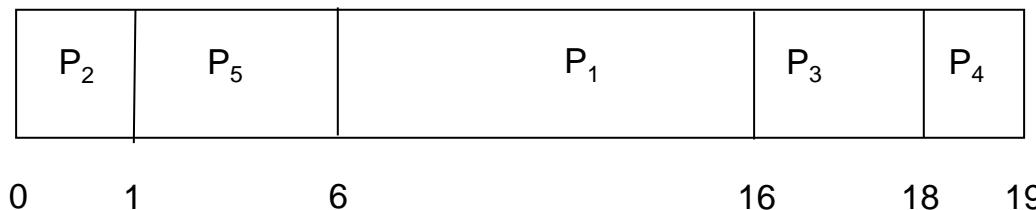
Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

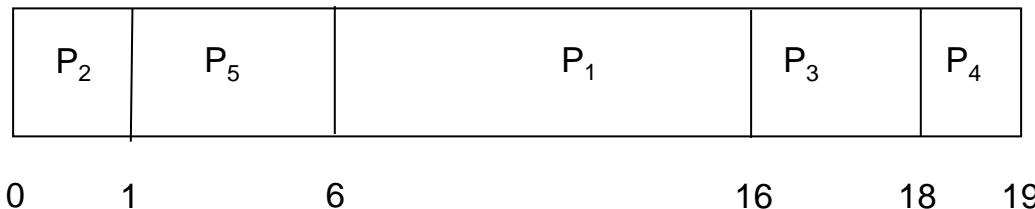
- Priority scheduling Gantt Chart



Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Waiting Time for $P_1 = 6 \text{ msec}$, $P_2 = 0 \text{ msec}$, $P_3 = 16 \text{ msec}$, $P_4 = 18 \text{ msec}$, $P_5 = 1 \text{ msec}$
- Turn Around Time for $P_1 = 16 \text{ msec}$, $P_2 = 1 \text{ msec}$, $P_3 = 18 \text{ msec}$, $P_4 = 19 \text{ msec}$, $P_5 = 6 \text{ msec}$
- Average waiting time = $(6+0+16+18+1)/5 = 8.2 \text{ msec}$
- Average turn around time = $(16+1+18+19+6) / 5 = 12 \text{ msec}$

Priority Scheduling

- Priorities can be defined either internally or externally.
- Internally defined priorities use some measurable quantity or quantities to compute the priority of a process.
- For example, **time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities.**
- External priorities are set by criteria outside the operating system, such as the **importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, etc**

Priority Scheduling

- Priority scheduling can be either preemptive or nonpreemptive.
- When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
- A preemptive priority scheduling algorithm **will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.**
- A nonpreemptive priority scheduling algorithm **will simply put the new process at the head of the ready queue.**

Draw Back of Priority Scheduling

- A major problem with priority scheduling algorithms is indefinite blocking, or starvation
- This can leave some low priority processes waiting indefinitely.
- In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.
- Generally, one of two things will happen.
- Either the process will eventually be run
- or the computer system will eventually crash and lose all unfinished low-priority processes

Solution

- A solution to the problem of indefinite blockage of low-priority processes is aging.
- Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.
- For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.
- Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed.
- In fact, it would take no more than 32 hours for a priority-127 process to age to a priority-0 process.

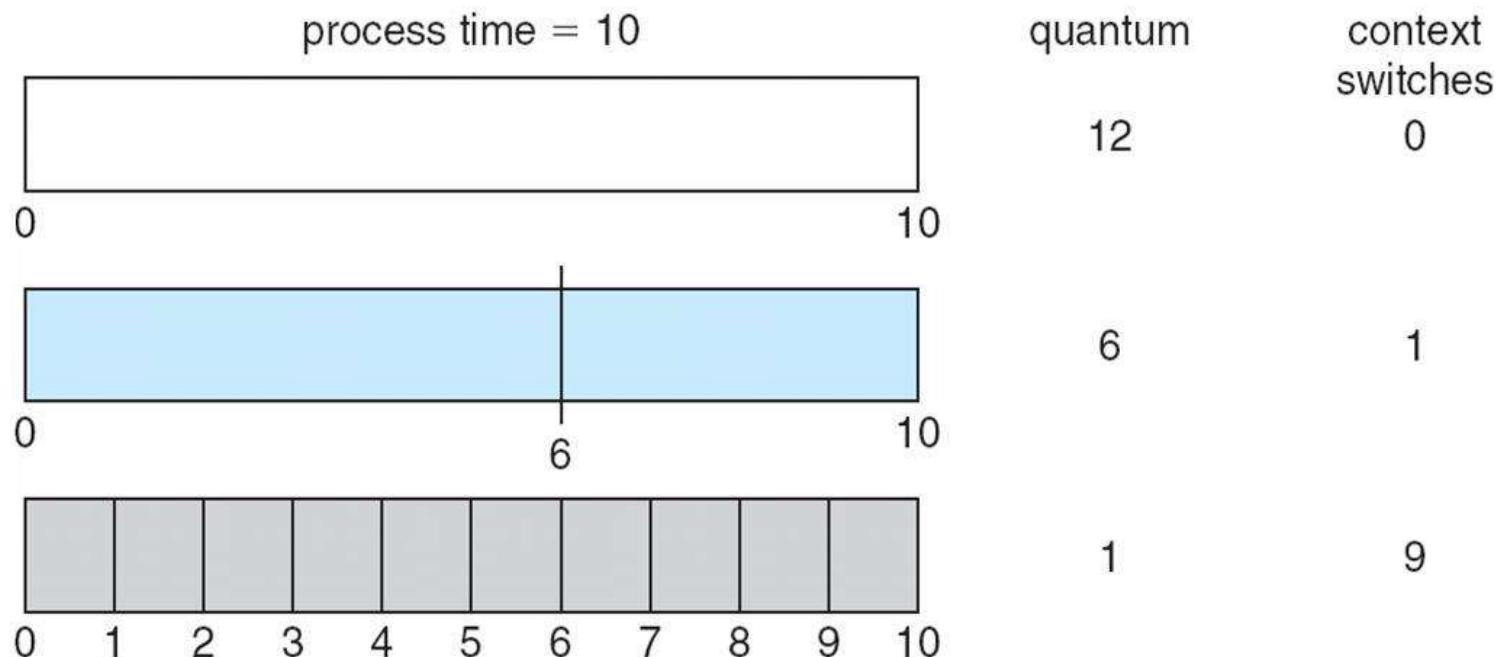
Round Robin (RR) Scheduling

- The round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
- It is similar to FCFS scheduling, but **with preemption**
- A small unit of time, called a time quantum or time slice, is defined which is generally ranging from 10 to 100 milliseconds in length.
- The ready queue is treated as a circular queue.
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

Round Robin (RR) Scheduling

- After this time quantum has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the CPU time in chunks of at most q time units at once.
- No process waits more than $(n-1)q$ time units.
- Timer interrupts every quantum to schedule next process
- Performance
 - q large \Rightarrow FIFO
 - q small \Rightarrow q must be large with respect to context switch, otherwise overhead is too high

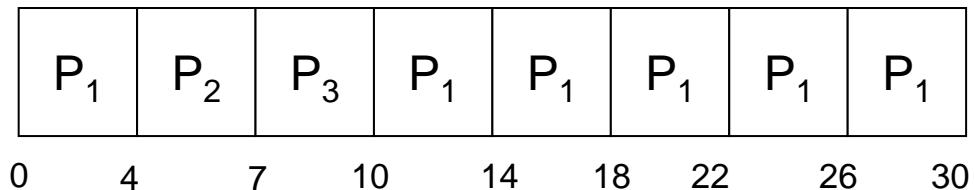
Time Quantum and Context Switch Time



Example of RR with Time Quantum = 4

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

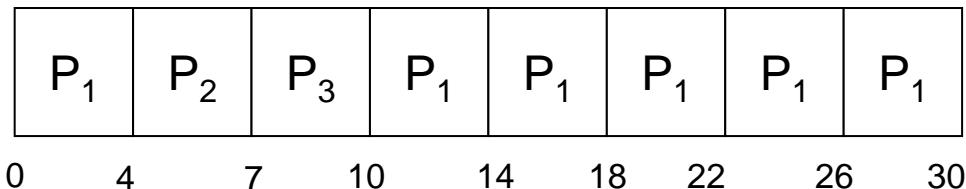


Typically, higher average turnaround than SJF, but better *response*
q should be large compared to context switch time
q usually 10ms to 100ms, context switch < 10 usec

Example of RR with Time Quantum = 4

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



P_1 waits for 6 ms (10- 4), P_2 waits for 4 ms, and P_3 waits for 7 ms.

Thus, the average waiting time is $17/3 = 5.66$ milliseconds.

Turn Around Time for $P_1 = 30$ msec , $P_2 = 7$ msec, $P_3 = 10$ msec

Average turn around time = $(30+7+10) / 3 = 15.67$ msec

Exercises

1. Consider the following set of processes, with the length of the CPU burst given in milliseconds

Process	Burst Time	Priority

Pt	10	3
p2	1	1
p3	2	3
p4	1	4
Ps	5	2

The processes are assumed to have arrived in the order P1, P2 , P3, P4, Ps, all at time 0.

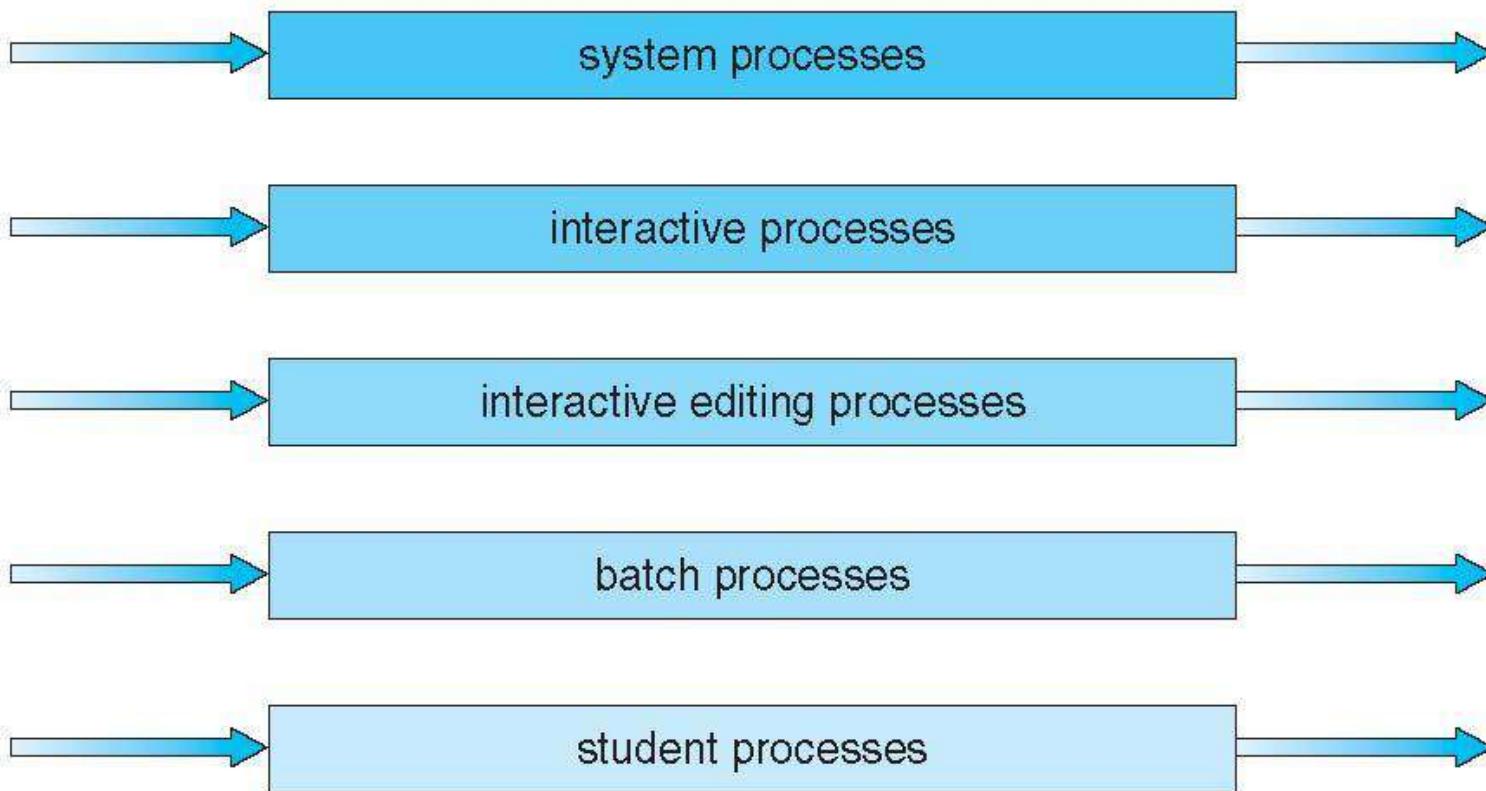
- a) Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, non preemptive priority (a smaller priority number implies a higher priority), and RR (quantum= 1).
- b) What is the turnaround time of each process for each of the scheduling algorithms in part a?
- c) What is the waiting time of each process for each of these scheduling algorithms?
- d) Which of the algorithms results in the minimum average waiting time (over all processes)?

Multilevel Queue Scheduling

- The class of scheduling algorithms for situations in which processes are classified into different groups.
- For example,
- **foreground (interactive) processes**
- **background (batch) processes.**
-
- These two types of processes have **different response-time requirements** and so may have **different scheduling needs**.
- In addition, foreground processes may have priority over background processes.

Multilevel Queue Scheduling

highest priority



lowest priority

Multilevel Queue Scheduling

- **First Possibility - Each queue has absolute priority over lower-priority queues.**
- The process in the batch queue, could run only if the queues for system processes, interactive processes, and interactive editing processes were all empty.
- If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted
- **This may lead to starvation – waiting for ever for CPU time**

Multilevel Queue Scheduling

- **Another possibility is to time-slice among the queues.**
- Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes.
- For instance, in the foreground-background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes,
- whereas the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

Multilevel Queue Scheduling - Drawback

- Normally, when the multilevel queue scheduling algorithm is used, **processes are permanently assigned to a queue when they enter the system.**
- If there are separate queues for foreground and background processes,
- **processes do not move from one queue to the other**, since processes do not change their foreground or background nature.
- This setup has the advantage of low scheduling overhead, **but it is inflexible.**

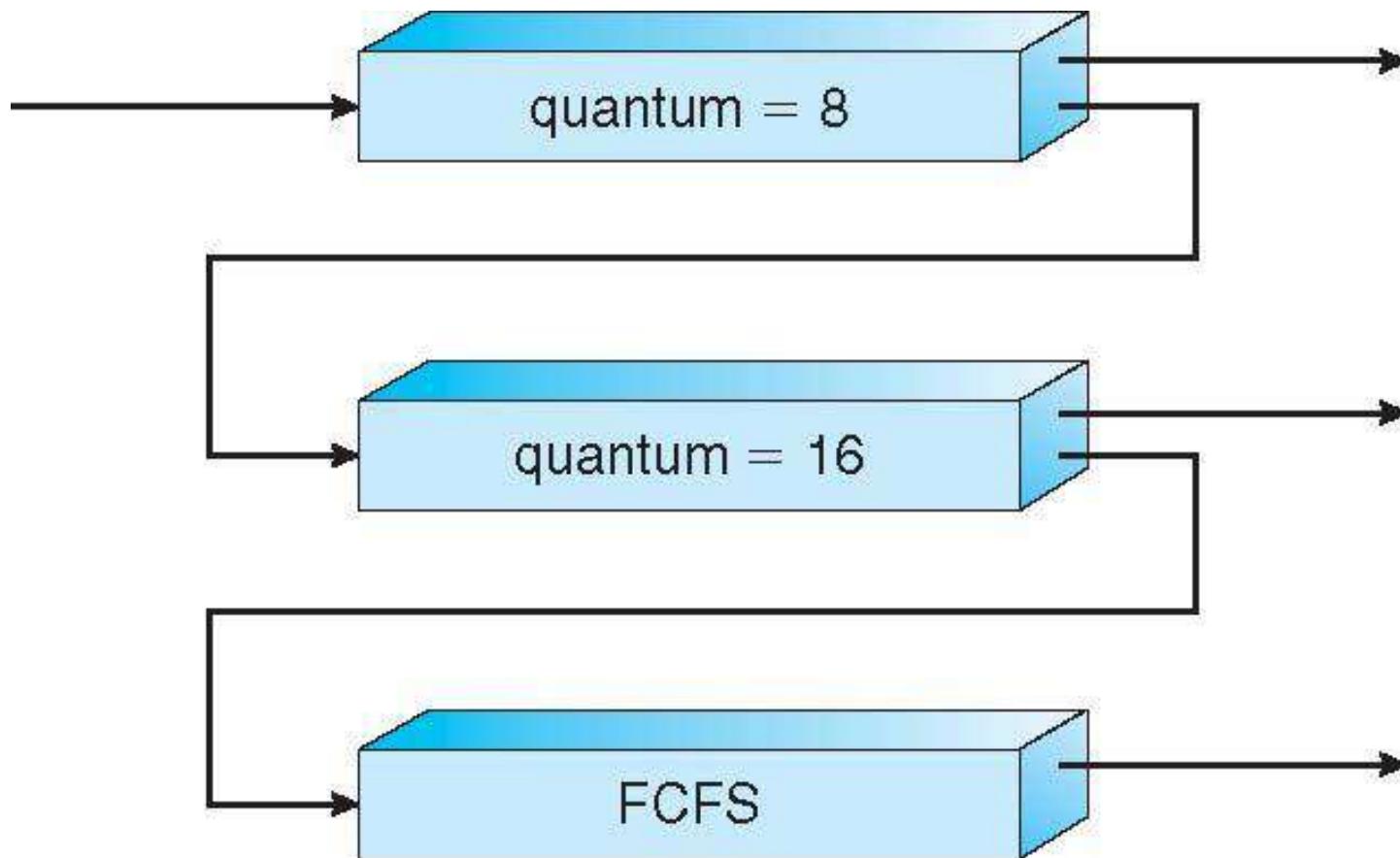
Multilevel feedback queue scheduling

- This allows a process to move between queues.
- The idea is to separate **processes according to the characteristics of their CPU bursts**.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- This scheme leaves **I/O-bound and interactive processes in the higher-priority queues**.
- In addition, a **process that waits too long in a lower-priority queue may be moved to a higher-priority queue**.
- This form of aging prevents starvation.

Multilevel feedback queue scheduling

- For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2
- The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1.
- Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2.
- A process in queue 1 will in turn be preempted by a process arriving for queue 0.

Multilevel Feedback Queues



Multilevel Feedback Queues

- A process entering the ready queue is put in queue 0.
- A process in queue 0 is given a time quantum of 8 milliseconds.
- If it does not finish within this time, it is moved to the tail of queue 1.
- If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds.
- If it does not complete, it is preempted and is put into queue 2.
- **Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.**

Multilevel Feedback Queues

- This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less.
- Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst.
- Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes.
- Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

Multilevel Feedback Queue

- Multilevel-feedback-queue scheduler defined by the following parameters:
 - number of queues
 - scheduling algorithms for each queue
 - method used to determine when to upgrade a process
 - method used to determine when to demote a process
 - method used to determine which queue a process will enter when that process needs service

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- **Contention Scope**
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on Light Weight Process
 - Known as **process-contention scope (PCS)** since scheduling competition is within the process
 - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Pthread Scheduling

- POSIX API allows specifying either PCS or SCS during thread creation
 - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
 - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Pthread Scheduling

- The Pthread IPC provides two functions for getting-and setting-the contention scope policy:
- **`pthread_attr_setscope(pthread_attr_t *attr, int scope)`**
- **`pthread_attr_getscope(pthread_attr_t *attr, int *scope)`**
- The first parameter for both functions contains a pointer to the attribute set for the thread.
- The second parameter for the `pthread_attr_setscope ()` function is passed either the **PTHREAD_SCOPE_SYSTEM** or the **PTHREAD_SCOPE_PROCESS** value, indicating how the contention scope is to be set.
- In the case of `pthread_attr_getscope ()`, this second parameter contains a pointer to an **int value that is set to the current value of the contention scope**.

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
    int i;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread_attr_setschedpolicy(&attr, SCHED_OTHER);
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);
    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    printf("I am a thread\n");
    pthread_exit(0);
}
```

we illustrate a Pthread scheduling API.

The program first determines the existing contention scope and sets it to PTHREAD_SCOPE_SYSTEM.

It then creates five separate threads that will run using the SCS scheduling policy.

Note that on some systems, only certain contention scope values are allowed.

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **If multiple CPUs are available, load sharing becomes possible;**
- **Homogeneous processors (Identical)** within a multiprocessor

Multiple-Processor Scheduling

- **Approaches to Multiple-Processor Scheduling**
- **One approach to CPU scheduling in a multiprocessor system** has all scheduling decisions, I/O processing, and other system activities handled **by a single processor-the master server.**
- The other processors ie slave processors execute only user code.
- **This asymmetric multiprocessing is simple because only one processor accesses the system data structures, reducing the need for data sharing.**

Multiple-Processor Scheduling

- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
 - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running
 - **soft affinity**
 - Processor affinity takes several forms.
 - When an operating system has a policy of attempting to keep a process running on the same processor-but not guaranteeing that it will do so-we have a situation known as soft affinity.
 - Here, it is possible for a process to migrate between processors

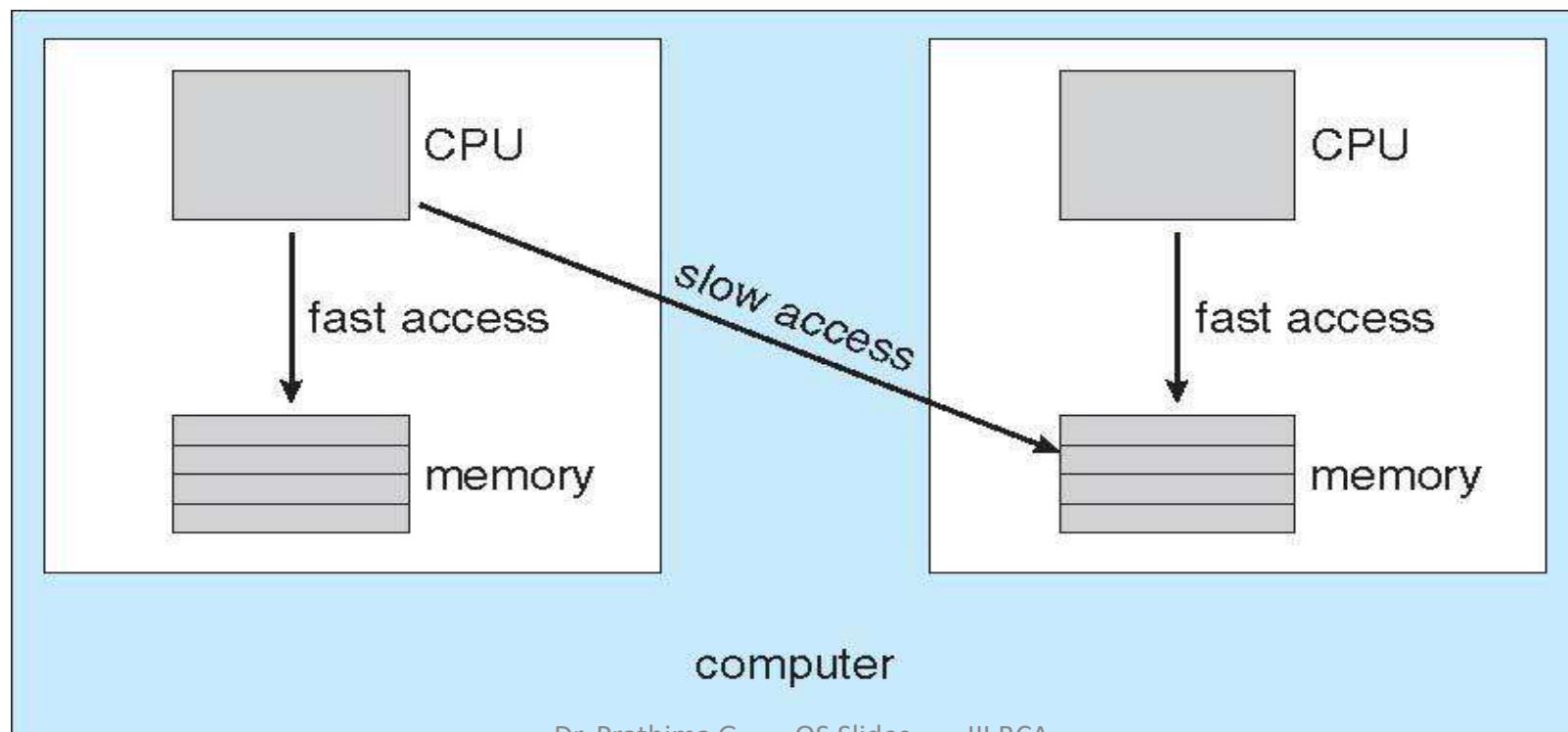
Multiple-Processor Scheduling

- **hard affinity**
- Some systems -such as **Linux** -also provide system calls that support hard affinity, thereby allowing a process to specify that it is not to migrate to other processors.
- **Solaris** allows processes to be assigned to limiting which processes can run on which CPUs.
- It also implements soft affinity.

NUMA and CPU Scheduling

The main-memory architecture of a system can affect processor affinity issues. This illustrates an architecture featuring **non-uniform memory access (NUMA)**, in which a CPU has faster access to some parts of main memory than to other parts.

Typically, this occurs in systems containing combined CPU and memory boards. The CPUs on a board can access the memory on that board with less delay than they can access memory on other boards in the system



Load Balancing

- On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor.
- Otherwise, one or more processors may sit idle while other processors have high workloads, along with lists of processes awaiting the CPU.
- Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system

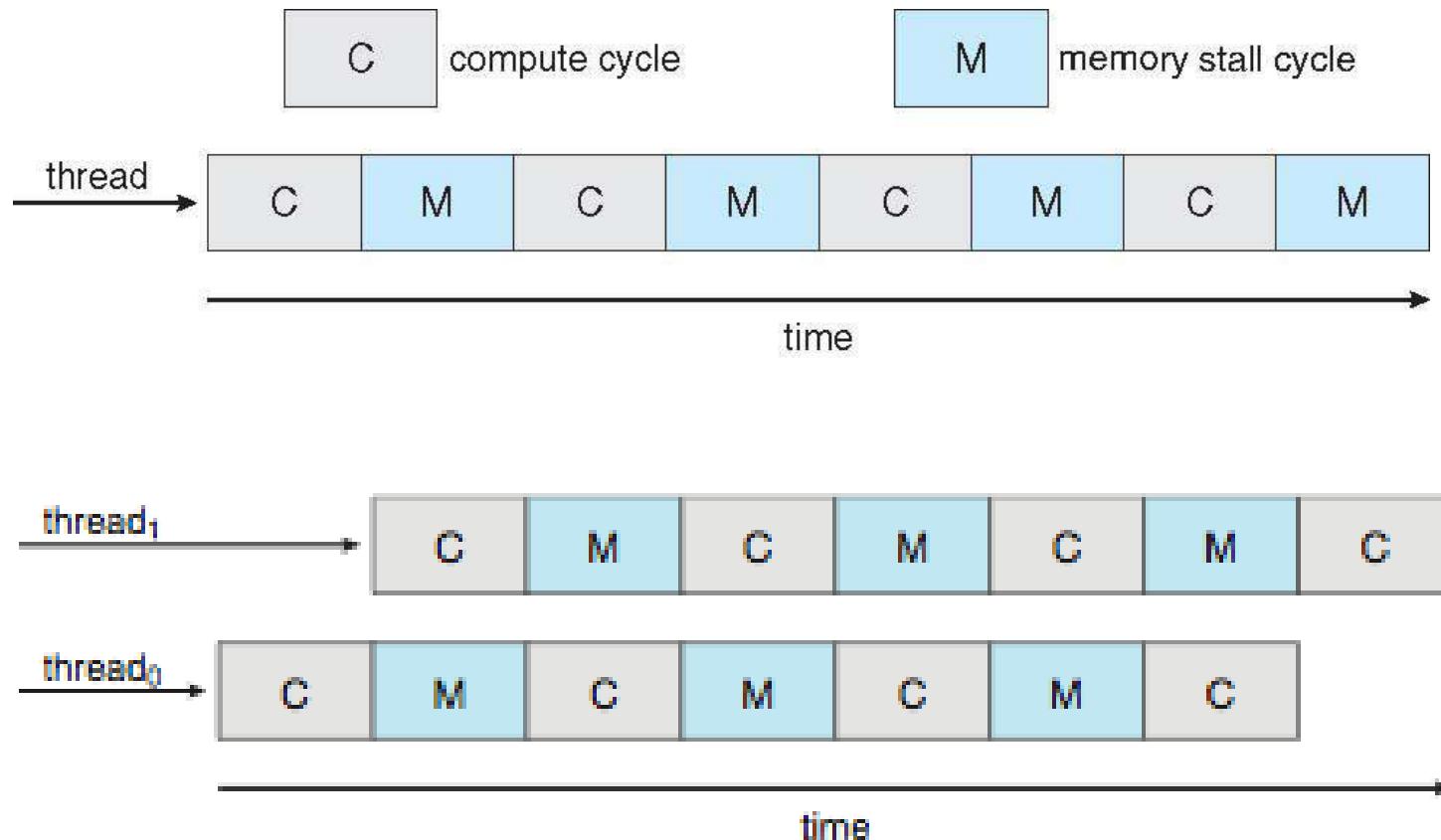
Load Balancing

- There are two general approaches to load balancing: **push migration and pull migration.**
- With **push migration**, a specific task periodically checks the load on each processor and
- -if it finds an imbalance-evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors.
- **Pull migration occurs when an idle processor pulls a waiting task from a busy processor.**
- Push and pull migration need not be mutually exclusive and are in fact often implemented in parallel on load-balancing systems.

Multicore Processors

- **Recent trend to place multiple processor cores on same physical chip**
- **Faster and consumes less power**
- Multiple threads per core also growing
 - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

Multithreaded Multicore System



Virtualization and Scheduling

- In general, though, most virtualized environments have **one host operating and many guest operating systems**.
- **The host operating system creates and manages the virtual machines**, and
- **Each virtual machine has a guest operating system installed and applications running within that guest**.

Virtualization and Scheduling

- Virtualization software schedules multiple guests onto CPU(s)
- Each guest doing its own scheduling
 - Not knowing it doesn't own the CPUs
 - Can result in poor response time
 - Can effect time-of-day clocks in guests

Virtualization and Scheduling

- **Poor Response Time** - The 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 that they are scheduling all of those cycles.
- **Time of Day clock** - 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 good scheduling-algorithm efforts of the operating systems within virtual machines.

Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

Example: Solaris Scheduling

- Solaris uses priority-based thread scheduling where each thread belongs to one of six classes:
- Six classes available
 - Time sharing (default)
 - Interactive
 - Real time
 - System
 - Fair Share
 - Fixed priority
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- For ex: Time sharing is multi-level feedback queue

Solaris Dispatch Table

The dispatch table contains the following fields:

Priority. The class-dependent priority for the time-sharing and interactive classes. A higher number indicates a higher priority.

Time quantum. The time quantum for the associated priority. The lowest priority (priority 0) has the highest time quantum (200 milliseconds), and the highest priority (priority 59) has the lowest time quantum (20 milliseconds).

Time quantum expired. The new priority of a thread that has used its entire time quantum without blocking. Such threads are considered CPU-intensive. As shown in the table, these threads have their priorities lowered.

Return from sleep. The priority of a thread that is returning from sleeping (such as waiting for I/O).

When I/O is available for a waiting thread, its priority is boosted to between 50 and 59, thus supporting the scheduling policy of providing good response time for interactive processes

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Solaris Scheduling

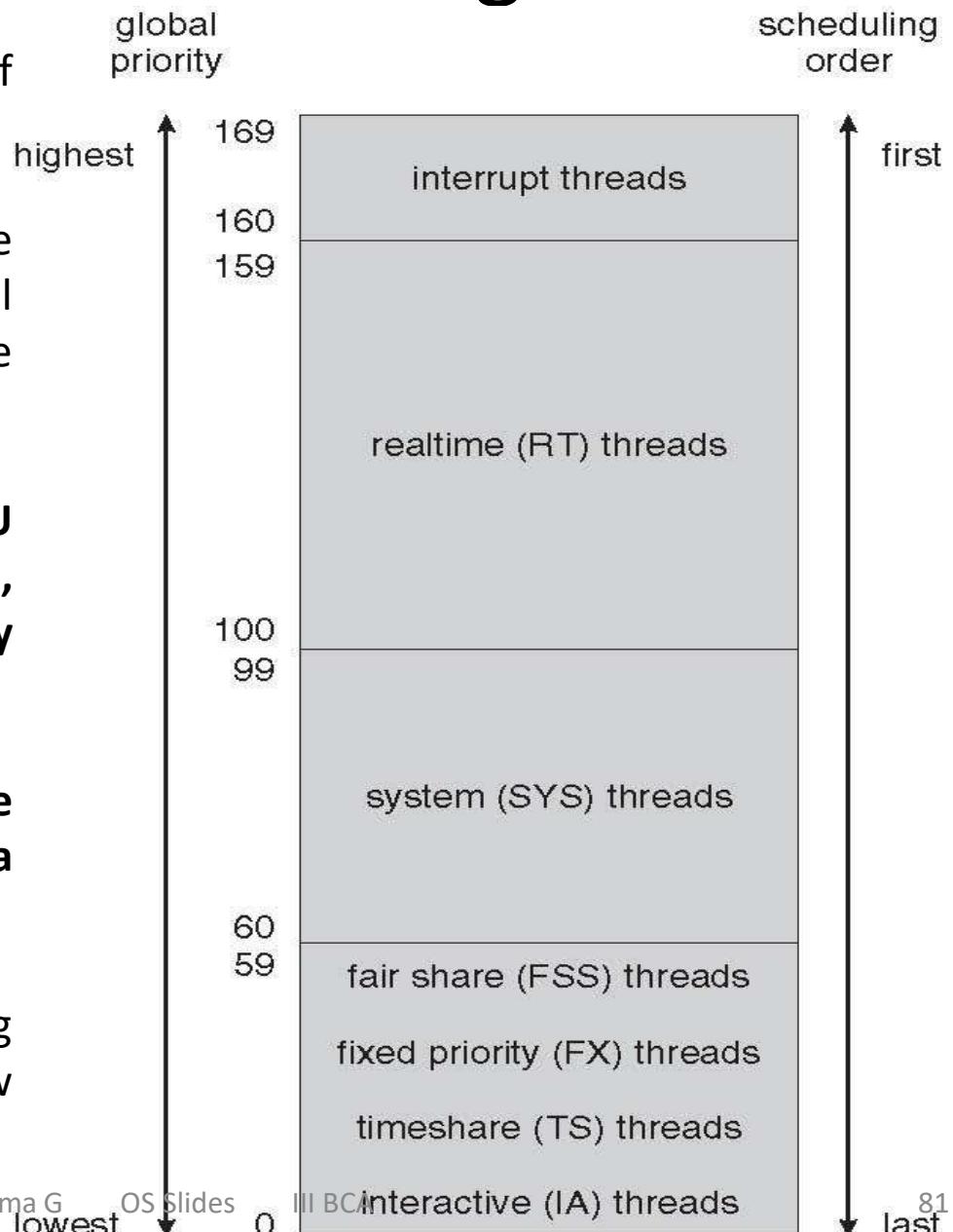
Each scheduling class includes a set of priorities.

However, the scheduler converts the class-specific priorities into global priorities and selects the thread with the highest global priority to run.

The selected thread runs on the CPU until it (1) blocks, (2) uses its time slice, or (3) is preempted by a higher-priority thread.

If there are multiple threads with the same priority, the scheduler uses a round-robin queue.

Figure illustrates how the six scheduling classes relate to one another and how they map to global priorities.



Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- *Dispatcher* is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time threads
- 32-level priority scheme
- Variable class is 1-15, real-time class is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs idle thread

Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong
 - **REALTIME_PRIORITY_CLASS**,
 - **HIGH_PRIORITY_CLASS**,
 - **ABOVE_NORMAL_PRIORITY_CLASS**,
 - **NORMAL_PRIORITY_CLASS**,
 - **BELOW_NORMAL_PRIORITY_CLASS**,
 - **IDLE_PRIORITY_CLASS**
 - All are variable except REALTIME
- A thread within a given priority class has a relative priority
 - **TIME_CRITICAL**,
 - **HIGHEST**,
 - **ABOVE_NORMAL**,
 - **NORMAL**,
 - **BELOW_NORMAL**,
 - **LOWEST**,
 - **IDLE**

Windows XP Priorities

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

For example, if the relative priority of a thread in the ABOVE_NORMAL_PRIORITY_CLASS is NORMAL, the numeric priority of that thread is 10.

By default, the base priority is the value of the NORMAL relative priority for that class. The base priorities for each priority class are:

REALTIME_PRIORITY_CLASS-24

HIGH_PRIORITY_CLASS-13

ABOVE_NORMAL_PRIORITY_CLASS-10

NORMAL_PRIORITY_CLASS-8

BELOW_NORMAL_PRIORITY_CLASS-6

IDLE_PRIORITY_CLASS-4

Windows Priority Classes

- When a user is running an interactive program, the system needs to provide especially good performance.
- For this reason, Windows XP has a special scheduling rule for processes in the NORMAL_PRIORITY_CLASS.
- **Windows XP distinguishes between the foreground process that is currently selected on the screen and the background processes that are not currently selected.**
- **When a process moves into the foreground, Windows XP increases the scheduling quantum by some factor-typically by 3.**

Linux Scheduling

- Constant order $O(1)$ scheduling time
- Preemptive, priority based
- **Two priority ranges: time-sharing and real-time**
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140
- Map into global priority with numerically lower values indicating higher priority
- Higher priority gets larger quantum

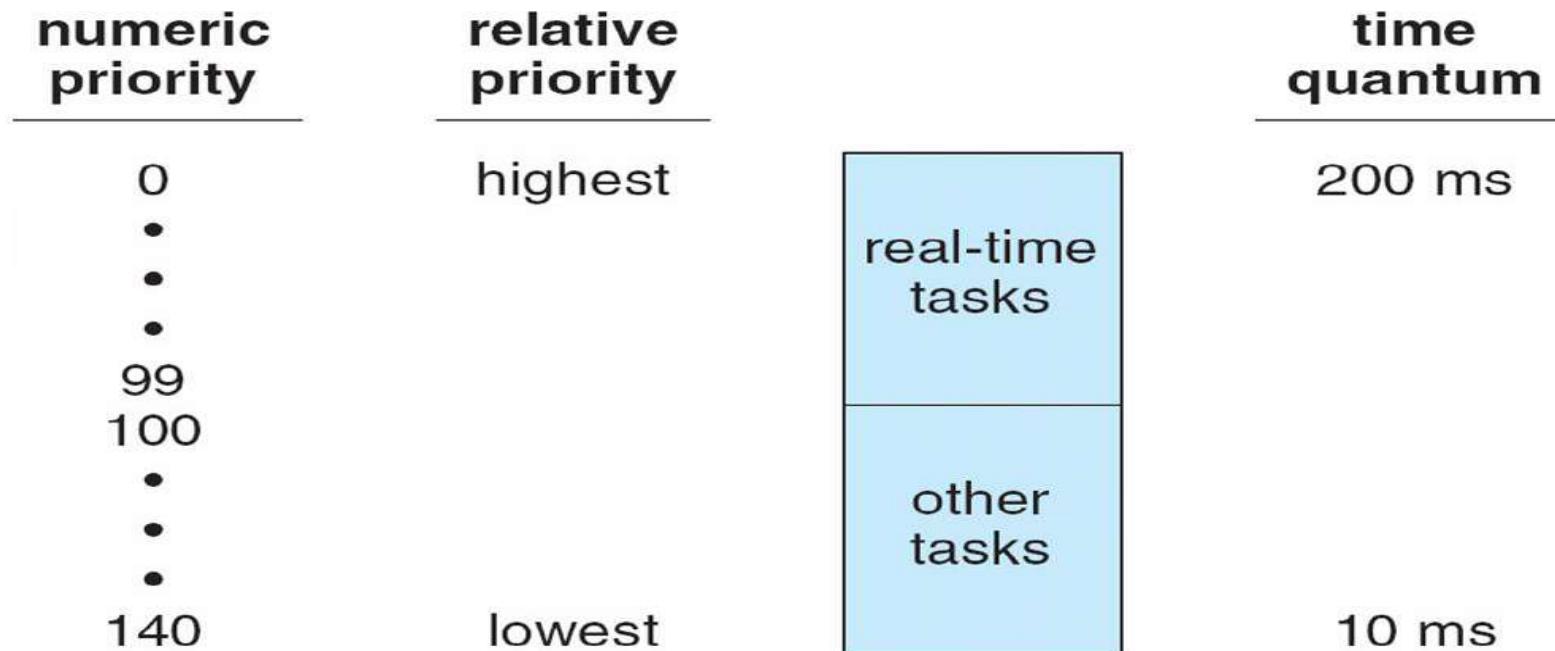


Figure : The relationship between priorities and time-slice length

Linux Scheduling

- Task run-able as long as time left in time slice (**active**)
- If no time left (**expired**), not run-able until all other tasks use their slices
- All run-able tasks tracked in per-CPU **runqueue** data structure
 - Two priority arrays (active, expired)
 - Tasks indexed by priority
 - When no more active, arrays are exchanged

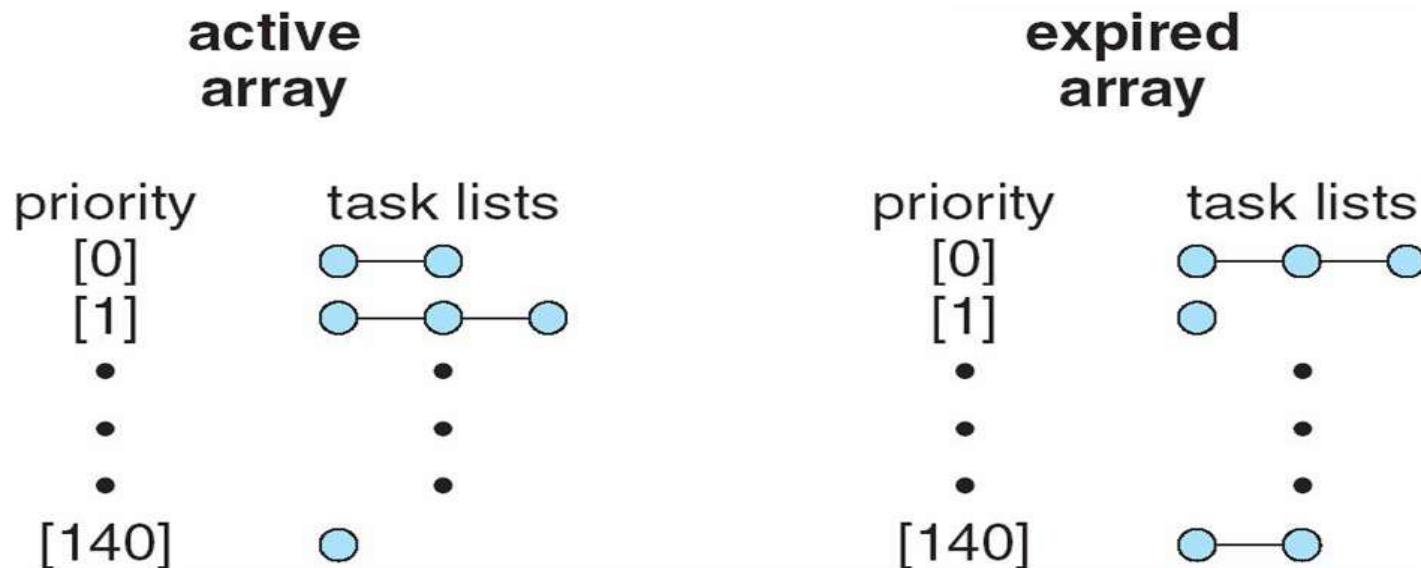


Figure : List of tasks indexed according to priority

Linux Scheduling

- Real-time scheduling according to POSIX.1b
 - Real-time tasks have static priorities
- All other tasks dynamic based on *nice* value plus or minus 5
 - Interactivity of task determines plus or minus
 - More interactive -> more minus
 - Priority recalculated when task expired
 - This exchanging arrays implements adjusted priorities

Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
 - Maximizing CPU utilization under the constraint that the maximum response time is 1 second
 - Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time

Deterministic Modeling

- One major class of evaluation methods is **analytic evaluation**.
- Analytic evaluation **uses the given algorithm and the system workload to produce a formula or**
- Number that evaluates the performance of the algorithm for that workload
- **Deterministic modeling** is one type of analytic evaluation
- This method **takes a particular predetermined workload** and
- Defines the **performance of each algorithm for that workload**.

Deterministic Modeling - Example

- For example, assume that we have the workload shown below. All five processes arrive at time 0, in the order given, with the length of the CPU burst given in milliseconds:
- Process Burst Time

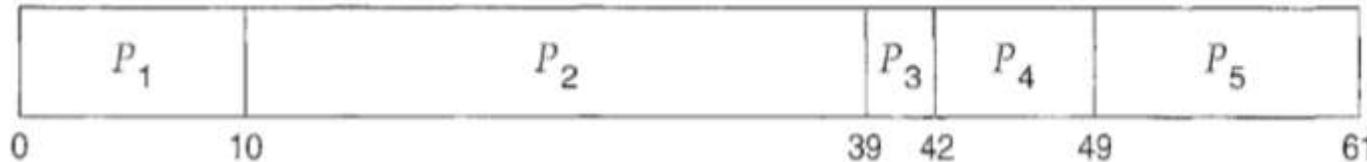
P1	10
P2	29
P3	3
p4	7
P5	12

Consider the FCFS, SJF, and RR (quantum = 10 milliseconds) scheduling algorithms for this set of processes.

Which algorithm would give the minimum average waiting time?

Deterministic Modeling

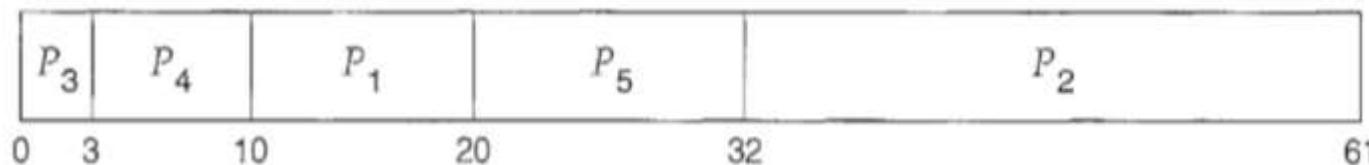
- For the FCFS algorithm, we would execute the processes as



- The waiting time is 0 milliseconds for process P1, 10 milliseconds for process P2, 39 milliseconds for process P3, 42 milliseconds for process P4, and 49 milliseconds for process P5.
- Thus, the average waiting time is $(0 + 10 + 39 + 42 + 49)/5 = 28$ milliseconds.

Deterministic Modeling

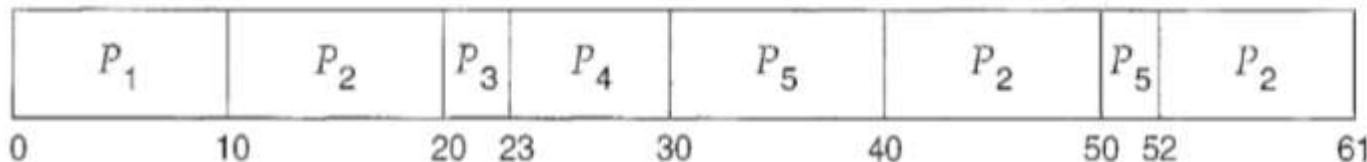
- With no preemptive SJF scheduling, we execute the processes as



- The waiting time is 10 milliseconds for process P1, 32 milliseconds for process P2, 0 milliseconds for process P3, 3 milliseconds for process P4, and 20 milliseconds for process P5.
- Thus, the average waiting time is $(10 + 32 + 0 + 3 + 20) / 5 = 13$ milliseconds.

Deterministic Modeling

- With the RR algorithm, we execute the processes as



- The waiting time is 0 milliseconds for process P1, 32 milliseconds for process P2, 20 milliseconds for process P3, 23 milliseconds for process P4, and 40 milliseconds for process P5.
- Thus, the average waiting time is $(0 + 32 + 20 + 23 + 40)/5 = 23$ milliseconds.

Deterministic Modeling - Analysis

- The average waiting time obtained with the SJF policy is less than half that obtained with FCFS scheduling
- The RR algorithm gives us an intermediate value.
- **Deterministic modeling is simple and fast. It gives us exact numbers, allowing us to compare the algorithms.**
- **However, it requires exact numbers for input, and its answers apply only to those cases.**
- The main uses of deterministic modeling are in describing scheduling algorithms and providing examples

Queueing Models

- On many systems, the processes run vary from day to day, so there is no static set of processes (or times) to use for deterministic modeling.
- **But we can determine the distribution of CPU and I/O bursts.**
- These distributions can be measured using a mathematical formula describing the probability of a particular CPU burst.
- **We can describe the distribution of times when processes arrive in the system (the arrival-time distribution).**
- From these distributions, **it is possible to compute the average throughput, utilization, waiting time, and so on for most algorithms.**

Queueing Models

- The computer system is described as a network of servers.
- Each server has a queue of waiting processes.
- The CPU is a server with its ready queue, as is the I/O system with its device queues.
- Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, and so on.
- This area of study is called queueing-network analysis.

Little's Formula

- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- **Little's law – in steady state, processes leaving queue must equal processes arriving, thus**
$$n = \lambda \times W$$
 - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds
- **Queueing models are limited bcos the mathematics of complicated algorithms and distributions can be difficult to work with.**

Simulations

- To get a more accurate evaluation of scheduling algorithms, we can use simulations.
- Running simulations involves programming a model of the computer system.
- The simulator has a variable representing a clock;
- As this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes, and the scheduler.
- As the simulation executes, statistics that indicate algorithm performance are gathered and printed.

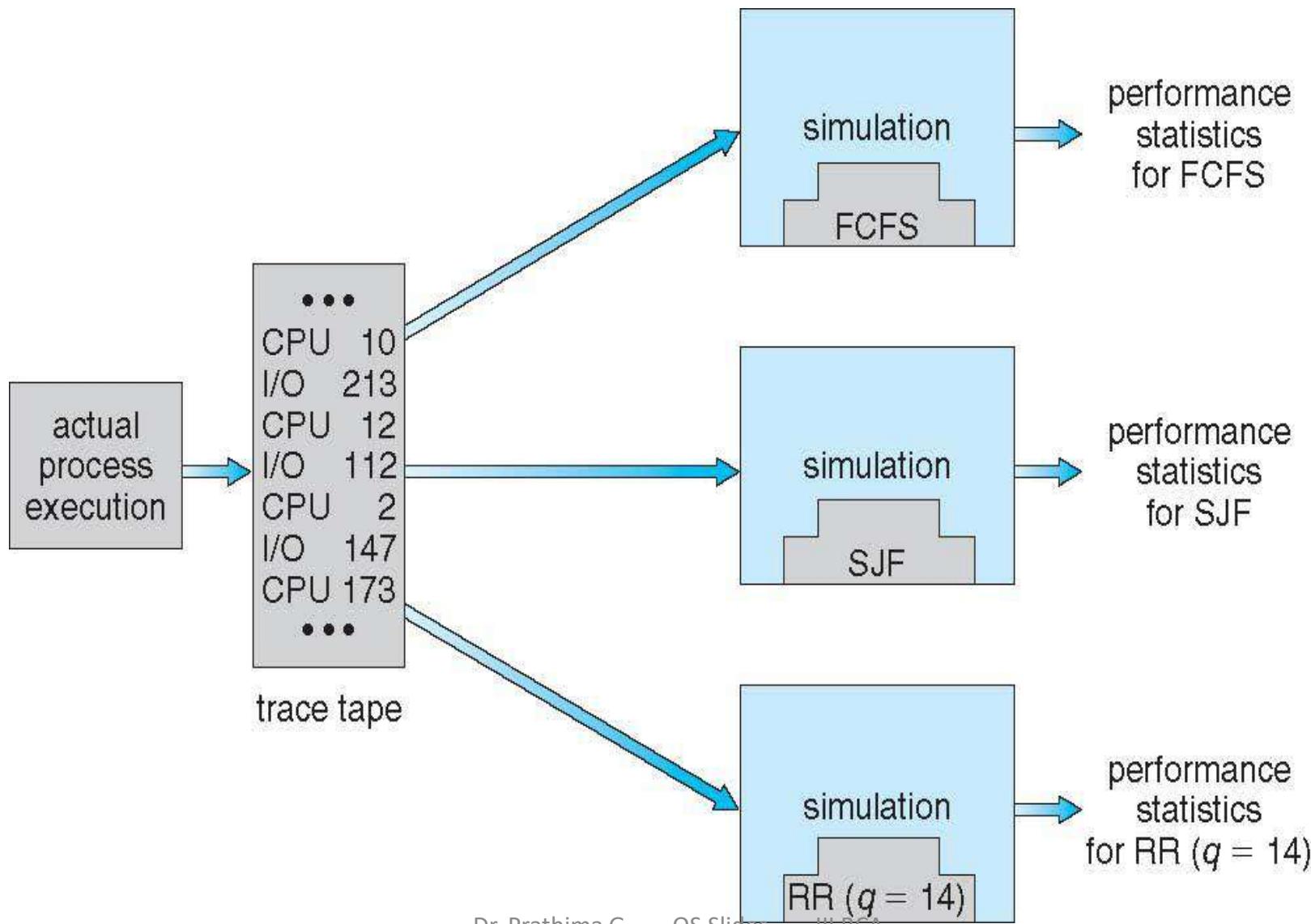
Simulations

The data to drive the simulation can be generated in several ways.

- The most common method uses a random-number generator that is programmed to generate processes, CPU burst times, arrivals, departures, and so on, according to probability distributions.
- The distributions can be defined mathematically (uniform, exponential, Poisson) or empirically.

- Trace tapes record sequences of real events in real systems
- Trace tapes provide an excellent way to compare two algorithms on exactly the same set of real inputs.
- This method can produce accurate results for its inputs. Simulations can be expensive, often requiring hours of computer time.
- A more detailed simulation provides more accurate results, but it also takes more computer time.
- In addition, trace tapes can require large amounts of storage space. Finally, the design, coding, and debugging of the simulator can be a major task

Evaluation of CPU Schedulers by Simulation



Implementation

- Even a simulation is of limited accuracy.
- **The only accurate way to evaluate a scheduling algorithm is to code it up, put it in the operating system, and see how it works.**
- **This approach puts the actual algorithm in the real system for evaluation under real operating conditions.**

- **The major difficulty with this approach is the high cost.**
- The expense is incurred not only in coding the algorithm and modifying the operating system to support it (along with its required data structures)
- but also in the reaction of the users to a constantly changing operating system.

Implementation

- Another difficulty is that the environment in which the algorithm is used will change.
-
- The environment will change not only in the usual way, as new programs are written and the types of problems change, but also as a result of the performance of the scheduler.
- If short processes are given priority, then users may break larger processes into sets of smaller processes.
- If interactive processes are given priority over non interactive processes, then users may switch to interactive use.
- Another approach is to use APIs that modify the priority of a process or thread.
- The Java, /POSIX, and /WinAPI/ provide such functions.
- The downfall of this approach is that performance-tuning a system or application most often does not result in improved performance in more general situations

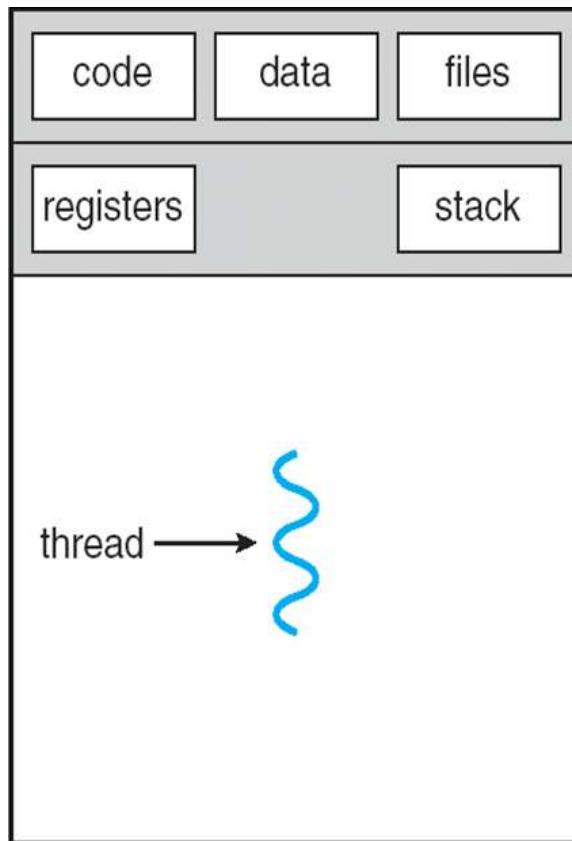
Threads

- Multithreading Models
- Thread Libraries
- Threading Issues
- threading in java

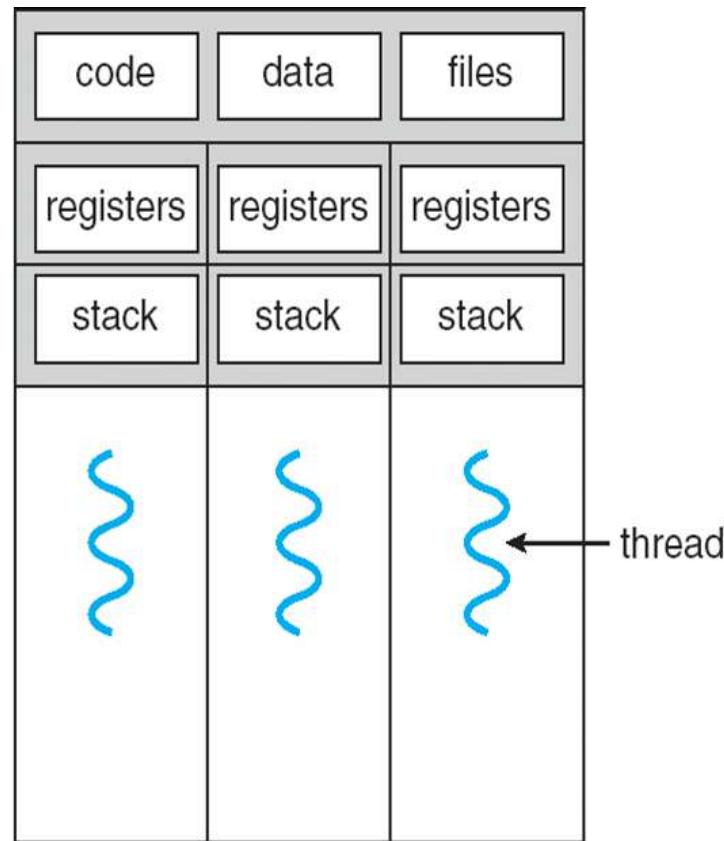
Motivation

- Threads run within application
- Multiple tasks within the application can be implemented by separate threads
 - Update display
 - Fetch data
 - Spell checking
 - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Single and Multithreaded Processes



single-threaded process



multithreaded process

Benefits

- **Responsiveness :**
- Multithreading an interactive application may allow a program to continue running even if part of it is blocked or
- is performing a lengthy operation,
- thereby increasing responsiveness to the user

Benefits

- **Resource Sharing**
- Processes may only share resources through techniques such as shared memory or message passing.
- Such techniques must be explicitly arranged by the programmer.
- However, threads share the memory and the resources of the process to which they belong by default.

Benefits

- **Economy**
Allocating memory and resources for process creation is costly.
- Because threads share the resources of the process to which they belong,
- It is more economical to create and context-switch threads

Benefits

- **Scalability**
- The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors

Multicore Programming

- Multicore systems putting pressure on programmers, challenges include:
 - **Dividing activities**
 - **Balance**
 - **Data splitting**
 - **Data dependency**
 - **Testing and debugging**

Multicore Programming

- **Dividing activities**
- This involves examining applications to find areas that can be divided into separate, concurrent tasks and thus can run in parallel on individual cores.

Multicore Programming

- Balance
- While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value

Multicore Programming

- **Data splitting**
- Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores

Multicore Programming

- Multicore systems putting pressure on programmers, challenges include:
 - **Dividing activities**
 - **Balance**
 - **Data splitting**
 - **Data dependency**
 - **Testing and debugging**

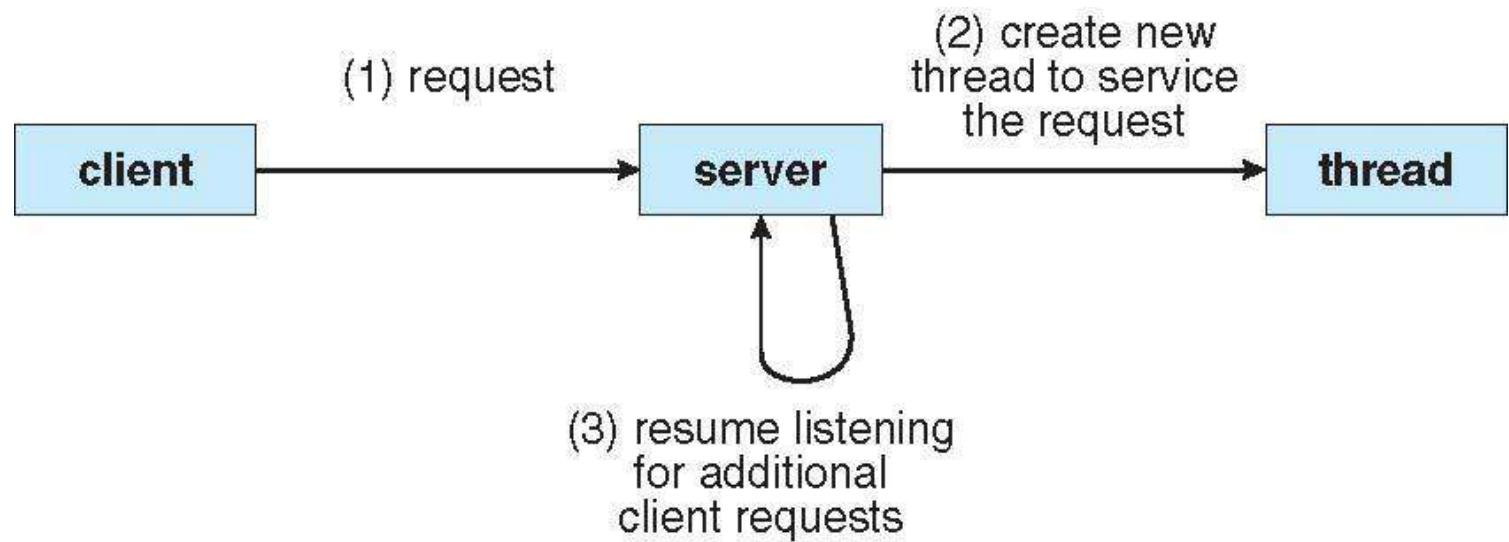
Multicore Programming

- **Data dependency**
- The data accessed by the tasks must be examined for dependencies between two or more tasks.
- In instances where one task depends on data from another,
- programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency.

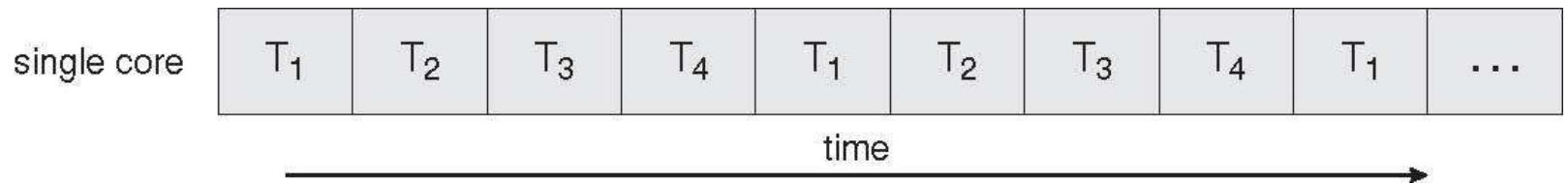
Multicore Programming

- **Testing and debugging**
- When a program is running in parallel on multiple cores, there are many different execution paths.
- Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications

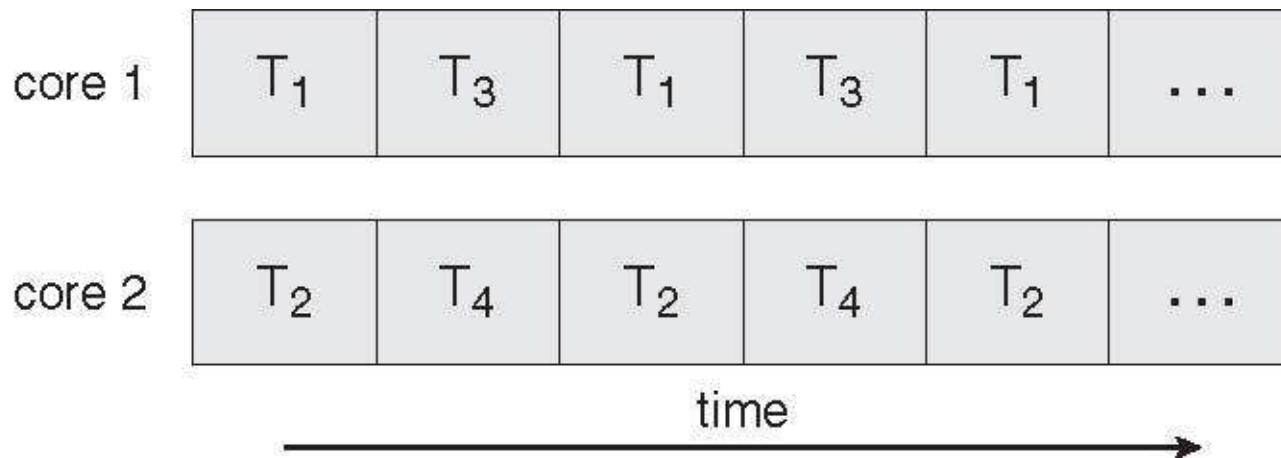
Multithreaded Server Architecture



Concurrent Execution on a Single-core System



Parallel Execution on a Multicore System



User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
 - POSIX **Pthreads**
 - Win32 threads
 - Java threads

Kernel Threads

- Supported by the Kernel
- Examples
 - Windows XP/2000
 - Solaris
 - Linux
 - Tru64 UNIX
 - Mac OS X

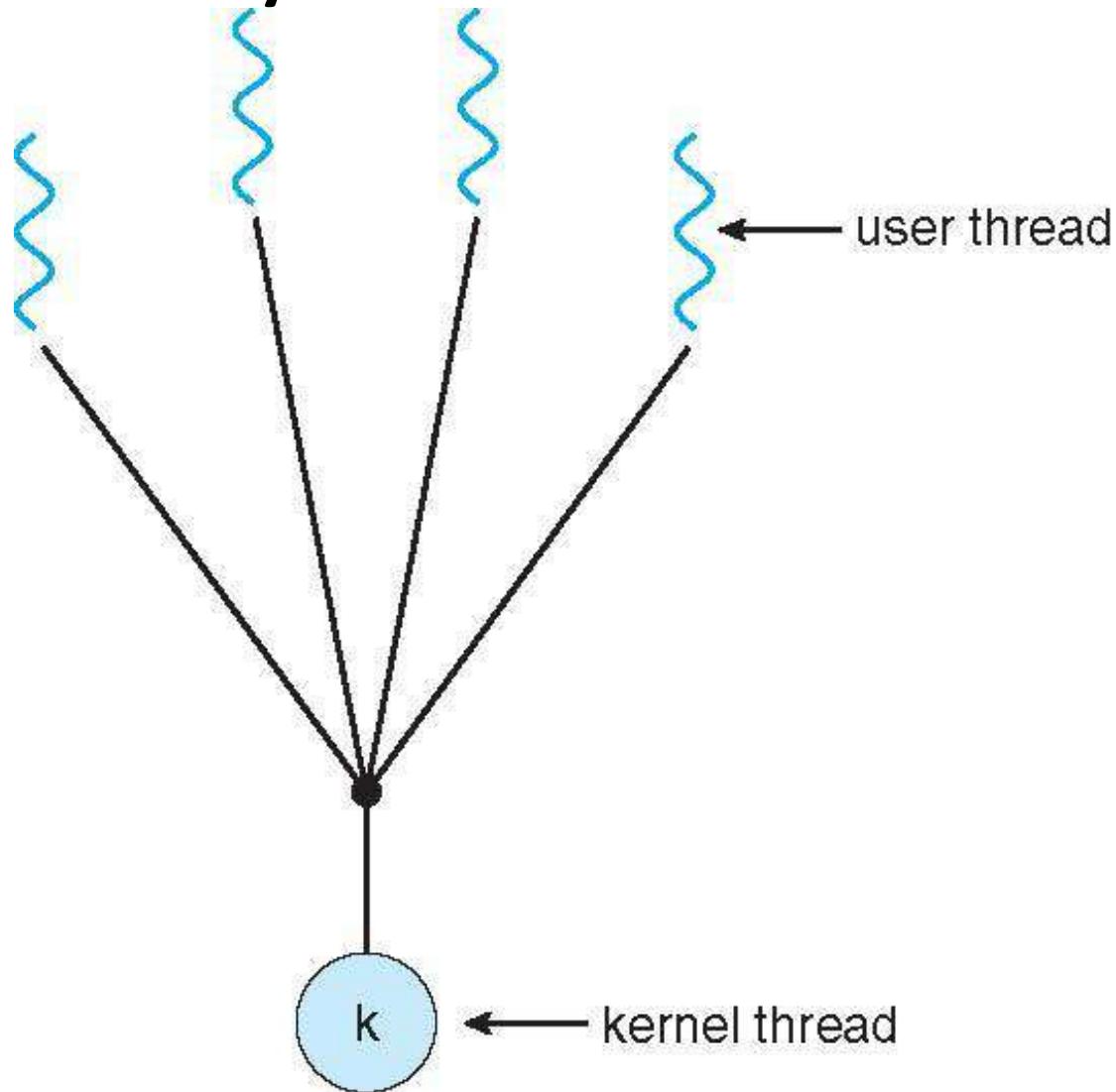
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread
- Thread management is done by the thread library in user space, so it is efficient
- Only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors
- Examples:
 - Solaris Green Threads
 - GNU Portable Threads

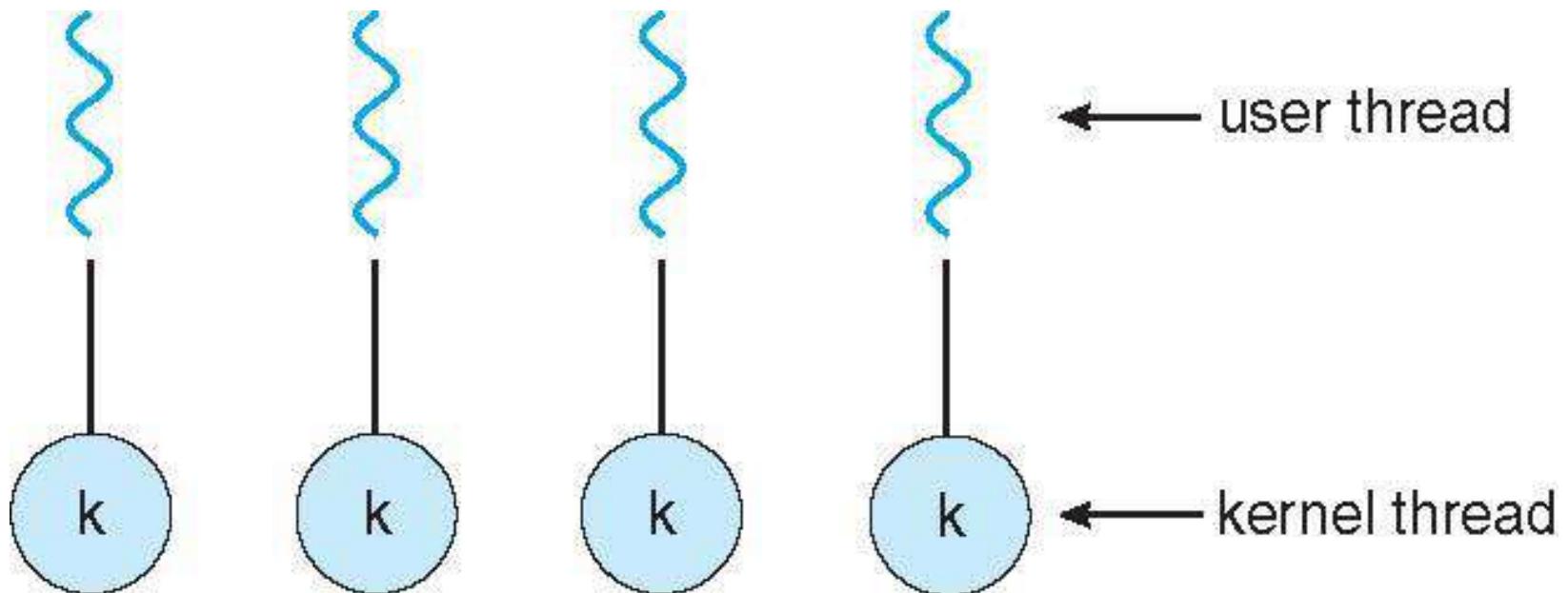
Many-to-One Model



One-to-One

- Each user-level thread maps to kernel thread
- It also allows multiple threads to run in parallel on multiprocessors.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.
- Because the overhead of creating kernel threads can burden the performance of an application
- Examples
 - Windows NT/XP/2000
 - Linux
 - Solaris 9 and later

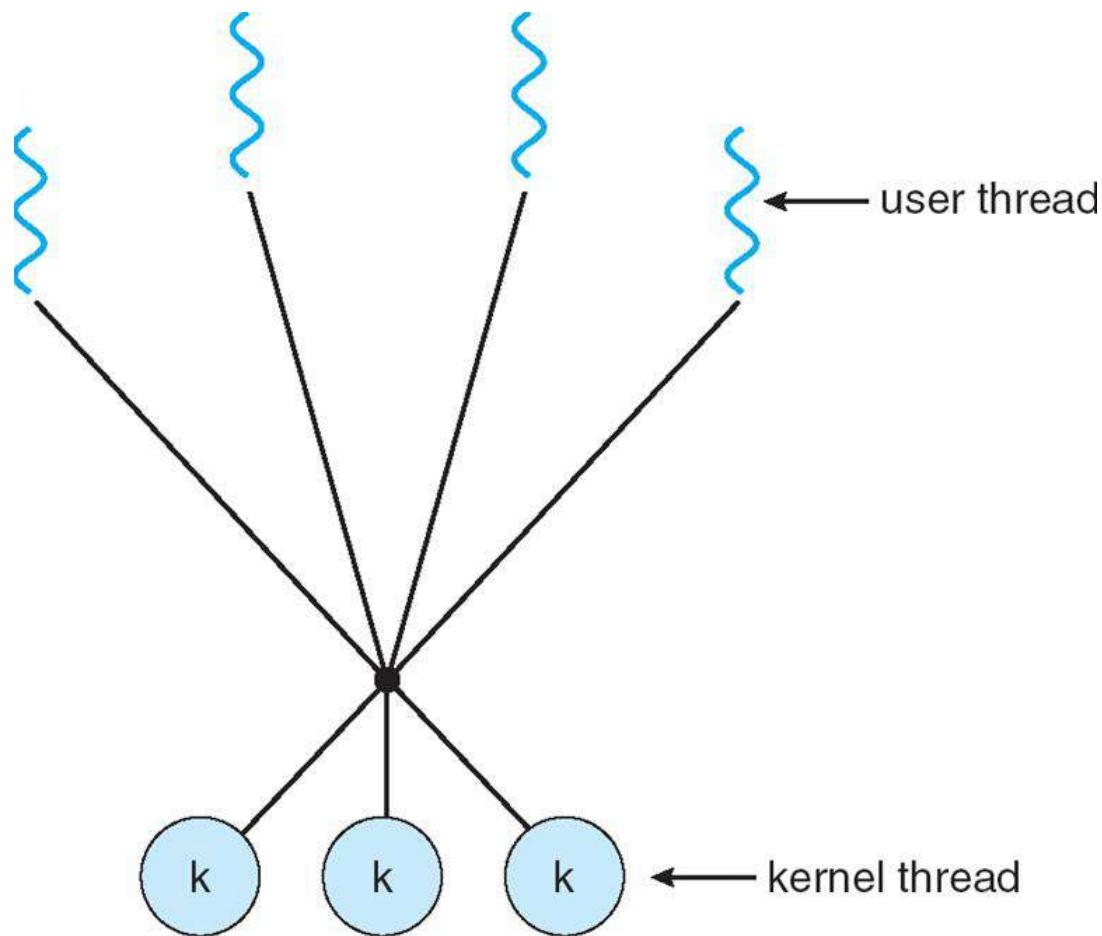
One-to-one Model



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package

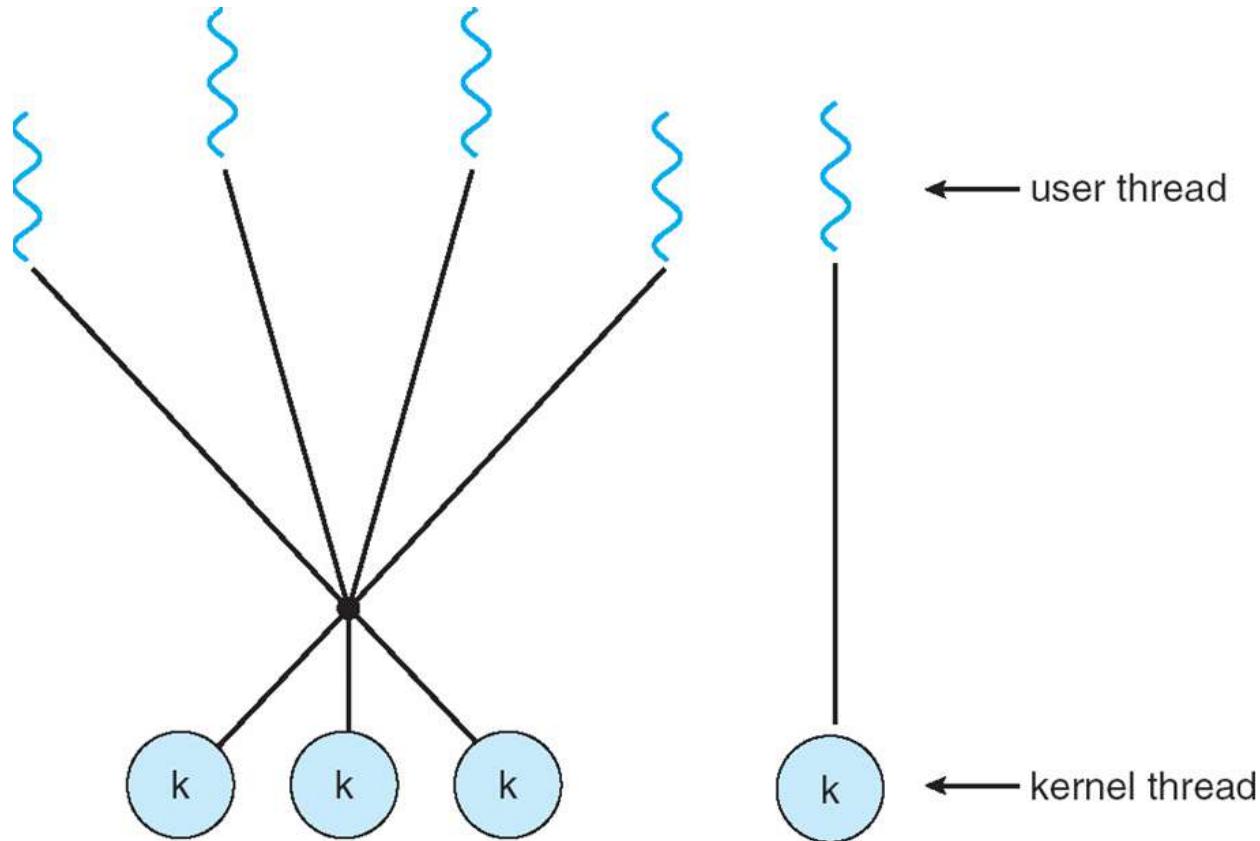
Many-to-Many Model



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
 - IRIX
 - HP-UX
 - Tru64 UNIX
 - Solaris 8 and earlier

Two-level Model



Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
 - Library entirely in user space
 - Kernel-level library supported by the OS

Types

- Three main thread libraries are in use today: (1) POSIX Pthreads, (2) Win32, and (3) Java.
- **Pthreads**, the threads extension of the POSIX standard
- The **Win32 thread** library is a kernel-level library available on Windows systems.
- The **Java thread** API allows threads to be created and managed directly in Java programs

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

PThreads Example

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr, "usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
        return -1;
    }
}
```

Pthreads Example

```
/* get the default attributes */
pthread_attr_init(&attr);
/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);
/* wait for the thread to exit */
pthread_join(tid,NULL);

printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

Figure 4.9 Multithreaded C program using the Pthreads API.

Win32 API Multithreaded C Program

- The technique for creating threads using the Win32 thread library is similar to the Pthreads technique.
- We illustrate the Win32 thread API in the C program shown in Figure

Win32 API Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* the thread runs in this separate function */

DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;
    return 0;
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;
    /* perform some basic error checking */
    if (argc != 2) {
        fprintf(stderr, "An integer parameter is required\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr, "An integer >= 0 is required\n");
        return -1;
    }
}
```

Win32 API Multithreaded C Program (Cont.)

```
// create the thread
ThreadHandle = CreateThread(
    NULL, // default security attributes
    0, // default stack size
    Summation, // thread function
    &Param, // parameter to thread function
    0, // default creation flags
    &ThreadId); // returns the thread identifier

if (ThreadHandle != NULL) {
    // now wait for the thread to finish
    WaitForSingleObject(ThreadHandle, INFINITE);

    // close the thread handle
    CloseHandle(ThreadHandle);

    printf("sum = %d\n", Sum);
}

}
```

Figure 4.10 Multithreaded C program using the Win32 API.

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
 - Extending Thread class
 - Implementing the Runnable interface

Java Multithreaded Program

```
class Sum
{
    private int sum;

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
}

class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run() {
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.setSum(sum);
    }
}
```

Java Multithreaded Program

```
public class Driver
{
    public static void main(String[] args) {
        if (args.length > 0) {
            if (Integer.parseInt(args[0]) < 0)
                System.err.println(args[0] + " must be >= 0.");
            else {
                // create the object to be shared
                Sum sumObject = new Sum();
                int upper = Integer.parseInt(args[0]);
                Thread thrd = new Thread(new Summation(upper, sumObject));
                thrd.start();
                try {
                    thrd.join();
                    System.out.println
                        ("The sum of "+upper+" is "+sumObject.getSum());
                } catch (InterruptedException ie) { }
            }
        }
        else
            System.err.println("Usage: Summation <integer value> ");
    }
}
```

Figure 4.11 Java program for the summation of a non-negative integer.

Threading Issues

1. Semantics of fork() and exec() system calls

- Some UNIX systems have chosen to have two versions of fork(),
- one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call
- if a thread invokes the exec() system call, the program specified in the parameter to exec () will replace the entire process-including all threads

Threading Issues

2.Thread cancellation of target thread

- This is the task of terminating a thread before it has completed

A thread that is to be canceled is often referred to as the Cancellation of a target thread may occur in two different scenarios:

- Asynchronous cancellation. One thread immediately terminates the target thread.
- Deferred cancellation. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

Threading Issues

- **3. Signal handling**
- A Signal is used in UNIX systems to notify a process that a particular event has occurred.
- A signal may be received either synchronously or asynchronously, depending on the source of and the reason for the event being signaled.
- All signals, whether synchronous or asynchronous, follow the same pattern:
 1. A signal is generated by the occurrence of a particular event.
 2. A generated signal is delivered to a process.
 3. Once delivered, the signal must be handled.

4. Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
 - Usually slightly faster to service a request with an existing thread than create a new thread
 - Allows the number of threads in the application(s) to be bound to the size of the pool

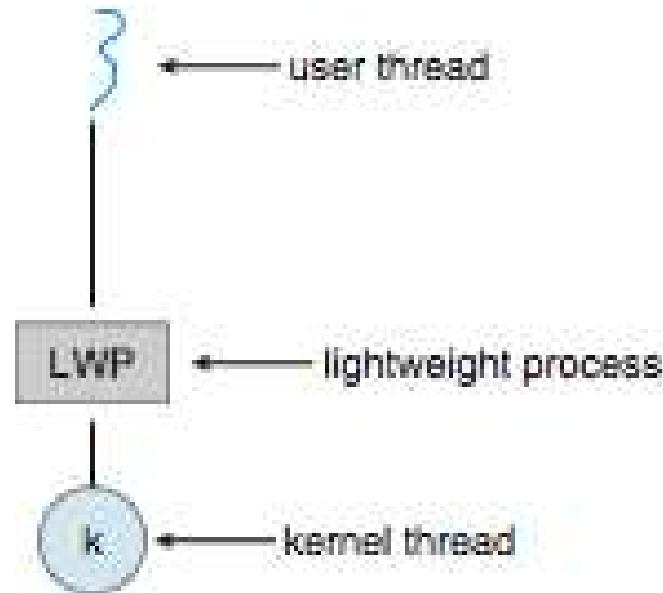
5. Thread Specific Data

- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

6. Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads

Lightweight Processes



Module 4

MEMORY MANAGEMENT

Topics

Main memory

1. Swapping,
2. Contiguous Memory Allocation,
3. Paging,
4. Structure Of Page Table,
5. Segmentation,
6. Examples.

Virtual memory

1. Demand Paging,
2. Copy On Write,
3. Page Replacement,
4. Allocation of Frames,
5. Thrashing,
6. Memory Mapped Files,
7. Allocating Kernel Memory,
8. Memory Management Utilities

Main Memory - Introduction

- Program must be brought (from disk) into memory and then brought into processor for execution
- Main memory and registers are only storage CPU can access directly
- 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

Figure : A base and a limit register define a logical address space

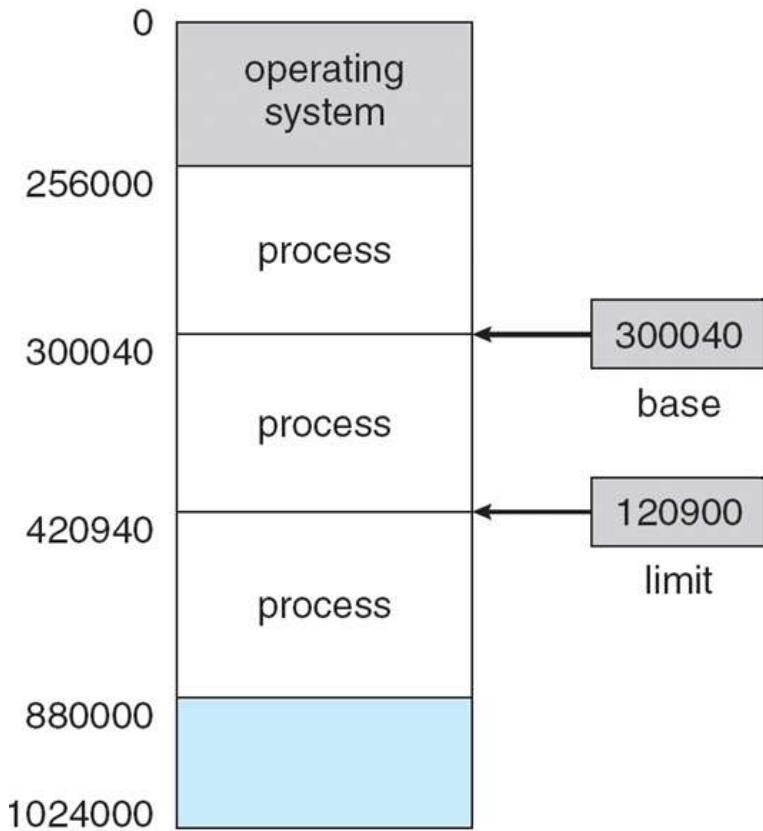
Each process has a separate memory space.

System has 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.

This protection is provided by using two registers, a base and a limit registers

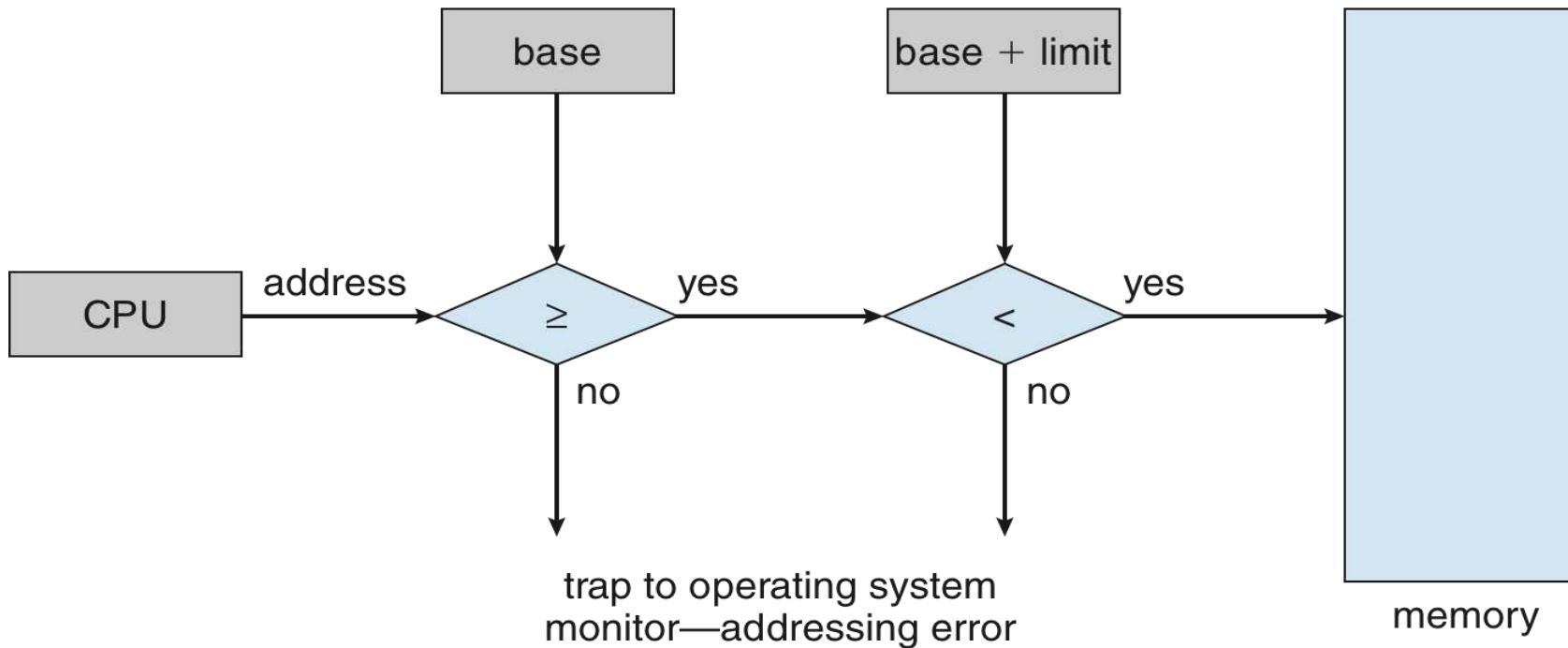
The base holds the smallest legal physical memory address;

Limit register the 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 of both).

Hardware Address Protection with Base and Limit Registers



Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.

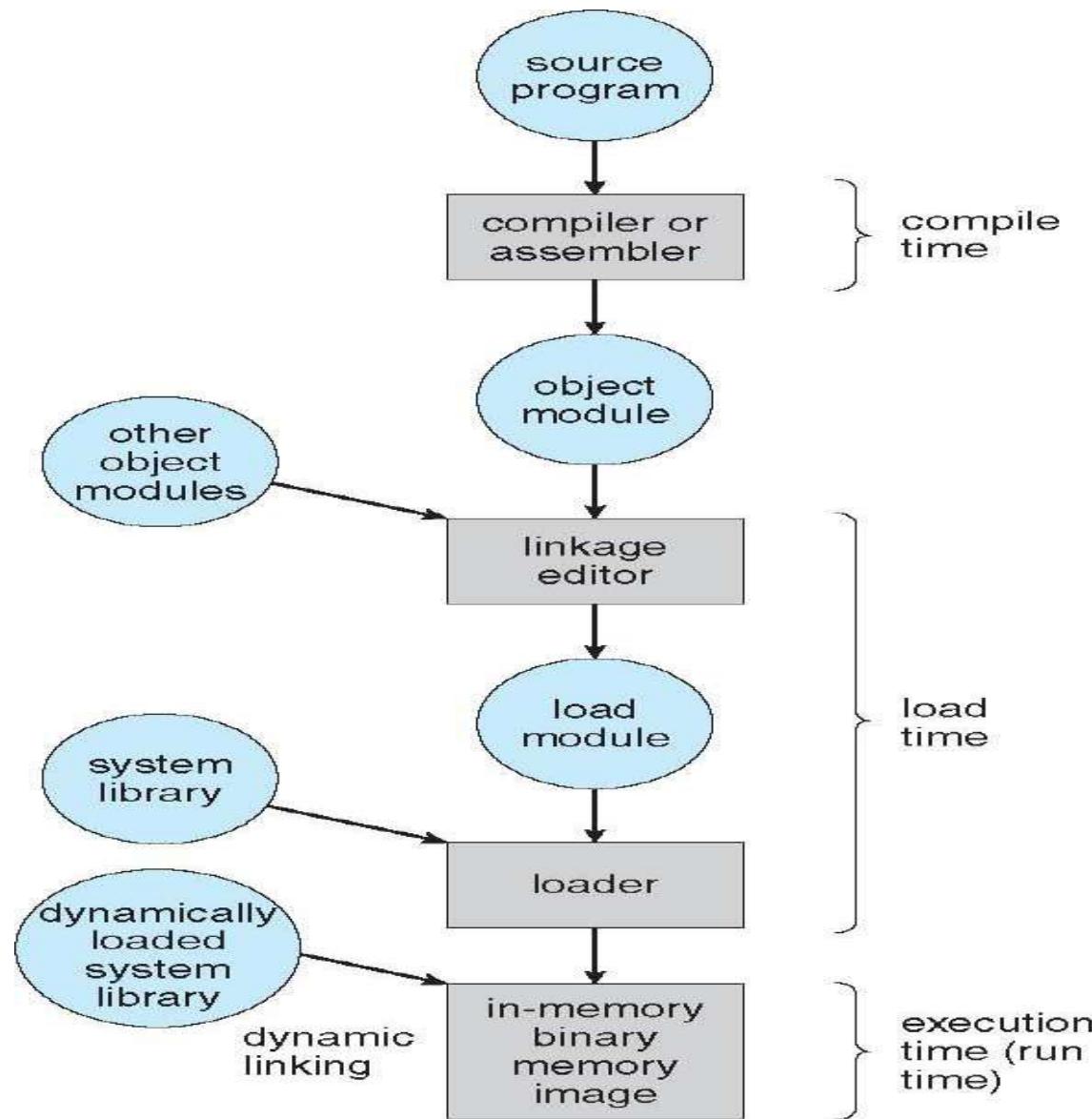
Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error .

This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

Address Binding

- Most systems allow a user process to reside in any part of the physical memory.
- Thus, although the address space of the computer starts at 00000, the first address of the user process need not be 00000.
- **Address binding of instructions and data to memory addresses** can happen at three different stages
 - **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
 - **Load time:** Must generate **relocatable code** if memory location is not known at compile time
 - **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another
 - Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



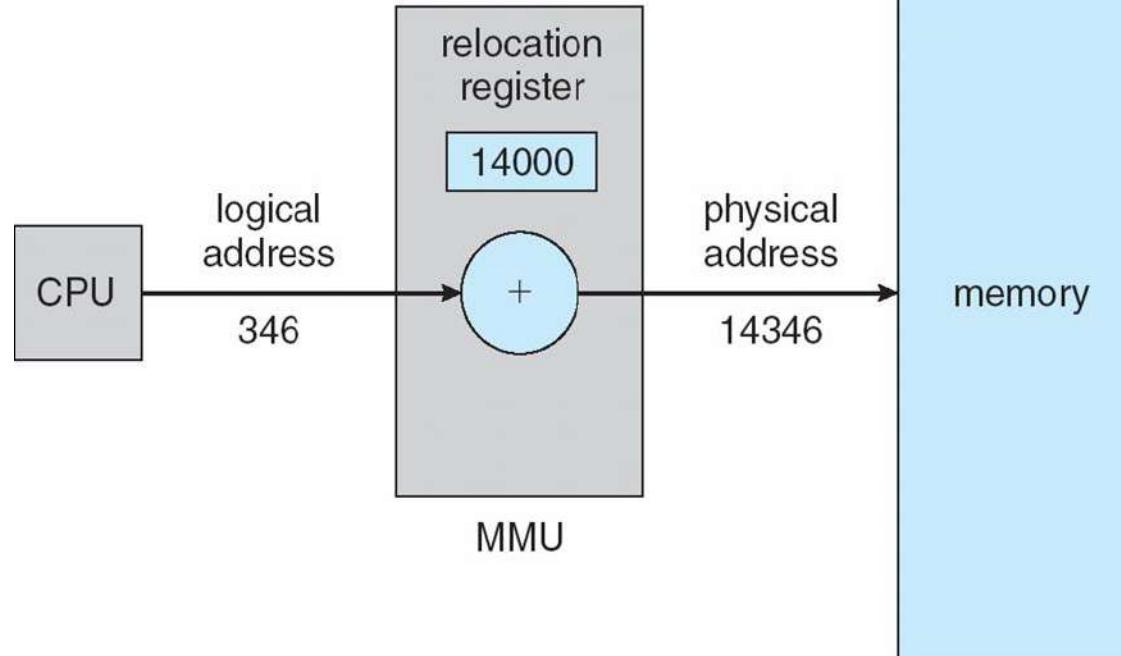
Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
 - **Logical address** – generated by the CPU; also referred to as **virtual address**
 - **Physical address** – address seen by the memory unit
- **Logical and physical addresses are the same in compile-time and load-time address-binding schemes**
- logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** the set of all physical addresses corresponding to these logical addresses

Memory-Management Unit

- **Hardware device that at run time maps virtual to physical address**
- Many methods possible
- Consider simple scheme where the **value in the relocation register is added to every address generated by a user process** at the time it is sent to memory
 - Base register now called **relocation register**
 - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
 - Execution-time binding occurs when reference is made to location in memory
 - Logical address bound to physical addresses

Dynamic relocation using a relocation register



- The base register is now called a 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.
- 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.

Dynamic Loading

- 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 overcome this in dynamic loading , routine is not loaded until it is called**
- **Better memory-space utilization; unused routine is never loaded**
- **All routines kept on disk in relocatable load format**
- **Useful when large amounts of code are needed to handle infrequently occurring cases**
- **No special support from the operating system is required**
 - Implemented through program design
 - OS can help by providing libraries to implement dynamic loading

Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- **Dynamic linking** –linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
 - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
 - Versioning may be needed

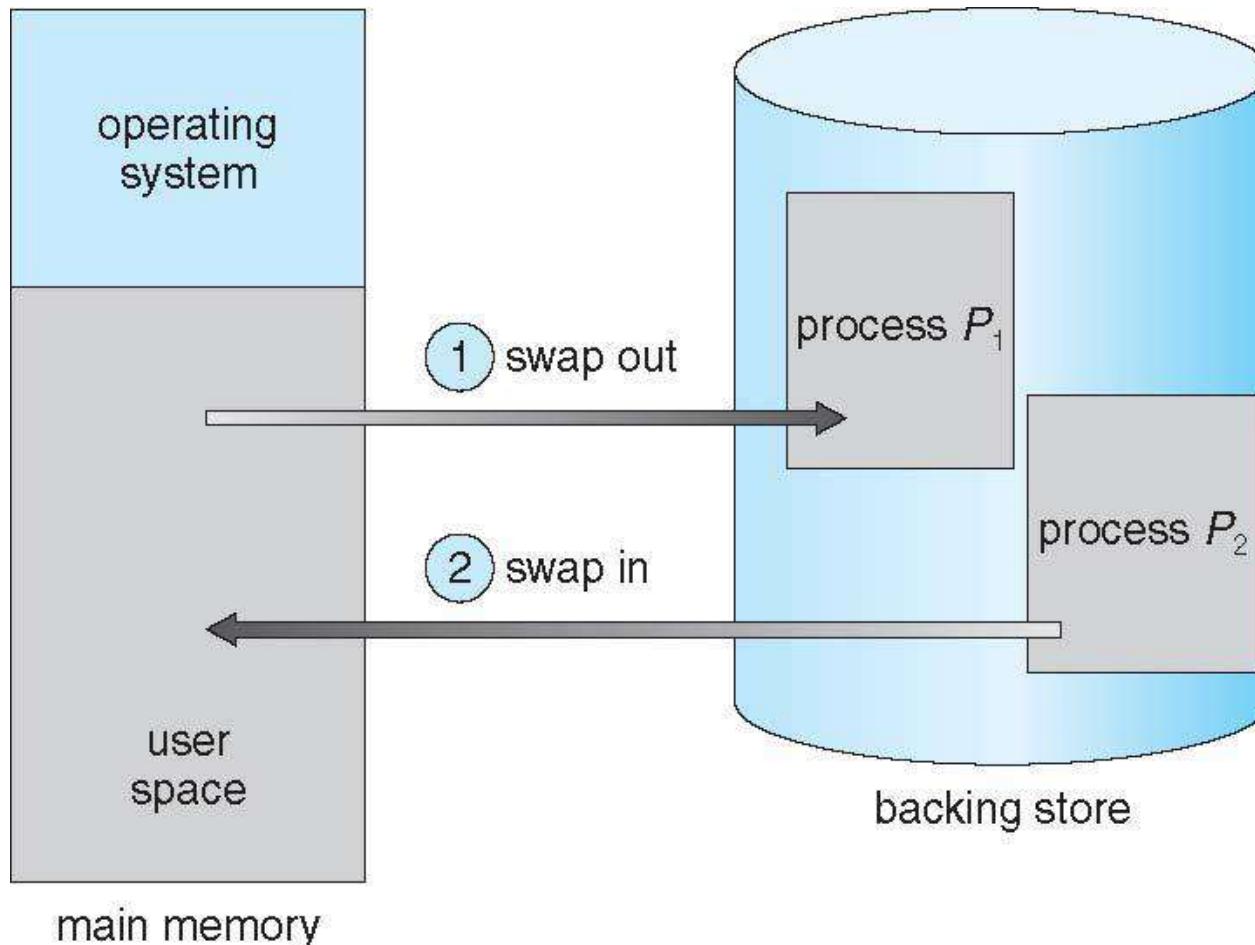
Swapping

- A process must be in memory to be executed.
- A process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.
- For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm
- When a quantum expires, the memory manager will start to swap out the process that just finished and
- to swap another process into the memory space that has been freed

Swapping

- In the meantime, the CPU scheduler will allocate a time slice to some other process in memory.
- When each process finishes its quantum, it will be swapped with another process.
- A variant of this swapping policy is used for priority-based scheduling algorithms
- If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process
- When the higher-priority process finishes, the lower-priority process can be swapped back in and continued
- **This variant of swapping is sometimes called roll out and roll in**

Schematic View of Swapping



Swapping

- Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously.
- This restriction is dictated by the method of address binding.
- If binding is done at assembly or load time, then the process cannot be easily moved to a different location.
- If execution-time binding is being used, however, then a process can be swapped into a different memory space,
- because the physical addresses are computed during execution time
- Swapping requires a backing store. The backing store is commonly a fast disk.

Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
- The actual transfer of the 100-MB process to or from main memory takes
- $100\text{MB}/50\text{MB \text{per second}} = 2 \text{ seconds}$
 - Plus disk latency of 8 ms
 - Swap out time of 2008 ms
 - Plus swap in of same sized process
 - Total context switch swapping component time of 4016ms (> 4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used-System calls to inform OS of memory use via request memory and release memory

Contiguous Allocation

- The memory is usually divided into two partitions: **one for the resident operating system and one for the user processes.**
- We can place the operating system in either low memory or high memory.
- The major factor affecting this decision is the location of the interrupt vector.
- Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well.

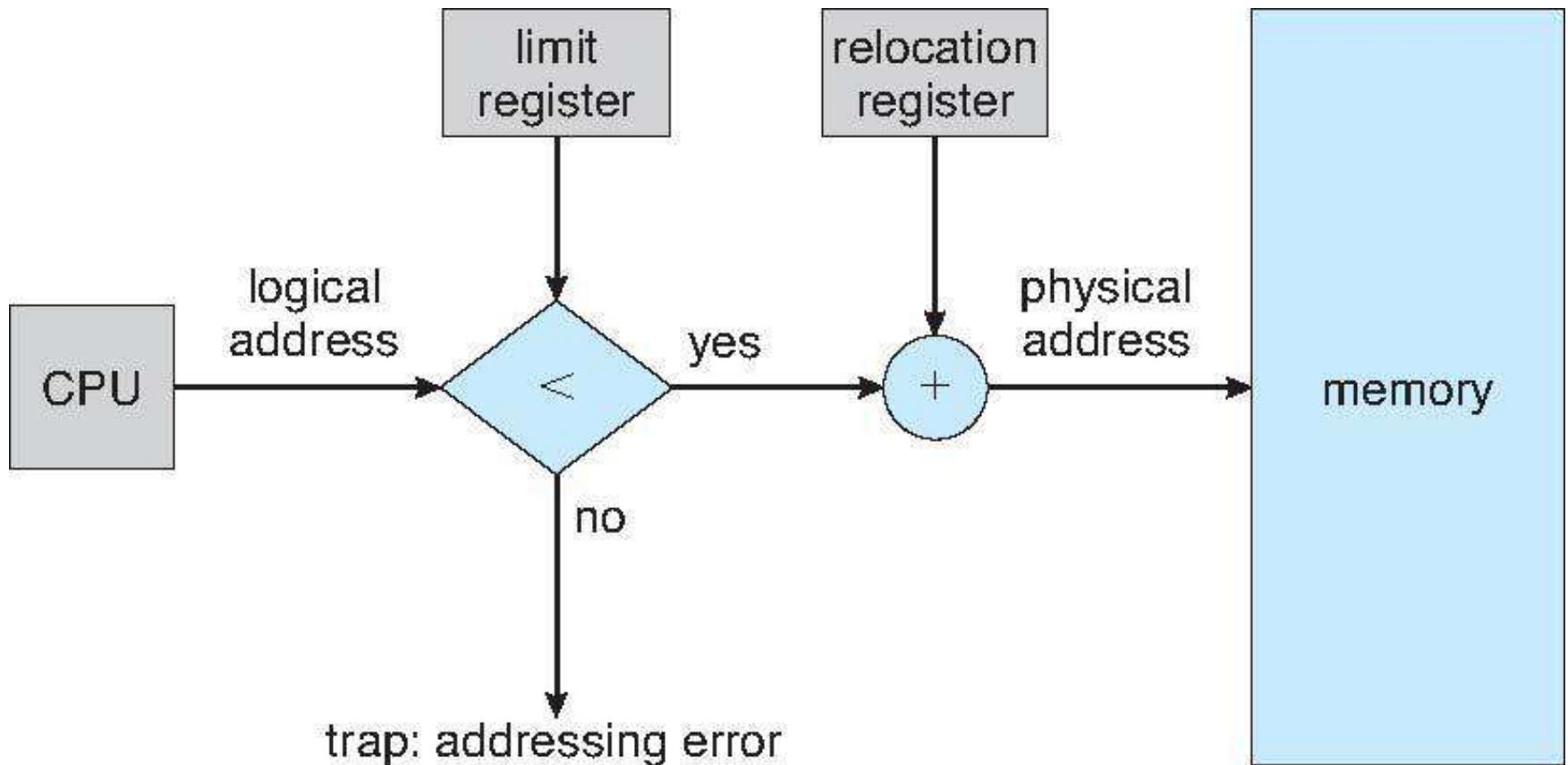
Contiguous Allocation

- 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 in the input queue waiting to be brought into memory
- In contiguous memory allocation, each process is contained in a single contiguous section of memory

Memory Mapping and Protection

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
 - Base register contains value of smallest physical address
 - Limit register contains range of logical addresses – each logical address must be less than the limit register
 - MMU maps logical address *dynamically*

Hardware Support for Relocation and Limit Registers

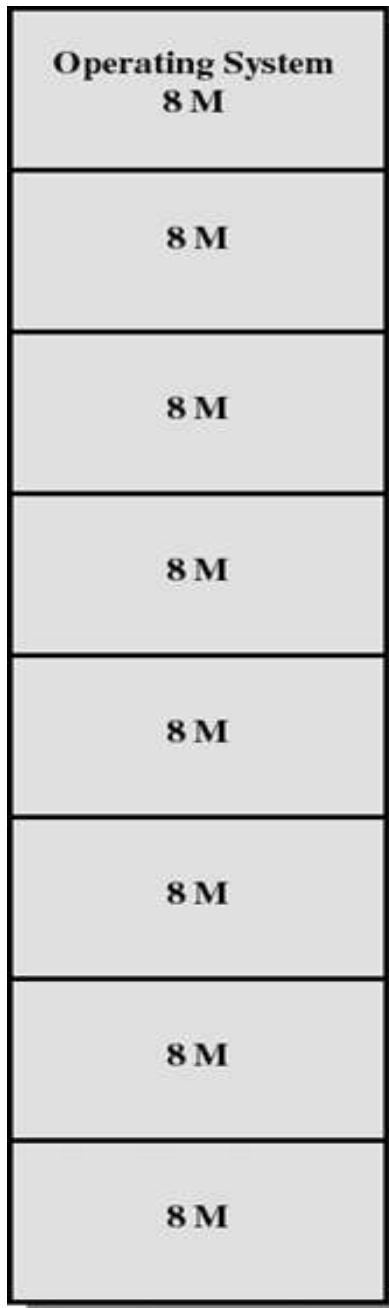


Memory Allocation – Fixed sized partitions

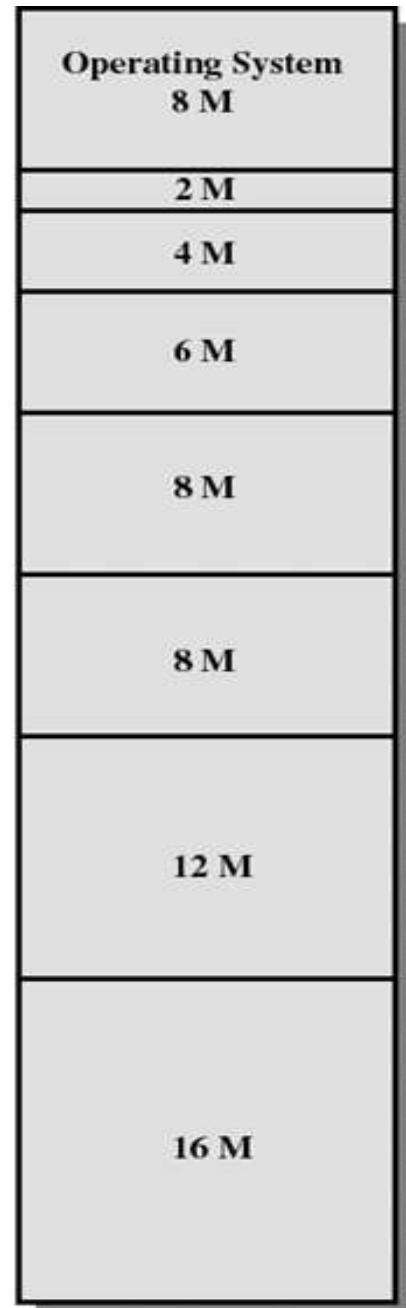
- One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions.
- Each partition may contain exactly one process.
- Thus, the degree of multiprogramming is bound by the number of partitions.
- In this when a partition is free, a process is selected from the input queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process.

Fixed partitions can be of equal or unequal sizes

This kind of partitions no longer in use as it suffers from internal fragmentation



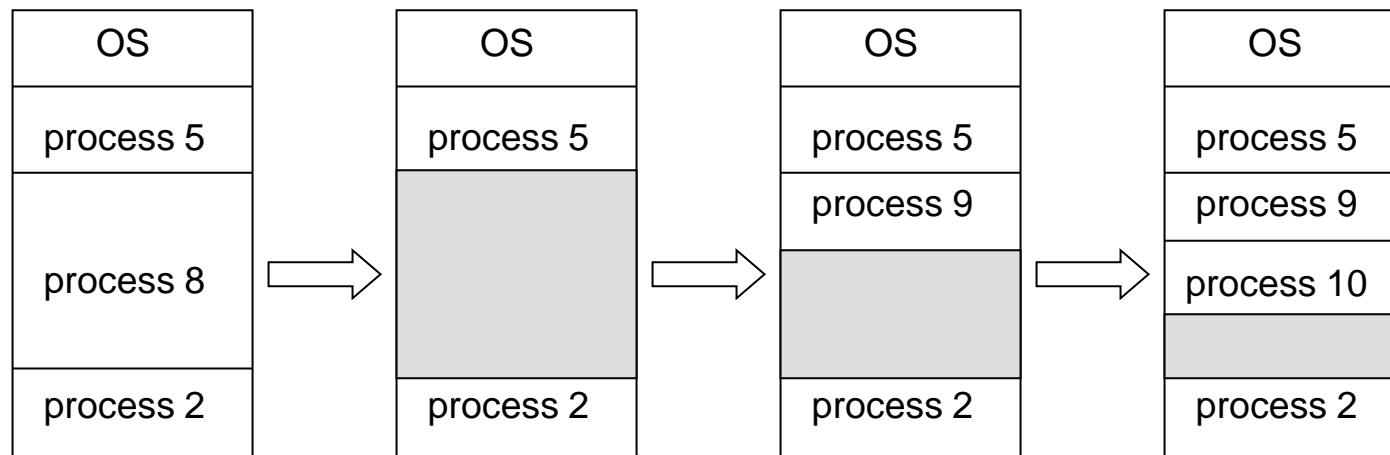
Dr. Prathibha Gopalakrishna Notes IIT Roorkee
Equal-size partitions



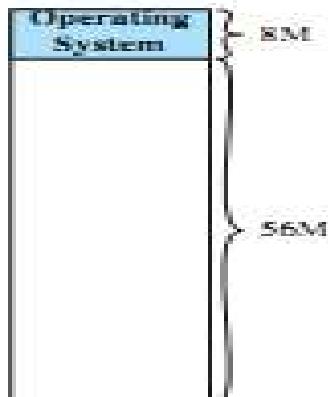
Unequal-size partitions 23

Memory Allocation – Variable sized partitions

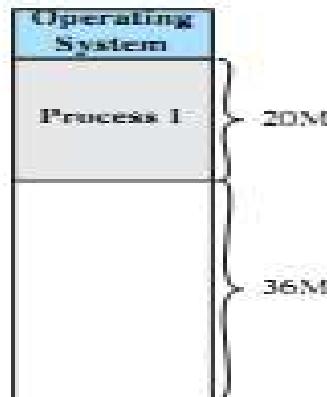
- In the scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.
- Initially, all memory is available for user processes and is considered one large block of available memory a hole.
- Eventually, memory contains a set of holes of various sizes



Variable Partitioning: example



(a)



(b)



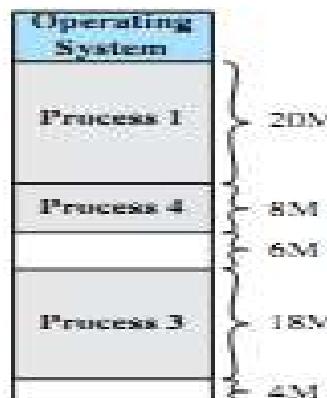
(c)



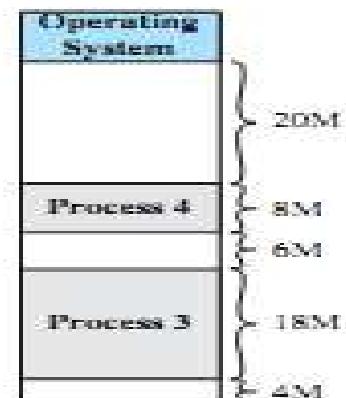
(d)



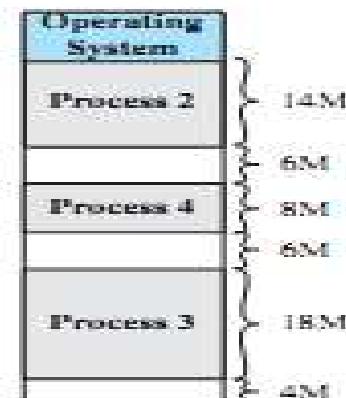
(e)



(f)



(g)



(h)

Variable sized partitions

- The memory blocks available comprise a set of partitions of various sizes scattered throughout memory.
- When a process arrives and needs memory, the system searches the set for a partition that is large enough for this process.
- If the partition 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 free partitions.

Variable sized partitions

- When a process terminates, it releases its block of memory, which is then placed back in the set of partitions.
- If the new partition is adjacent to other partitions, these adjacent partitions are merged to form one larger partition.
- At this point, the system may need to check whether there are processes waiting for memory and
- whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

Dynamic Storage-Allocation Problem

- This procedure is to select a free partition from the set of available free partitions.
- Three main strategies are
 - First Fit
 - Best Fit
 - Worst Fit

First Fit

- Allocate the first partition that is big enough.
- Searching can start either at the beginning of the set of partitions or
 - at the location where the previous first-fit search ended.
- We can stop searching as soon as we find a free partition that is large enough.

Best Fit

- Allocate the smallest partition that is big enough.
- Search the entire list, unless the list is ordered by size.
- This strategy produces the smallest leftover partition

Worst Fit

- Allocate the largest partition.
- Again, we must search the entire list, unless it is sorted by size.
- This strategy produces the largest leftover partition

Comparison

- Both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization.
- First fit is generally faster.
- Best fit gives better space utilization

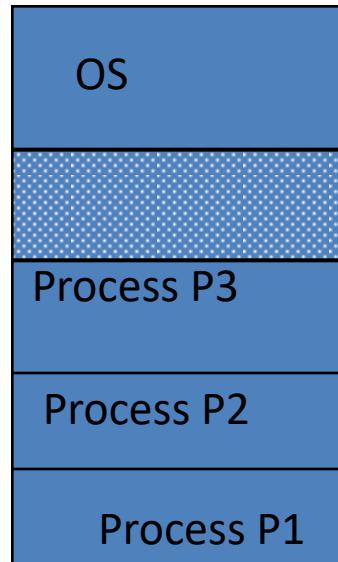
Fragmentation

- Wastage of memory space due to fixed sized and variable sized partitions in memory
- Two types
 1. Internal Fragmentation
 2. External Fragmentation

Internal Fragmentation

A program may not fit in a partition

Any program, no matter how small, occupies an entire partition.

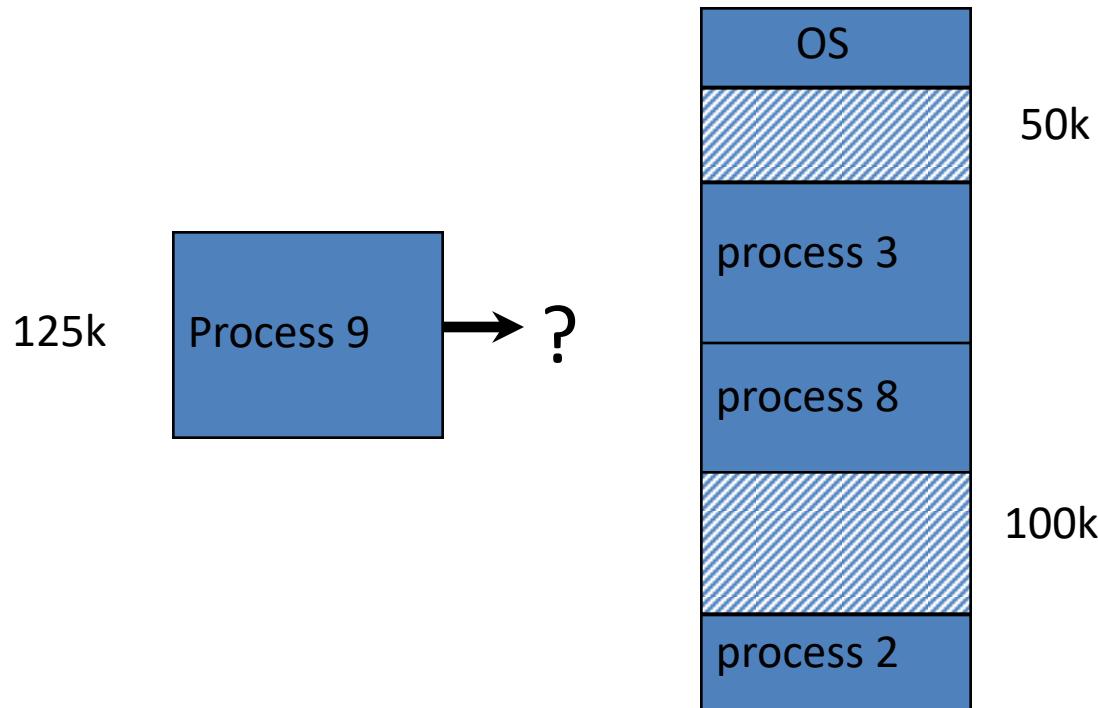


- Have some “empty” space for each processes

- **Internal Fragmentation** - allocated memory may be slightly larger than requested memory ;
- this size difference is memory internal to a partition, but not being used.

External Fragmentation

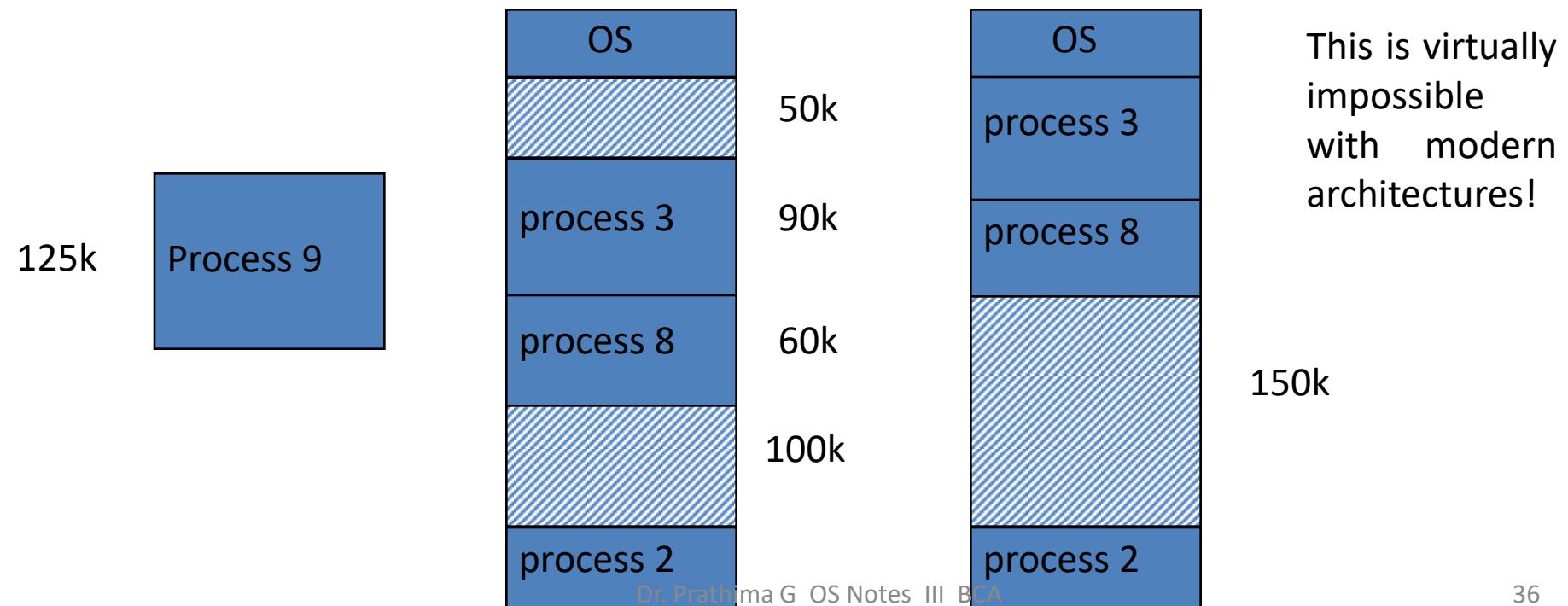
- External Fragmentation - total memory space exists to satisfy request but it is not contiguous



Memory external to all processes is fragmented

Compaction

- Reduce external fragmentation by **compaction**
 - Shuffle memory contents to place all free memory together in one large block
 - Compaction is possible *only* if relocation is dynamic, and is done at execution time
 - I/O problem
 - Latch job in memory while it is involved in I/O
 - Do I/O only into OS buffers

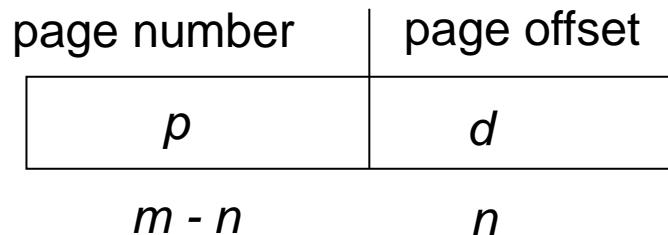


Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames**
 - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size N pages, need to find N free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation(only in the last page)

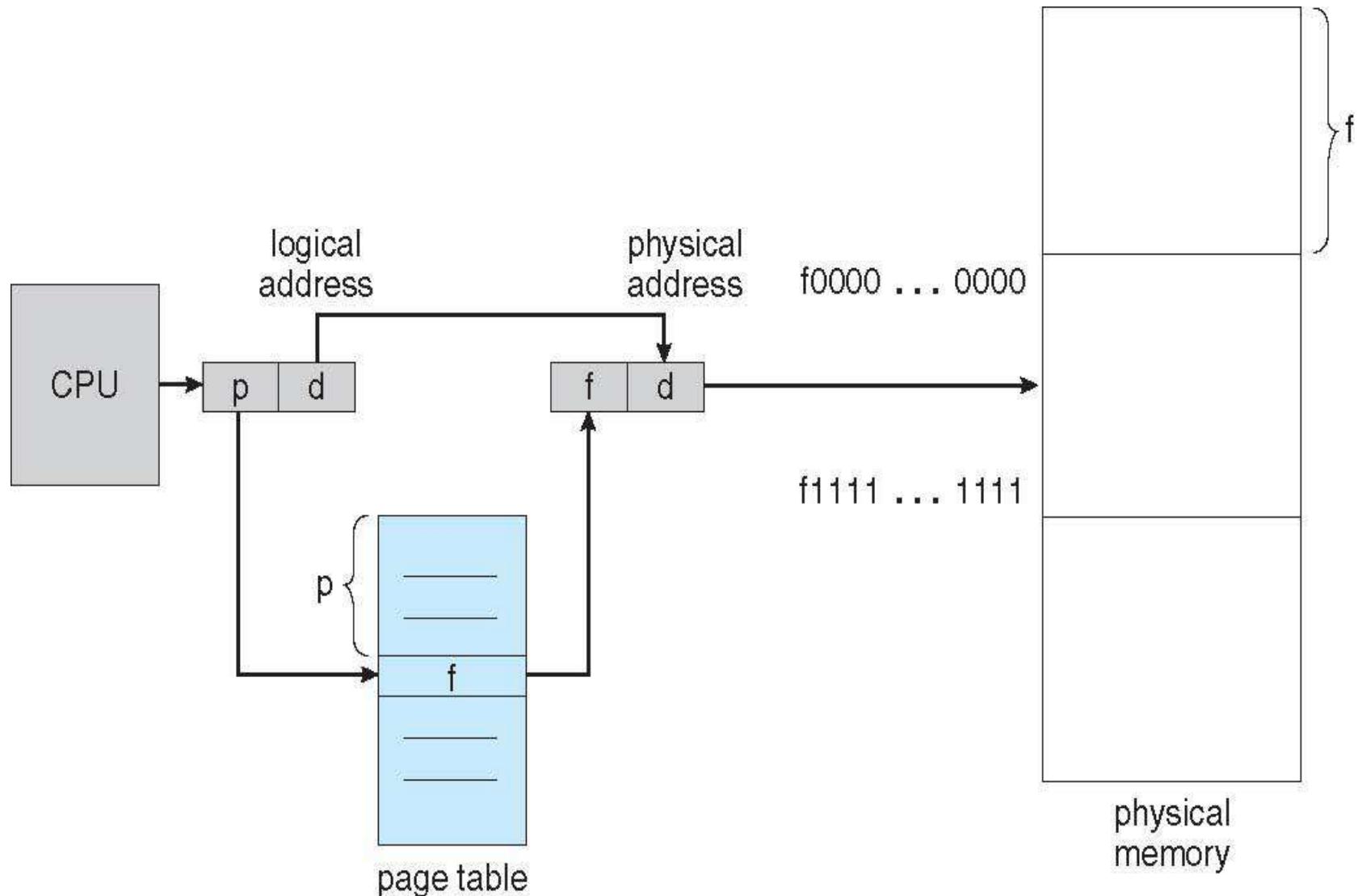
Address Translation Scheme

- Address generated by CPU is divided into:
 - **Page number (p)** – used as an index into a **page table** which contains base address of each page in physical memory
 - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit

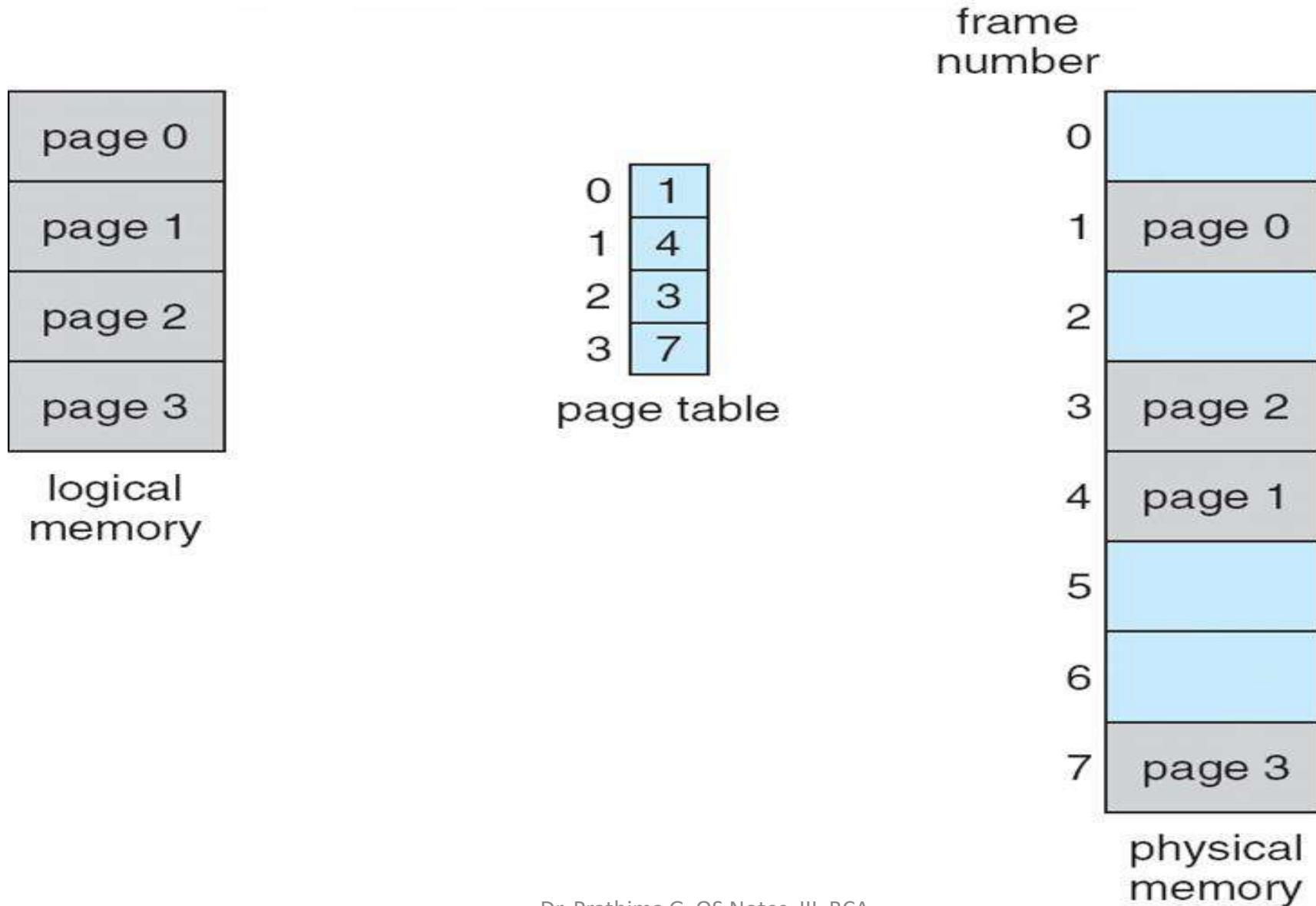


- If the size of the logical address space is 2^m , and a page size is 2^n addressing units (bytes or words) then
 - the high-order $m - n$ bits of a logical address designate the page number, and
 - the n low-order bits designate the page offset.

Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example – mapping logical address to Physical address

- Logical address, n= 2 and m = 4.
- Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages),
- 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 x 4) + 0].
- Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 x 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 x 4) + 0].
- Logical address 13 maps to physical address 9.

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	
8	
12	

4	i
5	j
6	k
7	l
8	m
9	n
10	o
11	p

12	
16	
20	
24	

16	
20	a
21	b
22	c
23	d

20	
21	e
22	f
23	g
24	h

physical memory

Paging Concept

- Paging is a form of dynamic relocation
- No external fragmentation: any free frame can be allocated to a process that needs it.
- However, we may have some internal fragmentation in the last frame
- Calculating internal fragmentation
 - Page size = 2,048 bytes
 - Process size = 72,766 bytes
 - 35 pages + 1,086 bytes
 - Internal fragmentation of $2,048 - 1,086 = 962$ bytes
 - Worst case fragmentation = 1 frame – 1 byte
 - On average fragmentation = 1 / 2 frame size

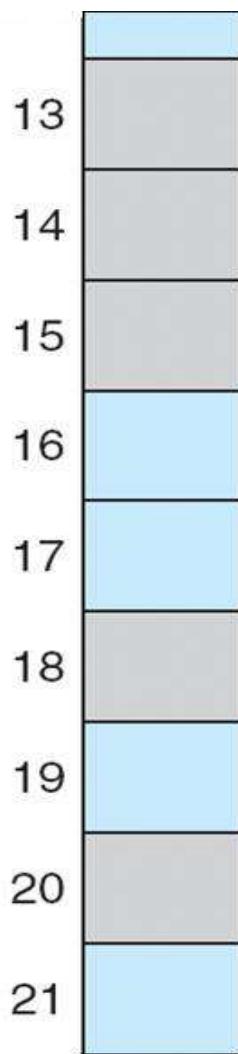
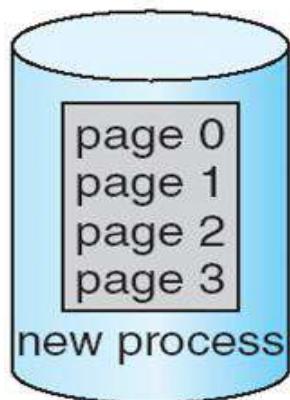
Allocating Free Frames

- 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

Free Frames

free-frame list

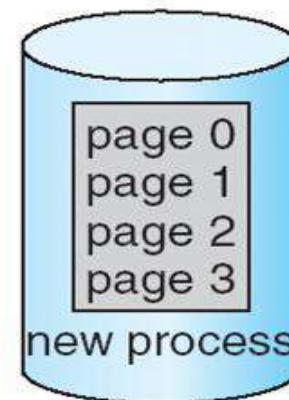
14
13
18
20
15



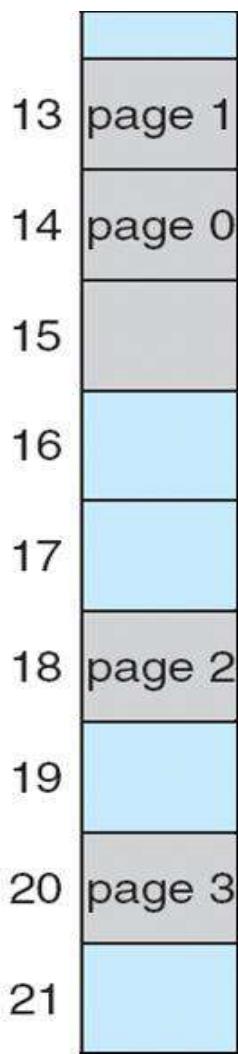
(a)

free-frame list

15



new-process page table



(b)

Before allocation

After allocation

Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
 - One for the page table and one for the data / instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**

Implementation of Page Table

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry
 - uniquely identifies each process to provide address-space protection for that process
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
 - Replacement policies must be considered
 - Some entries can be **wired down** for permanent fast access

Translation look-aside buffers (TLBs)

- The TLB contains only a few of the page-table entries.
- When a logical address is generated by the CPU, its page number is presented to the TLB.
- If the page number is found, its frame number is immediately available and is used to access memory.
- The whole task may take less than 10 percent longer than it would if an unmapped memory reference were used
- If the page number is not in the TLB (known as a a memory reference to the page table must be made.

Translation look-aside buffers (TLBs)

- When the frame number is obtained, we can use it to access memory
- 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, the operating system must select one for replacement.
- Replacement policies range from least recently used (LRU) to random.
- Furthermore, some TLBs allow certain entries to be meaning that they cannot be removed from the TLB. Typically, TLB entries for kernel code are wired down

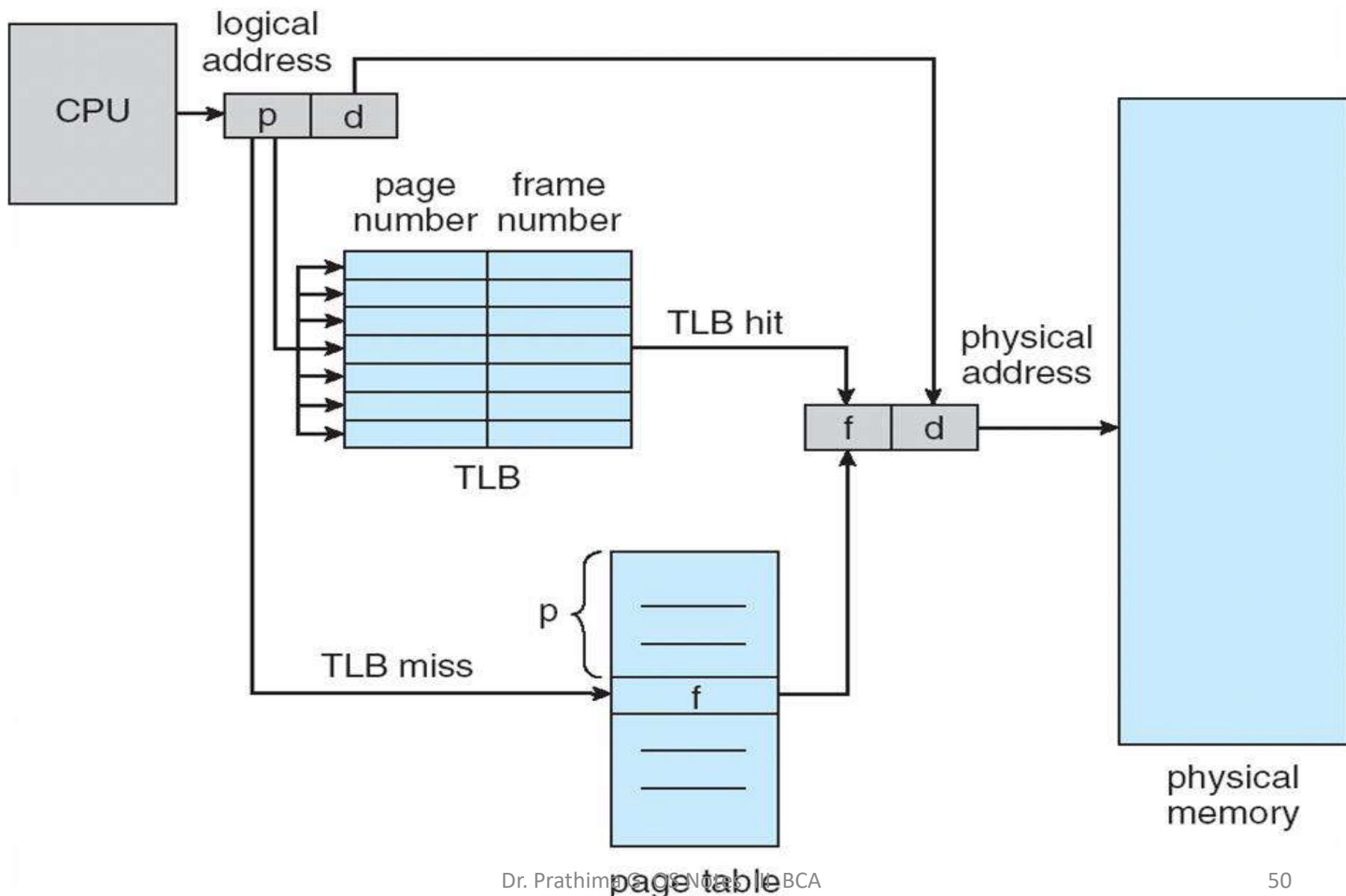
Associative Memory

- Associative memory – parallel search

Page #	Frame #

- Address translation (p, d)
 - If p is in associative register, get frame # out
 - Otherwise get frame # from page table in memory

Paging Hardware With TLB



Effective Access Time

- The percentage of times that a particular page number is found in the TLB is called the 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 20 nanoseconds to search the TLB and 100 nanoseconds to access memory,
- then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB.
- If we fail to find the page number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds) and
- then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds.

Effective Access Time

- To find the effective we weight the case by its probability:
- effective access time = $0.80 \times 120 + 0.20 \times 220 = 140$ nanoseconds.
- In this example, we suffer a 40-percent slowdown in memory-access time (from 100 to 140 nanoseconds).
- For a 98-percent hit ratio, we have
- effective access time = $0.98 \times 120 + 0.02 \times 220 = 122$ nanoseconds.
- This increased hit rate produces only a 22 percent slowdown in access time.

Memory Protection

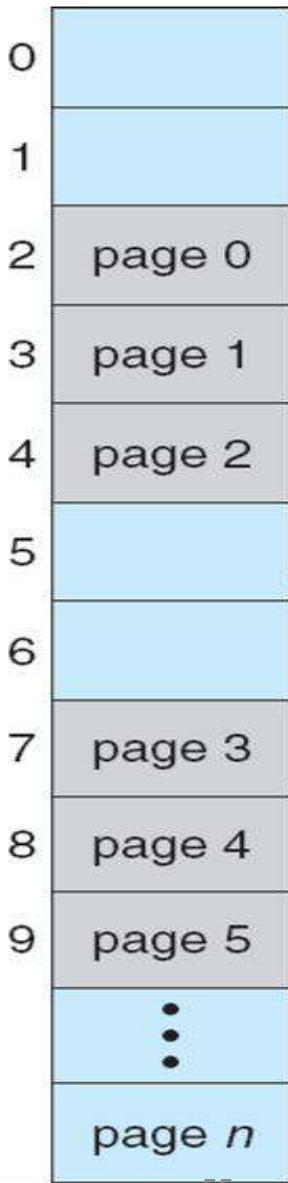
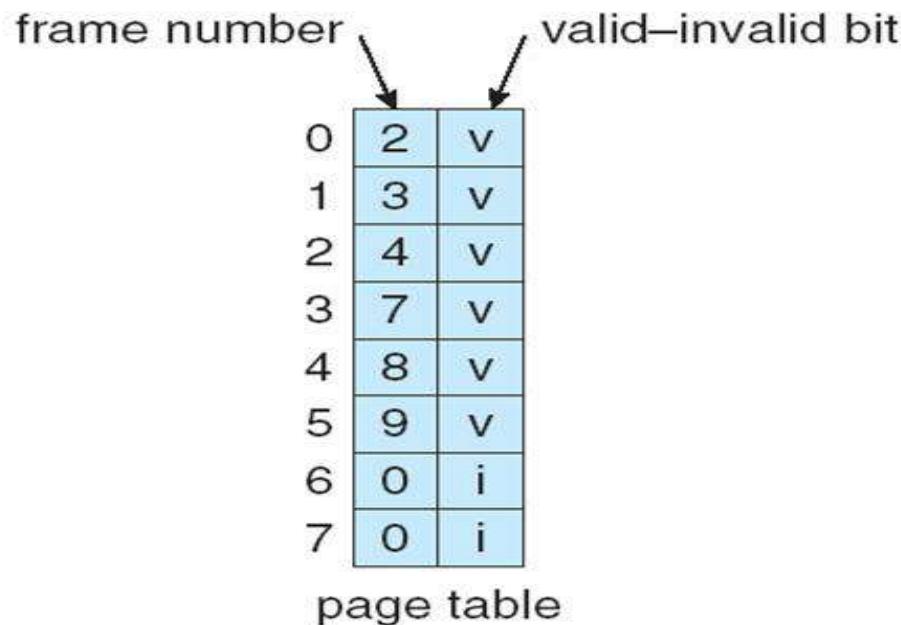
- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
 - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
 - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
 - “invalid” indicates that the page is not in the process’ logical address space
 - Or use PTLR
- Any violations result in a trap to the kernel

Example

- 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, 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 valid -invalid bit is set to invalid, and
- the computer will trap to flee operating system (invalid page reference).

Valid (v) or Invalid (i) Bit In A Page Table

00000	page 0
	page 1
	page 2
	page 3
	page 4
10,468	page 5
12,287	



Hardware Protection

- Some systems provide hardware, in the form of a page table length register(PTLR) length to indicate the size of the page table.
- 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

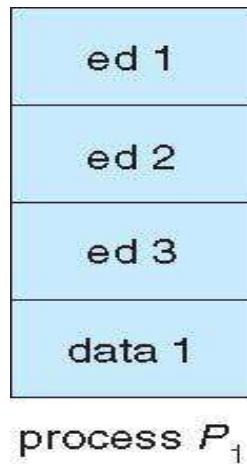
Shared Pages

- An advantage of paging is the possibility of sharing common code.
- This consideration is particularly important in a time-sharing environment.
- Consider a system that supports 40 users, each of whom executes a text editor.
- If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users.

Shared Pages

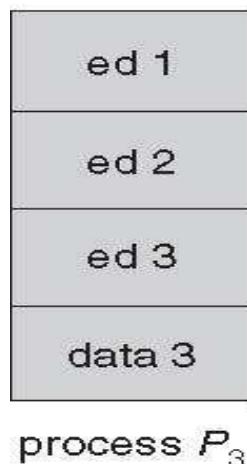
- If the code is reentrant (or pure code) however, it can be shared, as shown in Figure.
- Here we see a three-page editor-each page 50 KB in size being shared among three processes.
- Each process has its own data page.
- Reentrant code is non-self-modifying code: it never changes during execution.
- 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.

Shared Pages Example



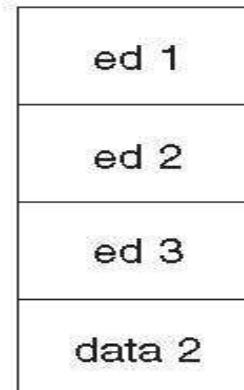
3
4
6
1

page table
for P_1



3
4
6
2

page table
for P_3



process P_2

3
4
6
7

page table
for P_2

0
1
2
3
4
5
6
7
8
9
10
11

Shared Pages

- Only one copy of the editor need be kept in physical memory
- Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.
- Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user.
- The total space required is now 2150 KB instead of 8,000 KB-a significant savings.
- Other heavily used programs can also be shared -compilers, window systems, run-time libraries, database systems, and so on.
- To be sharable, the code must be reentrant.

Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
 - Consider a 32-bit logical address space as on modern computers
 - Page size of 4 KB (2^{12})
 - Page table would have 1 million entries ($2^{32} / 2^{12}$)
 - If each entry is 4 bytes \rightarrow 4 MB of physical address space / memory for page table alone
 - That amount of memory used to cost a lot
 - Don't want to allocate that 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

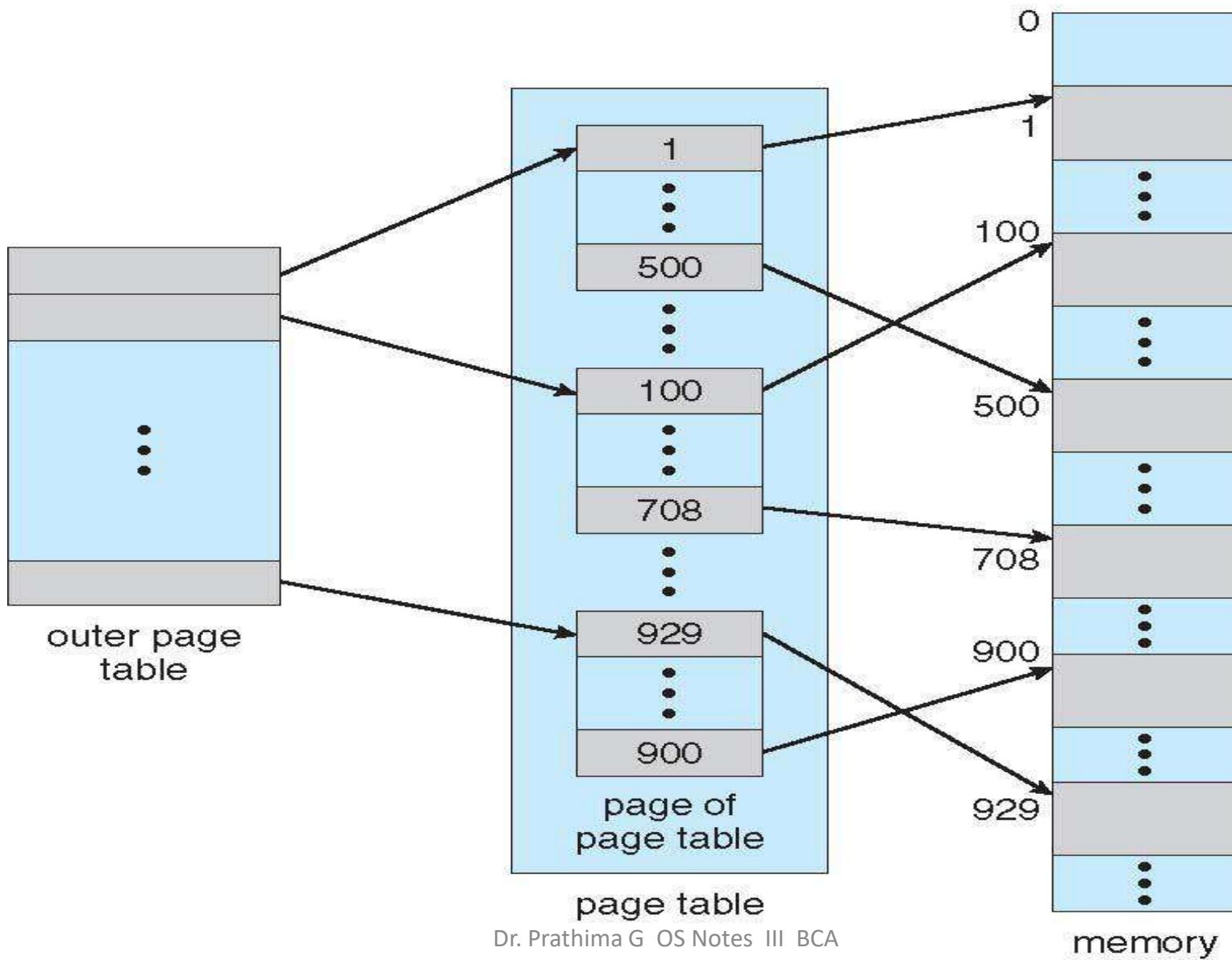
Structure of the Page Table

- Hierarchical Page Tables
- Hashed Page Tables
- Inverted Page Tables
- Hierarchical Paging-One way is to use a two-level paging algorithm,
- in which the page table itself is also paged .

Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

Two-Level Page-Table Scheme



Two-Level Paging Example

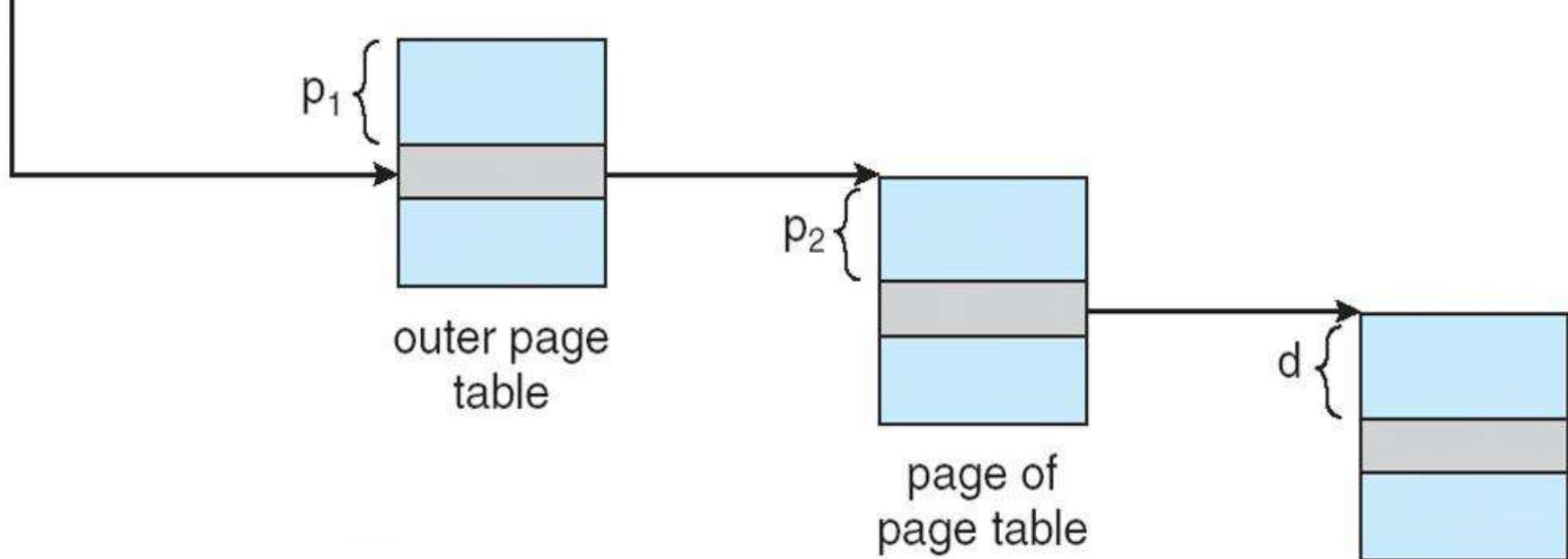
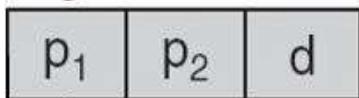
- A logical address (on 32-bit machine with 1K page size) is divided into:
 - a page number consisting of 22 bits
 - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
 - a 12-bit page number
 - a 10-bit page offset
- Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d

- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table
- Known as **forward-mapped page table**

Address-Translation Scheme

logical address



64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
 - Then page table has 2^{52} entries
 - If two level scheme, inner page tables could be 2^{10} 4-byte entries
 - Address would look like

outer page	inner page	page offset
p_1	p_2	d
42	10	12

- Outer page table has 2^{42} entries or 2^{44} bytes
- One solution is to add a 2nd outer page table
- But in the following example the 2nd outer page table is still 2^{34} bytes in size
 - And possibly 4 memory access to get to one physical memory location

Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

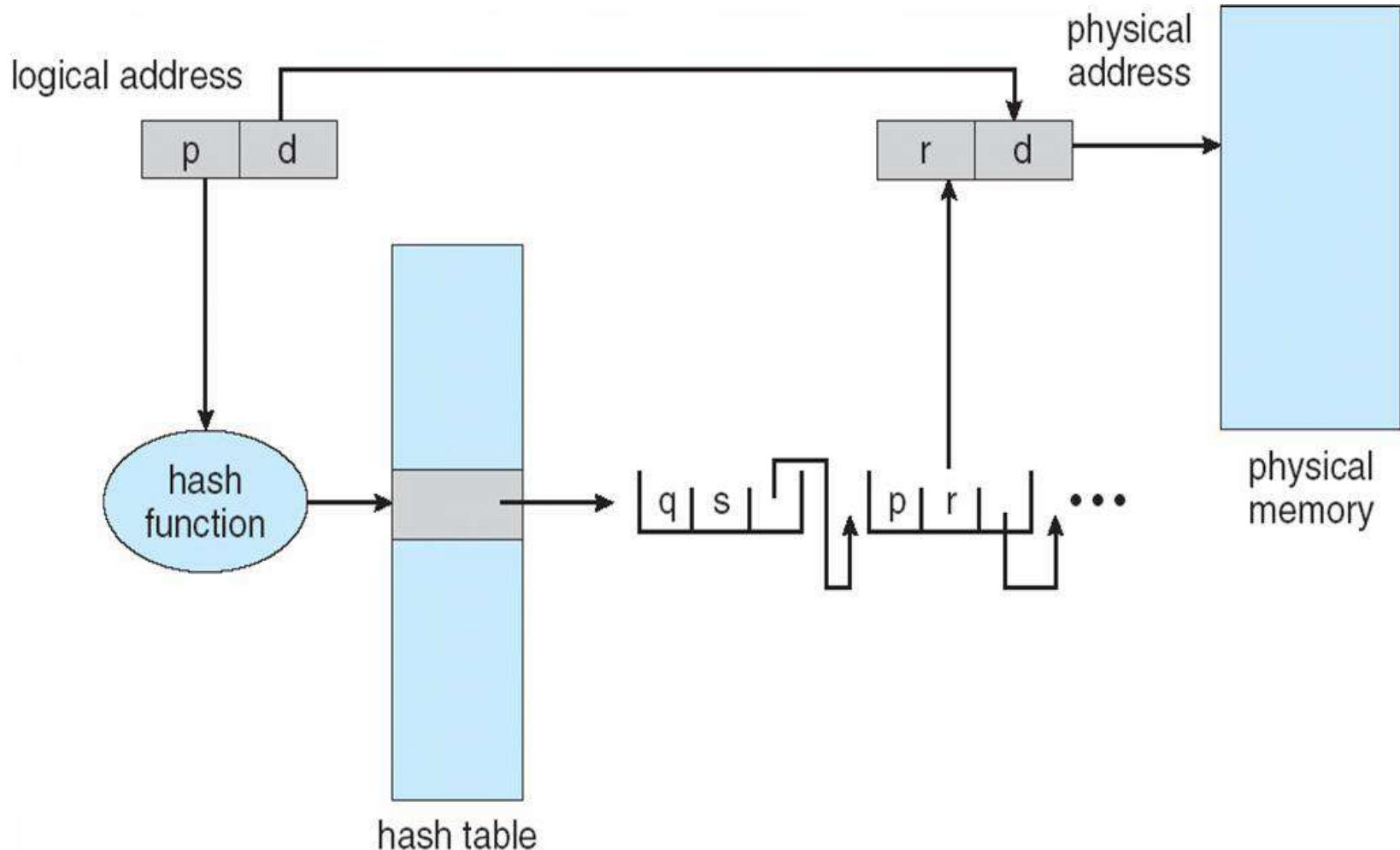
Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
 - This page table contains a chain of elements hashing to the same location
 - Each entry in the hash table contains a linked list of elements that hash to the same location

Hashed Page Tables

- Each element contains
 - (1) the virtual page number
 - (2) the value of the mapped page frame
 - (3) a pointer to the next element in the linked list

Hashed Page Table



Working

- 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

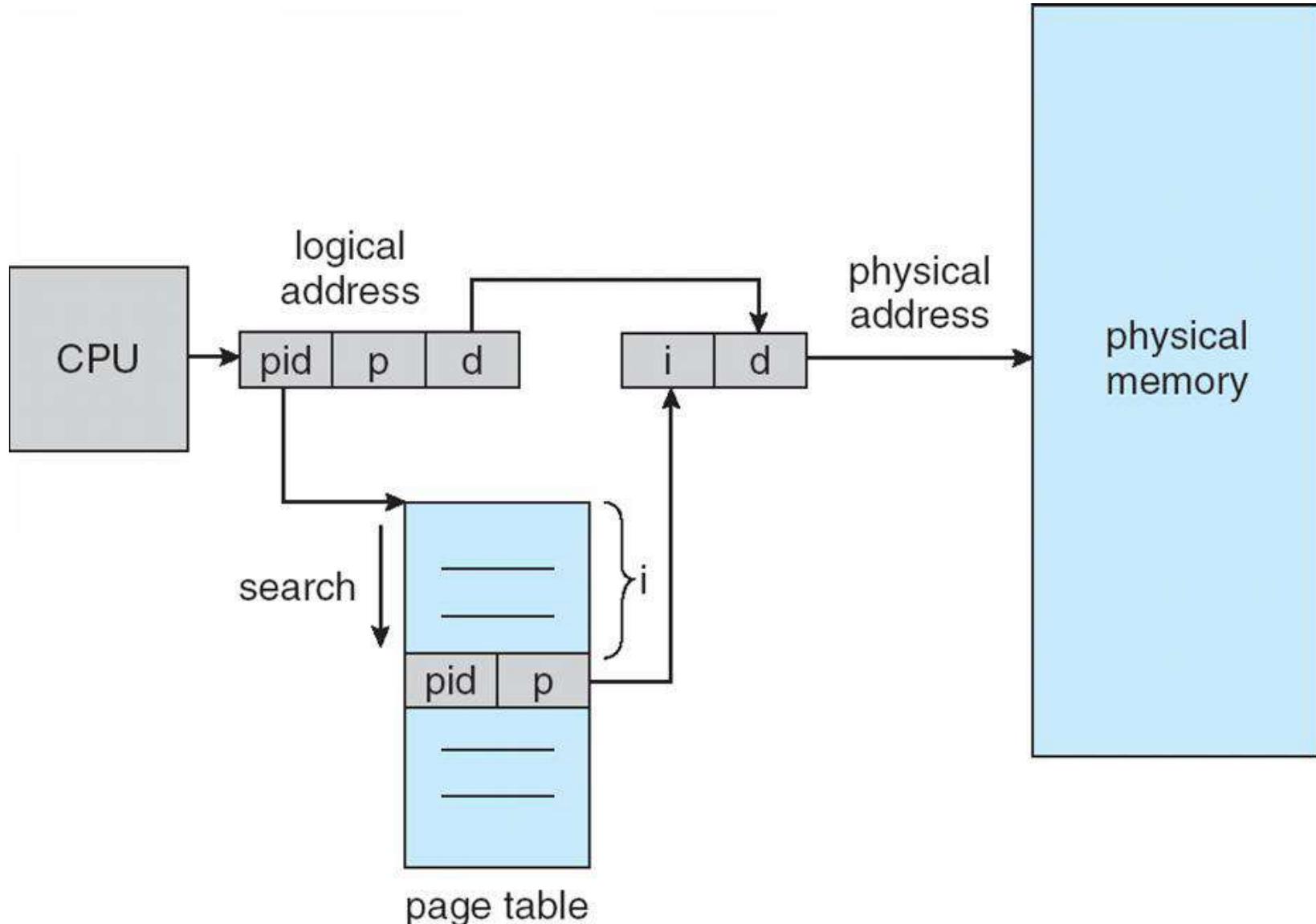
Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- 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

Inverted Page Table

- Each virtual address in the system consists of a triple:
- <process-id, page-number, offset>.
- Each inverted page-table entry is a pair <process-id, page-number>
- where the process-id assumes the role of the address-space identifier

Inverted Page Table Architecture



Inverted Page Table

- When a memory reference occurs,
- part of the virtual address, consisting of <process-id, pagenumber>, 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 <i, offset> is generated.
- If no match is found, then an illegal address access has been attempted.

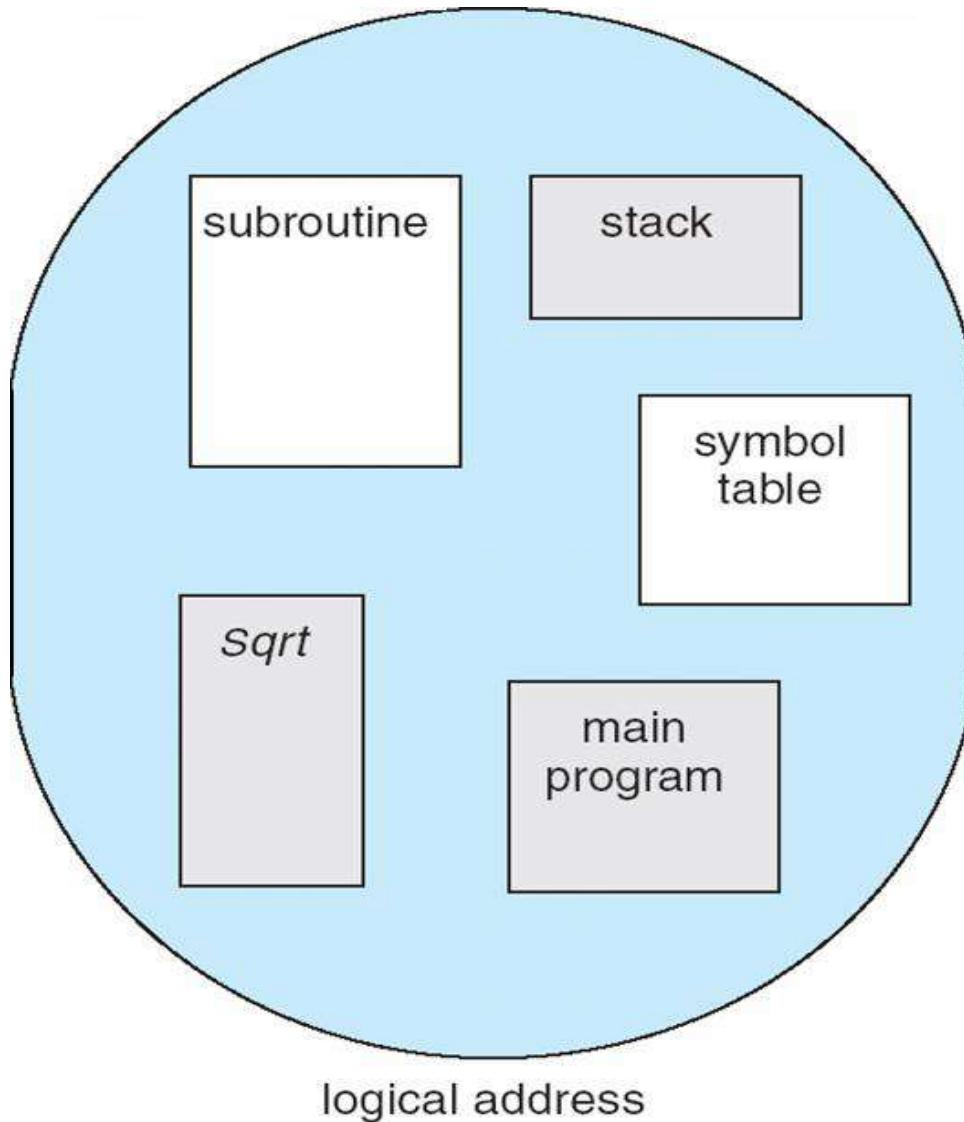
Inverted Page Table

- 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 for a match.
- This search would take far too long.
- To alleviate this problem, we use a hash table

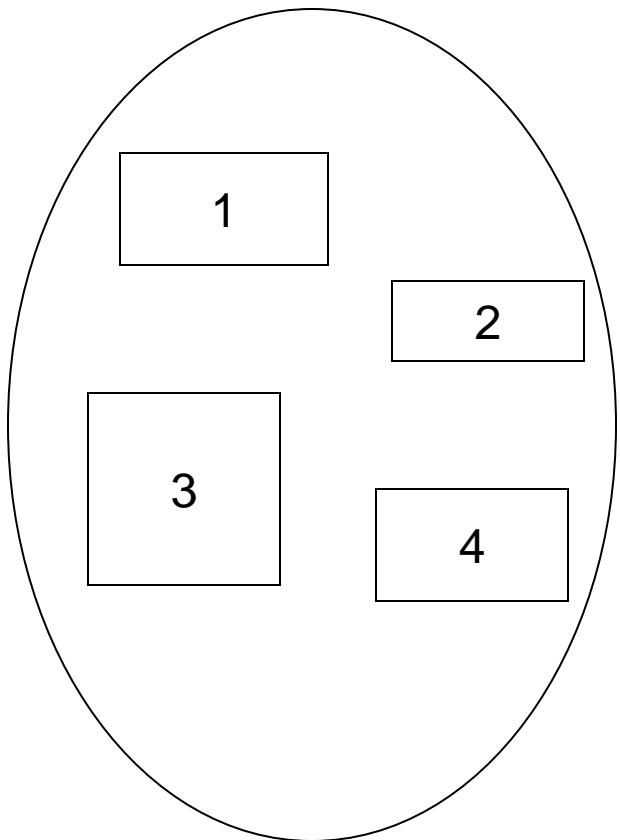
Segmentation

- **Memory-management scheme that supports user view of memory**
- A program is a collection of segments
 - A segment is a logical unit such as:
 - main program
 - procedure
 - function
 - method
 - object
 - local variables, global variables
 - common block
 - stack
 - symbol table
 - arrays

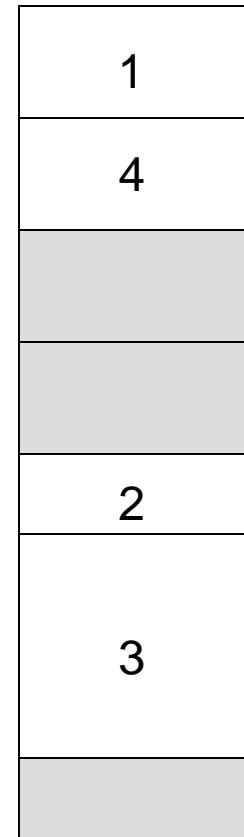
User's View of a Program



Logical View of Segmentation



user space

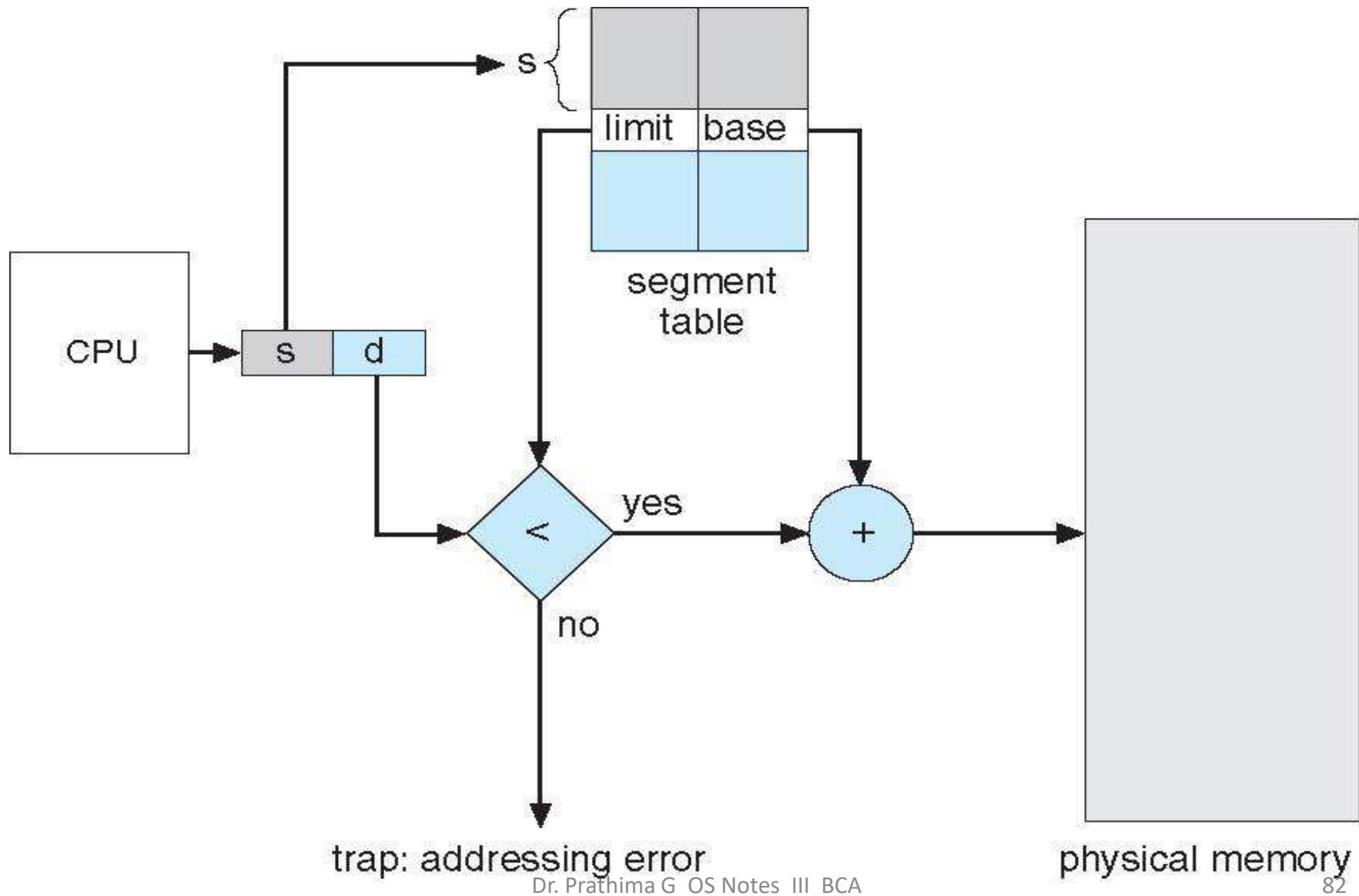


physical memory space

Segmentation Architecture

- Logical address consists of a two tuple:
 $\langle \text{segment-number}, \text{offset} \rangle,$
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
 - **base** – contains the starting physical address where the segments reside in memory
 - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;
segment number **s** is legal if **s < STLR**

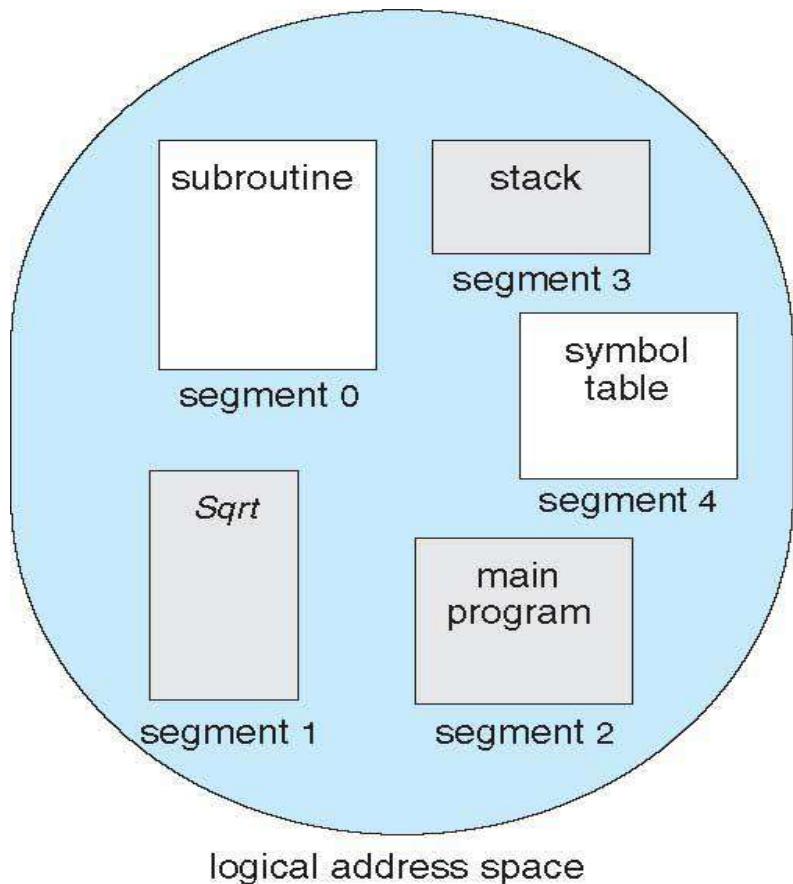
Segmentation Hardware



Segmentation Architecture

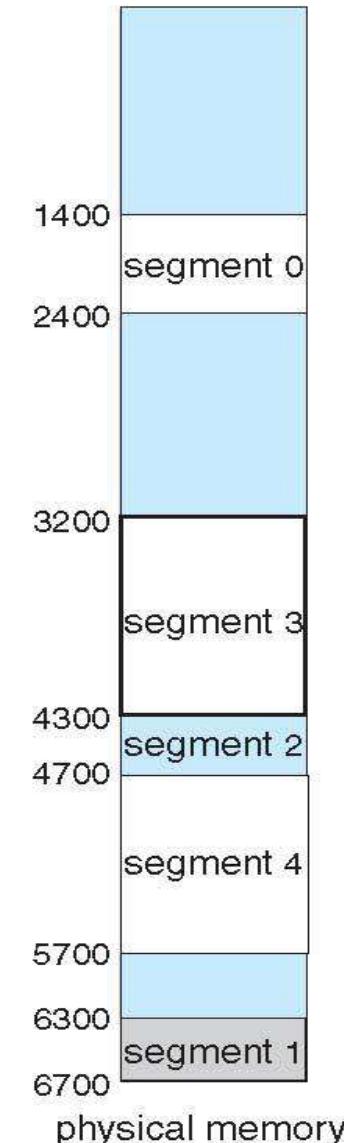
- Protection
 - With each entry in segment table associate:
 - validation bit = 0 \Rightarrow illegal segment
 - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Example of Segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

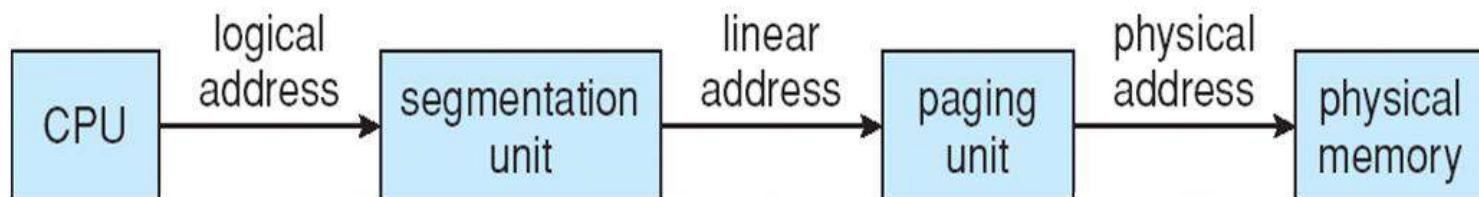


Example: The Intel Pentium

- Both paging and segmentation have advantages and disadvantages.
- In fact some architectures provide both.
- Intel Pentium architecture supports both pure segmentation and segmentation with paging

Example: The Intel Pentium

- In Pentium systems, 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).

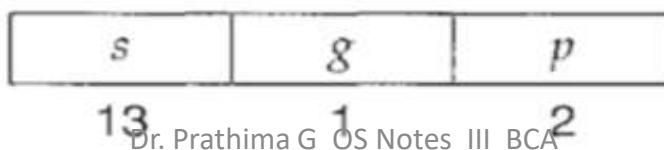


Pentium Segmentation

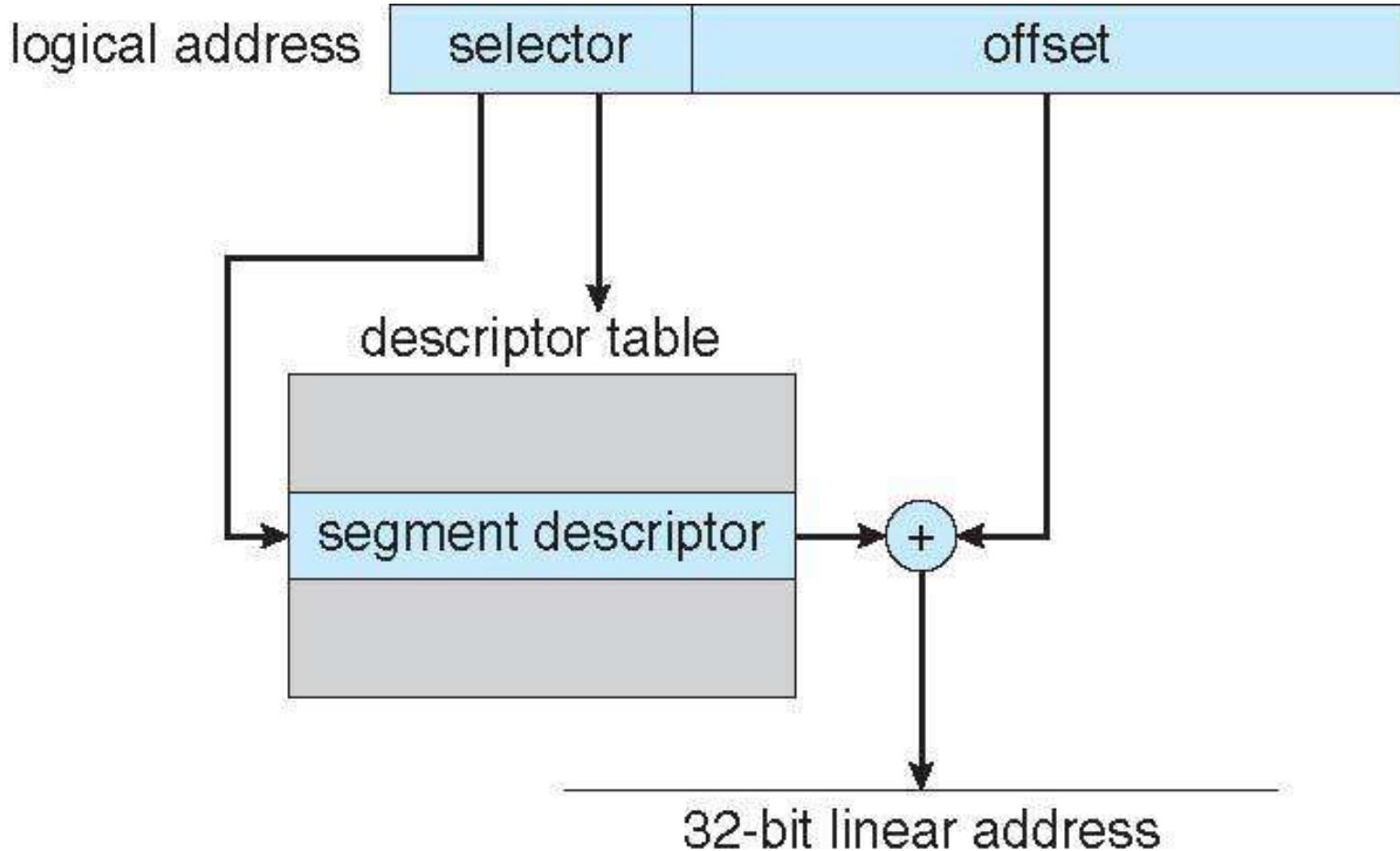
- Each segment can be 4 GB
- Up to 16 K segments per process
- Divided into two partitions
 - First partition of up to 8 K segments are private to process (kept in **local descriptor table LDT**)
 - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table GDT**)

Pentium Segmentation

- The logical address is a pair of (selector, offset) where the selector of 16 bits
- In which 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 (or word) within the segment



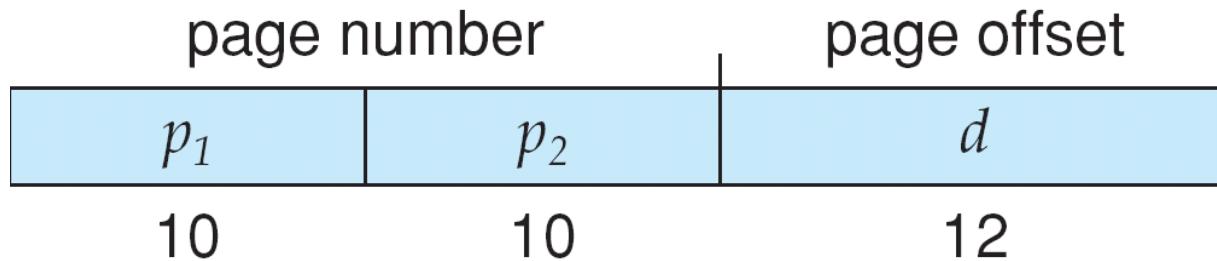
Intel Pentium Segmentation



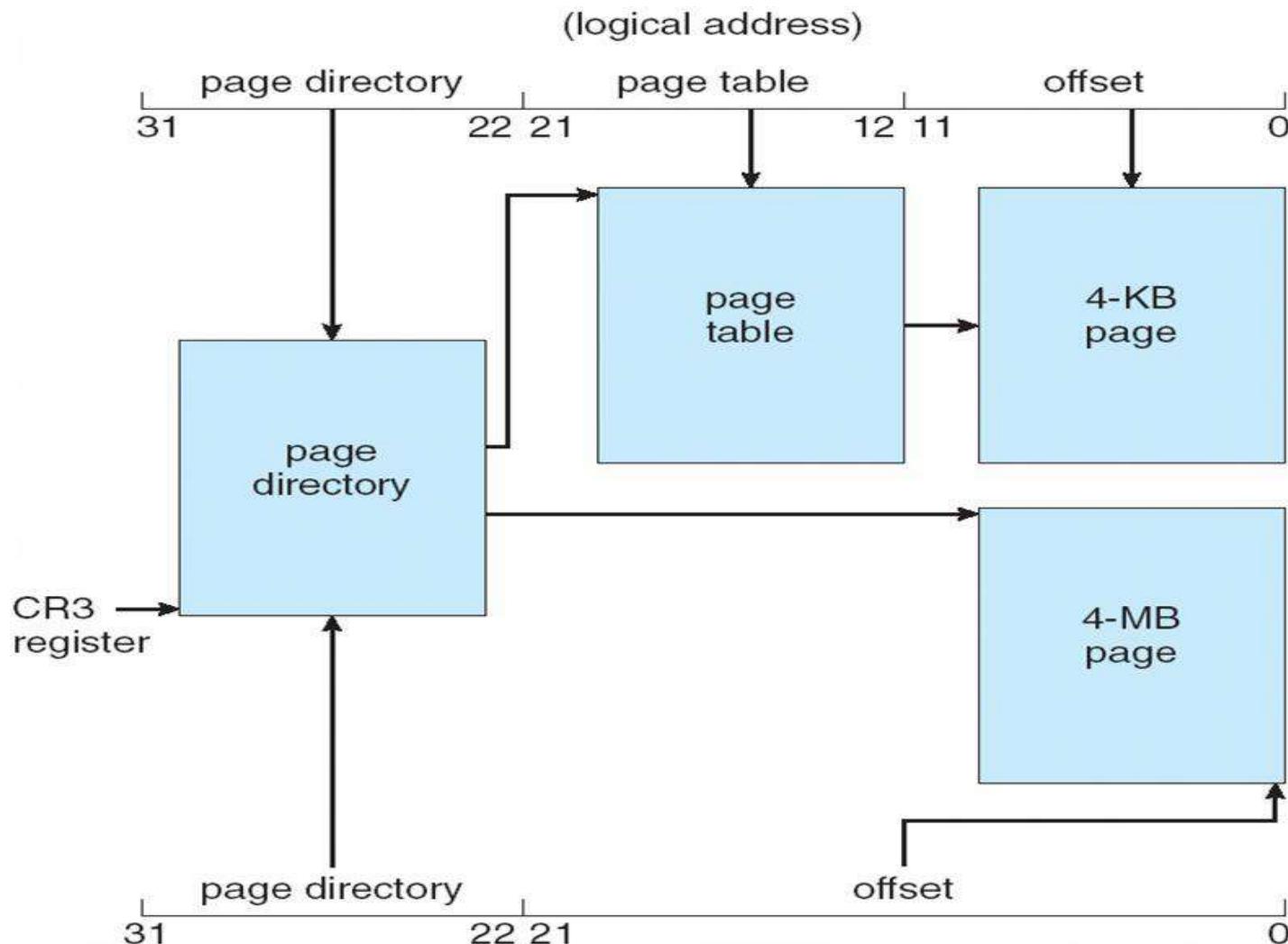
Pentium Paging

The Pentium architecture allows a page size of either 4 KB or 4 MB.

For 4-KB pages, the Pentium uses a two-level paging scheme in which the division of the 32-bit linear address is as follows:



Pentium Paging Architecture

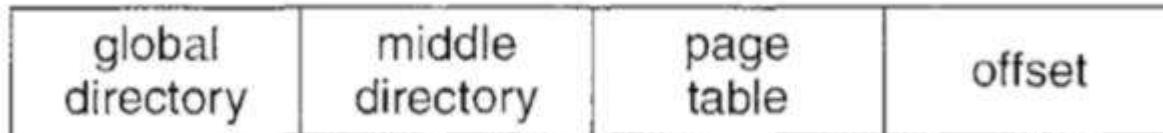


Intel Pentium address translation Mechanism

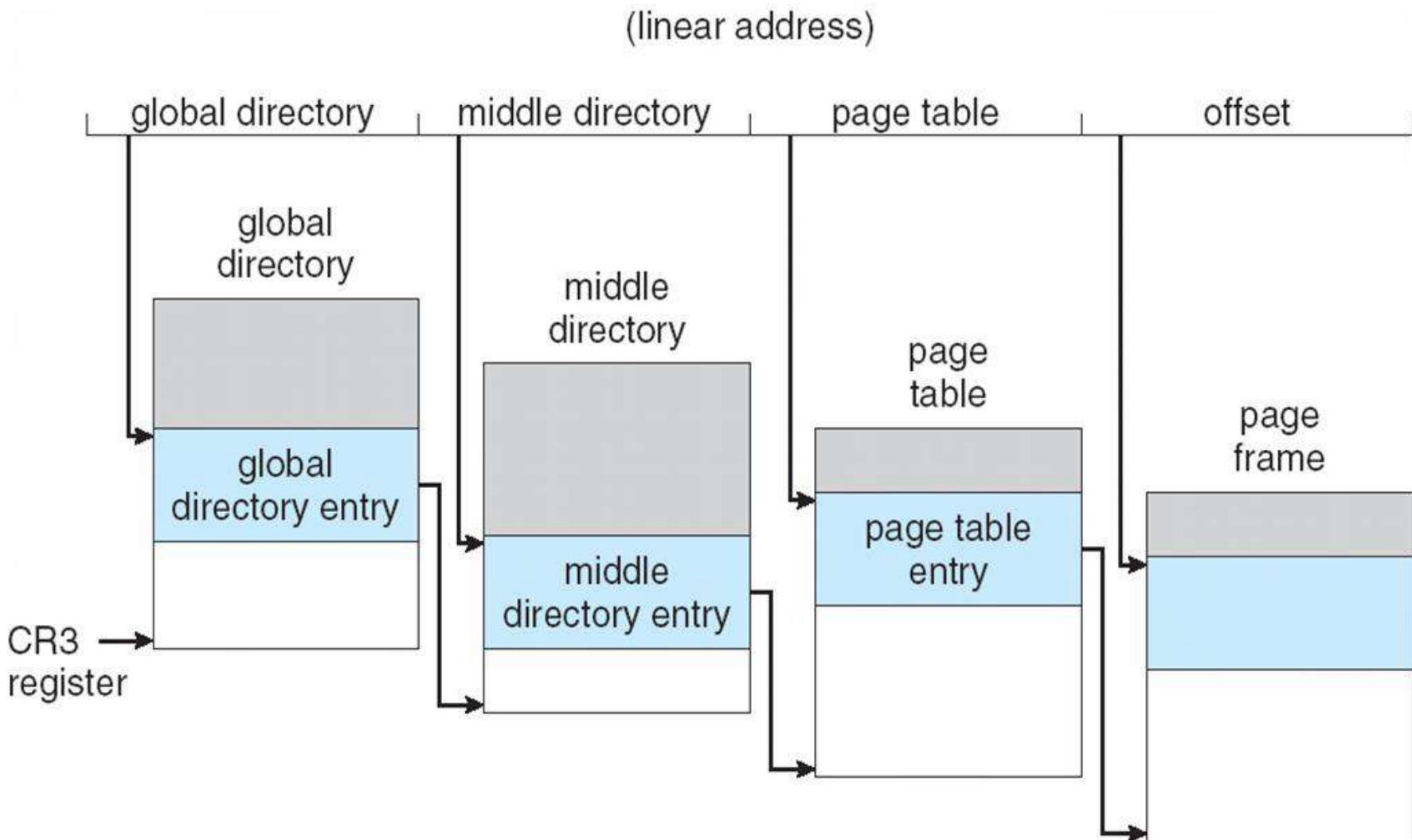
- The 10 high-order bits reference an entry in the outer most page table, which the Pentium terms the page directory.
- The page directory entry points to an inner page table that is indexed by the contents of the innermost 10 bits in the linear address.
- Finally, the low-order bits 0-11 refer to the offset in the 4-KB page pointed to in the page table.

Linux on Pentium Machine

- On the Pentium, Linux uses only six segments:
- A segment for kernel code
- A segment for kernel data
- A segment for user code
- A segment for user data
- A task-state segment (TSS)
- A default LDT segment
- The segments for user code and user data are shared by all processes running in user mode



Three-level Paging in Linux



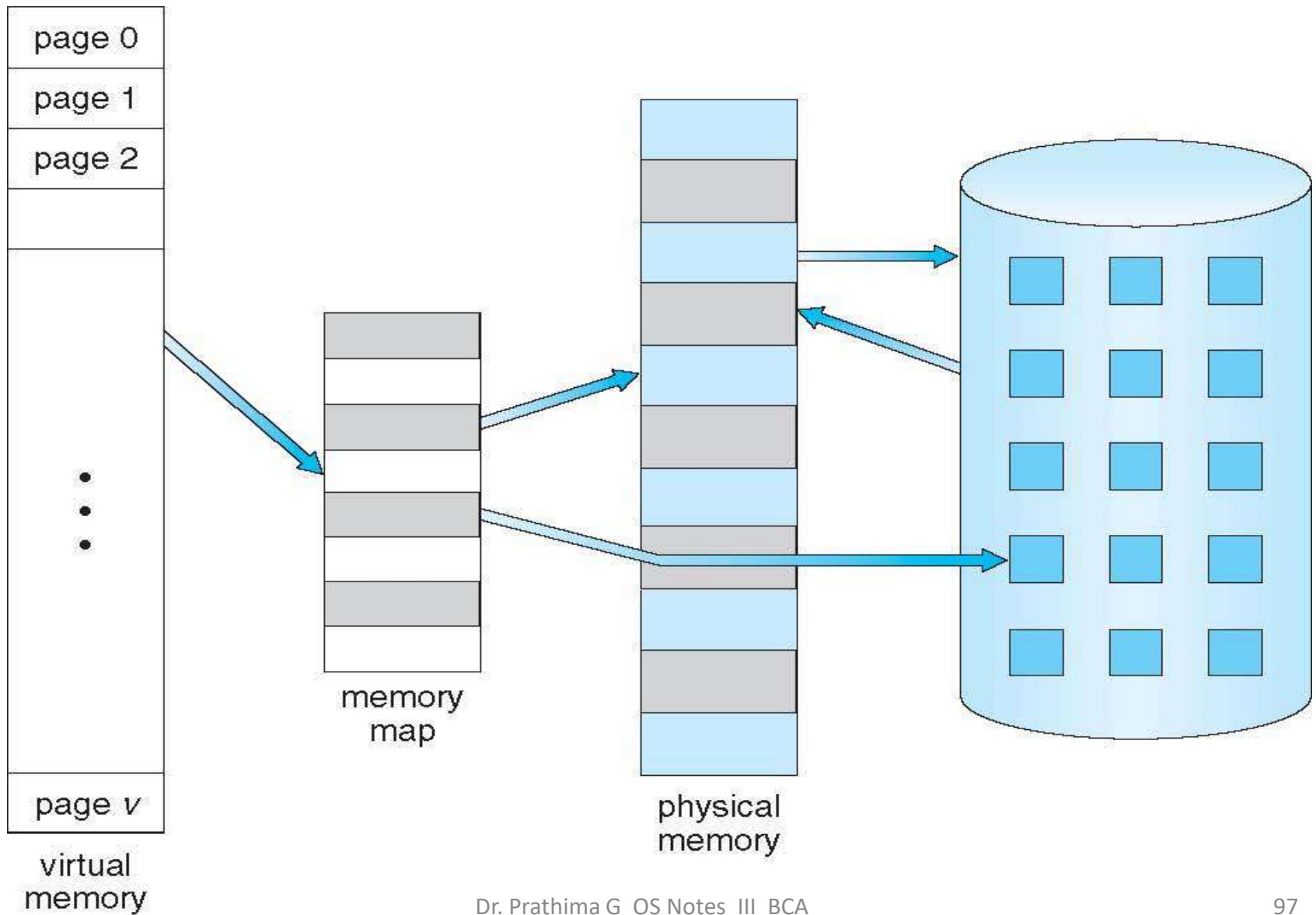
Virtual Memory

- Code needs to be in memory to execute, but entire program rarely used
 - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
 - Program no longer constrained by limits of physical memory
 - Program and programs could be larger than physical memory

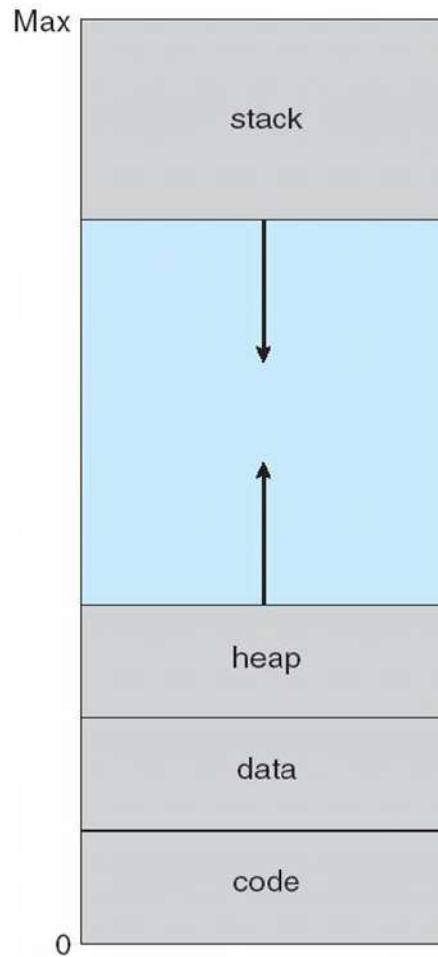
Virtual Memory

- **Virtual memory** – separation of user logical memory from physical memory
 - Only part of the program needs to be in memory for execution
 - Logical address space can therefore be much larger than physical address space
 - Allows address spaces to be shared by several processes
 - Allows for more efficient process creation
 - More programs running concurrently
 - Less I/O needed to load or swap processes
- Virtual memory can be implemented via:
 - Demand paging
 - Demand segmentation

Virtual Memory That is Larger Than Physical Memory



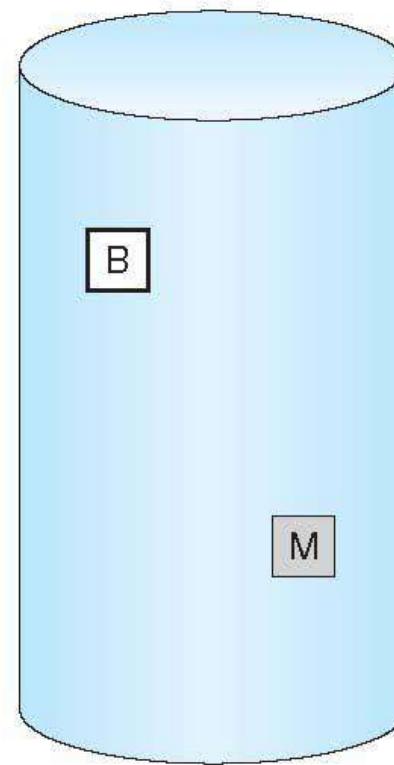
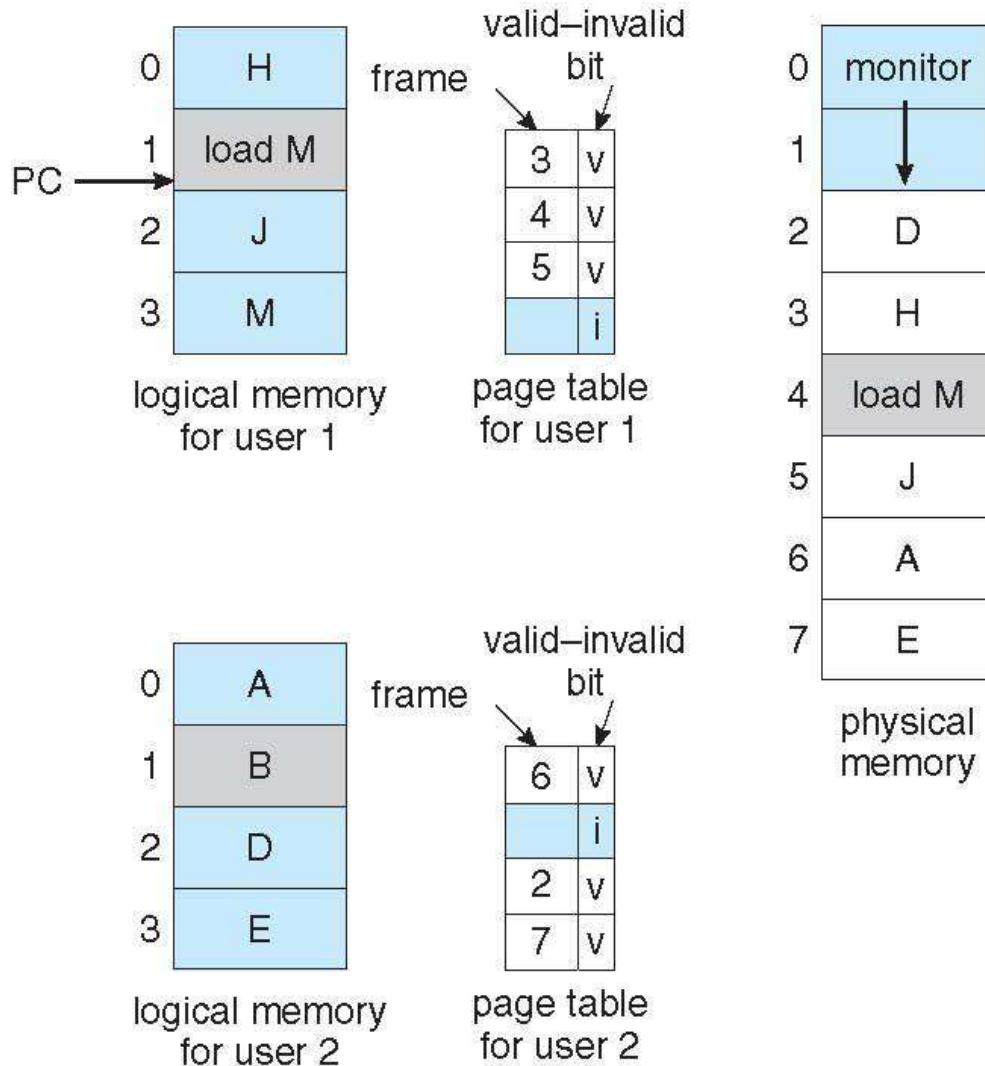
Virtual-address Space



Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Page replacement completes separation between logical memory and physical memory –
- large virtual memory can be provided on a smaller physical memory

Need For Page Replacement

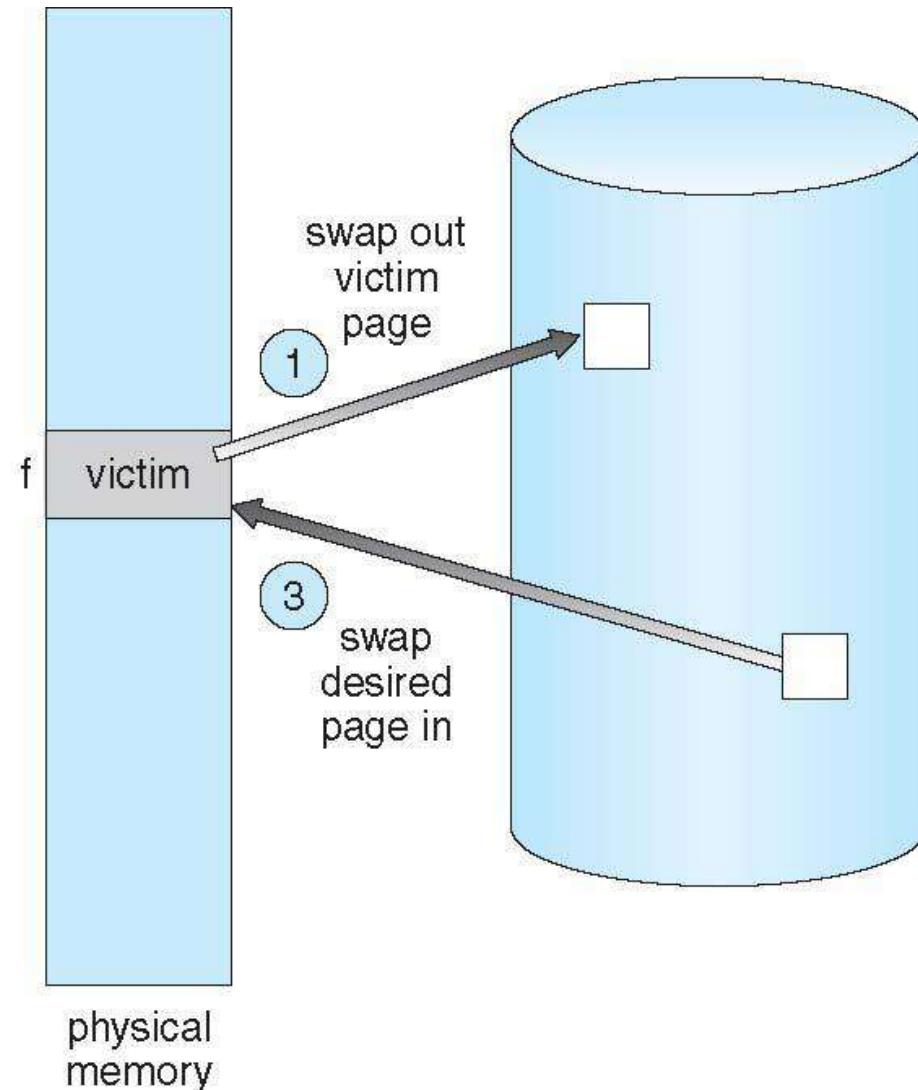
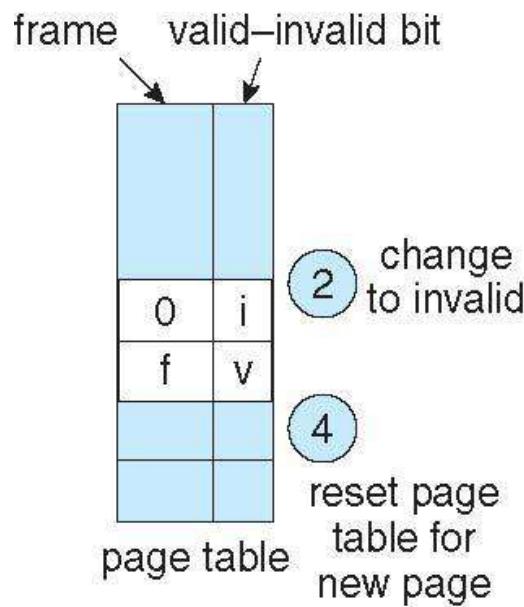


Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
 - If there is a free frame, use it
 - If there is no free frame, use a page replacement algorithm to select a **victim frame**
 - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing Effective Access Time

Page Replacement



Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
 - How many frames to give each process
 - Which frames to replace
- **Page-replacement algorithm**
 - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
 - String is just page numbers, not full addresses
 - Repeated access to the same page does not cause a page fault
- In all our examples, the reference string is

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
 - How many frames to give each process
 - Which frames to replace
- **Page-replacement algorithm**
 - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
 - String is just page numbers, not full addresses
 - Repeated access to the same page does not cause a page fault
- In all our examples, the reference string is

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

First-In-First-Out (FIFO) Algorithm

- Reference string:
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)

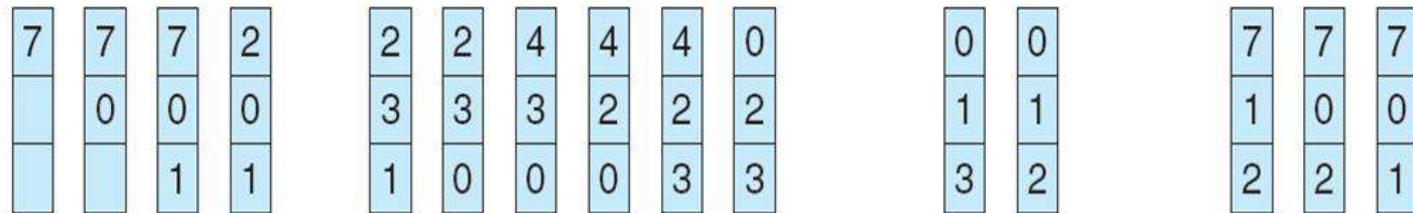
1	7	2	4	0	7	
2	0	3	2	1	0	15 page faults
3	1	0	3	2	1	

- Can vary by reference string: consider
1,2,3,4,1,2,5,1,2,3,4,5
 - Adding more frames can cause more page faults!
 - **Belady's Anomaly**
- How to track ages of pages?
 - Just use a FIFO queue

FIFO Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

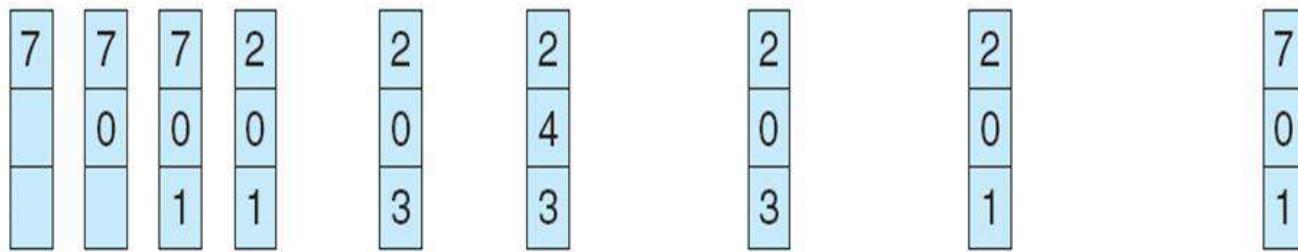
Optimal Algorithm

- Replace page that will not be used for longest period of time
 - 9 is optimal for the example on the next slide
- How do you know this?
 - Can't read the future
- Used for measuring how well your algorithm performs

Optimal Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

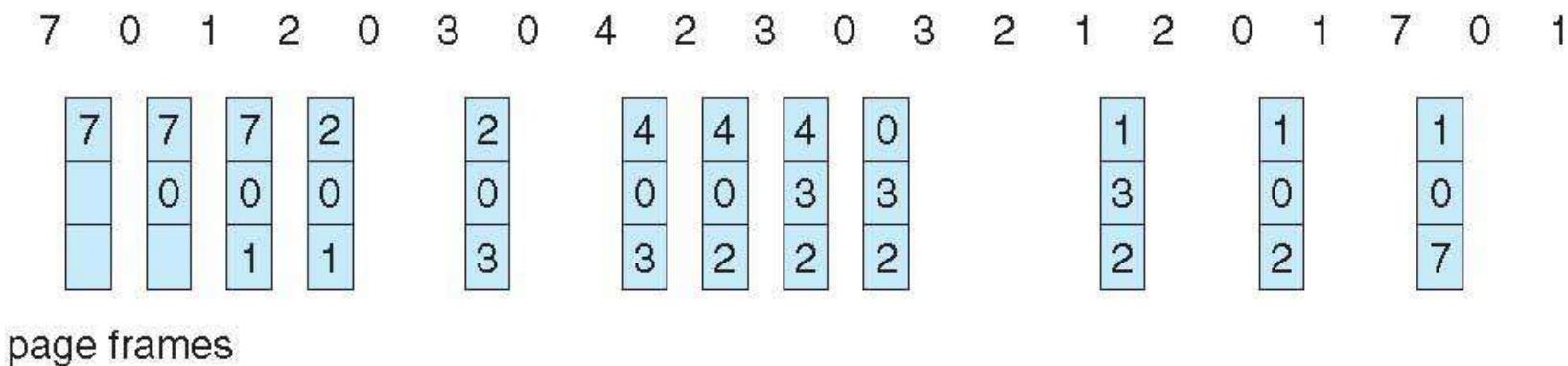


page frames

No of page faults = 9

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page
reference string



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used

Page Replacement Exercise

- How many page faults occur for FIFO, Optimal and LRU algorithms for the following reference string with four page frames?
- 1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.

Page Replacement Exercise

- Consider the following page reference string:
- 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.
- How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, and seven frames?
Remember that all frames are initially empty, so your first unique pages will cost one fault each.
- LRU replacement
- FIFO replacement
- Optimal replacement

Virtual Memory (Cont...)

Demand paging

- Consider how an executable program might be loaded from disk into memory.
- **One option is to load the entire program in physical memory at program execution time.**
- However, a problem with this approach is that we may not initially need the entire program in memory.
- Suppose a program starts with a list of available options from which the user is to select.
- Loading the entire program into memory results in loading the executable code for all options, regardless of whether an option is ultimately selected by the user or not.

Demand paging

- An alternative strategy is to load pages only as they are needed.
- This technique is known as **demand paging** and is commonly used in virtual memory systems.
- With **demand-paged virtual memory**, pages are only loaded when they are demanded during program execution;
- **pages that are never accessed are thus never loaded into physical memory.**

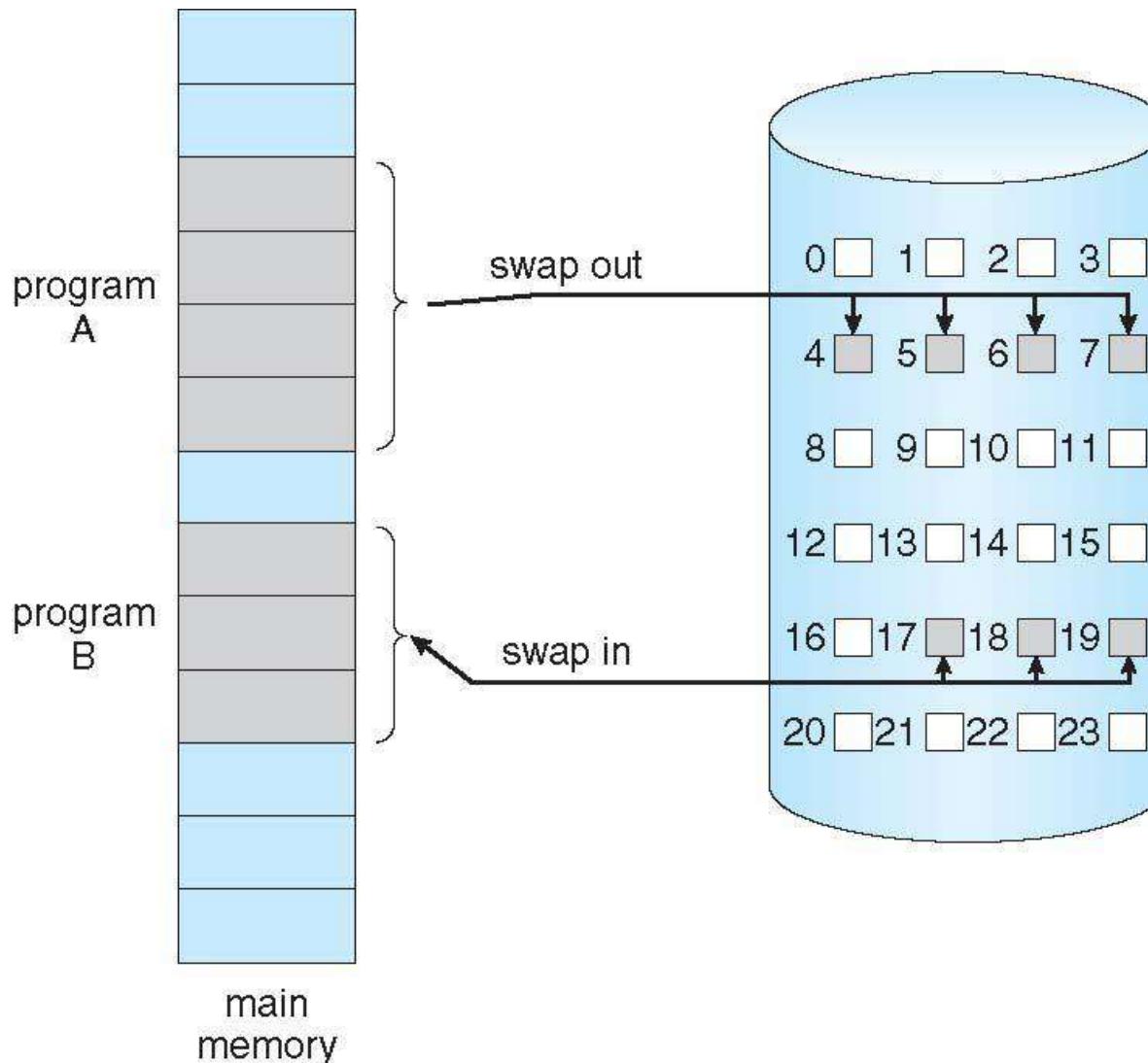
Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
 - Less I/O needed, no unnecessary I/O
 - Less memory needed
 - Faster response
 - More users
- **Lazy swapper** – never swaps a page into memory unless page will be needed
 - Swapper that deals with pages is a **pager**
 - A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory (usually a disk).
 - When we want to execute a process, we swap it into memory.
 - Rather than swapping the entire process into memory, however, we use a **A lazy swapper** never swaps a page into memory unless that page will be needed

Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
 - Less I/O needed, no unnecessary I/O
 - Less memory needed
 - Faster response
 - More users
- **Lazy swapper** – never swaps a page into memory unless page will be needed
 - Swapper that deals with pages is a **pager**
 - A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory (usually a disk).
 - When we want to execute a process, we swap it into memory.
 - Rather than swapping the entire process into memory, use a lazy swapper never swaps a page into memory unless that page will be needed

Transfer of a Paged Memory to Contiguous Disk Space



Demand Paging

- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again.
- Instead of swapping in a whole process, the pager brings only those pages into memory.
- Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.
- With this scheme, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk.

Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

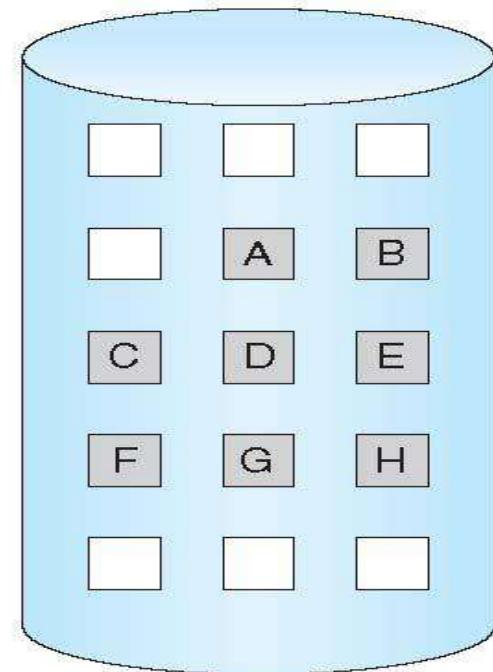
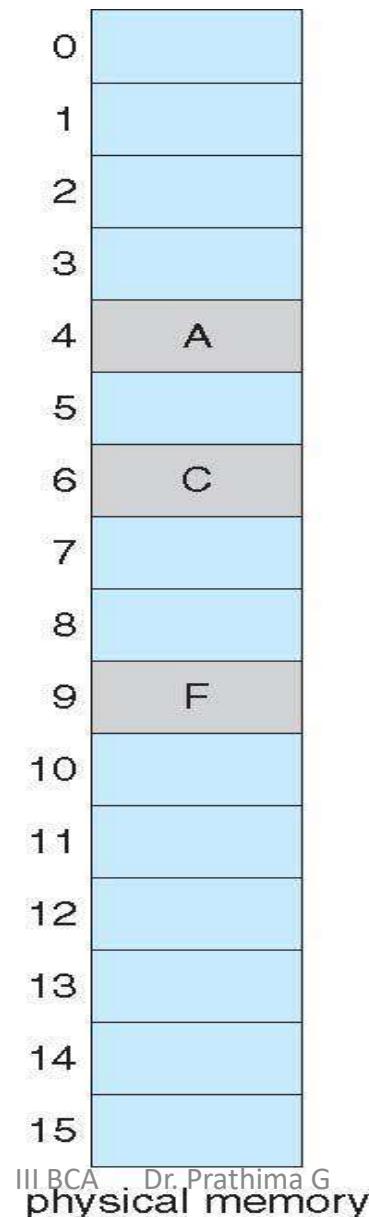
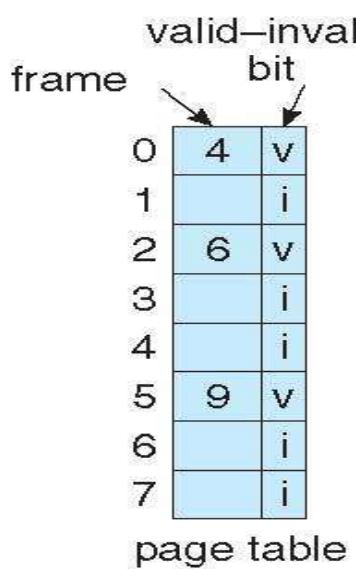
- | Frame # | valid-invalid bit |
|---------|-------------------|
| | v |
| | v |
| | v |
| | v |
| | i |
| | |
| | i |
| | i |

- During address translation, if valid–invalid bit in page table entry is **I** ⇒ page fault

Page Table When Some Pages Are Not in Main Memory

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

logical memory



Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

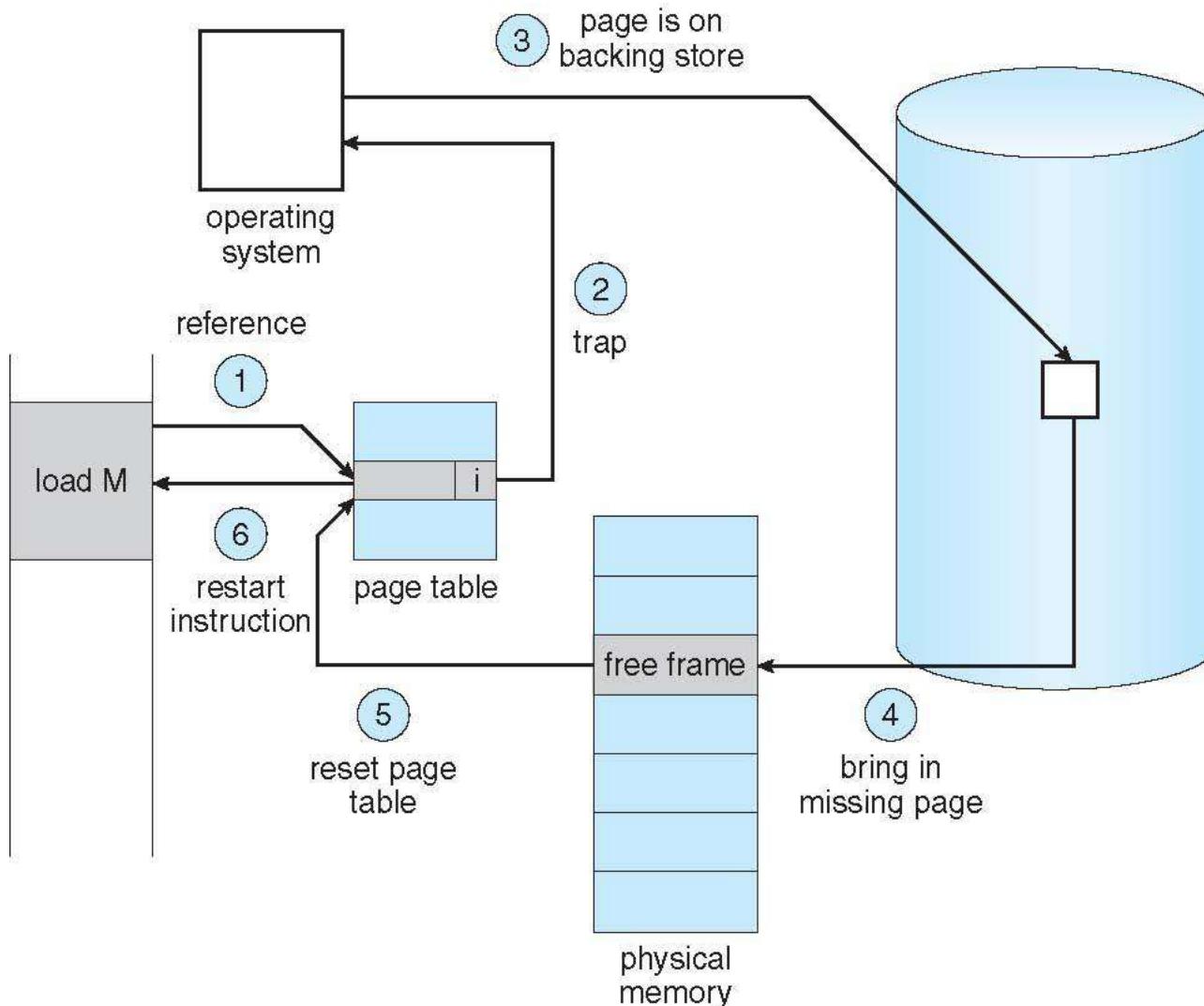
page fault

1. Operating system looks at another table to decide:
 - Invalid reference \Rightarrow abort
 - Just not in memory
2. Get empty frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
Set validation bit = **V**
5. Restart the instruction that caused the page fault

Aspects of Demand Paging

- **Extreme case** – start process with *no* pages in memory
 - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
 - And for every other process pages on first access
 - **Pure demand paging**
- **Actually, a given instruction could access multiple pages** -> multiple page faults
 - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
 - Page table with valid / invalid bit
 - Secondary memory (swap device with **swap space**)
 - Instruction restart

Steps in Handling a Page Fault



Performance of Demand Paging

- Stages in Demand Paging

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
 1. Wait in a queue for this device until the read request is serviced
 2. Wait for the device seek and/or latency time
 3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1$
 - if $p = 0$ no page faults
 - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)
$$\text{EAT} = (1 - p) \times \text{memory access} + p (\text{page fault overhead} + \text{swap page out} + \text{swap page in} + \text{restart overhead})$$

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $$\begin{aligned} \text{EAT} &= (1 - p) \times 200 + p (8 \text{ milliseconds}) \\ &= (1 - p) \times 200 + p \times 8,000,000 \\ &= 200 + p \times 7,999,800 \end{aligned}$$
- If one access out of 1,000 causes a page fault, then
 $\text{EAT} = 8.2 \text{ microseconds.}$
This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
 - $220 > 200 + 7,999,800 \times p$
 $20 > 7,999,800 \times p$
 - $p < .0000025$
 - < one page fault in every 400,000 memory accesses

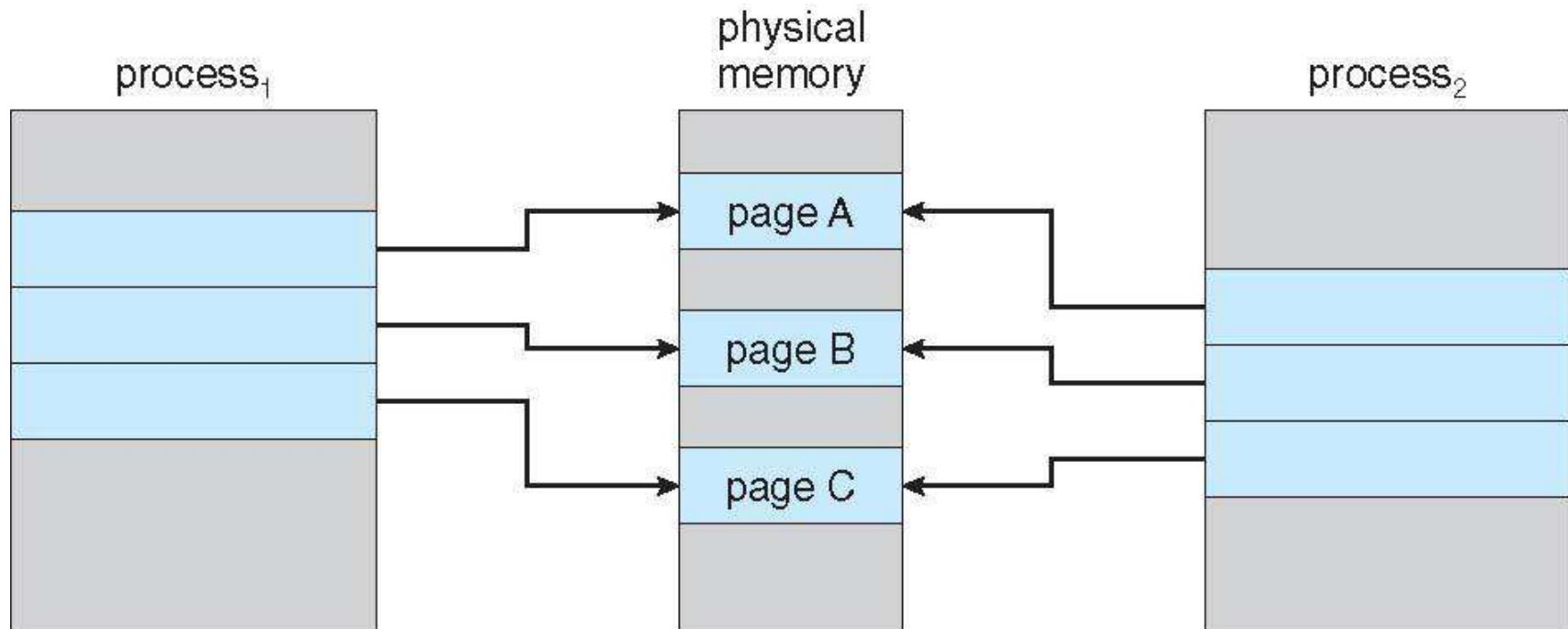
Demand Paging Optimizations

- Copy entire process image to swap space at process load time
 - Then page in and out of swap space
 - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
 - Used in Solaris and current BSD

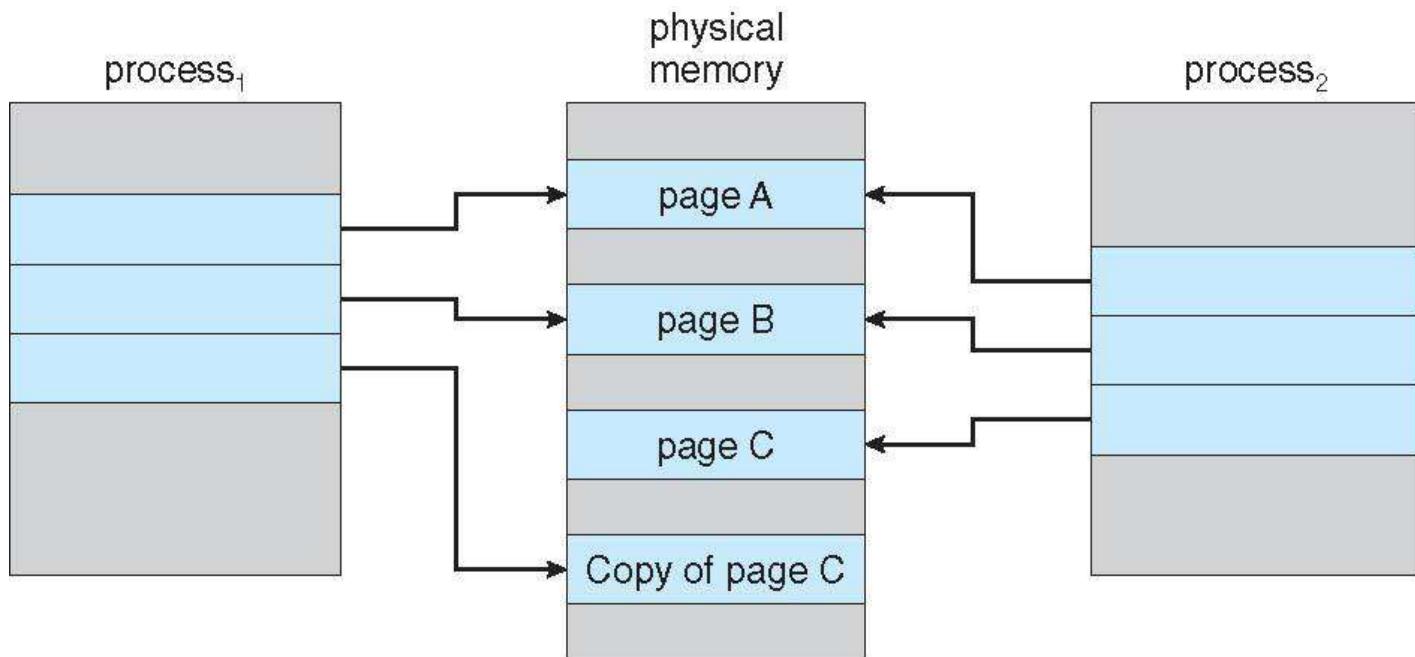
Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially *share* the same pages in memory
 - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
 - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
 - Designed to have child call `exec()`
 - Very efficient

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



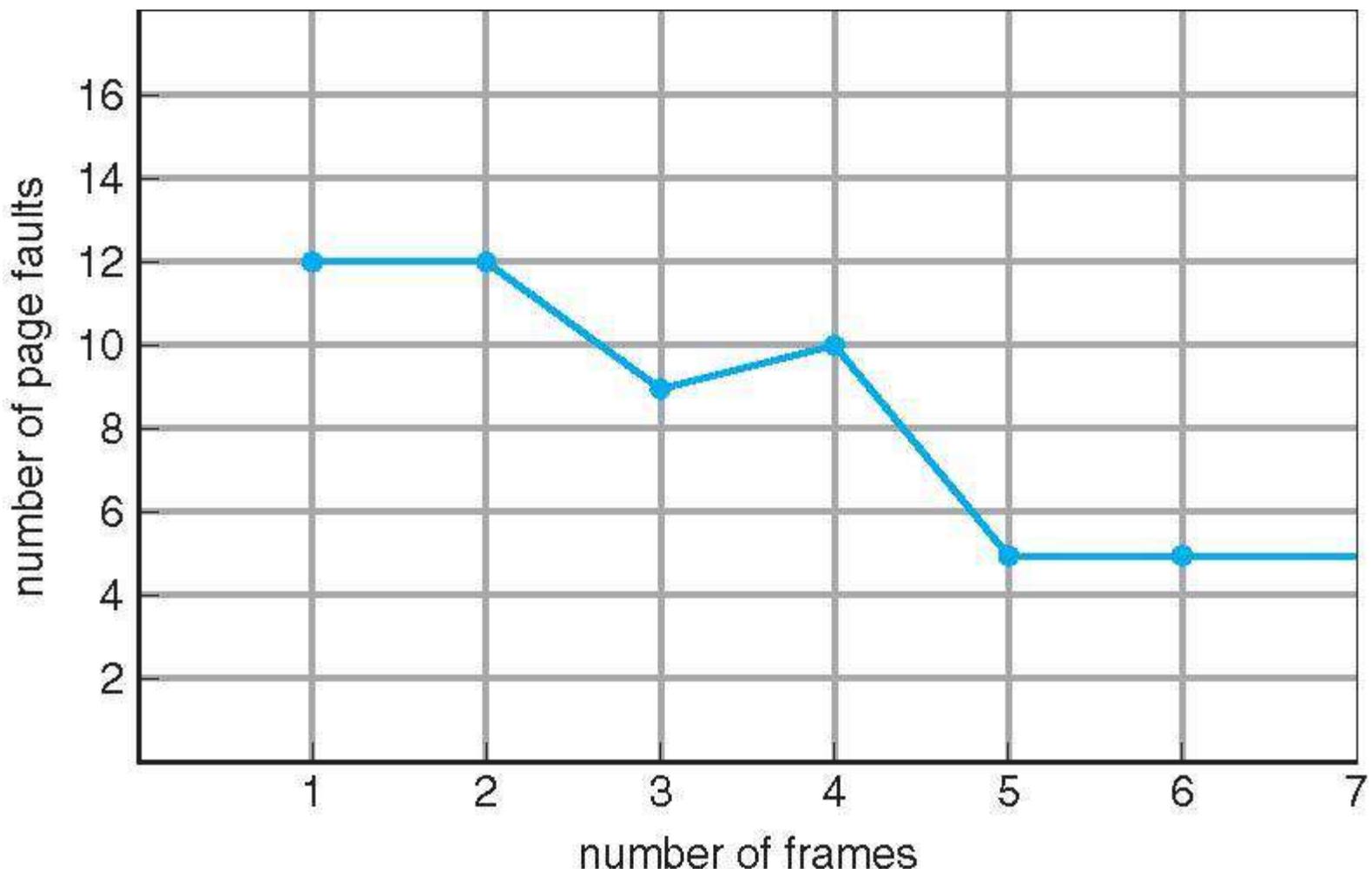
What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
 - Algorithm – terminate? swap out? replace the page?
 - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

FIFO Illustrating Belady's Anomaly



LRU Algorithm

- **Counter implementation**
 - Every page entry has a counter
 - every time page is referenced through this entry, copy the clock into the counter
 - When a page needs to be changed, look at the counters to find smallest value

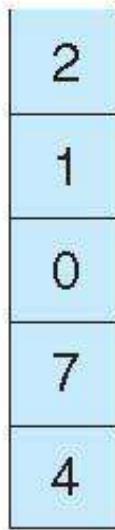
LRU Algorithm

- Stack implementation
 - Keep a stack of page numbers in a double link form:
 - Page referenced:
 - move it to the top
 - requires 6 pointers to be changed
 - But each update more expensive

Use Of A Stack to Record The Most Recent Page References

reference string

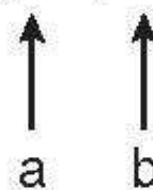
4 7 0 7 1 0 1 2 1 2 7 1 2



stack
before
a



stack
after
b



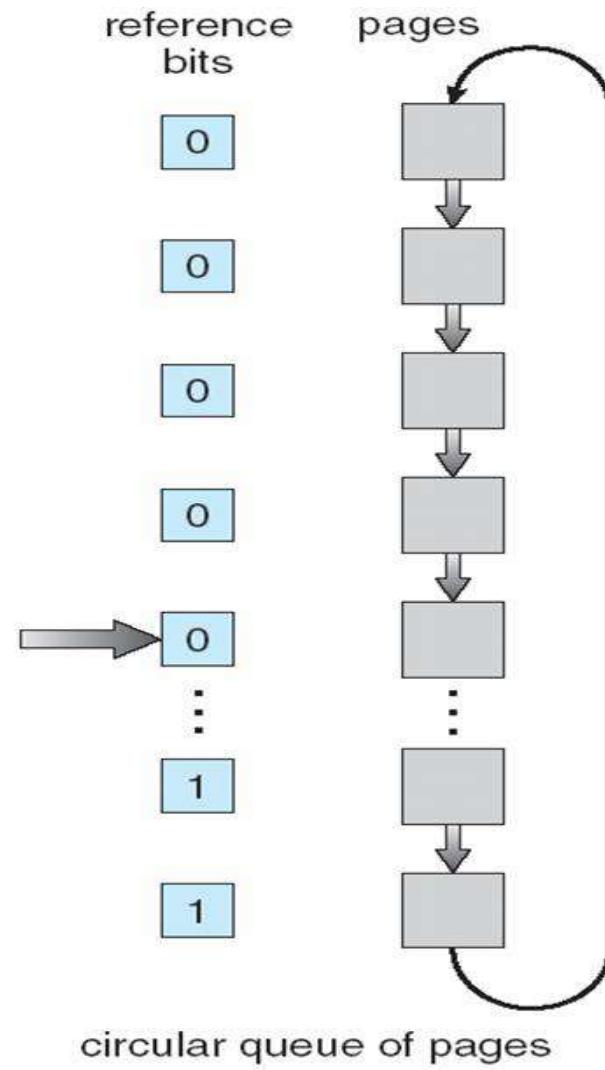
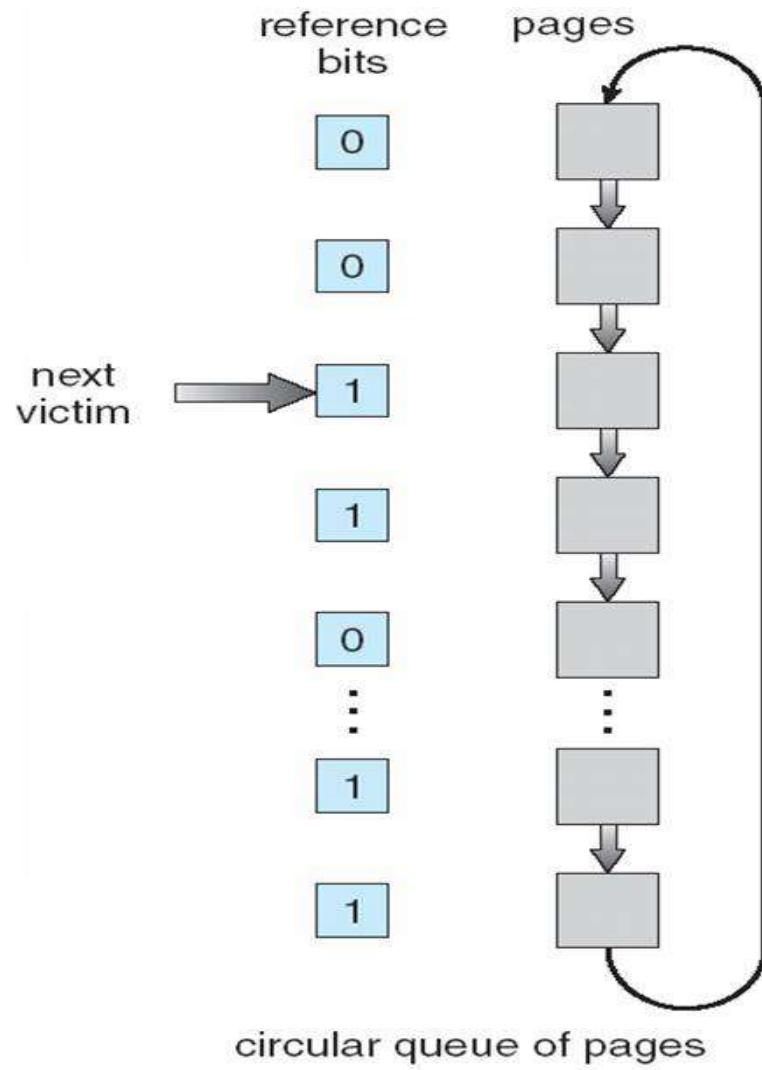
LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
 - With each page associate a bit, initially = 0
 - When page is referenced bit set to 1
 - Replace any with reference bit = 0 (if one exists)
 - We do not know the order

LRU Approximation Algorithms

- **Second-chance algorithm**
 - Generally FIFO, plus hardware-provided reference bit
 - Clock replacement
 - If page to be replaced has
 - Reference bit = 0 -> replace it
 - reference bit = 1 then:
 - set reference bit 0, leave page in memory
 - replace next page, subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



(a)

OS PPT NOTES

III BCA

Dr. Prathima G

(b)

28

Enhanced Second-Chance Algorithm

- Based on the **reference bit** and the **modify bit** as an ordered pair.
- Four possible classes:
- (0, 0) neither recently used nor modified -**best page to replace**
- (0, 1) not recently used but **modified-not quite as good, because the page will need to be written out before replacement**
- (1, 0) recently used but clean-**probably will be used again soon**
- (1, 1) recently used and modified -**probably will be used again soon, and the page will be need to be written out to disk before it can be replaced**

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
 - Not common
- **LFU(Least Frequently Used) Algorithm:** replaces page with smallest count
- **MFU(Most Frequently Used) Algorithm:**
- based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page-Buffering Algorithms

- **Keep a pool of free frames, always**
 - Then frame available when needed, not found at fault time
 - Read page into free frame and select victim to evict and add to free pool
 - When convenient, evict victim
- **Possibly, keep list of modified pages**
 - When backing store otherwise idle, write pages there and set to non-dirty
- **Possibly, keep free frame contents intact and note what is in them**
 - If referenced again before reused, no need to load contents again from disk
 - Generally useful to reduce penalty if wrong victim frame selected

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
 - OS keeps copy of page in memory as I/O buffer
 - Application keeps page in memory for its own work
- Operating system can give direct access to the disk, getting out of the way of the applications
 - Raw disk mode
- Bypasses file system services, such as file I/O demand paging, file locking, prefetching, space allocation, file names, and directories.

Allocation of Frames

- Each process needs *minimum* number of frames
- Example: IBM 370 – 6 pages to handle MOVE instruction:
 - instruction is 6 bytes, might span 2 pages
 - 2 pages to handle *from*
 - 2 pages to handle *to*
- *Maximum* is total frames in the system
- Two major allocation schemes
 - fixed allocation
 - priority allocation
- Many variations

Fixed Allocation

- **Equal allocation –**
- The easiest way to split m frames among n processes is **to give everyone an equal share, m/n frames.**
- For example, if there are 93 frames (after allocating frames for the OS) and 5 processes, give each process 18 frames
 - Keep some as free frame buffer pool , say 3 frames

Fixed Allocation

- **Proportional allocation** – Allocate according to the size of process
 - Dynamic as degree of multiprogramming, process sizes change

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

a_i = allocation for p_i = $\frac{s_i}{S} \times m$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation

- Use a proportional allocation scheme using solution,
- the ratio of frames depends not on the relative sizes of processes but rather on the priorities of processes or
- on a combination of size and priority.
- If process P_i generates a page fault,
 - select for replacement a frame from a process with lower priority number

Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames
- one process can take a frame from another
 - But then process execution time can vary greatly
 - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
 - More consistent per-process performance
 - But possibly underutilized memory

Non-Uniform Memory Access

- So far all memory accessed equally
- Many systems are NUMA – speed of access to memory varies-
- Slower than systems in which memory and CPUs are located on the same motherboard
 - Managing which page frames are stored at which locations can significantly affect performance in NUMA systems.
 - If we treat memory as uniform in such a system,
 - CPUs may wait significantly longer for memory access than if we modify memory allocation algorithms to take NUMA into account.

Non-Uniform Memory Access

- Optimal performance comes from allocating memory “close to” the CPU on which the process is scheduled
 - And modifying the scheduler to schedule the process on the same system board when possible
 - Solved by Solaris by creating **Igroups**

Non-Uniform Memory Access

- Each Igroup gathers together close CPUs and memory.
- Solaris tries to schedule all threads of a process and allocate all memory of a process within an Igroup.
- If that is not possible, it picks nearby Igroups for the rest of the resources needed.
- In this manner, **overall memory latency is minimized, and CPU cache hit rates are maximized.**

Thrashing

- If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault
- At this point, it must replace some page.
- Since all its pages are in active use, it must replace a page that will be needed again right away
- Consequently, it quickly faults again, and again, and again, replacing pages that it must back in immediately
- This high paging activity is called Thrashing
- A process is thrashing if it is spending more time on paging than executing.

Cause of Thrashing

- The operating system monitors CPU utilization.
- If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system.
- A global page-replacement algorithm is used;
- it replaces pages without regard to the process to which they belong.
- Now suppose that a process enters a new phase in its execution and needs more frames.
- It starts faulting and taking frames away from other processes.

Cause of Thrashing

- These processes need those pages, however, and so they also fault, taking frames from other processes.
- These faulting processes must use the paging device to swap pages in and out.
- As they queue up for the paging device, the ready queue empties.
- As processes wait for the paging device, CPU utilization decreases.

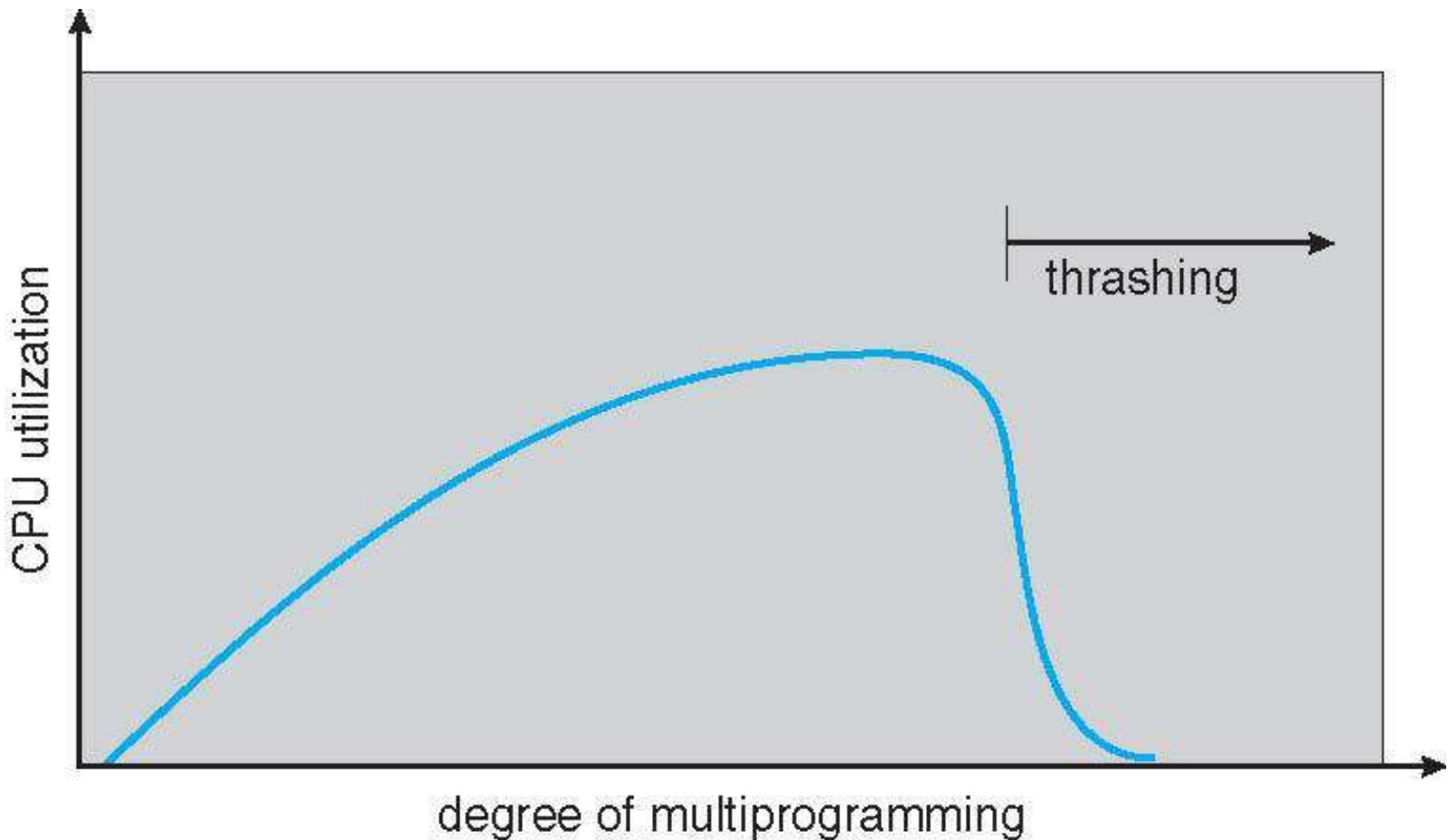
Cause of Thrashing

- The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result.
- The new process tries to get started by taking frames from running processes,
- causing more page faults and a longer queue for the paging device.
- As a result, CPU utilization drops even further, and
- the CPU scheduler tries to increase the degree of multiprogramming even more.

Cause of Thrashing

- **Thrashing has occurred, and system throughput plunges.**
- **The page fault rate increases tremendously.**
- **As a result, the effective memory-access time increases**
- **No work is getting done, because the processes are spending all their time paging.**

Thrashing



Prevent Thrashing

- To prevent Thrashing, we must provide a process with as many frames as it needs.
- But how do we know how many frames it "needs"?
- There are several techniques.
- The working-set strategy starts by looking at how frames a process is actually using.
- This approach defines the locality of process execution.
- The locality model states that, as a process executes, it moves from locality to locality.
- A locality is a set of pages that are actively used together

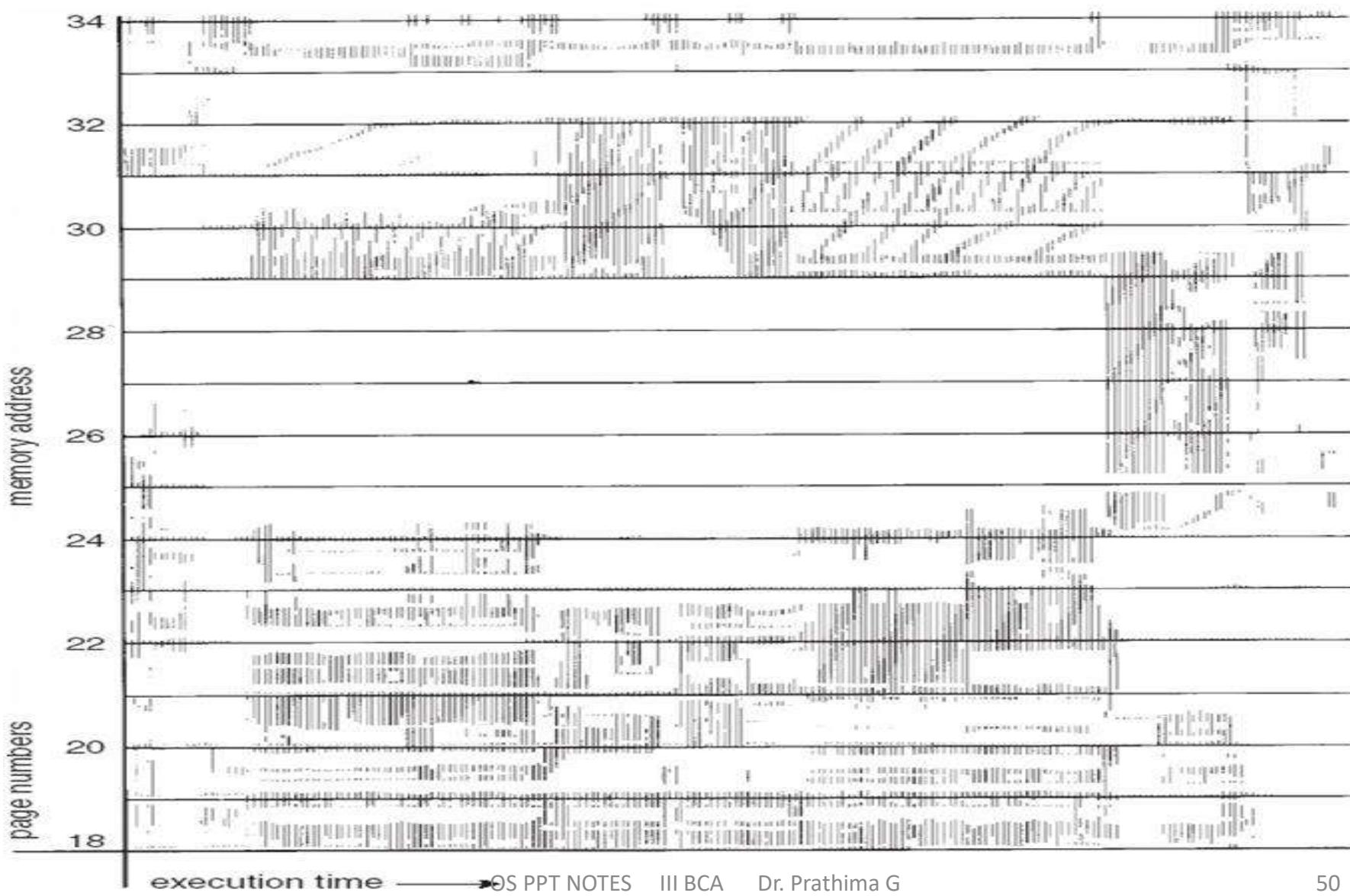
Prevent Thrashing

- A program is generally composed of several different localities, which may overlap.
- For example, when a function is called, it defines a new locality.
- In this locality, memory references are made to the instructions of the function call, its local variables, and a subset of the global variables.
- When we exit the function, the process leaves this locality,
- since the local variables and instructions of the function are no longer in active use.

Prevent Thrashing

- Suppose we allocate enough frames to a process to accommodate its current locality.
- It will fault for the pages in its locality until all these pages are in memory;
- It will not fault again until it changes localities.
- If we do not allocate enough frames to accommodate the size of the current locality,
- The process will thrash, since it cannot keep in memory all the pages that it is actively using.

Locality In A Memory-Reference Pattern

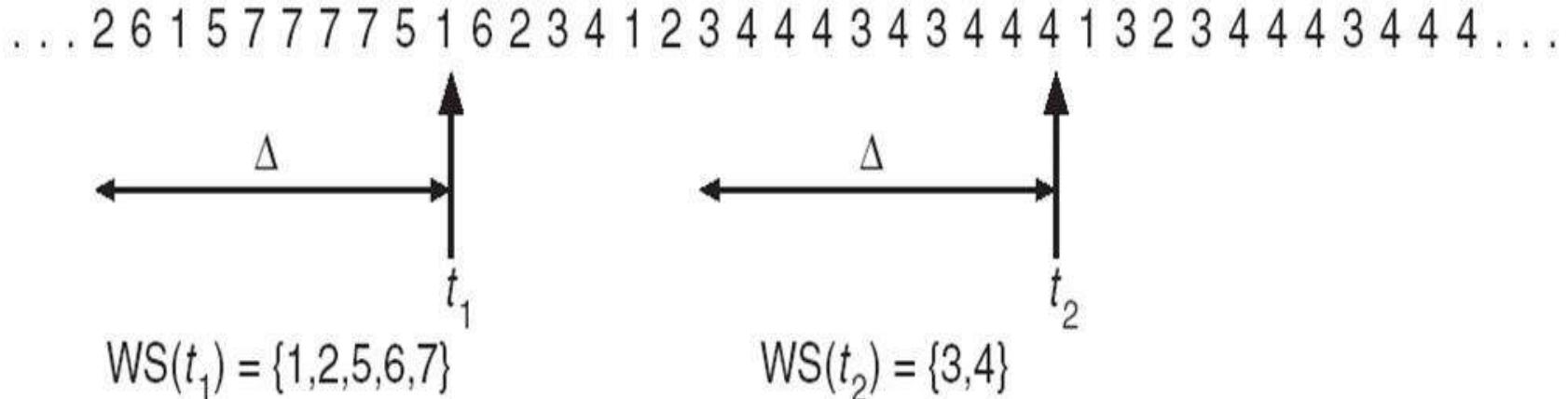


Working-Set Model

- This is based on the assumption of locality.
- This model uses a parameter Δ to define the working set window.
- The idea $\Delta \equiv$ working-set window \equiv a fixed number of page references

Working-set model

page reference table



For example, given the sequence of memory references shown in Figure, if $\Delta = 10$ memory references,

then the working set at time t_1 is $\{1, 2, 5, 6, 7\}$.

By time t_2 , the working set has changed to $\{3, 4\}$.

Working-Set Model

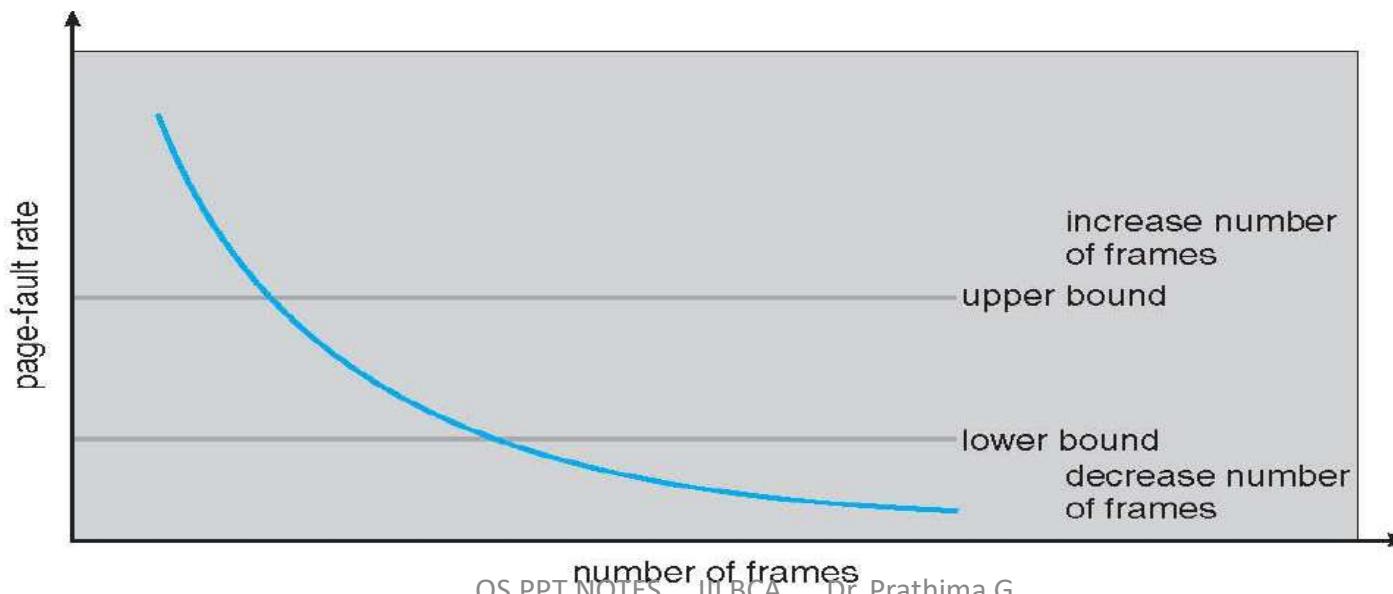
- WSS_i (working set of Process P_i) =
total number of pages referenced in the most recent Δ (varies in time)
 - if Δ too small will not encompass entire locality
 - if Δ too large will encompass several localities
 - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \sum WSS_i$ = total demand frames
 - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes

Page-Fault Frequency(PFF)

- The working-set model is successful to prevent thrashing but it seems a clumsy way to control thrashing.
- PFF takes a more direct approach.
- Thrashing has a high page-fault rate.
- We want to control the page-fault rate.
- When it is too high, we know that the process needs more frames.
- Conversely, if the page-fault rate is too low, then the process may have too many frames.

Page-Fault Frequency

- Establish upper and lower bounds on the desired page-fault rate
- **The actual page-fault rate exceeds the upper limit, we allocate the process another frame;**
- **If the page-fault rate falls below the lower limit, we remove a frame from the process.**
- Thus, we can directly measure and control the page-fault rate to prevent thrashing.



Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging
 - A page-sized portion of the file is read from the file system into a physical page
 - Subsequent reads/writes to/from the file are treated as ordinary memory accesses

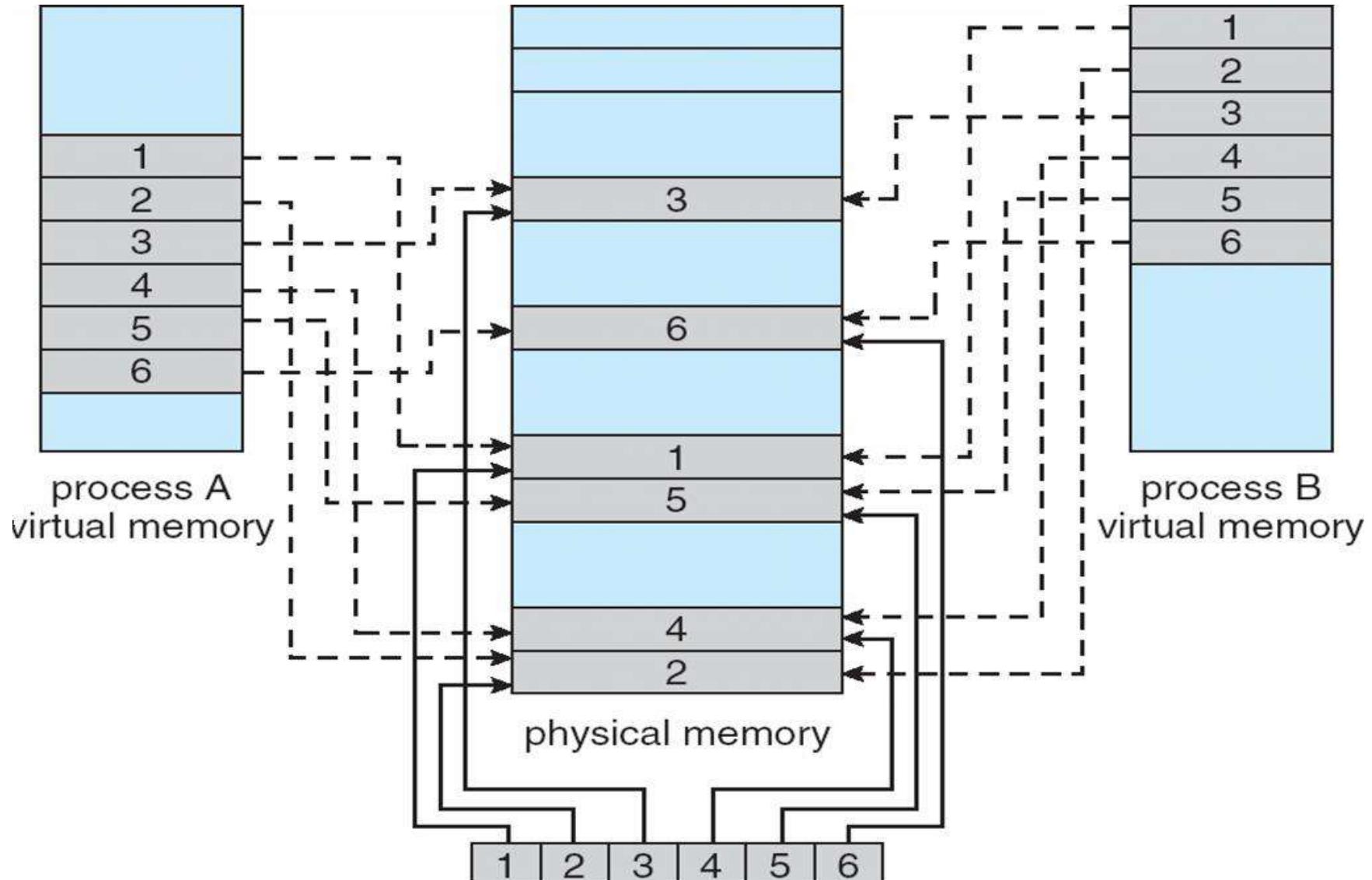
Memory-Mapped Files

- Simplifies and speeds file access by driving file I/O through memory rather than `open()`, `read()` and `write()` system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
 - Periodically and / or at file `close()` time
 - For example, when the pager scans for dirty pages

Memory-Mapped File Technique for all I/O

- Multiple processes may be allowed to map the same file concurrently, to allow sharing of data.
- The virtual memory map of each sharing process points to the same page of physical memory-the page that holds a copy of the disk block .
- **The memory-mapping system calls can also support copy-on-write functionality,**
- **allowing processes to share a file in read-only mode but to have their own copies of any data they modify.**

Memory Mapped Files



Memory-Mapped Shared Memory in Windows

The general outline for creating a region of shared memory using memory mapped files in the Win32 API .

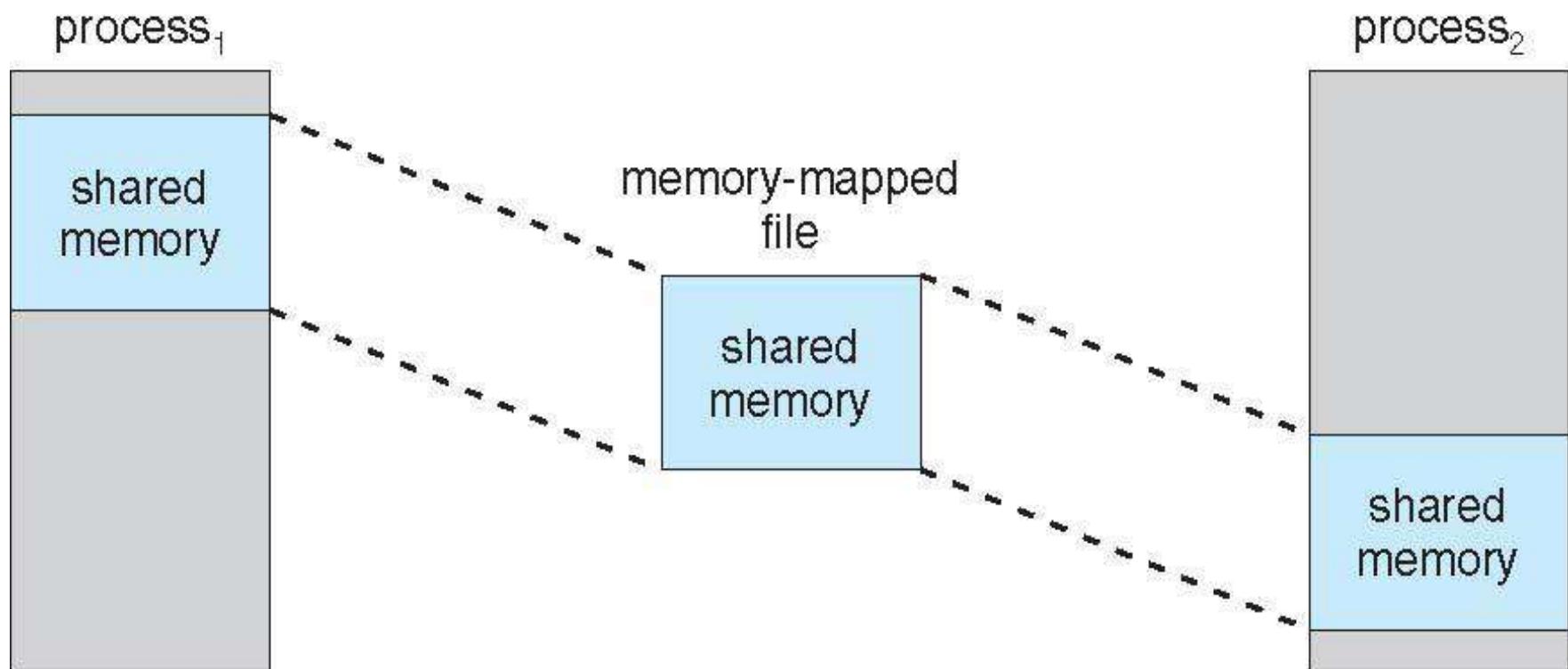
This involves first creating a file mapping for the file to be mapped and

Then establishing a view of the mapped file in a process's virtual address space

A second process can then open and create a view of the mapped file in its virtual address space.

Memory-Mapped Shared Memory in Windows

The mapped file represents the shared-memory object that will enable communication to take place between the processes.



Allocating Kernel Memory

- When a process running in user mode requests additional memory,
- Pages are allocated from the list of free page frames maintained by the kernel.
- Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes.
- There are two primary reasons for this:

Allocating Kernel Memory

- 1. The kernel requests memory for data structures of varying sizes, some of which are less than a page in size.
- As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation.
- This is especially important because many operating systems do not subject kernel code or data to the paging system.

Allocating Kernel Memory

- **2. Pages allocated to user-mode processes do not necessarily have to be in contiguous physical memory.**
- However certain hardware devices interact directly with physical memory-
- without the benefit of a virtual memory interface-
- and consequently may require memory residing in physically contiguous pages.

Allocating Kernel Memory

- Two strategies for managing free memory that is assigned to kernel processes:
 - **1. The "buddy system."**
 - **2. The slab allocation.**

Buddy System

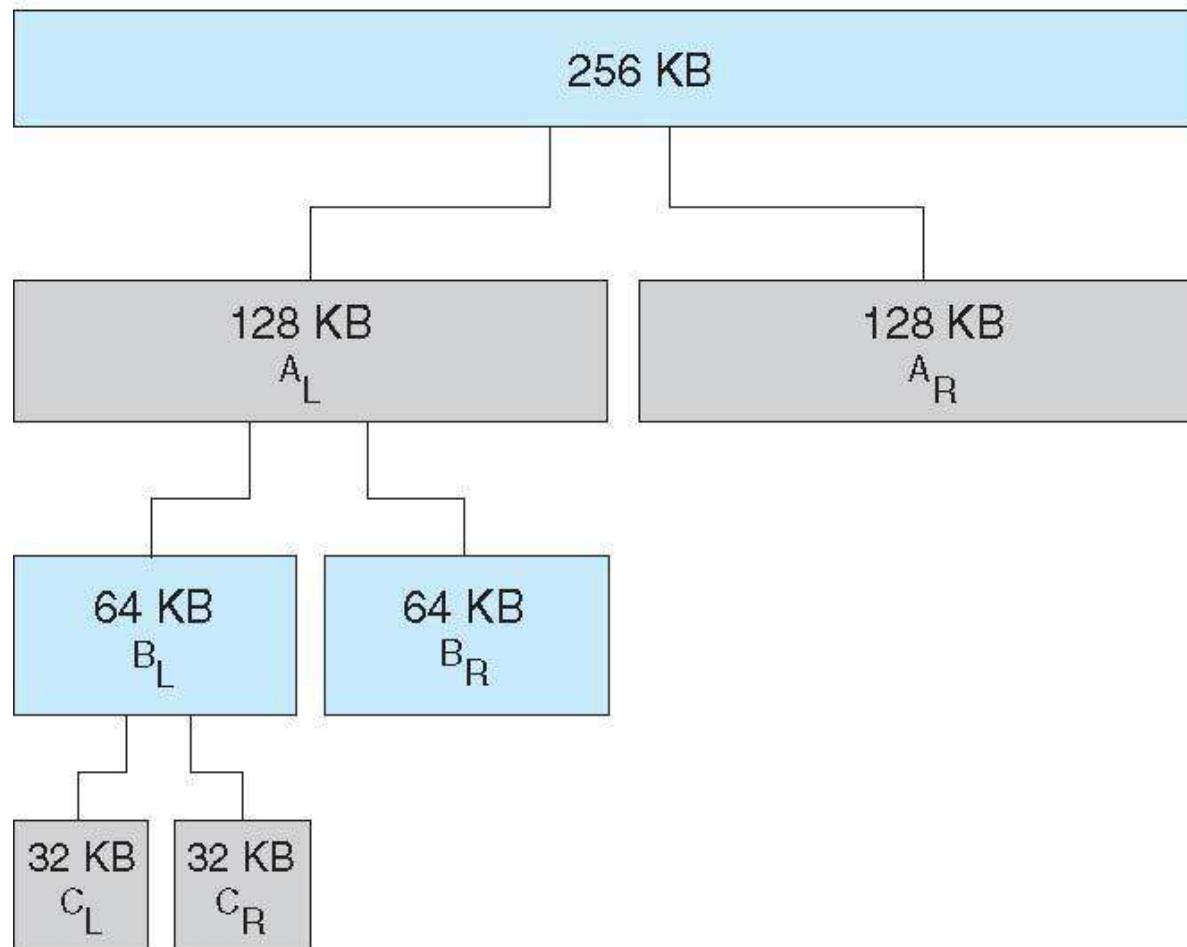
- **Allocates memory from fixed-size segment consisting of physically-contiguous pages**
- Memory allocated using **power-of-2 allocator**
 - Satisfies requests in units sized as power of 2
 - Request rounded up to next highest power of 2
 - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
 - Continue until appropriate sized chunk available

Buddy System

- For example, assume 256KB chunk available, kernel requests 21KB
 - Split into A_L and A_r of 128KB each
 - One further divided into B_L and B_R of 64KB
 - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage – quickly coalesce unused chunks into larger chunk
- Disadvantage - fragmentation

Buddy System Allocator

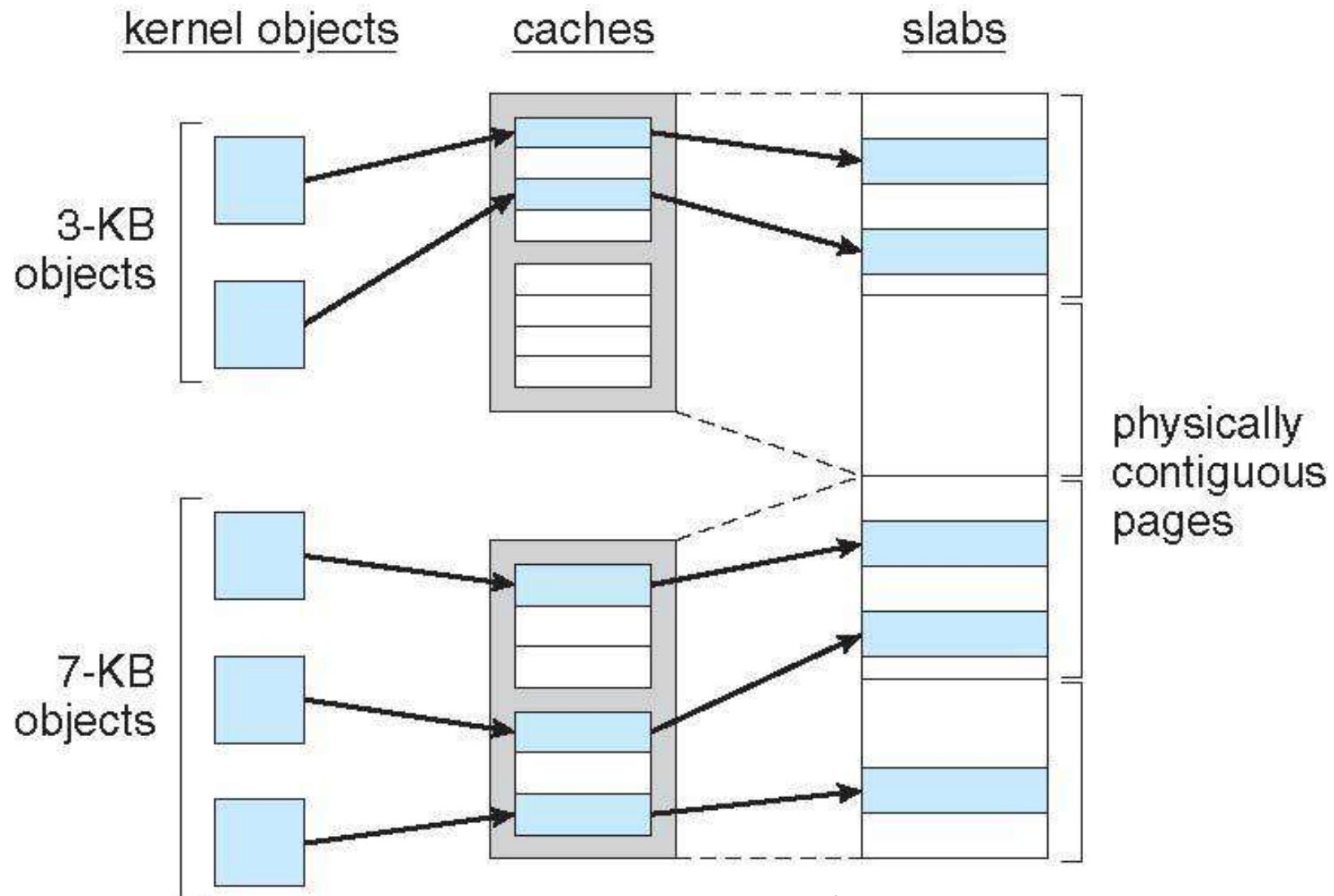
physically contiguous pages



Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
 - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- **If slab is full of used objects, next object allocated from empty slab**
 - If no empty slabs, new slab allocated
- **Benefits include no fragmentation, fast memory request satisfaction**

Slab Allocation



DEADLOCKS

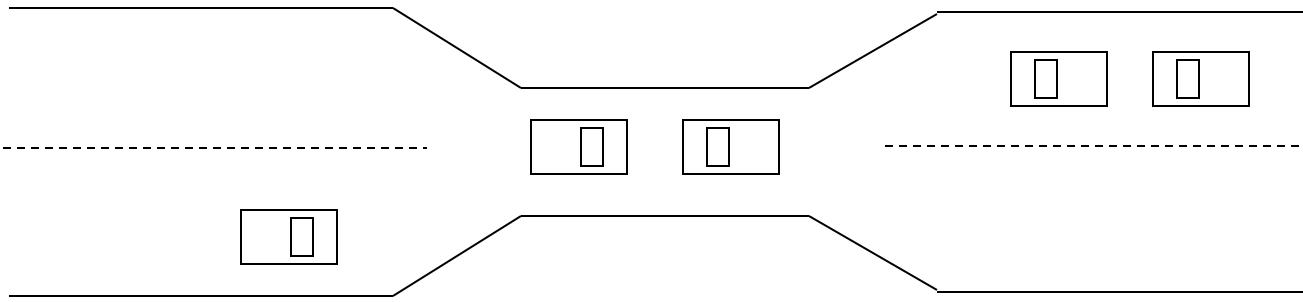
Outline

- Resources
- Principles of deadlock
- Methods For Handling Deadlock – Ostrich And Bankers Algorithm
- Deadlock Prevention
- Detection And Recovery
- Deadlock Avoidance

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example
 - System has 2 disk drives
 - P_1 and P_2 each hold one disk drive and each needs another one

Bridge Crossing Example



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs

System Model

- Resource types R_1, R_2, \dots, R_m
CPU cycles, memory space, I/O devices – DVDs, Printers, Scanners, disk
- Each resource type R_i has W_i instances- If a system has two CPUs, then the resource type CPU has two instances.
- Similarly, the resource type printer may have five instances.
- Each process utilizes a resource as follows:
 - **request**
 - **use**
 - **release**

System Model

- **Request.** The process requests the resource.
- If the request cannot be granted immediately
- (for example, if the resource is being used by another process),
- Then the requesting process must wait until it can acquire the resource.
- **Use.** The process can operate on the resource
- (for example, if the resource is a printer, the process can print on the printer).
- **Release.** The process releases the resource.

Deadlock Characterization

- In a deadlock, processes never finish executing, and
- System resources are tied up, preventing other jobs from starting.
- Here, we look more closely at features that characterize deadlocks.

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** Resources cannot be preempted, a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that
 - P_0 is waiting for a resource that is held by P_1 ,
 - P_1 is waiting for a resource that is held by P_2 , ...,
 - P_{n-1} is waiting for a resource that is held by P_n , and
 - P_n is waiting for a resource that is held by P_0 .

Resource-Allocation Graph

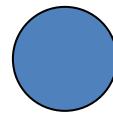
- Deadlocks can be described more precisely in terms of a directed graph called a system resource allocation graph.
- This graph consists of a set of vertices V and a set of edges E .
- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system

Resource-Allocation Graph

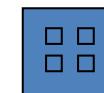
- A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$
- It signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource.
- A directed edge from resource type R_j to process P_i ;
- Is denoted by $R_j \rightarrow P_i$, it signifies that an instance of resource type R_j has been allocated to process P_i ;
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph

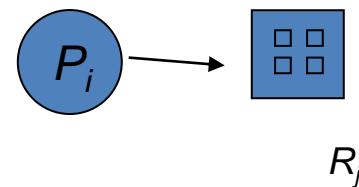
- Process



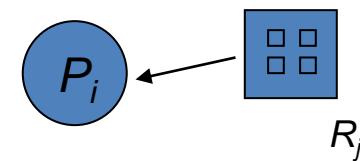
Resource Type with 4 instances



- P_i requests instance of R_j



- P_i is holding an instance of R_j



Example of a Resource Allocation Graph

The resource-allocation graph shown in Figure depicts the following situation.

The sets P, Rand E:

$$P = \{P_1, P_2, P_3\}$$

$$R = \{R_1, R_2, R_3, R_4\}$$

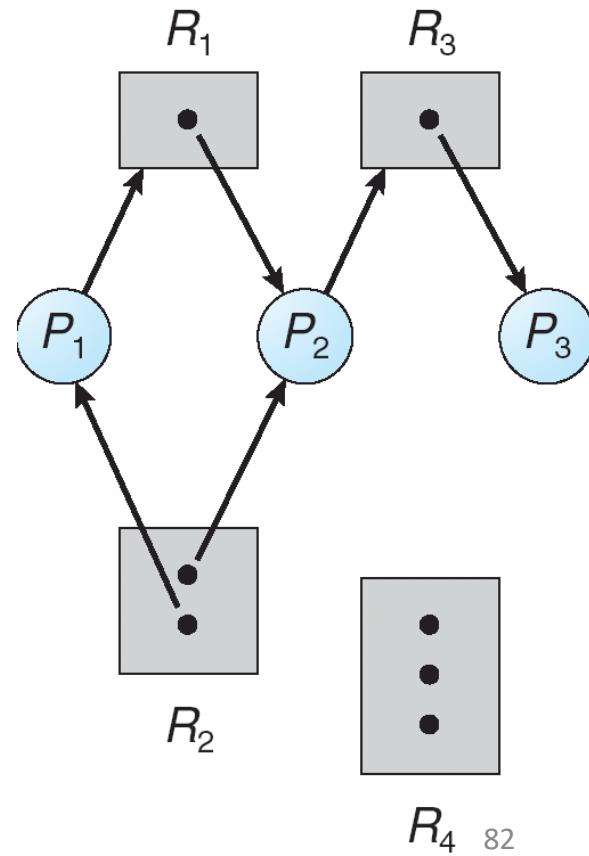
$$E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$$

Resource instances:

- One instance of resource type R1
- Two instances of resource type R2
- One instance of resource type R3
- Three instances of resource type R4

Process states:

- Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.
- Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
- Process P3 is holding an instance of R3.



Resource Allocation Graph With A Deadlock

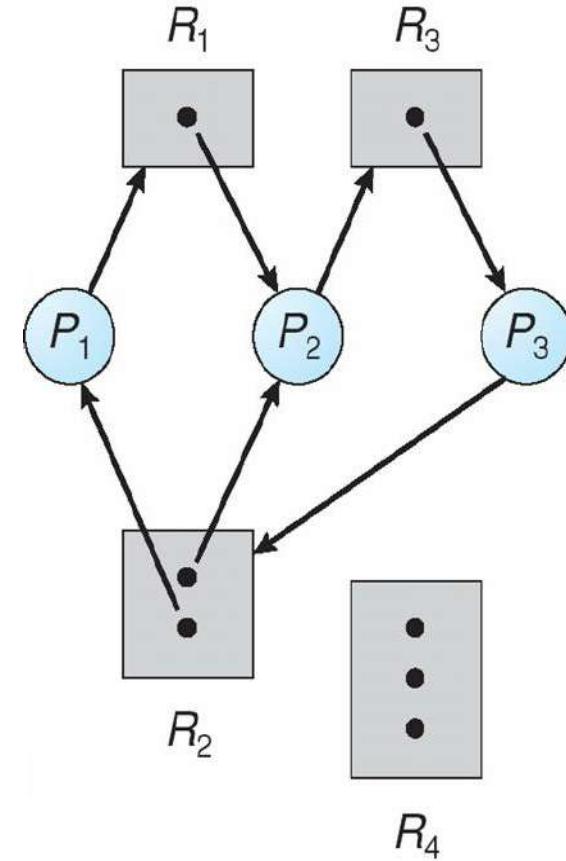
For a resource-allocation graph, if the graph contains no cycles, then no process in the system is deadlocked.

If the graph does contain a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.

Each process involved in the cycle is deadlocked.

In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

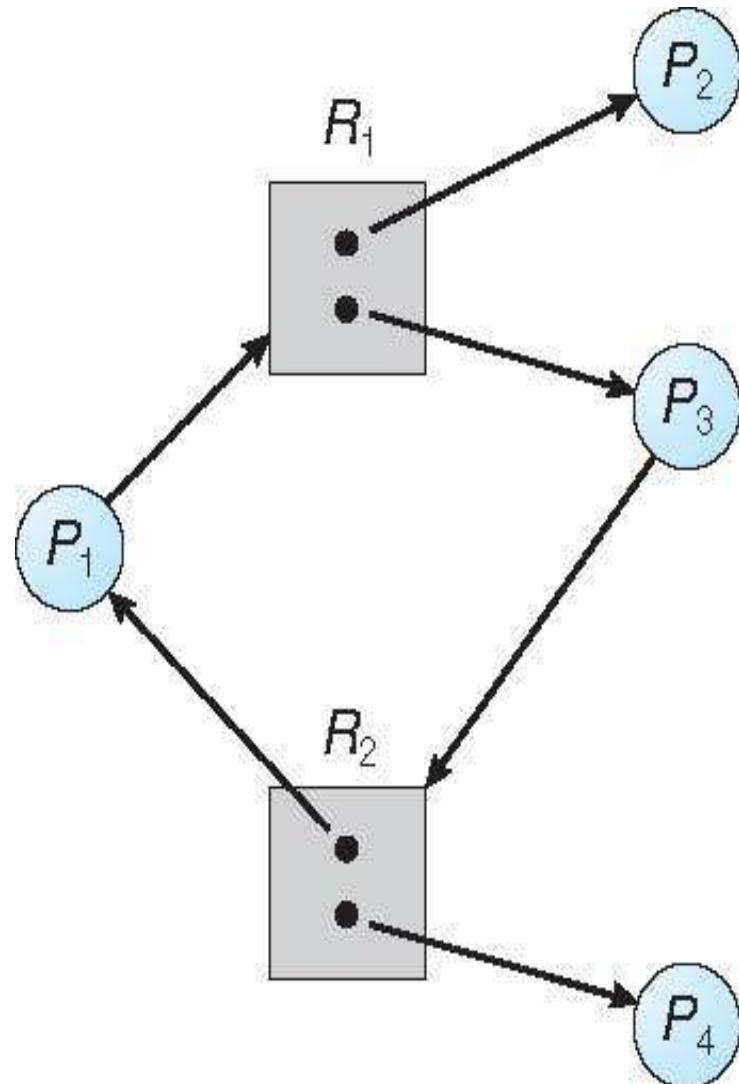


Graph With A Cycle But No Deadlock

If each resource type has several instances,

Then a cycle does not necessarily imply that a deadlock has occurred.

In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.



Basic Facts

- If graph contains no cycles \Rightarrow no deadlock
- If graph contains a cycle
 - If only one instance per resource type, then deadlock
 - If several instances per resource type, possibility of deadlock but not necessarily deadlock

Methods for Handling Deadlocks

- We can use a protocol **to prevent or avoid deadlocks**, ensuring that the system will never enter a deadlocked state.
- We can allow the system **to enter a deadlocked state, detect it, and recover**.
- We can ignore the problem altogether and **pretend that deadlocks never occur in the system**
- The third solution is the one used by most operating systems, including UNIX and Windows;
- It is then up to the application developer to write programs that handle deadlocks.

Deadlock Prevention

- For a deadlock to occur, each of the four necessary conditions must hold.
- By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

Deadlock Prevention

- The **mutual-exclusion** condition must hold for non sharable resources
- For example, a printer cannot be simultaneously shared by several processes
- Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock
- Read-only files are a good example of a sharable resource
- If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file
- A process never needs to wait for a sharable resource

Deadlock Prevention

- **Hold and Wait** – ensure that this condition will not be met
ie:
- must guarantee that whenever a process requests a resource, it does not hold any other resources
 - Require process to request and be allocated all its resources before it begins execution, or
 - allow process to request resources only when the process has none
 - **Drawbacks - Low resource utilization- since resources may be allocated but unused for a long period**
 - **starvation possible- A process that needs several popular resources may have to wait indefinitely**

Deadlock Prevention

- **No Preemption** –To ensure that this condition does not hold,
 - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
 - Preempted resources are added to the list of resources for which the process is waiting
 - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Deadlock Prevention

- **Circular Wait** – One way to ensure that this condition never holds
- To impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.
- To illustrate, we let $R = \{ R_1, R_2, \dots, R_m \}$ be the set of resource types.
- We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.
- For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:
 - $F(\text{tape drive}) = 1 \quad F(\text{disk drive}) = 5 \quad F(\text{printer}) = 12$
 - Like this no circular wait

Deadlock Avoidance

Requires that the system has some additional *a priori* information available

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need
- The **deadlock-avoidance algorithm** dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- **Resource-allocation state** is defined by the number of available and allocated resources, and the maximum demands of the processes

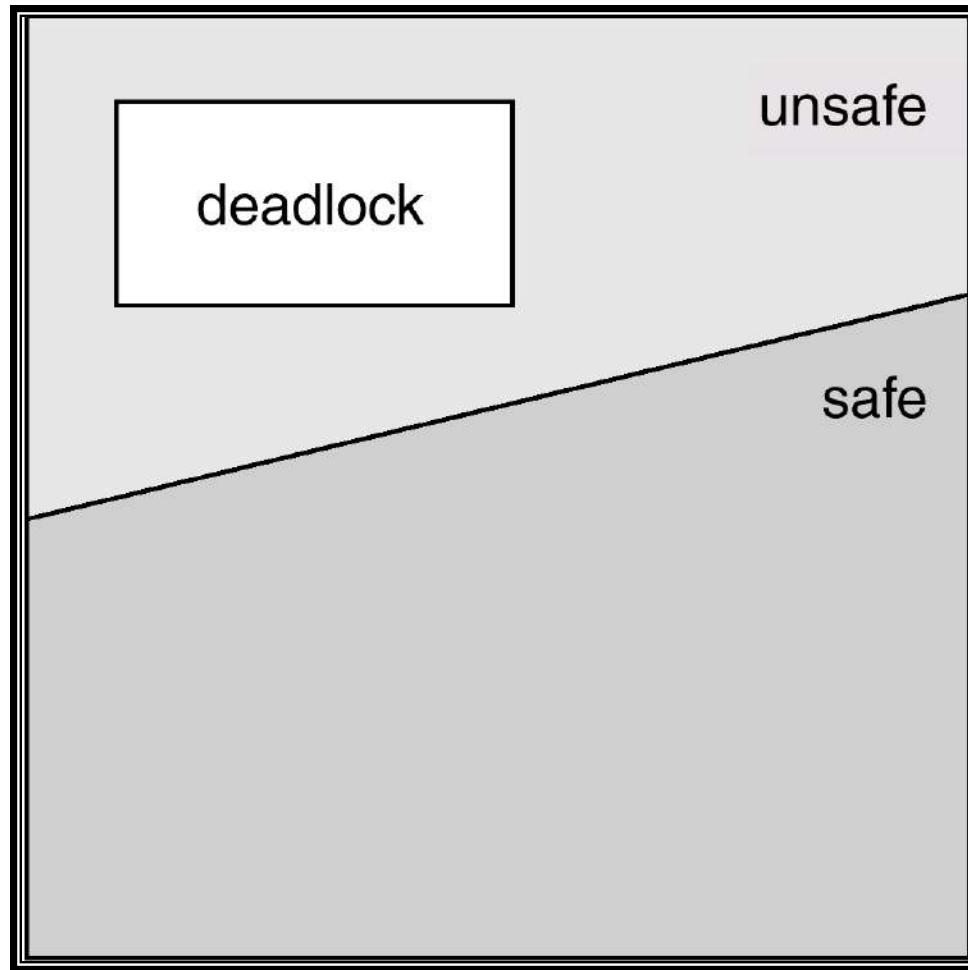
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a *safe state*.
- System is in safe state if there exists a safe sequence of all processes.
- Sequence $\langle P_1, P_2, \dots, P_n \rangle$ is safe if for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j , with $j < i$.
 - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
 - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
 - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Basic Facts

- If a system is in safe state \Rightarrow no deadlocks.
- If a system is in unsafe state \Rightarrow possibility of deadlock.
- Avoidance \Rightarrow ensure that a system will never enter an unsafe state.

Safe, Unsafe , Deadlock State



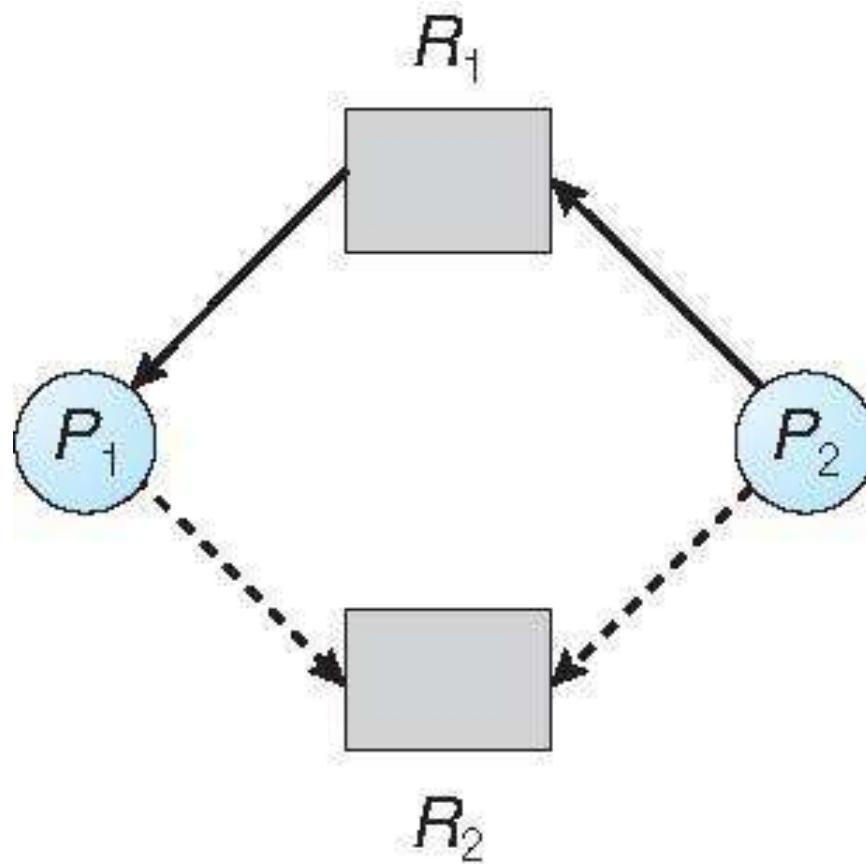
Avoidance algorithms

- Single instance of a resource type
 - **Use a resource-allocation graph**
- Multiple instances of a resource type
 - **Use the banker's algorithm**

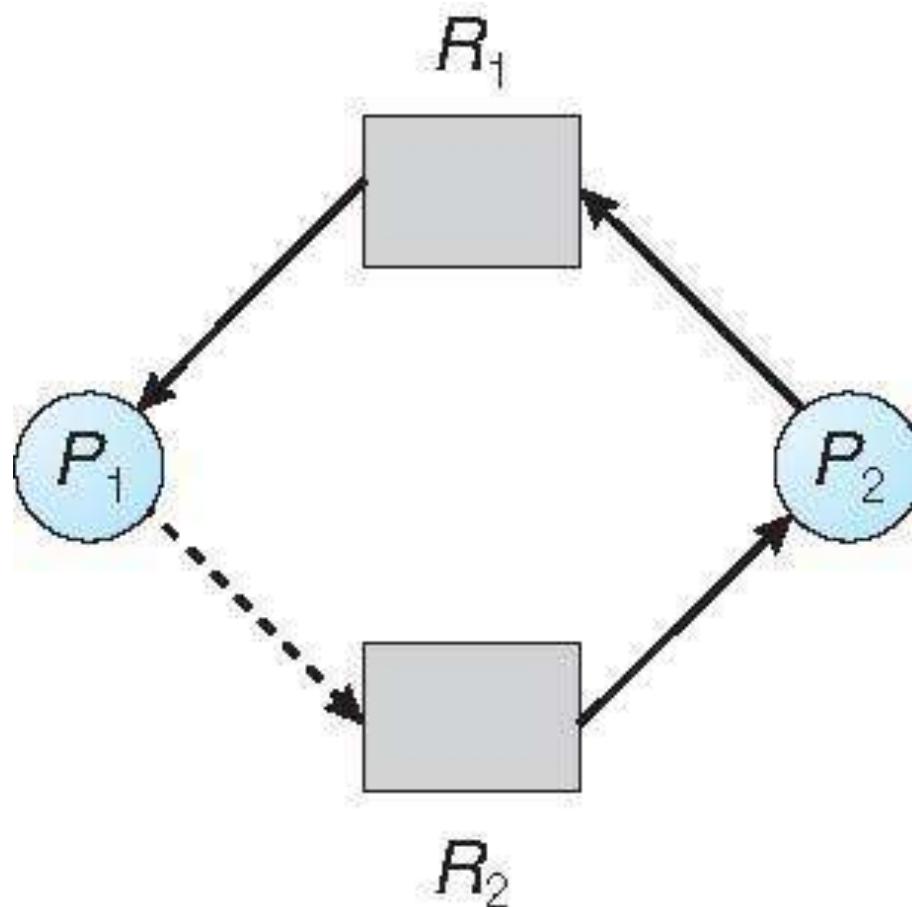
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j ; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

- Suppose process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

Let n = number of processes,

m = number of resource types

- **Available:** Vector of length m . If $\text{Available}[j] = k$, there are k instances of resource type R_j **currently available**
- **Max:** $n \times m$ matrix. If $\text{Max}[i,j] = k$, then process P_i will request at most k instances of resource type R_j .
- **Alloc:** $n \times m$ matrix. If $\text{Alloc}[i,j] = k$ then P_i is currently allocated (i.e. holding) k instances of R_j .
- **Need:** $n \times m$ matrix. If $\text{Need}[i,j] = k$, then P_i may need k more instances of R_j to complete its task.

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Alloc}[i,j].$$

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length m and n , respectively. Initialize:

Work := Available

Finish [i] == false for i = 1,2, ..., n.

2. Find an i such that both:

Finish [i] == false

Need_i ≤ Work

If no such i exists, go to step 4.

3. *Work := Work + Allocation_i*

(Resources freed when process completes)

Finish[i] := true

go to step 2.

4. If *Finish [i] = true for all i*, then the system is in a safe state.

Resource-Request Algorithm for Process P_i

Request_i = request vector for P_i .

$\text{Request}_i[j] = k$ means process P_i wants k instances of resource type R_j .

1. If $\text{Request}_i \leq \text{Need}_i$, go to step 2. Otherwise, error (process exceeded its maximum claim).
2. If $\text{Request}_i \leq \text{Available}$, go to step 3. Otherwise P_i must wait, (resources not available).
3. “Allocate” requested resources to P_i as follows:

$\text{Available} := \text{Available} - \text{Request}_i$

$\text{Alloc}_i := \text{Alloc}_i + \text{Request}_i$

$\text{Need}_i := \text{Need}_i - \text{Request}_i$

If safe \Rightarrow the resources are allocated to P_i .

If unsafe \Rightarrow restore the old resource-allocation state and block P_i

Example of Banker's Algorithm

5 processes P_0 through P_4

3 resource types A (10 units), B (5 units), and C (7 units).

Snapshot at time T_0 :

	<u>Allocation</u>			<u>Max</u>	<u>Available</u>	
	A	B	C	A	B	C
P_0	0	1	0	7	5	3
P_1	2	0	0	3	2	2
P_2	3	0	2	9	0	2
P_3	2	1	1	2	2	2
P_4	0	0	2	4	3	3

Example (cont)

Need = Max – Allocation

	<u>Need</u>		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria.

Now P_1 requests (1,0,2)

Check that Request \leq Available
(that is, $(1,0,2) \leq (3,3,2)$) \Rightarrow true.

	<u>Allocation</u>			<u>Need</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

Executing safety algorithm shows that sequence $< P_1, P_3, P_4, P_0, P_2 >$ satisfies safety requirement

Can request for (3,3,0) by P_4 be granted?

Can request for (0,2,0) by P_0 be granted?

Deadlock Detection

- If a system does not employ either a deadlock-prevention or a dead lock avoidance algorithm,
- Then a deadlock situation may occur.
- In this environment, the system may provide:
- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

Single Instance of Each Resource Type

- If all resources have only a single instance,
- Then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, **called a wait-for graph**.
- Graph is obtained from resource-allocation graph **by removing the resource nodes and collapsing the appropriate edges**.

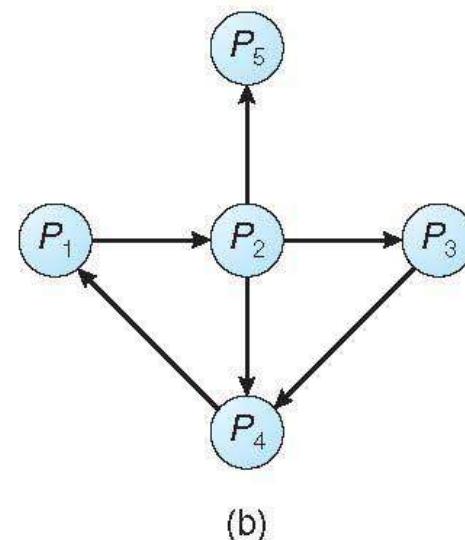
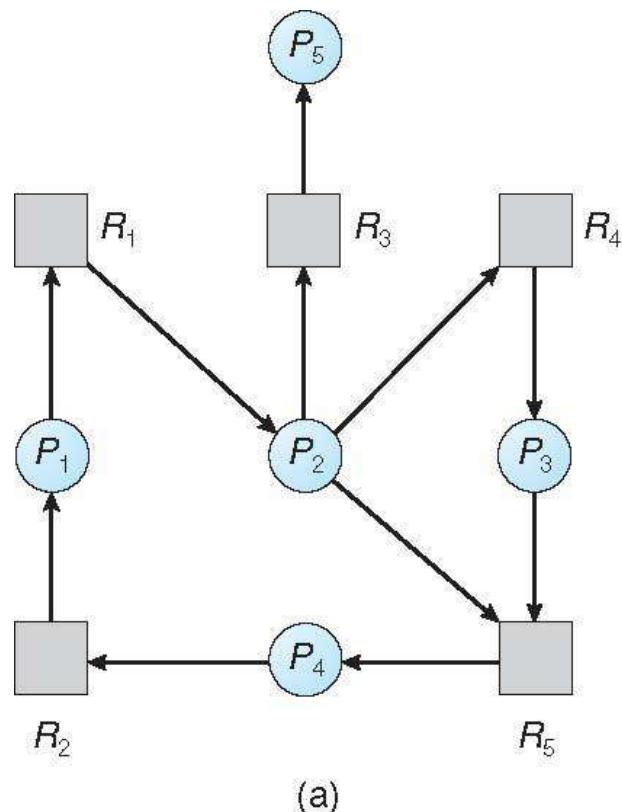
Single Instance of Each Resource Type

- Maintain **wait-for** graph
- An edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs.
- An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_j$ and $R_j \rightarrow P_i$ for some resource R_j
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j

Single Instance of Each Resource Type

- Periodically invoke an algorithm that searches for a cycle in the graph.
- If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph

Corresponding wait-for graph

Several Instances of a Resource Type

- The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.
- We turn now to a deadlock detection algorithm that is applicable to such a system.
- The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm

Several Instances of a Resource Type

- **Available:** A vector of length m indicates the number of available resources of each type
- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process
- **Request:** An $n \times m$ matrix indicates the current request of each process.
- If $\text{Request}[i][j] = k$, then process P_i is requesting k more instances of resource type R_j .

Detection Algorithm

The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed.

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively Initialize:

- (a) ***Work = Available***
- (b) For $i = 1, 2, \dots, n$, if $\text{Allocation}_i \neq 0$, then
Finish[i] = false; otherwise, ***Finish[i] = true***

2. Find an index ***i*** such that both:

- (a) ***Finish[i] == false***
- (b) ***Request_i ≤ Work***

If no such ***i*** exists, go to step 4

Detection Algorithm

3. $\text{Work} = \text{Work} + \text{Allocation}_i$,
 $\text{Finish}[i] = \text{true}$
go to step 2
4. If $\text{Finish}[i] == \text{false}$, for some i , $1 \leq i \leq n$, then the system
is in deadlock state.
Moreover, if $\text{Finish}[i] == \text{false}$, then P_i is deadlocked

Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

- Five processes P_0 through P_4 ; three resource types A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time T_0 :

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

- Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = true$ for all i

Example

- P_2 requests an additional instance of type **C**

Request

	A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

- State of system?

- Can reclaim resources held by process P_0 , but insufficient resources to fulfill other processes' requests
- Deadlock exists, consisting of processes P_1 , P_2 , P_3 , and P_4

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
 - How often a deadlock is likely to occur?
 - How many processes will need to be rolled back?
- If detection algorithm is invoked arbitrarily,
- There may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from deadlock

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
- One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- Another possibility is to let the system recover from the deadlock automatically.
- There are two options for breaking a deadlock
- **One is simply to abort one or more processes to break the circular wait.**
- **The other is to preempt some resources from one or more of the deadlocked processes.**

1. Process Termination

- To eliminate deadlocks by aborting a process, we use one of two methods.
- In both methods, the system reclaims all resources allocated to the terminated processes.
- **Abort all deadlocked processes.** This method clearly will break the deadlock cycle, but at great expense
 - The deadlocked processes may have computed for a long time, and
 - The results of these partial computations must be discarded and
 - Probably will have to be recomputed later.

1. Process Termination

- **Abort one process at a time until the deadlock cycle is eliminated.**
- This method incurs considerable overhead, since after each process is aborted,
- A deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Which process to abort?

- Many factors may affect which process is chosen, including:
- What the priority of the process is
- How long the process has computed and how much longer the process will compute before completing its designated task
- How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
- How many more resources the process needs in order to complete
- How many processes will need to be terminated
- Whether the process is interactive or batch

2. Resource Preemption

- To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and
- Give these resources to other processes until the deadlock cycle is broken.
- **If preemption is required to deal with deadlocks, then three issues need to be addressed:**

2. Resource Preemption

- 1. **Selecting a victim.** Which resources and which processes are to be preempted?
- As in process termination, we must determine the order of preemption to minimize cost.
- Cost factors may include such parameters as the number of resources a deadlocked process is holding and
- The amount of time the process has thus far consumed during its execution.

2. Resource Preemption

2. Rollback. If we preempt a resource from a process, what should be done with that process?

- Clearly, it cannot continue with its normal execution as it is missing some needed resource.
- We must roll back the process to some safe state and restart it from that state.
- Since, in general, it is difficult to determine what a safe state is,
- **The simplest solution is a total rollback: abort the process and then restart it.**
- Although it is more effective to roll back the process only as far as necessary to break the deadlock,
- This method requires the system to keep more information about the state of all running processes.

2. Resource Preemption

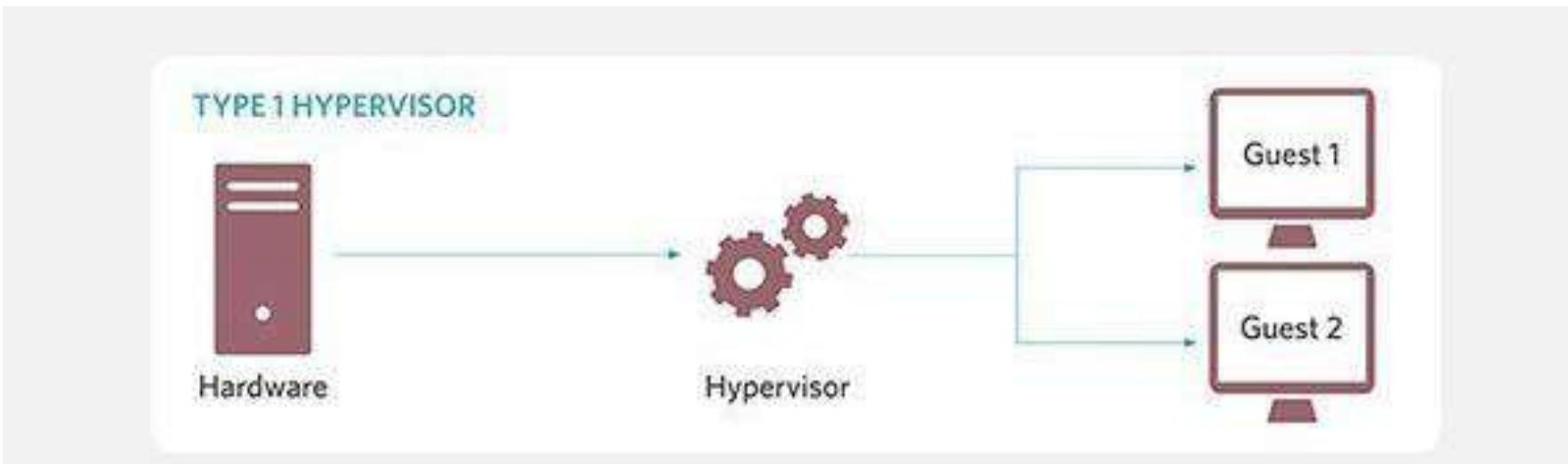
- **3. Starvation.** How do we ensure that starvation will not occur?
- That is, how can we guarantee that resources will not always be preempted from the same process?
- In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim.
- As a result, this process never completes its designated task, a starvation situation that must be dealt with in any practical system.
- **Clearly, we must ensure that a process can be picked as a victim" only a (small) finite number of times.**

HYPERVERISOR

What is hypervisor?

- ▶ A hypervisor also known as virtual machine monitor(VMM) is a piece of computer software,firmware or hardware that create and run virtual machine.
- ▶ The hypervisor is a software program that provides the layer of abstraction, handles the translations between physical and virtual resources -- such as physical vs. virtual CPUs or memory -- and manages the creation and support of virtual machines (VMs).
- ▶ The physical hardware that a hypervisor runs on is typically referred to as a host machine, whereas the VMs the hypervisor creates and supports are collectively called guest machines.

Type-1 Hypervisor Representation



Type-1 Hypervisor

- ▶ A Type 1 hypervisor runs directly on the host machine's physical hardware, and it's referred to as a bare-metal hypervisor; it doesn't have to load an underlying OS first.
- ▶ With direct access to the underlying hardware and no other software
- ▶ Type 1 hypervisors are regarded as the most efficient and best-performing hypervisors available for enterprise computing.
- ▶ VMware ESXi, Microsoft Hyper-V server and open source KVM are examples of Type 1 hypervisors.

Features of Type-1 Hypervisor

- ▶ Hypervisors that run directly on physical hardware are also highly secure.
- ▶ logical isolation of every guest VM against malicious software and activity.
- ▶ Expanding hardware capabilities, allowing each single machine to do more simultaneous work
- ▶ Efforts to control costs and to simplify management through consolidation of servers
- ▶ The improved security, reliability, and device independence possible from hypervisor architectures
- ▶ The ability to run complex, OS-dependent applications in different hardware or OS environments
- ▶ the virtualized system hosts at least one VM with an OS and management software, which enables admins to manage the physical system using system management tools such as Microsoft System Center.

ORIGIN

- ▶ The first hypervisors providing [full virtualization](#) were the test tool [SIMMON](#) and IBM's one-off research [CP-40](#) system, which began production use in January 1967, and became the first version of IBM's [CP/CMS](#) operating system.
- ▶ CP-40 ran on a [S/360-40](#) that was modified at the IBM [Cambridge Scientific Center](#) to support dynamic address translation, a feature that enabled virtualization.
- ▶ With CP-40, the hardware's *supervisor state* was virtualized as well, allowing multiple operating systems to run concurrently in separate [virtual machine](#) contexts.
- ▶ [CP/CMS](#) formed part of IBM's attempt to build robust [time-sharing](#) systems for its [mainframe](#) computers. By running multiple operating systems concurrently, the hypervisor increased system robustness and stability.
- ▶ IBM announced its [System/370](#) series in 1970 without any virtualization features, but added [virtual memory](#).

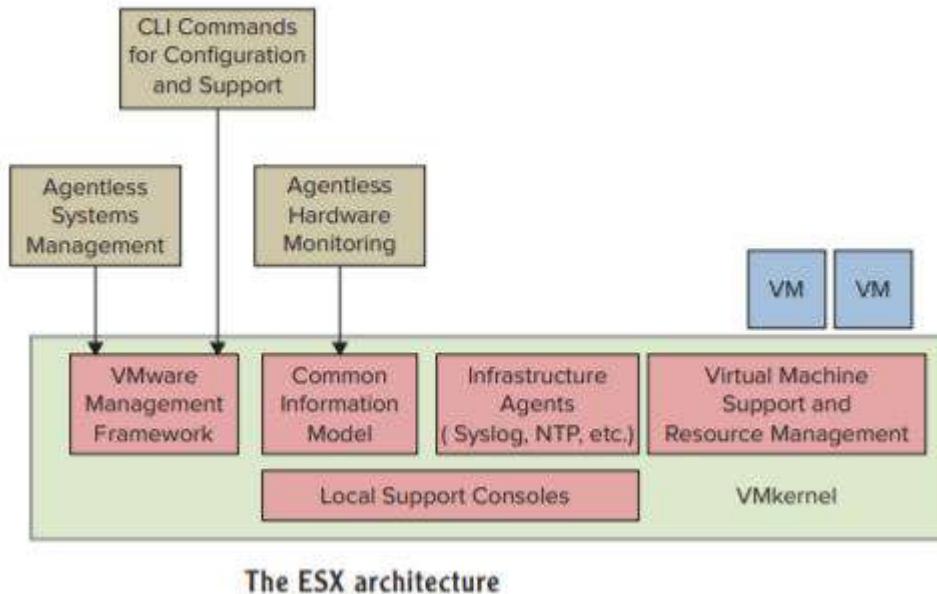
Embedded hypervisors

- ▶ Embedded hypervisors, targeting embedded systems and certain real-time operating system (RTOS) environments, are designed with different requirements when compared to desktop and enterprise systems, including robustness, security and real-time capabilities.
- ▶ The resource-constrained nature of many embedded systems, especially battery-powered mobile systems, imposes a further requirement for small memory-size and low overhead.
- ▶ The embedded world uses a wider variety of architectures and less standardized environments.
- ▶ Support for virtualization requires memory protection and a distinction between user mode and privileged mode, which rules out most microcontrollers.

Todays Type-1 hypervisors

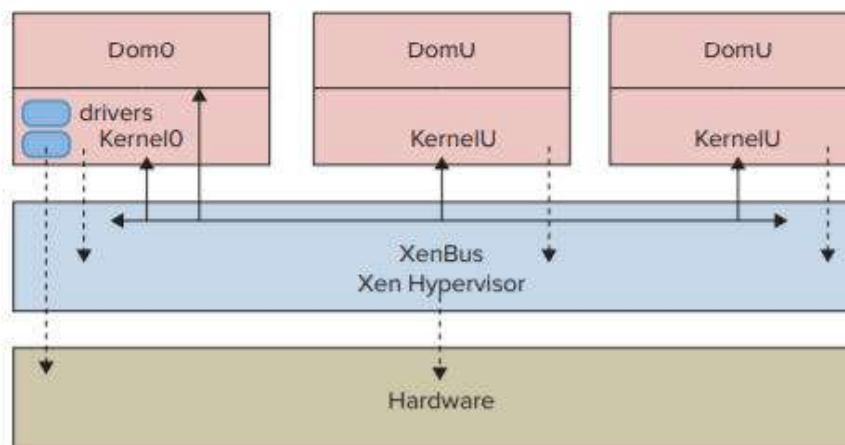
VMware ESX

- ▶ VmwareEsx was the first type-1 hypervisor released in 2001 by VMware.
- ▶ The original architecture of ESX was made up of two parts, the actual hypervisor, which did the virtualization work, and a Linux-based console module that sat alongside the hypervisor and acted as a management interface to the hypervisor.



Citrix Xen

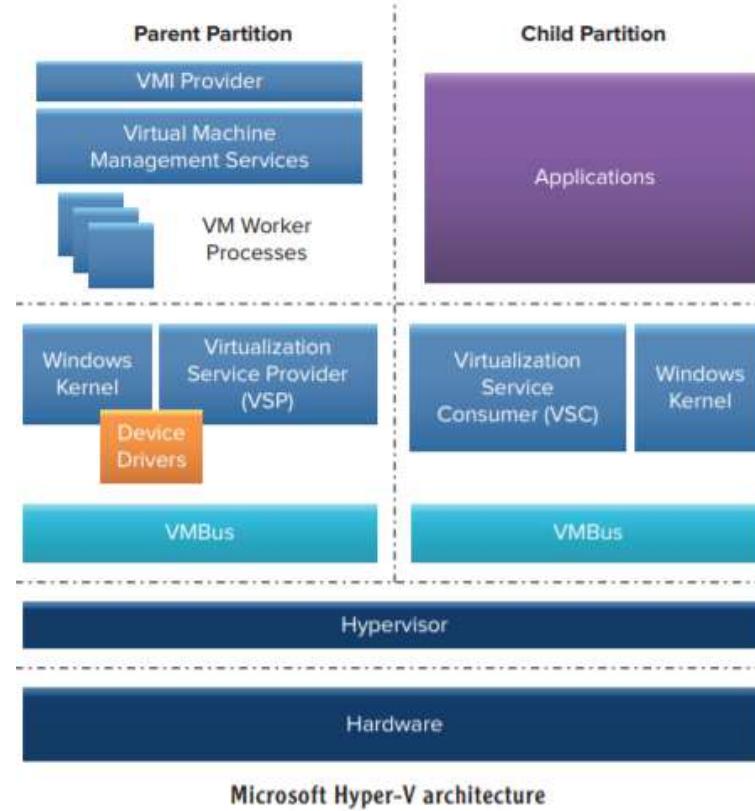
- ▶ The Xen hypervisor began as a research project in the late 1990s at the University of Cambridge.
- ▶ The Xen model has a special guest called Domain 0, also referred to as Dom0.
- ▶ This guest gets booted when the hypervisor is booted
- ▶ it has direct access to the hardware, it handles all of the I/O for the individual guests.
- ▶ When additional guests make requests of the underlying hardware resources, those requests go through the hypervisor



The Xen hypervisor architecture

Microsoft Hyper-V

- ▶ Microsoft began in the virtualization space with Virtual Server in 2005.
- ▶ rather than guests, the virtualized workloads are called partitions. Similar to the Xen model, it requires a special parent partition that has direct access to the hardware resources. Like Dom0.
- ▶ the parent partition runs an operating system—in this case, Windows Server 2008.



VIRTUALIZATION REQUIREMENTS

- ▶ The three elements to consider when selecting virtualization hardware
 - ▶ Cpu
 - ▶ Memory
 - ▶ network I/O capacity

VIRTUALIZATION REQUIREMENTS

<Cpu>

- ▶ For most processes, VMWare recommends a single virtual CPU.
- ▶ This reduces the management overhead on both the guest system and VMWare.
- ▶ A single virtual CPU however has the same impact as a single physical CPU.
- ▶ Only a single thread at a time can be executed.
- ▶ VM with two virtual CPUs can never use more than two physical CPUs at the same time.

VIRTUALIZATION REQUIREMENTS

<Memory>

- ▶ Virtualization depends heavily on ram. So ensure you select a machine that has enough ram to support your application usage.
- ▶ 32GB of RAM is recommended.
- ▶ one way to avoid wasting memory is to know the upper memory limits that each operating system supports.
 - ▶ For example, 32-bit operating systems can address a maximum of 4 GB of memory. Assigning anything over 4 GB to a virtual machine running a 32-bit operating system would be a waste.

VIRTUALIZATION REQUIREMENTS

<I/O>

- ▶ I/O virtualization provides greater flexibility, greater utilization and faster provisioning when compared to traditional NIC and HBA card architectures.
- ▶ Virtual I/O lowers costs and enables simplified server management by using fewer cards, cables, and switch ports, while still achieving full network I/O performance.
- ▶ In a virtualized I/O environment, only one cable is needed to connect servers to both storage and network traffic.
- ▶ I/O virtualization increases the practical density of I/O by allowing more connections to exist within a given space.

Virtualization by Hypervisor

- ▶ **Virtualization of the CPU:** If an OS tries to consume all of the CPU, the hypervisor will prevent it and allow other processes to execute
- ▶ **Virtualization of the memory:** a running OS has its own virtual address space that the hypervisor maps to physical memory to give the process the illusion that it is the only user of RAM.
- ▶ Each execution environment is called a guest and the computing platform on which they execute is called the host.
- ▶ The Hypervisor runs on the host and acts as a bridge between the host and the guests;

Security implications

- ▶ The use of hypervisor technology by malware and rootkits installing themselves as a hypervisor below the operating system, known as hyperjacking
- ▶ This can make them more difficult to detect because the malware could intercept any operations of the operating system (such as someone entering a password) without the anti-malware software necessarily detecting it (since the malware runs below the entire operating system).

Module 5

WINDOWS OS

Case Study: Windows OS

- History
- Design Principles
- System Components
- Environmental Subsystems
- File system
- Networking
- Programmer Interface

Windows 2000

- 32-bit preemptive multitasking operating system for Intel microprocessors
- Key goals for the system:
 - portability
 - security
 - POSIX compliance
 - multiprocessor support
 - extensibility
 - international support
 - compatibility with MS-DOS and MS-Windows applications
- Uses a micro-kernel architecture
- Available in four versions, Professional, Server, Advanced Server, National Server
- New version – Windows 2003, is now available

History

- In 1988, Microsoft decided to develop a “new technology” (NT) portable operating system that supported both the OS/2 and POSIX APIs.
- Originally, NT was supposed to use the OS/2 API as its native environment but during development NT was changed to use the Win32 API, reflecting the popularity of Windows 3.0.

Design Principles

- Extensibility — layered architecture
 - Executive, which runs in protected mode, provides the basic system services.
 - On top of the executive, several server subsystems operate in user mode.
 - Modular structure allows additional environmental subsystems to be added without affecting the executive.
- Portability — 2000 can be moved from one hardware architecture to another with relatively few changes.
 - Written in C and C++
 - Processor-dependent code is isolated in a dynamic link library (DLL) called the “hardware abstraction layer” (HAL).

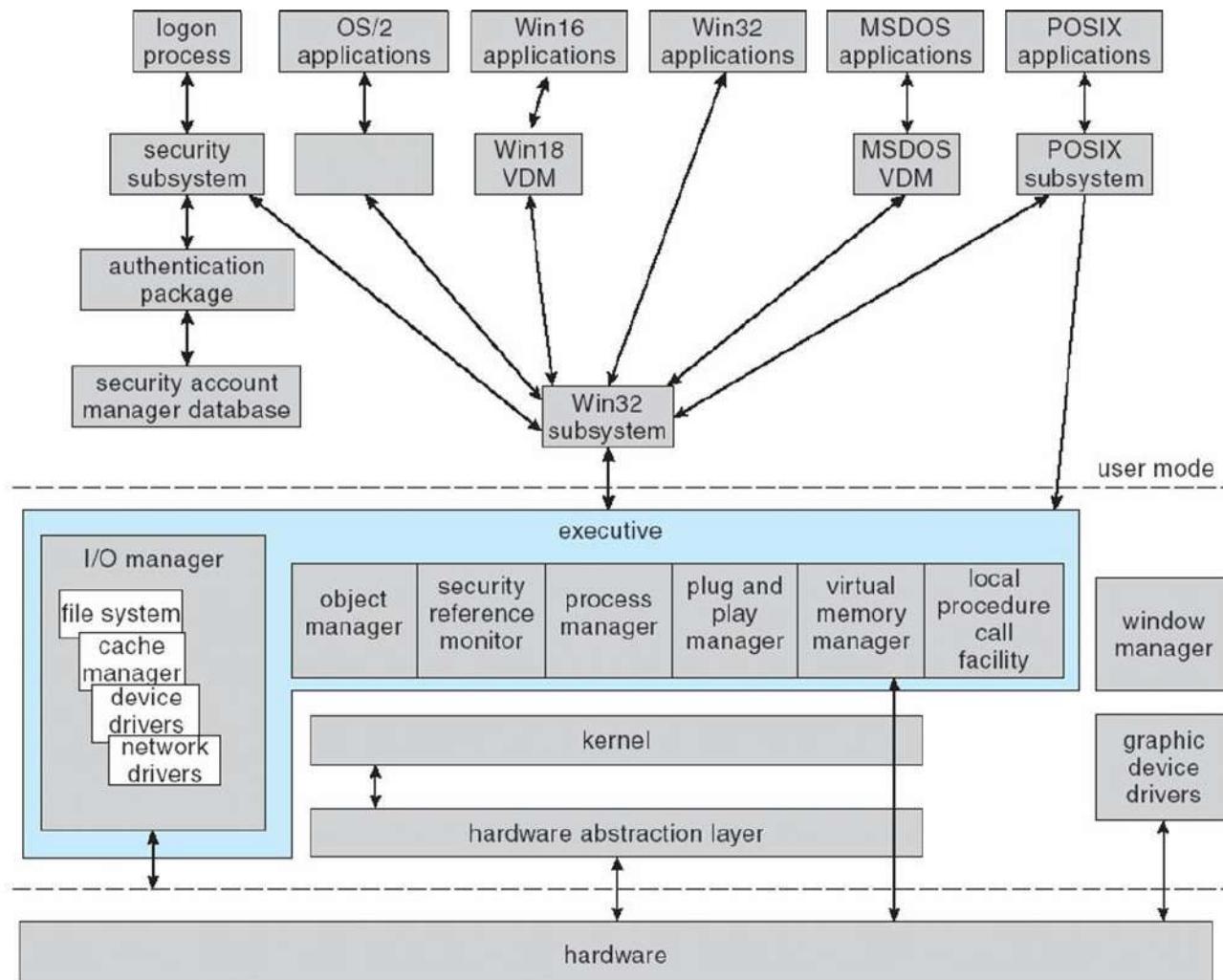
Design Principles (Cont.)

- Reliability — 2000 uses hardware protection for virtual memory, and software protection mechanisms for operating system resources.
- Compatibility — applications that follow the IEEE 1003.1 (POSIX) standard can be complied to run on 2000 without changing the source code.
- Performance — 2000 subsystems can communicate with one another via high-performance message passing
 - Preemption of low priority threads enables the system to respond quickly to external events
 - Designed for symmetrical multiprocessing
- International support — supports different locales via the national language support (NLS) API.

2000 Architecture

- Layered system of modules
- Protected mode — HAL, kernel, executive
- User mode — collection of subsystems
 - Environmental subsystems emulate different operating systems
 - Protection subsystems provide security functions

Depiction of 2000 Architecture



System Components — Kernel

- Foundation for the executive and the subsystems
- Never paged out of memory; execution is never preempted
- Four main responsibilities:
 - thread scheduling
 - interrupt and exception handling
 - low-level processor synchronization
 - recovery after a power failure
- Kernel is object-oriented, uses two sets of objects
 - **dispatcher objects** control dispatching and synchronization (events, mutants, mutexes, semaphores, threads and timers).
 - **control objects** (asynchronous procedure calls, interrupts, power notify, power status, process and profile objects)

Kernel — Process and Threads

- The process has a virtual memory address space, information (such as a base priority), and an affinity for one or more processors.
- Threads are the unit of execution scheduled by the kernel's dispatcher.
- Each thread has its own state, including a priority, processor affinity, and accounting information.
- A thread can be one of six states: *ready*, *standby*, *running*, *waiting*, *transition*, and *terminated*.

Kernel — Scheduling

- The dispatcher uses a 32-level priority scheme to determine the order of thread execution.
- Priorities are divided into two classes:
 - The real-time class contains threads with priorities ranging from 16 to 31
 - The variable class contains threads having priorities from 0 to 15
- Characteristics of 2000's priority strategy
 - Trends to give very good response times to interactive threads that are using the mouse and windows
 - Enables I/O-bound threads to keep the I/O devices busy
 - Complete-bound threads soak up the spare CPU cycles in the background.

Kernel – Scheduling (Cont.)

- Scheduling can occur when a thread enters the ready or wait state, when a thread terminates, or when an application changes a thread's priority or processor affinity.
- Real-time threads are given preferential access to the CPU; but 2000 does not guarantee that a real-time thread will start to execute within any particular time limit.
 - This is known as **soft realtime**.

Windows 2000 Interrupt Request Levels

interrupt levels	types of interrupts
31	machine check or bus error
30	power fail
29	interprocessor notification (request another processor to act; e.g., dispatch a process or update the TLB)
28	clock (used to keep track of time)
27	profile
3–26	traditional PC IRQ hardware interrupts
2	dispatch and deferred procedure call (DPC) (kernel)
1	asynchronous procedure call (APC)
0	passive

Kernel – Trap Handling

- The kernel provides trap handling when exceptions and interrupts are generated by hardware or software.
- Exceptions that cannot be handled by the trap handler are handled by the kernel's *exception dispatcher*.
- The interrupt dispatcher in the kernel handles interrupts by calling either an interrupt service routine (such as in a device driver) or an internal kernel routine.
- The kernel uses spin locks that reside in global memory to achieve multiprocessor mutual exclusion.

Executive — Object Manager

- 2000 uses objects for all its services and entities; the object manager supervises the use of all the objects.
 - Generates an object **handle**
 - Checks security
 - Keeps track of which processes are using each object
- Objects are manipulated by a standard set of methods, namely `create`, `open`, `close`, `delete`, `query name`, `parse` and `security`

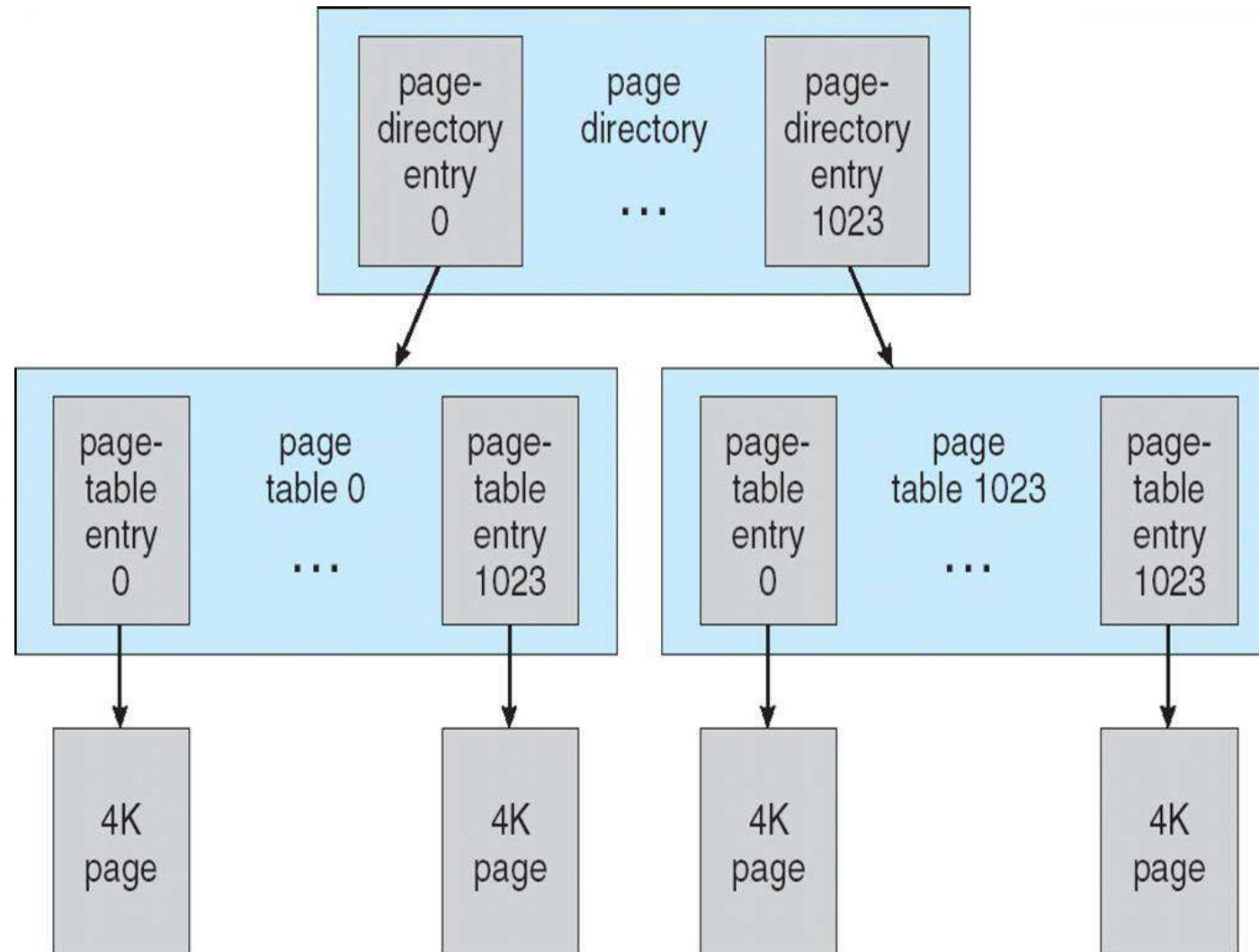
Executive – Naming Objects

- The 2000 executive allows any object to be given a name, which may be either permanent or temporary.
- Object names are structured like file path names in MS-DOS and UNIX.
- 2000 implements a **symbolic link object**, which is similar to **symbolic links** in UNIX that allow multiple nicknames or aliases to refer to the same file.
- A process gets an object handle by creating an object by opening an existing one, by receiving a duplicated handle from another process, or by inheriting a handle from a parent process.
- Each object is protected by an access control list.

Executive — Virtual Memory Manager

- The design of the VM manager assumes that the underlying hardware supports virtual to physical mapping a paging mechanism, transparent cache coherence on multiprocessor systems, and virtual addressing aliasing.
- The VM manager in 2000 uses a page-based management scheme with a page size of 4 KB.
- The 2000 VM manager uses a two step process to allocate memory:
 - The first step reserves a portion of the process's address space.
 - The second step commits the allocation by assigning space in the 2000 paging file.

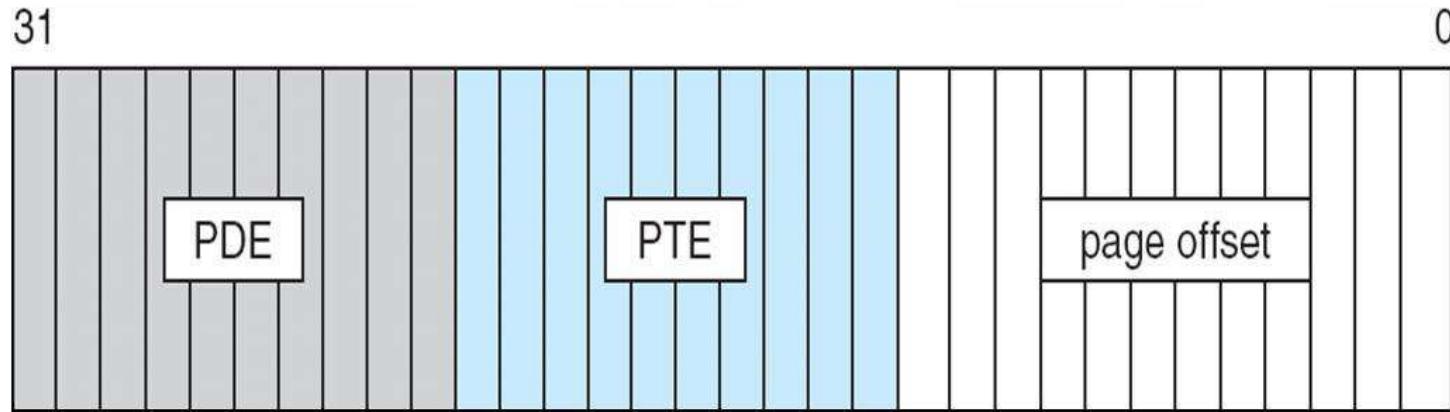
Virtual-Memory Layout



Virtual Memory Manager (Cont.)

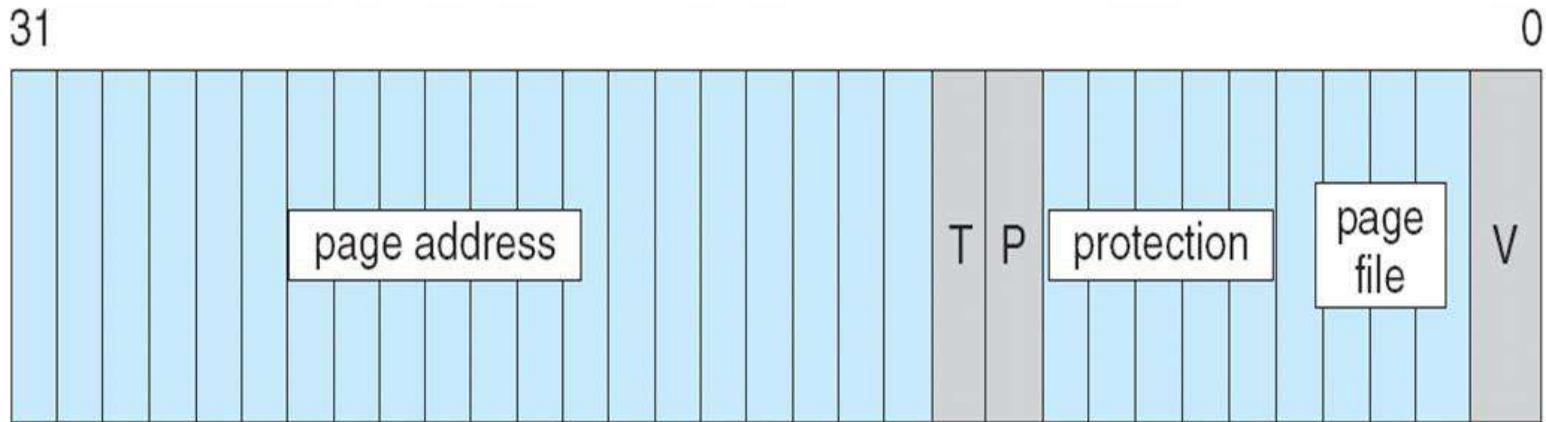
- The virtual address translation in 2000 uses several data structures:
 - Each process has a **page directory** that contains 1024 **page directory entries** of size 4 bytes
 - Each page directory entry points to a *page table* which contains 1024 **page table entries** (PTEs) of size 4 bytes
 - Each PTE points to a 4 KB **page frame** in physical memory
- A 10-bit integer can represent all the values from 0 to 1023, therefore, can select any entry in the page directory, or in a page table.
- This property is used when translating a virtual address pointer to a byte address in physical memory.
- A page can be in one of six states: *valid, zeroed, free standby, modified* and *bad*.

Virtual-to-Physical Address Translation



- 10 bits for page directory entry, 20 bits for page table entry, and 12 bits for byte offset in page

Page File Page-Table Entry



5 bits for page protection, 20 bits for page frame address, 4 bits to select a paging file, and 3 bits that describe the page state $V = 0$

Executive — Process Manager

- Provides services for creating, deleting, and using threads and processes
- Issues such as parent/child relationships or process hierarchies are left to the particular environmental subsystem that owns the process

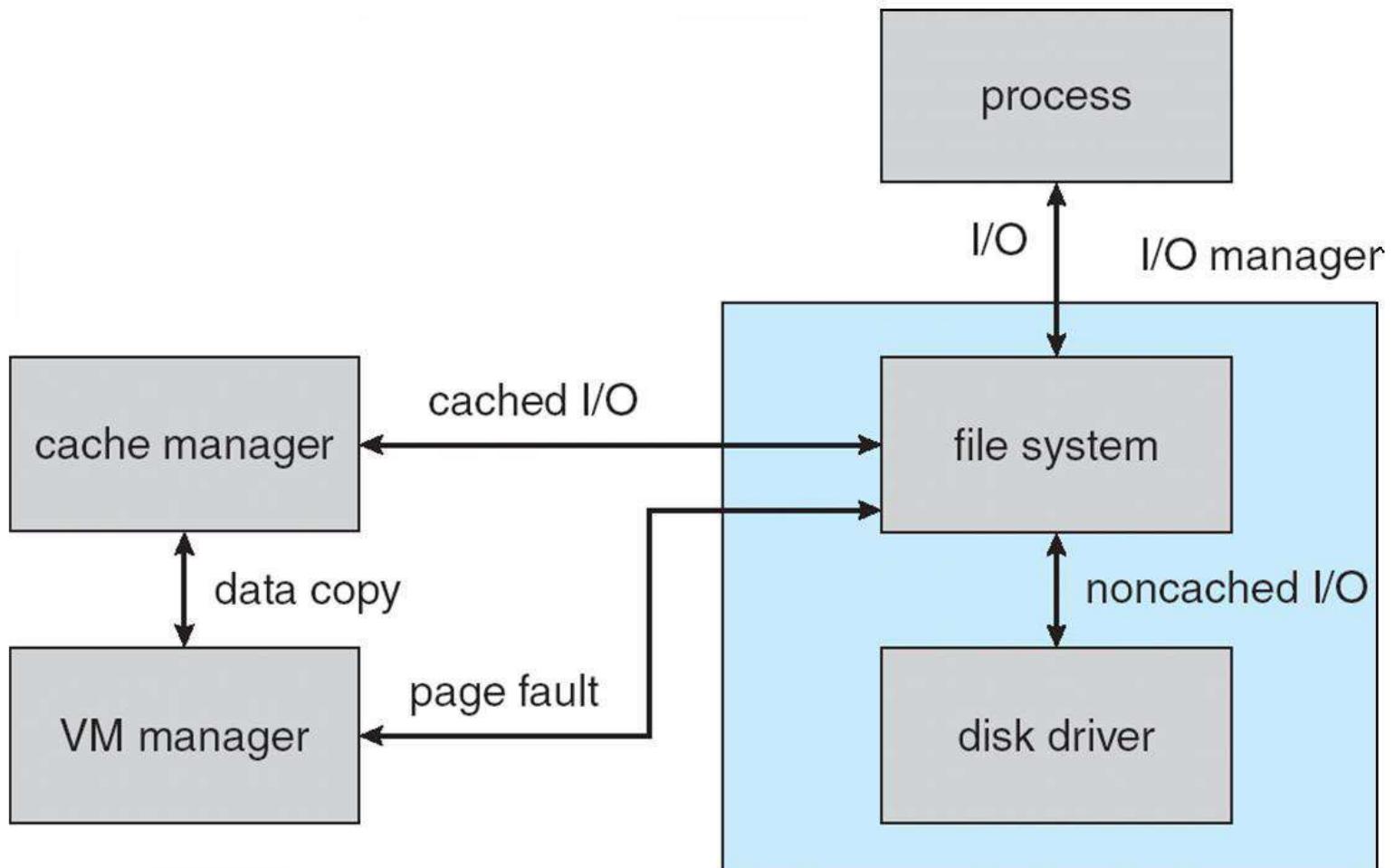
Executive — Local Procedure Call Facility

- The LPC passes requests and results between client and server processes within a single machine.
- In particular, it is used to request services from the various 2000 subsystems.
- When a LPC channel is created, one of three types of message passing techniques must be specified.
 - First type is suitable for small messages, up to 256 bytes; port's message queue is used as intermediate storage, and the messages are copied from one process to the other.
 - Second type avoids copying large messages by pointing to a shared memory section object created for the channel.
 - Third method, called *quick LPC* was used by graphical display portions of the Win32 subsystem.

Executive – I/O Manager

- The I/O manager is responsible for:
 - file systems
 - cache management
 - device drivers
 - network drivers
- Keeps track of which installable file systems are loaded, and manages buffers for I/O requests
- Works with VM Manager to provide memory-mapped file I/O
- Controls the 2000 cache manager, which handles caching for the entire I/O system
- Supports both synchronous and asynchronous operations, provides time outs for drivers, and has mechanisms for one driver to call another

File I/O



Executive — Security Reference Monitor

- The object-oriented nature of 2000 enables the use of a uniform mechanism to perform runtime access validation and audit checks for every entity in the system.
- Whenever a process opens a handle to an object, the security reference monitor checks the process's security token and the object's access control list to see whether the process has the necessary rights.

Executive – Plug-and-Play Manager

- Plug-and-Play (PnP) manager is used to recognize and adapt to changes in the hardware configuration.
- When new devices are added (for example, PCI or USB), the PnP manager loads the appropriate driver.
- The manager also keeps track of the resources used by each device.

Environmental Subsystems

- User-mode processes layered over the native 2000 executive services to enable 2000 to run programs developed for other operating system.
- 2000 uses the Win32 subsystem as the main operating environment; Win32 is used to start all processes.
 - It also provides all the keyboard, mouse and graphical display capabilities.
- MS-DOS environment is provided by a Win32 application called the *virtual dos machine* (VDM), a user-mode process that is paged and dispatched like any other 2000 thread.

Environmental Subsystems (Cont.)

- 16-Bit Windows Environment:
 - Provided by a VDM that incorporates *Windows on Windows*
 - Provides the Windows 3.1 kernel routines and sub routines for window manager and GDI functions
- The POSIX subsystem is designed to run POSIX applications following the POSIX.1 standard which is based on the UNIX model.

Environmental Subsystems (Cont.)

- OS/2 subsystems runs OS/2 applications.
- Logon and Security Subsystems authenticates users logging to Windows 2000 systems.
 - Users are required to have account names and passwords.
- The authentication package authenticates users whenever they attempt to access an object in the system.
 - Windows 2000 uses Kerberos as the default authentication package.

File System

- The fundamental structure of the 2000 file system (NTFS) is a *volume*.
 - Created by the 2000 disk administrator utility
 - Based on a logical disk partition
 - May occupy portions of a disk, an entire disk, or span across several disks
- All *metadata*, such as information about the volume, is stored in a regular file.
- NTFS uses *clusters* as the underlying unit of disk allocation.
 - A cluster is a number of disk sectors that is a power of two.
 - Because the cluster size is smaller than for the 16-bit FAT file system, the amount of internal fragmentation is reduced.

File System — Internal Layout

- NTFS uses logical cluster numbers (LCNs) as disk addresses.
- A file in NTFS is not a simple byte stream, as in MS-DOS or UNIX, rather, it is a structured object consisting of attributes.
- Every file in NTFS is described by one or more records in an array stored in a special file called the Master File Table (MFT).
- Each file on an NTFS volume has a unique ID called a file reference.
 - 64-bit quantity that consists of a 48-bit file number and a 16-bit sequence number
 - Can be used to perform internal consistency checks
- The NTFS name space is organized by a hierarchy of directories; the index root contains the top level of the B+ tree.

File System — Recovery

- All file system data structure updates are performed inside transactions that are logged.
 - Before a data structure is altered, the transaction writes a log record that contains redo and undo information.
 - After the data structure has been changed, a commit record is written to the log to signify that the transaction succeeded.
 - After a crash, the file system data structures can be restored to a consistent state by processing the log records.

File System — Recovery (Cont.)

- This scheme does not guarantee that all the user file data can be recovered after a crash, just that the file system data structures (the metadata files) are undamaged and reflect some consistent state prior to the crash.
- The log is stored in the third metadata file at the beginning of the volume.
- The logging functionality is provided by

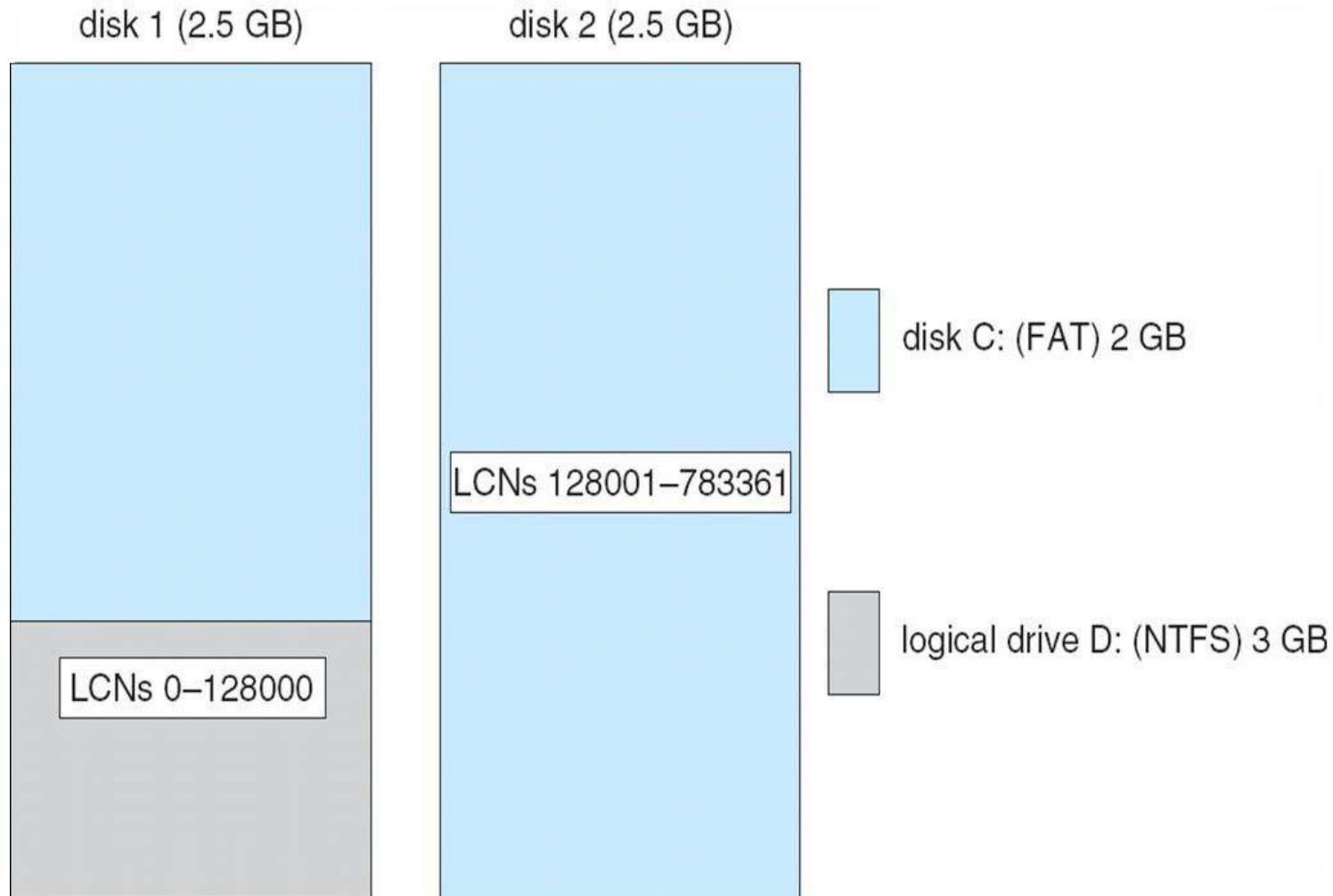
File System — Security

- Security of an NTFS volume is derived from the 2000 object model.
- Each file object has a security descriptor attribute stored in this MFT record.
- This attribute contains the access token of the owner of the file, and an access control list that states the access privileges that are granted to each user that has access to the file.

Volume Management and Fault Tolerance

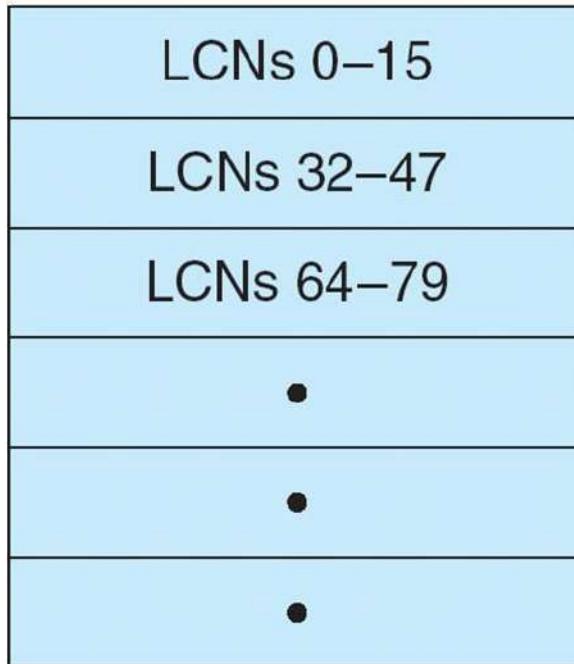
- FtDisk, the fault tolerant disk driver for 2000, provides several ways to combine multiple SCSI disk drives into one logical volume.
- Logically concatenate multiple disks to form a large logical volume, a **volume set**.
- Interleave multiple physical partitions in round-robin fashion to form a **stripe set** (also called RAID level 0, or “disk striping”)
 - Variation: **stripe set with parity**, or RAID level

Volume Set On Two Drives

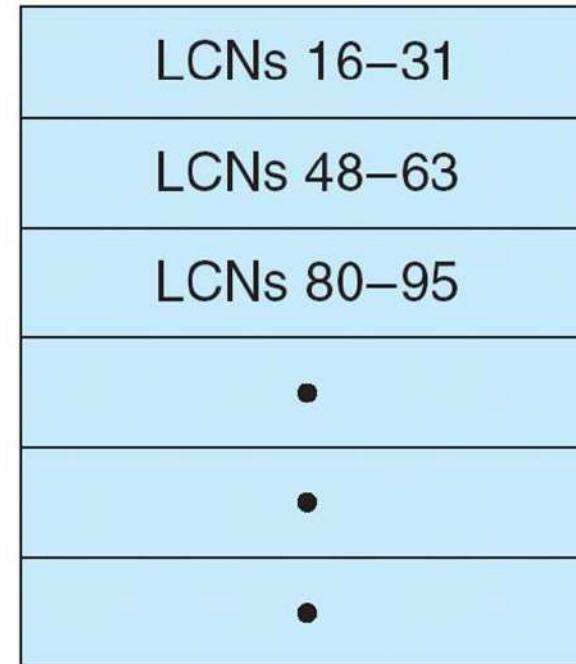


Stripe Set on Two Drives

disk 1 (2 GB)

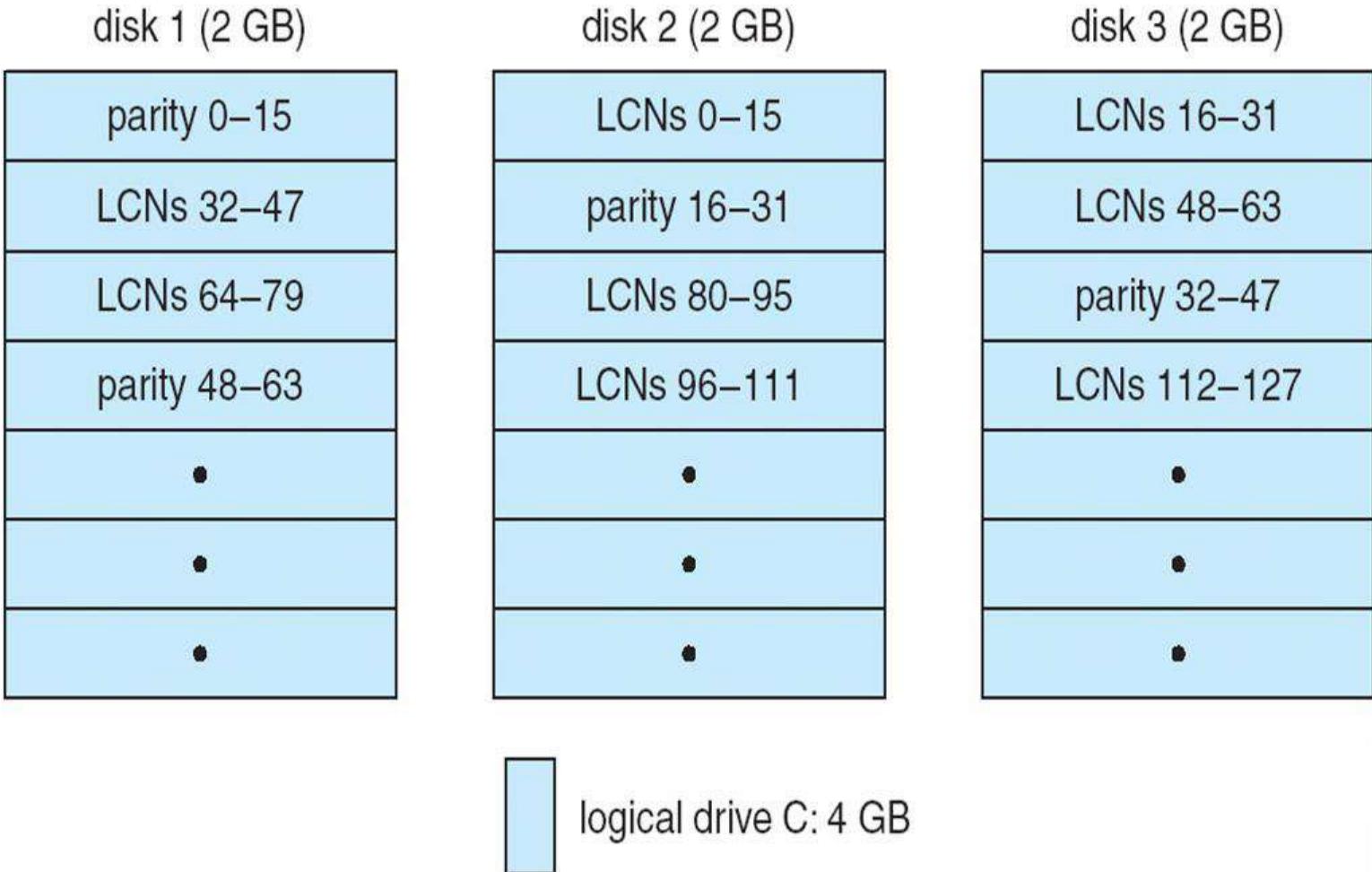


disk 2 (2 GB)

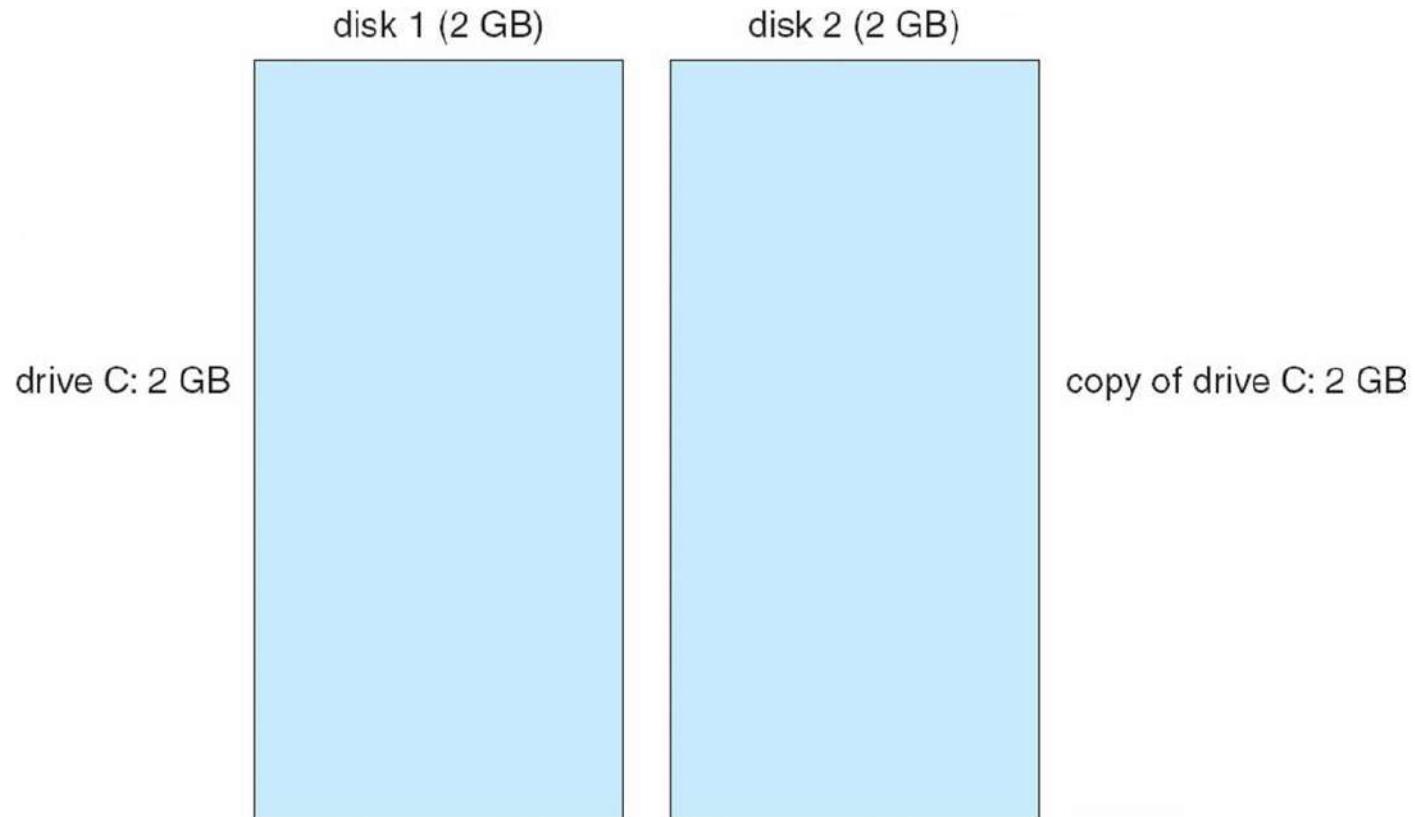


logical drive C: 4 GB

Stripe Set With Parity on Three Drives



Mirror Set on Two Drives



File System — Compression

- To compress a file, NTFS divides the file's data into *compression units*, which are blocks of 16 contiguous clusters.
- For sparse files, NTFS uses another technique to save space:
 - Clusters that contain all zeros are not actually allocated or stored on disk
 - Instead, gaps are left in the sequence of virtual cluster numbers stored in the MFT entry for the file

File System — Reparse Points

- A reparse point returns an error code when accessed. The reparse data tells the I/O manager what to do next.
- Reparse points can be used to provide the functionality of UNIX mounts.
- Reparse points can also be used to access files that have been moved to offline storage.

Networking

- 2000 supports both peer-to-peer and client/server networking; it also has facilities for network management.
- To describe networking in 2000, we refer to two of the internal networking interfaces:
 - NDIS (Network Device Interface Specification)
 - Separates network adapters from the transport protocols so that either can be changed without affecting the other.
 - TDI (Transport Driver Interface) — Enables any application to communicate with the network.

Networking — Protocols

- The server message block (SMB) protocol is used to send I/O requests over the network. It has four message types:
 - Session control
 - File
 - Printer
 - Message
- The network basic Input/Output system (NetBIOS) is a hardware abstraction interface for networks. Used to:

Networking — Protocols (Cont.)

- NetBEUI (NetBIOS Extended User Interface): default protocol for Windows 95 peer networking and Windows for Workgroups; used when 2000 wants to share resources with these networks.
- 2000 uses the TCP/IP Internet protocol to connect to a wide variety of operating systems and hardware platforms.
- PPTP (Point-to-Point Tunneling Protocol) is

Networking — Protocols (Cont.)

- The Data Link Control protocol (DLC) is used to access IBM mainframes and HP printers that are directly connected to the network.
- 2000 systems can communicate with Macintosh computers via the Apple Talk protocol if an 2000 Server on the network is running the Windows 2000 Services for Macintosh package.

Networking — Dist. Processing Mechanisms

- 2000 supports distributed applications via named NetBIOS, named pipes and mailslots, Windows Sockets, Remote Procedure Calls (RPC), and Network Dynamic Data Exchange (NetDDE).
- NetBIOS applications can communicate over the network using NetBEUI, NWLink, or TCP/IP.
- Named pipes are connection-oriented messaging mechanism that are named via

Distributed Processing Mechanisms (Cont.)

- The 2000 RPC mechanism follows the widely-used Distributed Computing Environment standard for RPC messages, so programs written to use 2000 RPCs are very portable.
 - RPC messages are sent using NetBIOS, or Winsock on TCP/IP networks, or named pipes on LAN Manager networks.
 - 2000 provides the Microsoft *Interface Definition Language* to describe the remote procedure names, arguments, and results.

Networking — Redirectors and Servers

- In 2000, an application can use the 2000 I/O API to access files from a remote computer as if they were local, provided that the remote computer is running an MS-NET server.
- A *redirector* is the client-side object that forwards I/O requests to remote files, where they are satisfied by a server.
- For performance and security, the

Access to a Remote File

- The application calls the I/O manager to request that a file be opened (we assume that the file name is in the standard UNC format).
- The I/O manager builds an I/O request packet.
- The I/O manager recognizes that the access is for a remote file, and calls a driver called a Multiple Universal Naming Convention Provider (MUP).
- The MUP sends the I/O request packet

Access to a Remote File (Cont.)

- The redirector sends the network request to the remote system.
- The remote system network drivers receive the request and pass it to the server driver.
- The server driver hands the request to the proper local file system driver.
- The proper device driver is called to access the data.
- The results are returned to the server driver which sends the data back to the

Networking — Domains

- NT uses the concept of a domain to manage global access rights within groups.
- A domain is a group of machines running NT server that share a common security policy and user database.
- 2000 provides three models of setting up trust relationships:
 - *One way, A trusts B*
 - *Two-way, transitive, A trusts B, B trusts C so A trusts C*

Name Resolution in TCP/IP Networks

- On an IP network, name resolution is the process of converting a computer name to an IP address
 - e.g., www.bell-labs.com resolves to 135.104.1.14
- 2000 provides several methods of name resolution:
 - Windows Internet Name Service (WINS)
 - broadcast name resolution
 - domain name system (DNS)

Name Resolution (Cont.)

- WINS consists of two or more WINS servers that maintain a dynamic database of name to IP address bindings, and client software to query the servers.
- WINS uses the Dynamic Host Configuration Protocol (DHCP), which automatically updates address configurations in the WINS database, without user or administrator intervention.

Programmer Interface – Access to Kernel Obj.

- A process gains access to a kernel object named XXX by calling the CreateXXX function to open a *handle* to XXX; the handle is unique to that process.
- A handle can be closed by calling the CloseHandle function; the system may delete the object if the count of processes using the object drops to 0.
- 2000 provides three ways to share objects between processes.

Programmer Interface – Process Management

- Process is started via the CreateProcess routine which loads any dynamic link libraries that are used by the process, and creates a *primary thread*.
- Additional threads can be created by the CreateThread function.
- Every dynamic link library or executable file that is loaded into the address space of a process is identified by an *instance handle*.

Process Management (Cont.)

- Scheduling in Win32 utilizes four priority classes:
 - IDLE_PRIORITY_CLASS (priority level 4)
 - NORMAL_PRIORITY_CLASS (level 8 – typical for most processes)
 - HIGH_PRIORITY_CLASS (level 13)
 - REALTIME_PRIORITY_CLASS (level 24)
- To provide performance levels needed for interactive programs, 2000 has a special scheduling rule for processes in the NORMAL_PRIORITY_CLASS

Process Management (Cont.)

- The kernel dynamically adjusts the priority of a thread depending on whether it is I/O-bound or CPU-bound.
- To synchronize the concurrent access to shared objects by threads, the kernel provides synchronization objects, such as semaphores and mutexes.
 - In addition, threads can synchronize by using the `WaitForSingleObject` or `WaitForMultipleObjects` functions.

Process Management (Cont.)

- A fiber is user-mode code that gets scheduled according to a user-defined scheduling algorithm.
 - Only one fiber at a time is permitted to execute, even on multiprocessor hardware.
 - 2000 includes fibers to facilitate the porting of legacy UNIX applications that are written for a fiber execution model.

Programmer Interface — Interprocess Comm.

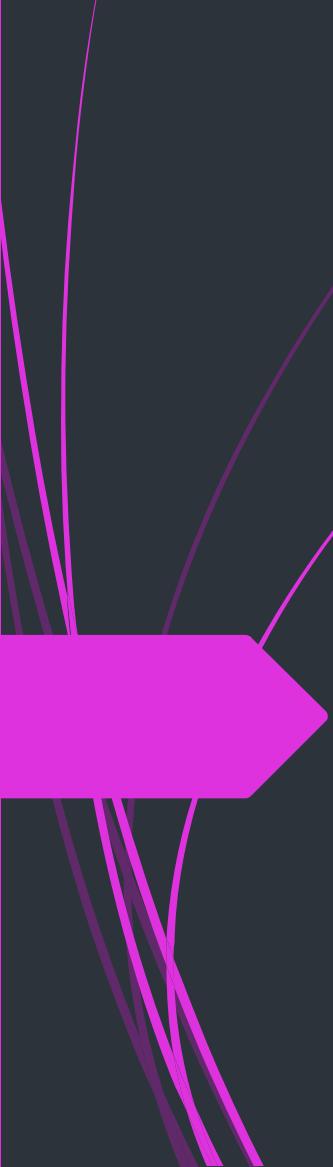
- Win32 applications can have interprocess communication by sharing kernel objects.
- An alternate means of interprocess communications is message passing, which is particularly popular for Windows GUI applications.
 - One thread sends a message to another thread or to a window
 - A thread can also send data with the message
- Every Win32 thread has its own input queue from which the thread receives messages.
- This is more reliable than the shared input queue of 16-bit windows, because with separate queues, one stuck application cannot block input to the other application

Programmer Interface – Memory Management

- Virtual memory:
 - VirtualAlloc reserves or commits virtual memory
 - VirtualFree decommits or releases the memory
 - These functions enable the application to determine the virtual address at which the memory is allocated.
- An application can use memory by memory mapping a file into its address space.
 - Multistage process
 - Two processes share memory by mapping the same file into their virtual memory.

Memory Management (Cont.)

- A heap in the Win32 environment is a region of reserved address space.
 - A Win 32 process is created with a 1 MB default heap.
 - Access is synchronized to protect the heap's space allocation data structures from damage by concurrent updates by multiple threads
- Because functions that rely on global or static data typically fail to work properly in a multithreaded environment, the thread-local storage mechanism allocates global storage on a per-thread basis.
 - The mechanism provides both dynamic and static methods of creating thread-local storage.



VIRTUAL MACHINES ON MULTICORE CPU'S

&

LICENSING ISSUES



VIRTUAL MACHINES

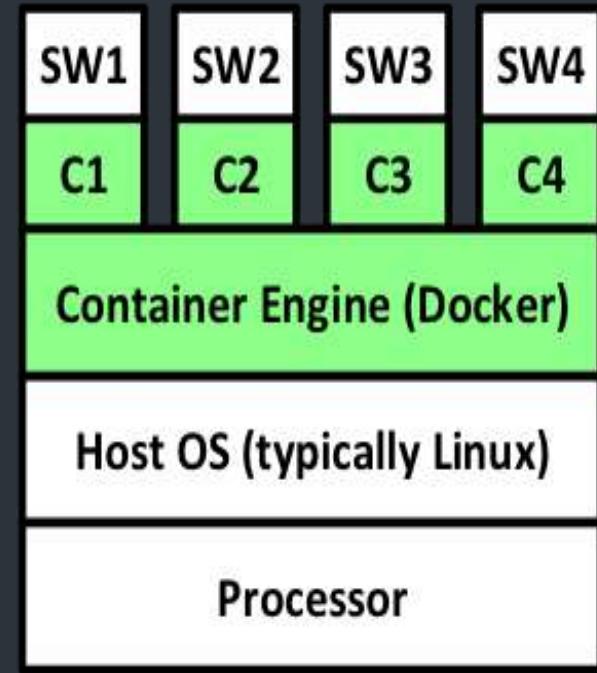
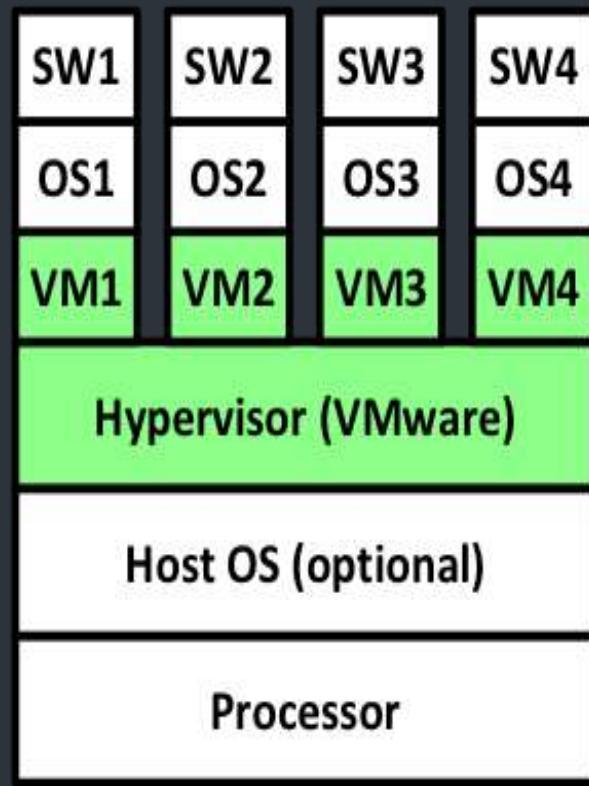
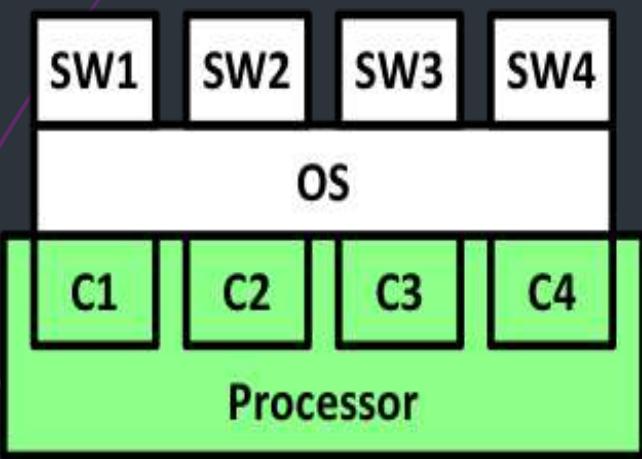
- ▶ A virtual machine is a program that acts as a virtual computer. It runs on your current operating system (the host operating system) and provides virtual hardware to guest operating systems. The guest OS runs in a window on your host OS, just like any other program on your computer.
- ▶ Virtual machines (VMs) allow you to run other operating systems within your current OS. The virtual OS will run as if it's just another program on your computer.

VIRTUAL MACHINES ON MULTICORE CPU

- Multicore processing and virtualization are rapidly becoming ubiquitous (appearing) in software development. They are widely used in the commercial world, especially in large data centers supporting cloud-based computing to
 - i. isolate application software from hardware and operating systems
 - ii. decrease hardware costs by enabling different applications to share underutilized (unused) computers or processors
 - iii. enhance scalability and responsiveness through the use of actual and virtual concurrency in architectures



► The combination of virtual machines and multicore CPUs opens a whole new world in which the number of CPUs available can be set in software. If there are, say, four cores, and each one can be used to run, for example, up to eight virtual machines, a single (desktop) CPU can be configured as a 32-node multicomputer if need be, but it can also have fewer CPUs, depending on the needs of the software. Never before has it been possible for an application designer to first choose how many CPUs he wants and then write the software accordingly. This clearly represents a new phase in computing.

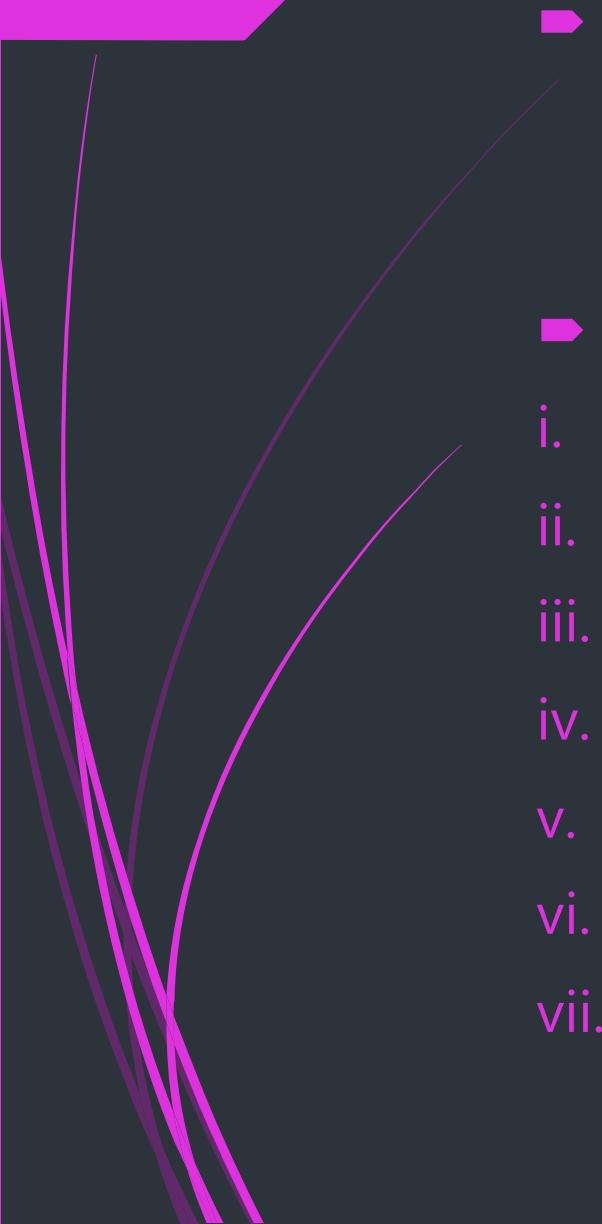


- 
- Multicore processing provides *multiple real hardware platforms*. Virtualization by VMs sits on top of multicore processing and provides *multiple virtual hardware platforms*. Virtualization by containers can sit on top of virtual machines where it can provide *multiple virtual operating systems*.
 - i. Multiple software applications can be allocated to a single container.
 - ii. Multiple containers can be allocated to a single virtual machine.
 - iii. Multiple virtual machines can be allocated to a single core.
 - iv. Multiple cores are contained by a single multicore processor.



LICENSING ISSUES ON VIRTUAL MACHINES

- ▶ Software developers make a profit by selling us the best products they can create. When selling pieces of their software in bulk, they offer licensing packages, so we don't have to buy multiple copies of it and face problems.
- ▶ Most software is licensed on a per-CPU basis. In other words, when you buy a program, you have the right to run it on just one CPU.
- ▶ There are a couple of different issues that can make licensing difficult in a virtualized environment. One such issue is that the organization may have to purchase several different license types in order to remain compliant.

- 
- ▶ Another issue that can complicate licensing is that virtualized environments tend to be highly dynamic. A guest operating system might be created on one host server and then immediately migrated to a different host.
 - ▶ Some of the required license types are:
 - i. Hypervisor License
 - ii. Management server licenses
 - iii. Managed server licenses
 - iv. Guest OS server licenses
 - v. Guest OS client access licenses
 - vi. Application licenses
 - vii. Application client access licenses

- 
- ▶ For example,

Suppose that you wanted to run Windows Server 2012 R2 on top of VMware. You would obviously have to license the VMware hypervisor, but you would also have to purchase a Windows Server license (typically the best option for all but the smallest virtualized environments is Datacenter Edition).

- ▶ Licensing can be complicated in virtualized environments. As you review the license requirements for your organization, be sure to check the requirements for your library servers to see if there are any licenses required for software installed within an image that is used to generate new VMs.

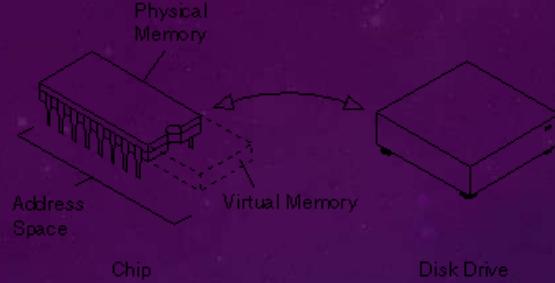
What is virtualization ?

Virtualization can be defined as one physical computer performing the same function as multiple computers. The process of separating the software layer of a computer or server from the hardware layer of a computer or server.

WHAT IS VIRTUAL MEMORY OR MEMORY VIRTUALIZATION?

Virtual memory is a memory management capability of an operating system (OS) that uses hardware and software to allow a computer to compensate for physical memory shortages by temporarily transferring data from random access memory (RAM) to disk storage.

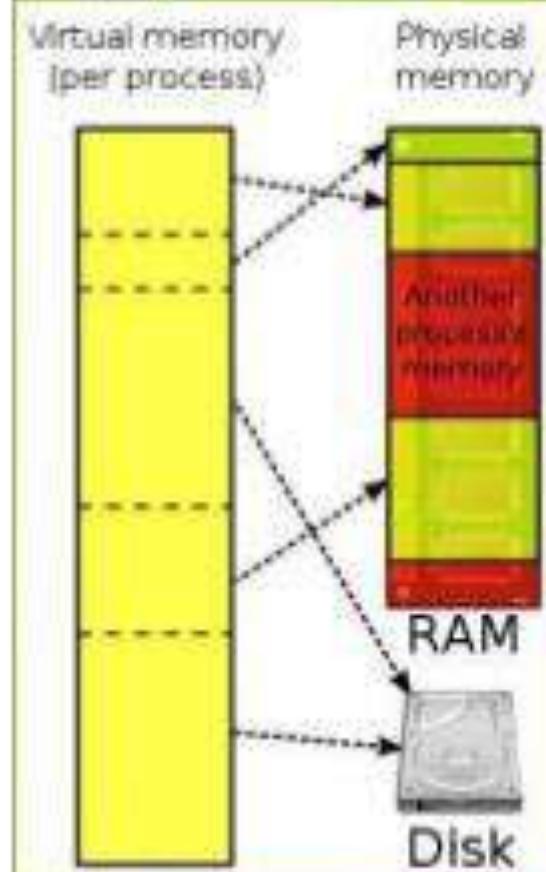
While copying virtual memory into physical memory, the OS divides memory into page files or swap files with a fixed number of addresses. Each page is stored on a disk and when the page is needed, the OS copies it from the disk to main memory and translates the virtual addresses into real addresses.



- * The purpose of virtual memory is to enlarge the address space, the set of addresses a program can utilize.
- * You can think of virtual memory as an alternate set of memory addresses. Programs use these *virtual addresses* rather than real addresses to store instructions and data. When the program is actually executed, the virtual addresses are converted into real memory addresses.
- * To facilitate copying virtual memory into real memory, the operating system divides virtual memory into pages, each of which contains a fixed number of addresses. Each page is stored on a disk until it is needed. When the page is needed, the operating system copies it from disk to main memory, translating the virtual addresses into real addresses.
- * The process of translating virtual addresses into real addresses is called *mapping*. The copying of virtual pages from disk to main memory is known as paging or swapping.

Virtual Memory

- ▶ Main memory and virtual memory are divided into equal sized pages.
- ▶ The entire address space required by a process need not be in memory at once. Some parts can be on disk, while others are in main memory.
- ▶ Further, the pages allocated to a process do not need to be stored contiguously-- either on disk or in memory.
- ▶ In this way, only the needed pages are in memory at any time, the unnecessary pages are in slower disk storage.



Importance of Virtual Memory

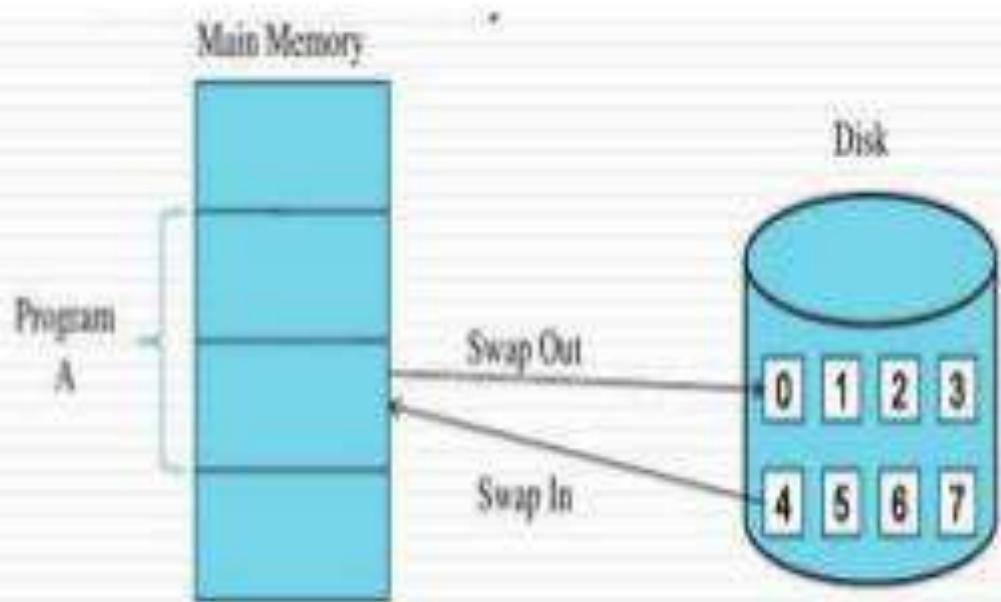
- ▶ When the computer runs out of physical memory it writes what it needs to remember to the hard disk in a swap file as virtual memory.
- ▶ If a computer running windows requires more memory or RAM, then it is installed in the system to run a program , it uses a small section of hard drive for this purpose.



Demand Paged Virtual Memory

- ▶ The fundamental approach in implementing virtual memory is paging.
- ▶ To facilitate copying virtual memory into real memory, the operating system divides virtual memory into *pages*, each of which contains a fixed number of addresses.
- ▶ To accomplish this, the virtual address is divided into two fields: A *page* field, and an *offset* field.
- ▶ The page field determines the page location of the address, and the offset indicates the location of the address within the page.

Demand Paged Virtual Memory



- ▶ When the pages are needed to execute a particular program, they are loaded.
- ▶ Pages that are never accessed are thus never loaded into the memory.
- ▶ This technique is known as Demand paging.

I/o virtualization

Input/output (I/O) virtualization is a methodology to simplify management, lower costs and improve performance of servers in enterprise environments. I/O virtualization environments are created by abstracting the upper layer protocols from the physical connections.^[1]

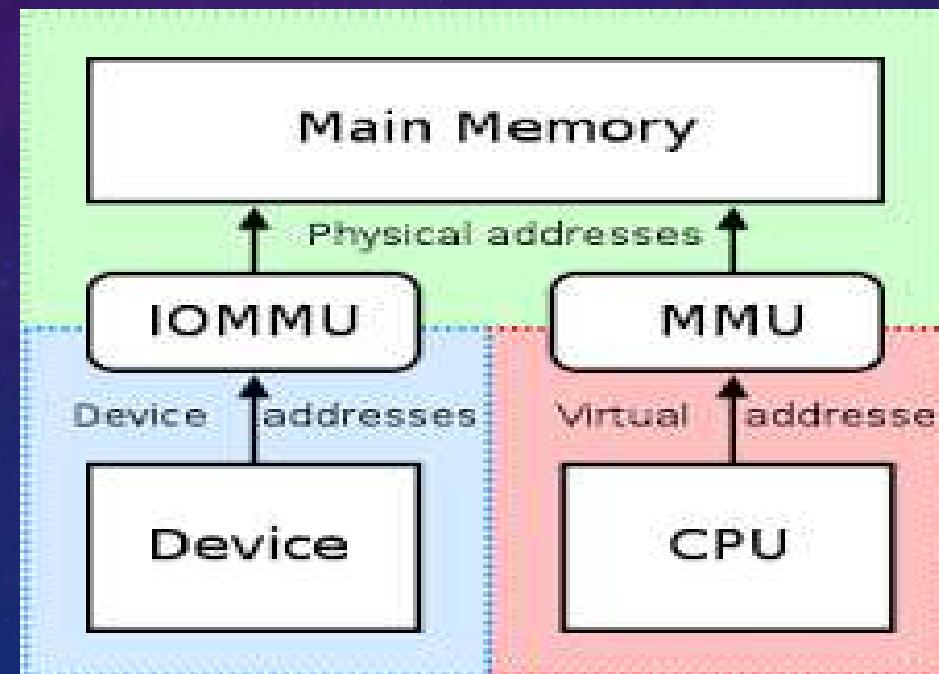
There are three kinds of I/o virtualization probs we can deal with

I/o MMU

DEVICE DOMAINS

SINGLE I/O VIRTUALIZATION

I/O MMUs Another I/O problem that must be solved somehow is the use of DMA, which uses absolute memory addresses. As might be expected, the hypervisor has to intervene here and remap the addresses before the DMA starts. However, hardware already exists with an I/O MMU, which virtualizes the I/O the same way the MMU virtualizes the memory. I/O MMU exists in different forms and shapes for many processor architectures. Even if we limit ourselves to the x86, Intel and AMD have slightly different technology. Still, the idea is the same. This hardware eliminates the DMA problem



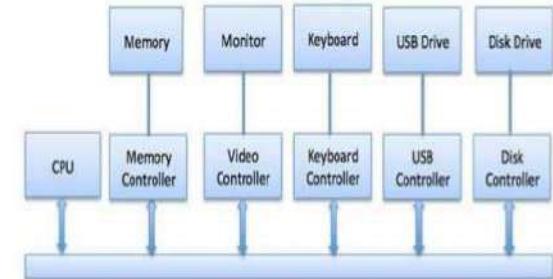
Direct Memory Access (DMA)

- ▶ Slow devices like keyboards will generate an interrupt to the main CPU after each byte is transferred. If a fast device such as a disk generated an interrupt for each byte, the operating system would spend most of its time handling these interrupts. So a typical computer uses direct memory access (DMA) hardware to reduce this overhead.
- ▶ Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module itself controls exchange of data between main memory and the I/O device. CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred.
- ▶ Direct Memory Access needs a special hardware called DMA controller (DMAC) that manages the data transfers and arbitrates access to the system bus. The controllers are programmed with source and destination pointers (where to read/write the data), counters to track the number of transferred bytes, and settings, which includes I/O and memory types, interrupts and states for the CPU cycles.

Device Domains

A different approach to handling I/O is to dedicate one of the virtual machines to run a standard operating system and reflect all I/O calls from the other ones to it. This approach is enhanced when paravirtualization is used, so the command being issued to the hypervisor actually says what the guest OS wants (e.g., read block 1403 from disk 1) rather than being a series of commands writing to device registers, in which case the hypervisor has to play Sherlock Holmes and figure out what it is trying to do.

Diagram:

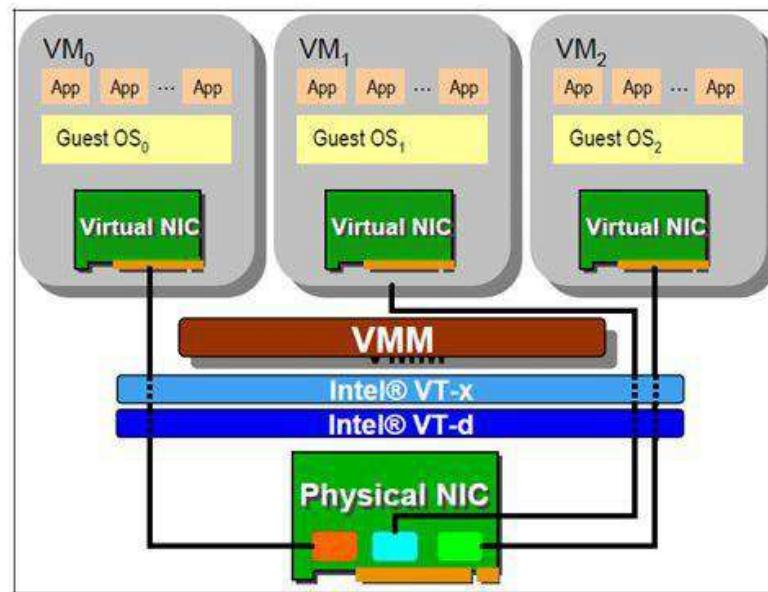


SINGLE ROOT I/O VIRTUALIZATION

SR-IOV is a specification that allows a single Peripheral Component Interconnect Express (PCIe) physical device under a single root port to appear to be multiple separate physical devices to the hypervisor or the guest operating system.

Single Root – IO Virtualization

- New industrial standard
 - Instead of implementing virtualization in CPU or memory only, industry comes up with new IO virtualization standard in PCI Express devices.
 - Advantages
 - Fully collaboration with physical hardware devices.
 - Improve system scalability.
 - Improve system agility.
 - Disadvantages
 - IO devices must implement with new specification.



SR-IOV Overview

- There are two kinds of functions:
 - Physical functions (PFs)
 - Have full configuration resources such as discovery, management and manipulation
 - Virtual functions (VFs)
 - Viewed as “light weighted” PCIe function.
 - Have only the ability to move data in and out of devices.
- SR-IOV specification shows that each device can have up to 256 VFs.

TOPIC:

VIRTUAL APPLIANCES AND VIRTUAL APPLIANCES ON MULTICORE CPU'S

WHAT IS AN APPLIANCE ?

- An appliance is a device or piece of equipment which is designed to perform a specific task.

VIRTUAL APPLIANCES

1. WHAT IS A VIRTUAL APPLIANCE ?

A VIRTUAL APPLIANCE IS A PRE-CONFIGURED VIRTUAL MACHINE IMAGE/SOFTWARE WHICH IS READY TO RUN ON A HYPERVISOR. VIRTUAL APPLIANCES ARE A SUBSET OF THE LARGER CLASS OF SOFTWARE APPLIANCES. INSTALLATION OF A SOFTWARE APPLIANCE ON A VIRTUAL MACHINE AND PACKAGING THAT INTO AN IMAGE CREATES A VIRTUAL APPLIANCE.

THE NEED FOR A VIRTUAL APPLIANCE

Virtual machines offer an interesting solution to a problem that has been hindering users , especially users of open source software.

The problem is that many applications are dependant on numerous other applications and libraries , which are themselves dependant on a host of other software packages.

USE OF A VIRTUAL APPLIANCE

By using an application as a virtual appliance we can eliminate problems with installation and configuration, such as software or driver compatibility issues.

With virtual machines now available , a software developer can carefully construct a virtual appliance , load it with the required operating system , compilers , libraries and application code , and make them ready to run.

After the developer is finished with developing of the virtual machine image , it can now be put in a CD –ROM or a website for download. This approach means that only the developer has to understand all the dependencies and the customer gets a complete package that works independently on any operating system that the user desires.

ADVANTAGES OF VIRTUAL APPLIANCES

- Can use multiple operating system environments on the same computer.
- When you create your virtual machine, you create a virtual hard disk. So, everything on that machine can crash, but if it does, it won't affect the host machine.

DISADVANTAGES OF VIRTUAL APPLIANCES.

- ▶ Virtual machines are less efficient than real machines because they access the hardware indirectly.
- ▶ When several virtual machines are running on the same host, performance may be hindered if the computer is lacking sufficient power.

VIRTUAL APPLIANCES ON MULTI-CORE CPU'S

What is a multi-core processor ?

A multi-core processor is a computer processor with two or more separate processing units, called cores, each of which reads and executes program instructions, as if the computer had several processors.

There are different types based on the number of cores present.

1. Dual Core (2 cores)
2. Quad Core (4 cores)
3. Hexa-Core (6 cores)
4. Octa-Core (8 cores)

THE NEED FOR VIRTUALISATION IN MULTI-CORE CPU'S

- ▶ The combination of virtual machines and multi-core CPU's have created a whole new world where , the number of CPU's available can be set by a software and this is possible only due to the implementation of virtualisation in multi core CPU's.
- ▶ It was never possible for an application designer to first choose how many CPU's he wants and write the software accordingly.

ABOUT:

- Virtual machines can share memory. Memory sharing is already available in de-duplication
- De-duplication refers to the process of not storing the same data twice.
- If virtual machines can share memory , then a single computer becomes a virtual multi-processor. Since all the cores in the processor share the same RAM , a single quad core chip could be easily configured as a 32-node multiprocessor.

ADVANTAGES OF IMPLEMENTING VIRTUALISATION ON MULTI-CORE CPU'S

- ▶ Multiple software applications can be allocated to a single container.
- ▶ Multiple virtual machines can be allocated to a single core.
- ▶ Multiple cores are contained by a single multicore processor.

DISADVANTAGES OF IMPLEMENTING VIRTUALISATION ON MULTI-CORE CPU'S

- ▶ It adds more complexity to the architecture which affects the performance , security and safety.
- ▶ Overheads may increase significantly.
- ▶ System becomes slower and less deterministic.

Classic synchronization problems –

Dining Philosophers Problem
Readers Writers' Problem
and
its programming solutions

Readers-Writers Problem

- Suppose that a database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- We distinguish between these two types of processes by referring to the former as readers and to the latter as writers.
- If two readers access the shared data simultaneously, no adverse effects will result.
- However, if a writer and some other process (either a reader or a writer) access the database simultaneously, conflict may arise

Readers-Writers Problem

- To ensure that these difficulties do not arise,
- The writers have exclusive access to the shared database while writing to the database.
- This synchronization problem is referred to as the readers-writers problem.

Readers-Writers Problem Variations

- The simplest one, referred to as the first readers-writers problem,
- Requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object.
- In other words, no reader should wait for other readers to finish simply because a writer is waiting.

Readers-Writers Problem Variations

- The second readers writers problem requires that, once a writer is ready, that writer performs its write as soon as possible.
- In other words, if a writer is waiting to access the object, no new readers may start reading.
- A solution to either problem may result in starvation.
- In the first case, writers may starve in the second case, readers may starve.

Solution to Readers-Writers Problem

```
semaphore mutex, wrt;  
int readcount;
```

The semaphores **mutex** and **wrt** are initialized to 1;
readcount is initialized to 0.

The semaphore **wrt** is common to both reader and writer processes.
The **mutex** semaphore is used to ensure mutual exclusion when the variable **readcount** is updated.

The **readcount** variable keeps track of how many processes are currently reading the object.

The semaphore **wrt** functions as a mutual-exclusion semaphore for the writers.

It is also used by the first or last reader that enters or exits the critical section.

Solution to Readers-Writers Problem

The structure of a reader process

- The structure of a writer process

```
do {  
    wait (wrt) ;  
  
    // writing is  
    performed  
  
    signal (wrt) ;  
} while (TRUE);
```

```
do {  
    wait (mutex) ;  
    readcount ++ ;  
    if (readcount == 1)  
        wait (wrt) ;  
    signal (mutex)  
  
    // reading is performed  
  
    wait (mutex) ;  
    readcount - - ;  
    if (readcount == 0)  
        signal (wrt) ;  
    signal (mutex) ;  
} while (TRUE);
```

Solution to Readers-Writers Problem

- Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and $n - 1$ readers are queued on mutex.
- Also observe that, when a writer executes signal (wrt), we may resume the execution of either the waiting readers or a single waiting writer.
- The selection is made by the scheduler.

Dining-Philosophers Problem



- Consider five philosophers who spend their lives thinking and eating.
- The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
- In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.
- When a philosopher thinks, he does not interact with his colleagues.

From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to him (the chopsticks that are between him and his left and right neighbors).

A philosopher may pick up only one chopstick at a time.

Obviously, he can not pick up a chopstick that is already in the hand of a neighbor.

When a hungry philosopher has both his chopsticks at the same time, he eats without releasing his chopsticks.

When he is finished eating, he puts down both of his chopsticks and starts thinking again

Dining-Philosophers Problem

- The dining-philosophers problem is considered a classic synchronization problem
- Because it is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.
- One simple solution is to represent each chopstick with a semaphore.
- A philosopher tries to grab a chopstick by executing await () operation on that semaphore;
- He releases his chopsticks by executing the signal() operation on the appropriate semaphores.
- The shared data - semaphore chopstick[5];
- Where all the elements of chopstick are initialized to 1.

Dining-Philosophers Problem

- The structure of philosopher as is shown in Figure
- Although this solution guarantees that no two neighbors are eating simultaneously,
- It nevertheless must be rejected because it could create a deadlock.
- Suppose that all five philosophers become hungry simultaneously and each grabs his left chopstick.
- All the elements of chopstick will now be equal to 0.
- When each philosopher tries to grab his right chopstick, he will be delayed forever

Dining-Philosophers Problem

Algorithm

- The structure of Philosopher *i*:

```
do {  
    wait ( chopstick[i] );  
    wait ( chopStick[ (i + 1) % 5] );  
  
    // eat  
  
    signal ( chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
  
    // think  
  
} while (TRUE);
```

- What is the problem with this algorithm?

Remedies

- Several possible remedies to the deadlock problem are listed next.
-
- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up his chopsticks only if both chopsticks are available
- Use an asymmetric solution;
- that is, an odd philosopher picks up first his left chopstick and then his right chopstick,
- whereas an even philosopher picks up his right chopstick and then his left chopstick

Monitor

- A abstract data type- or ADT- encapsulates private data with public methods to operate on that data.
- A monitor type is an ADT which presents a set of programmer-defined operations that are provided mutual exclusion within the monitor.
- The monitor type also contains the declaration of variables whose values define the state of an instance of that type,
- Along with the bodies of procedures or functions that operate on those variables.

Monitors

- A **high-level abstraction that provides a convenient and effective mechanism for process synchronization**
- ***Abstract data type*, internal variables only accessible by code within the procedure**
- **Only one process may be active within the monitor at a time**
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { .... }
    procedure Pn (...) {.....}
```

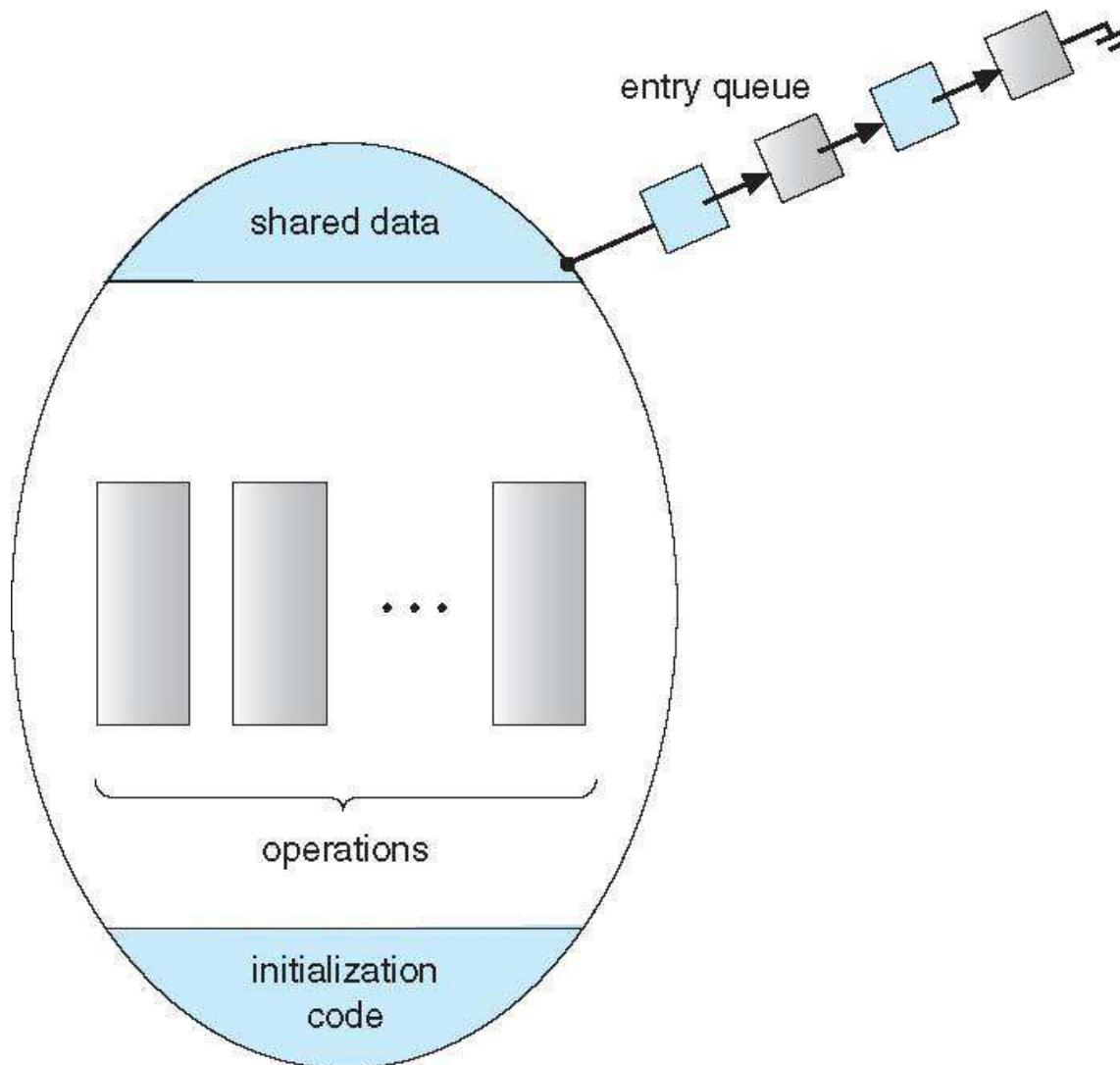
```
Initialization code (...) { ... }
```

```
}
```

Monitors

- The syntax of a monitor type is shown in Figure.
- The representation of a monitor type cannot be used directly by the various processes.
- Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- Similarly, the local variables of a monitor can be accessed by only the local procedures.

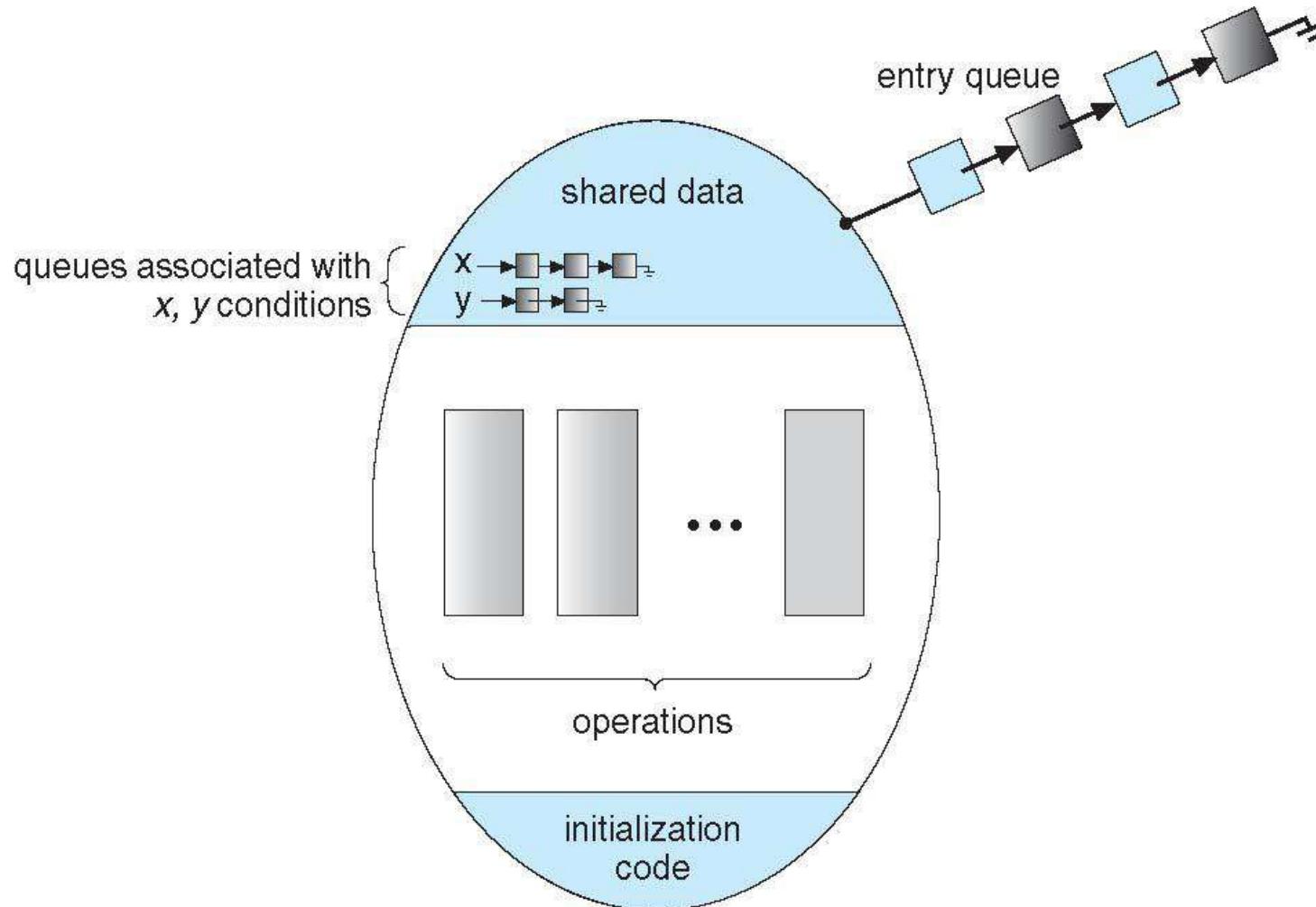
Schematic view of a Monitor



Condition Variables

- condition x, y;
- Two operations on a condition variable:
 - `x.wait ()` – a process that invokes the operation is suspended until `x.signal ()`
 - `x.signal ()` – resumes one of processes (if any) that invoked `x.wait ()`
 - If no `x.wait ()` on the variable, then it has no effect on the variable

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes `x.signal ()`, with Q in `x.wait ()` state, what should happen next?
 - If Q is resumed, then P must wait
- Options include
 - **Signal and wait** – P waits until Q leaves monitor or waits for another condition
 - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition
 - Both have pros and cons – language implementer can decide
 - Monitors implemented in Concurrent Pascal compromise
 - P executing signal immediately leaves the monitor, Q is resumed
 - Implemented in other languages including Mesa, C#, Java

Dining-Philosophers Solution Using Monitors

- Next, we illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem.
- **This solution imposes the restriction that a philosopher may pick up his chopsticks only if both of them are available.**
- To code this solution, we need to distinguish among three states in which we may find a philosopher.
- For this purpose, we introduce the following data structure:
- `enum{THINKING, HUNGRY, EATING}state[5];`

Dining-Philosophers Solution Using Monitors

- Philosopher i can set the variable state $[i] = \text{EATING}$ only if her two neighbors are not eating: $(\text{state}[(i+4) \% 5] \neq \text{EATING})$ and $(\text{state}[(i+1) \% 5] \neq \text{EATING})$.
- We also need to declare condition $\text{self}[5]$;
- In which philosopher i can delay himself when he is hungry but is unable to obtain the chopsticks he needs.
- The distribution of the chopsticks is controlled by the monitor Dining Philosophers, whose definition is shown in Figure

Solution to Dining Philosophers

monitor DiningPhilosophers

```
{  
    enum { THINKING, HUNGRY, EATING} state [5] ;  
    condition self [5];
```

```
void pickup (int i) {  
    state[i] = HUNGRY;  
    test(i);  
    if (state[i] != EATING) self [i].wait;  
}
```

```
void putdown (int i) {  
    state[i] = THINKING;  
    // test left and right neighbors  
    test((i + 4) % 5);  
    test((i + 1) % 5);  
}
```

Solution to Dining Philosophers

```
void test (int i) {  
    if ( (state[(i + 4) % 5] != EATING) &&  
        (state[i] == HUNGRY) &&  
        (state[(i + 1) % 5] != EATING) ) {  
        state[i] = EATING ;  
        self[i].signal () ;  
    }  
}
```

```
initialization_code() {  
    for (int i = 0; i < 5; i++)  
        state[i] = THINKING;  
}
```

Solution to Dining Philosophers

- Each philosopher i invokes the operations `pickup()` and `putdown()` in the following sequence:

`DiningPhilosophers.pickup (i);`

`EAT`

`DiningPhilosophers.putdown (i);`

- No deadlock, but starvation is possible

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex; // (initially = 1)  
semaphore next; // (initially = 0)  
int next-count = 0;
```

- Each procedure *F* will be replaced by

```
wait(mutex);  
...  
body of F;  
...  
if (next_count > 0)  
    signal(next)  
else  
    signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Monitor Implementation – Condition Variables

- For each condition variable **x**, we have:

```
semaphore x_sem; // (initially = 0)  
int x-count = 0;
```

- The operation **x.wait** can be implemented as:

```
x-count++;  
if (next_count > 0)  
    signal(next);  
else  
    signal(mutex);  
wait(x_sem);  
x-count--;
```

Monitor Implementation

- The operation `x.signal` can be implemented as:

```
if (x-count > 0) {  
    next_count++;  
    signal(x_sem);  
    wait(next);  
    next_count--;  
}
```

Resuming Processes within a Monitor

- If several processes queued on condition x , and $x.signal()$ executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form
 $x.wait(c)$
 - Where c is **priority number**
 - Process with lowest number (highest priority) is scheduled next

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }
    void release() {
        busy = FALSE;
        x.signal();
    }
    initialization code() {
        busy = FALSE;
    }
}
```