**Chapter 1: Introduction**

1. An operating system is software that manages a computer’s hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware.
2. The hardware— the central processing unit (CPU), the memory, and the input/output (I/O) devices—provides the basic computing resources for the system.
3. The application programs—such as word processors, spreadsheets,compilers, and web browsers—define the ways in which these resources are used to solve users’ computing problems.
4. The user’s view of the computer varies according to the interface being used.
5. From the computer’s point of view, the operating system is the program most

intimately involved with the hardware. In this context, we can view an operating

system as a resource allocator.

1. The operating system is the one program running at all times on the computer, usually called the kernel.

Along with the kernel, there are two other types of programs: system

programs, which are associated with the operating system but are not necessarily part of the kernel, and application programs, which include all programs

not associated with the operation of the system.

1. Bus that provides access between components and shared memory.
2. When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located.

The interrupt service routine executes; on completion, the CPU resumes the interrupted computation.

1. General-purpose computers run most of their programs from rewritable memory, called main memory (also called random-access memory, or RAM). Main memory commonly is implemented in a semiconductor technology called dynamic random-access memory (DRAM), volatile memory.
2. The most common secondary-storage devices are hard-disk drives (HDDs) and nonvolatile memory (NVM) devices, which provide storage for both programs and data. Most programs (system and application) are stored in secondary storage until they are loaded into memory.
3. On modern computers, from mobile devices to servers, multiprocessor systems now dominate the landscape of computing. Traditionally, such systemshave two (or more) processors, each with a single-core CPU.
4. Multiprogramming increases CPU utilization, as well as keeping users satisfied, by organizing programs so that the CPU always has one to execute.

A program in execution is termed a process.

Multitasking is a logical extension of multiprogramming. In multitasking systems, the CPU executes multiple processes by switching among them, but the switches occur frequently, providing the user with a fast response time.

1. Caching is an important principle of computer systems.

Information is normally kept in some storage system (such as main memory).

As it is used, it is copied into a faster storage system— the cache—on a temporary basis. When we need a particular piece of information, we first check whether it is in the cache.

If it is, we use the information directly from the cache.

If it is not, we use the information from the source, putting a copy in the cache

under the assumption that we will need it again soon.

1. Virtualization is a technology that allows us to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate environment is running on its own private computer.
2. Singly linked list, doubly linked list, circularly linked list, stack, queue, tree, hash function, bitmaps are kernel data structures.

The data structures used in the Linux kernel are available in the kernel source

code.

1. Traditional computing, as name suggests, is a possess of using physical data centers for storing digital assets and running complete networking system for daily operations.

Mobile computing refers to computing on handheld smartphones and tablet

computers. These devices share the distinguishing physical features of being

portable and lightweight.

Client-Server Coputing is a contemporary network architecture features arrangements in which server systems satisfy requests generated by client systems. This form of specialized distributed system, called a client–server system.

The peer-to-peer computing (P2P) system model where all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service. Peer-to-peer systems offer an advantage over traditional client–server systems.

Cloud computing is a type of computing that delivers computing, storage,

and even applications as a service across a network.

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

**Chapter 2: Operating-System Structures**

1. User interface. Almost all operating systems have a user interface (UI).This interface can take several forms. Most commonly, a graphical user interface (GUI) is used.
   1. a command-line interface, or command interpreter, that allows users to directly enter commands to be performed by the operating system.
   2. userfriendly graphical user interface, or GUI , users employ a mouse-base windowand-menu system characterized by a desktop metaphor.
   3. touch-screen interface
2. System calls provide an interface to the services made available by an operating system. These calls are generally available as functions written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions.
3. System calls can be grouped roughly into six major categories: process control, fil management, device management, information maintenance, communications, and protection.
4. System services, also known as system utilities, provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls. Others are considerably more complex.
5. Source files are compiled into object files that are designed to be loaded into any physical memory location, a format known as an relocatable object fil . Next, the linker combines these relocatable object files into a single binary executable file.

A loader is used to load the binary executable file into memory, where it is

eligible to run on a CPU core.

An activity associated with linking and loading is relocation, which assigns final addresses to the program parts and adjusts code and data in the program to match those addresses so that.

1. The simplest structure for organizing an operating system is no structure at all. That is, place all of the functionality of the kernel into a single, static binary file that runs in a single address space. This approach—known as a monolithic structure—is a common technique for designing operating systems.
2. A system can be made modular in many ways. One method is the layered approach, in which the operating system is broken into a number of layers(levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.
3. The microkernel approach for designing operating systems uses a minimal kernel; most services run as user-level applications. Communication takes place via message passing.
4. A modular approach for designing operating systems provides operatingsystem services through modules that can be loaded and removed during run time. Many contemporary operating systems are constructed as hybrid systems using a combination of a monolithic kernel and modules.
5. 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.”

1. A boot loader loads an operating system into memory, performs initialization, and begins system execution.
2. The performance of an operating system can be monitored using either counters or tracing. Counters are a collection of system-wide or perprocess statistics, while tracing follows the execution of a program through the operating system.

**Chapter 3: Processes**

1. A process is represented by the value of the program counter and the contents of the processor’s registers. The memory layout of a process is typically divided into multiple sections

Text section— the executable code

Data section—global variables

Heap section—memory that is dynamically allocated during program run time

Stack section— temporary data storage when invoking functions (such as

function parameters, return addresses, and local variables).

1. As a process executes, it changes state. The state of a process is defined in part by the current activity of that process.

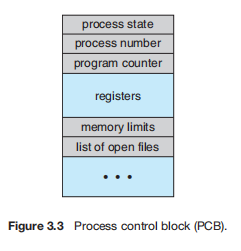
New. The process is being created.

Running. Instructions are being executed.

Waiting. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

Ready. The process is waiting to be assigned to a processor.

Terminated. The process has finished execution.

1. Each process is represented in the operating system by a process control block (PCB)—also called a task control block.
2. Process state. The state may be new, ready, running, waiting, halted, and so on.
3. Program counter. The counter indicates the address of the next instruction to be executed for this process.
4. CPU registers. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.
5. CPU-scheduling information. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
6. Memory-management information. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.
7. Accounting information. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers,and so on.
8. I/O status information. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

TO BE CONTINUED…

Operation on processes

1. Process creation

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

Most operating systems (including UNIX, Linux, and Windows) identify

processes according to a unique process identifie (or pid), which is typically an integer number. The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.

On UNIX and Linux systems, we can obtain a listing of processes by using

the ps command. For example, the command will list complete information for all processes currently active in the system.

ps -el

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

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

1. The parent continues to execute concurrently with its children.

2. The parent waits until some or all of its children have terminated.

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

1. The child process is a duplicate of the parent process (it has the

same program and data as the parent).

2. The child process has a new program loaded into it.

A new process is created by the fork() system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.

Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference: the return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

The exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec() system call) and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways.

The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait() system call to move itself off the ready queue until the termination of the child. Because the call to exec() overlays the process’s address space with a new program, exec() does not return control unless an error occurs.

1. Process Termination

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

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, TerminateProcess() in Windows).

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

• The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)

• The task assigned to the child is no longer required.

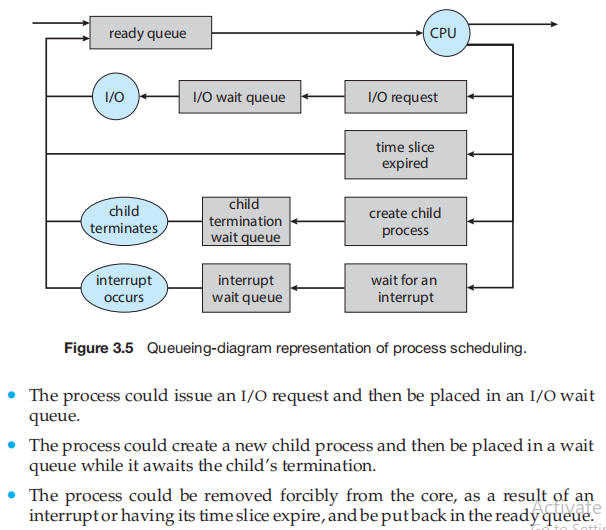
• The parent is exiting, and the operating system does not allow a child to

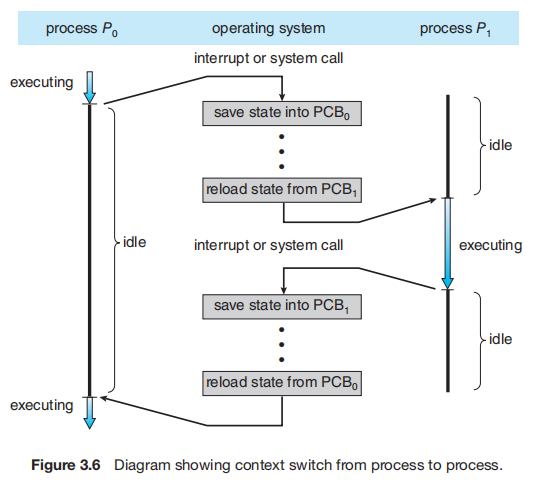
continue if its parent terminates.

When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls wait(), because the process table contains the process’s exit status.

* A process that has terminated, but whose parent has not yet called wait(), is known as a zombie process. All processes transition to this state when they terminate, but generally they exist as zombies only briefly. Once the parent calls wait(), the process identifier of the zombie process and its entry in the process table are released.
* Now consider what would happen if a parent did not invoke wait() and instead terminated, thereby leaving its child processes as orphans. Traditional UNIX systems addressed this scenario by assigning the init process as the new parent to orphan processes. The init process periodically invokes wait(), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan’s process identifier and process-table entry

CONTINUE…

1. Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. This feature is especially beneficial on multicore systems, where multiple threads can run in parallel.
2. If there are more processes than cores, excess processes will have to wait until a core is free and can be rescheduled. The number of processes currently in memory is known as the degree of multi programming.
3. As processes enter the system, they are put into a ready queue, where they are ready and waiting to execute on a CPU’s core This queue is generally stored as a linked list; a ready-queue header contains pointers to the first PCB in the list, and each PCB includes a pointer field that points to the next PCB in the ready queue.
4. 
5. interrupts cause the operating system to change a CPU core from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems. When an interrupt occurs, the system needs to save the current context of the process running on the CPU core so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state, and memory-management information. Generically, we perform a state save of the current state of the CPU core, be it in kernel or user mode, and then a state restore to resume operations.
6. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

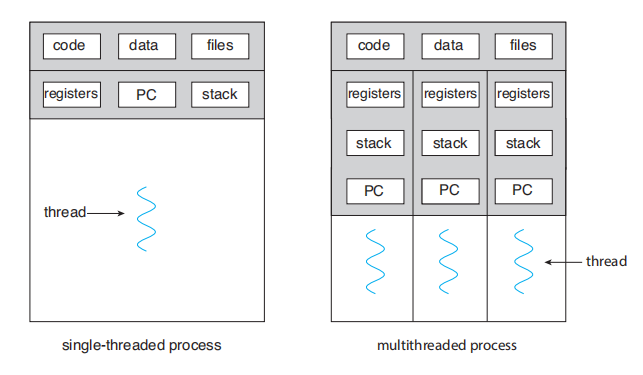


1. Context-switch times are highly dependent on hardware support. For instance, some processors provide multiple sets of registers. A context switch here simply requires changing the pointer to the current register set. Of course,if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the greater the amount of work that must be done during a context switch.
2. A process is independent if it does not share data with any other processes executing in the system. A process is cooperating if it can affect or be affected by the other processes executing in the system.
3. A pipe acts as a conduit allowing two processes to communicate. Pipes were one of the first IPC mechanisms in early UNIX systems. They typically provide one of the simpler ways for processes to communicate with one another, although they also have some limitations.
4. There are several reasons for providing an environment that allows process cooperation:
   1. Information sharing. Since several applications may be interested in the same piece of information (for instance, copying and pasting), we must provide an environment to allow concurrent access to such information.
   2. Computation speedup. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Noticet that such a speedup can be achieved only if the computer has multiple processing cores.
   3. Modularity. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
5. A socket is defined as an endpoint for communication. A pair of processes communicating over a network employs a pair of sockets—one for each process.A socket is identified by an IP address concatenated with a port number. In general, sockets use a client–server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection.
6. Two common forms of client–server communication are sockets and remote procedure calls (RPCs).

Sockets allow two processes on different machines to communicate over a network.

RPCs abstract the concept of function (procedure) calls in such a way that a function can be invoked on another process that may reside on a separate computer.

**Chapter 4: Threads & Concurrency**



A thread is a basic unit of CPU utilization; it comprises a thread ID, a program

counter (PC), a register set, and a stack. It shares with other threads belonging

to the same process its code section, data section, and other operating-system

resources, such as open files and signals.

Multi-thread processing is a mechanism to accomplish high performance by partitioning and processing the data in multiple threads parallelly.

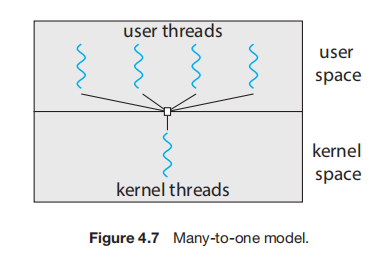
The number of threads will be determined automatically under multiple conditions such as the size of the read data or the number of CPU cores.

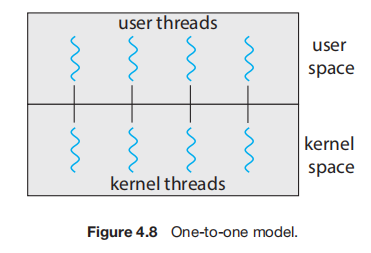
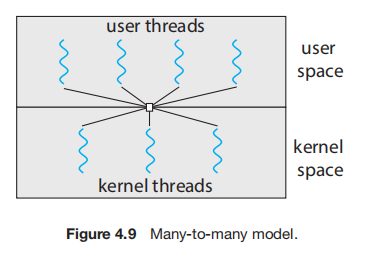
1. The benefits of multithreaded programming can be broken down into four major categories:
   1. Responsiveness. Multithreading an interactive application may allow a

program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces.

* 1. Resource sharing The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
  2. Ecnomy. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Additionally, context switching is typically faster between threads than between processes.
  3. Scalability. The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

1. We look at three common ways of establishing such a relationship: the many-to-one model, the one-toone model, and the many-to-many model.



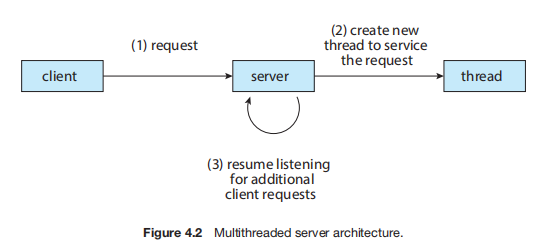
* 1. Many-to-One model maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient .However, the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.
  2. One-to-One model maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. 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, and a large number of kernel threads may burden the performance of a system.
  3. Many-to-Many model multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine.The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and he corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

1. A thread library provides an API for creating and managing threads. Three common thread libraries include Windows, Pthreads, and Java threading.

We introduce two general strategies for creating multiple threads: asynchronous threading and synchronous threading.

With asynchronous threading, once the parent creates a child thread, the parent resumes its execution, so that the parent and child execute concurrently and independently of one another. Because the threads are independent, there is typically little data sharing between them

Asynchronous threading is the strategy used in the multithreaded server illustrated in Figure 4.2 and is also commonly used for designing responsive user interfaces.



Synchronous threading occurs when the parent thread creates one or more children and then must wait for all of its children to terminate before it resumes.

Here, the threads created by the parent perform work concurrently, but the

parent cannot continue until this work has been completed. Once each thread

has finished its work, it terminates and joins with its parent. Only after all of the children have joined can the parent resume execution

1. Threads may be terminated using either asynchronous or deferred cancellation.
   1. Asynchronous cancellation stops a thread immediately, even if it is in the middle of performing an update.
   2. Deferred cancellation informs a thread that it should terminate but allows the thread to terminate in an orderly fashion. In most circumstances, deferred cancellation is preferred to asynchronous termination.

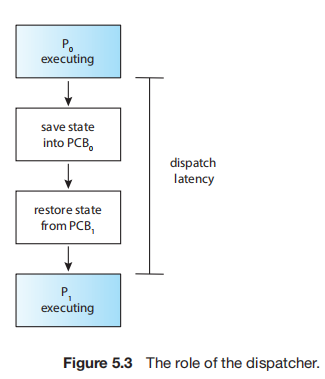
**Chapter 5: CPU Scheduling**

1. CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.
2. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
3. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively.Every time one process has to wait, another process can take over use of the CPU. On a multicore system, this concept of keeping the CPU busy is extended to all processing cores on the system.
4. Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.

The selection process is carried out by the CPU scheduler, which selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

1. CPU-scheduling decisions may take place under the following four circumstances:
   1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process)
   2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
   3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)
   4. When a process terminates.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is nonpreemptive (ko nhường) or cooperative. Otherwise, it is preemptive (có nhường).



1. Another component involved in the CPU-scheduling function is the dispatcher.
2. The dispatcher is the module that gives control of the CPU’s core to the process selected by the CPU scheduler. This function involves the following:
   1. Switching context from one process to another
   2. Switching to user mode
   3. Jumping to the proper location in the user program to resume that program
3. The dispatcher should be as fast as possible, since it is invoked during every context switch. The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency and is illustrated in Figure 5.3.
4. Scheduling Criteria (Tiêu chí định thời)

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

CPU utilization. We want to 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 loaded system). (CPU utilization can be obtained by using the top command on Linux, macOS, and UNIX systems.)

Throughput. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed 5.3 Scheduling Algorithms 205 per time unit, called throughput. For long processes, this rate may be one process over several seconds; for short transactions, it may be tens of processes per second.

Turnaround time. From the point of view of a particular process, the important criterion is how long it takes to execute that 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 in the ready queue, executing on the CPU, and doing I/O.

Waiting time. The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

Response time. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

1. **Scheduling algorithm**
   1. By far the simplest CPU-scheduling algorithm is the first-come first serve (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 a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is the removed from the queue. On the negative side, the average waiting time under the FCFS policy is often quite long
   2. A different approach to CPU scheduling is the shortest-job-firs (SJF) scheduling algorithm. This algorithm associates with each process the length of the process’s next CPU burst. 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. Note that a more appropriate term for this scheduling method would be the shortest-next-CPU-burst algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.
   3. The round-robin (RR) scheduling algorithm is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a time quantum or time slice, is defined. A time quantum is generally from 10 to 10 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.
   4. The SJF algorithm is a special case of the general priority-scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.
   5. With both priority and round-robin scheduling, all processes may be placed in a single queue, and the scheduler then selects the process with the highest priority to run. Depending on how the queues are managed, an O(n) search may be necessary to determine the highest-priority process. In practices it is often easier to have separate queues for each distinct priority, and priority scheduling simply schedules the process in the highest-priority queue. This is illustrated in Figure 5.7. This approach—known as multilevel queue.

If there are multiple processes in the highest-priority queue, they are executed in round-robin order. In the most generalized form of this approach, a priority is assigned statically to each process, and a process remains in the same queue for the duration of its runtime.

* 1. The multilevel feedback queue scheduling algorithm, in contrast, 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—which are typically characterized by short CPU bursts —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.

Preemptive Priority

In Preemptive Priority Scheduling, at the time of arrival of a process in the ready queue, its Priority is compared with the priority of the other processes present in the ready queue as well as with the one which is being executed by the CPU at that point of time. The One with the highest priority among all the available processes will be given the CPU next.

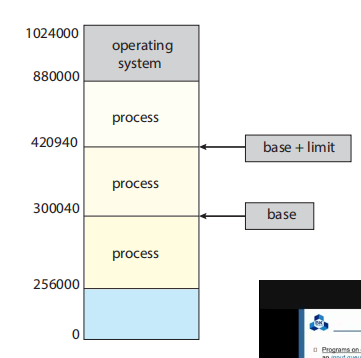
The difference between preemptive priority scheduling and non preemptive priority scheduling is that, in the preemptive priority scheduling, the job which is being executed can be stopped at the arrival of a higher priority job.

Once all the jobs get available in the ready queue, the algorithm will behave as non-preemptive priority scheduling, which means the job scheduled will run till the completion and no preemption will be done.

**Chapter 6: Synchronization Tools**

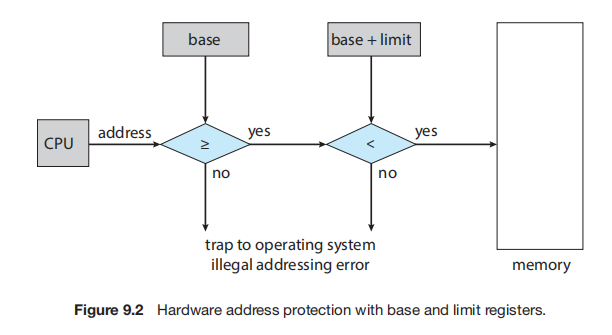
**Chapter 9: Main Memory**

1. A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data).
2. We first need to make sure that each process has a separate memory space. Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution. To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit, as illustrated in Figure 9.1.

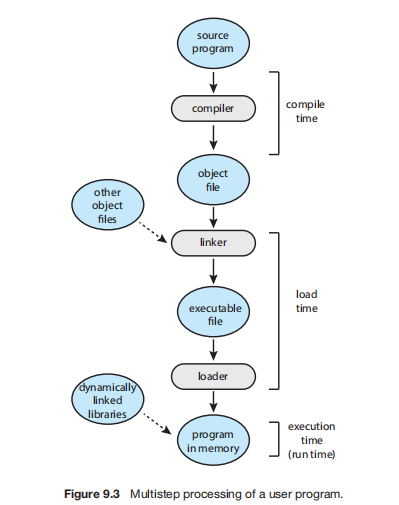


The base register holds the smallest legal physical memory address; the limit register specifies the size of the range. For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

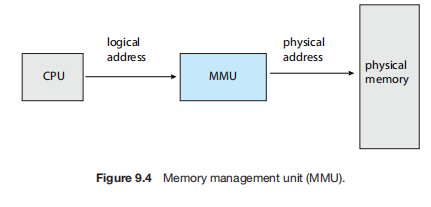
1. Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users’ memory results in a trap to the operating system, which treats the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



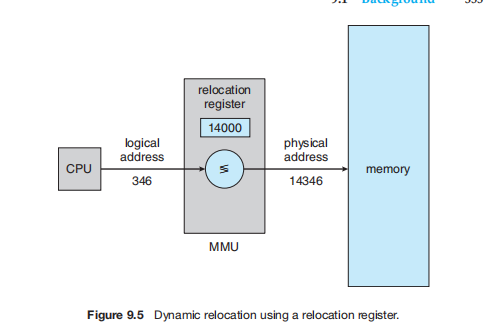
1. Classically, the binding of instructions and data to memory addresses can be done at any step along the way:
   1. Compile time. If you know at compile time where the process will reside in memory, then absolute code can be generated. For example, if you know that a user process will reside starting at location R, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.
   2. Load time. If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.
   3. Execution time. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work, as will be discussed in Section 9.1.3. Most operating systems use this method.



1. An address generated by the CPU is commonly referred to as a logical address, whereas an address seen by the memory unit— that is, the one loaded into the memory-address register of the memory—is commonly referred to as a physical address.



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



1. The user program never accesses the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses—all as the number 346. Only when it is used as a memory address (in an indirect load or store, perhaps) is it relocated relative to the base register
2. The user program deals with logical addresses. The memory mapping hardware converts logical addresses into physical addresses. This form of execution-time binding was discussed in Section 9.1.2. The final location of a referenced memory address is not determined until the reference is made.
3. We now have two different types of addresses: logical addresses (in the range 0 to max) and physical addresses (in the range R + 0 to R + max for a base value R).
4. Dynamic Loading (Nạp động)

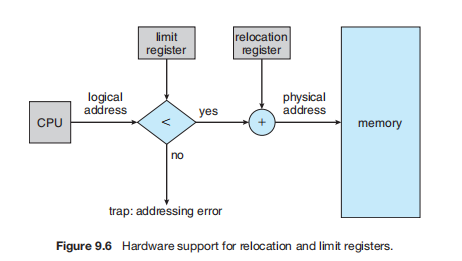
In our discussion so far, it has been necessary for the entire program and all data of a process to be in physical memory for the process to execute. The size of a process has thus been limited to the size of physical memory. To obtain better memory-space utilization, we can use dynamic loading.

The entire program does need to be in memory to execute.

1. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format.
2. Memory management discussed thus far has required the physical address space of a process to be contiguous. We now introduce paging, a memorymanagement scheme that permits a process’s physical address space to be non contiguous. Paging avoids external fragmentation and the associated need for compaction, two problems that plague contiguous memory allocation. Because it offers numerous advantages, paging in its various forms is used in most operating systems, from those for large servers through those for mobile devices. Paging is implemented through cooperation between the operating system and the computer hardware.
3. Contiguous allocaton ( Cấp phát liên tục)

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

1. We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are waiting to be brought into memory. In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.



We can prevent a process from accessing memory that it does not own by

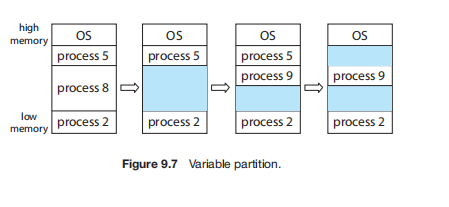
combining two ideas previously discussed.

If we have a system with a relocation register (Section 9.1.3), together with a limit register (Section 9.1.1), we accomplish our goal.

The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600). Each logical address must fall within the range specified by the limit register. The MMU maps the logical address dynamically by adding the value in the relocation register.

This mapped address is sent to memory (Figure 9.6).

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



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

1. Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation. As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes.

(phân mảnh ngoại)

1. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself.

The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is internal fragmentation —unused memory that is internal to a partition.

( phân mảnh trong).

1. Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem. Statistical analysis of first fit, for instance, reveals that, even with some optimization, given N allocated blocks, another 0.5 N blocks will be lost to fragmentation. That is, one-third of memory may be unusable! This property is known as the 50-percent rule.
2. One solution to the problem of external fragmentation is compaction. The goal is to shuffle the memory contents so as to place all free memory together in one large block.

Compaction is not always possible, however. If relocation

is static and is done at assembly or load time, compaction cannot be done. It is possible only if relocation is dynamic and is done at execution time. If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address. When compaction is possible, we must determine its cost.

1. We now introduce paging, a memorymanagement scheme that permits a process’s physical address space to be noncontiguous. Paging avoids external fragmentation and the associated need for compaction, two problems that plague contiguous memory allocation.

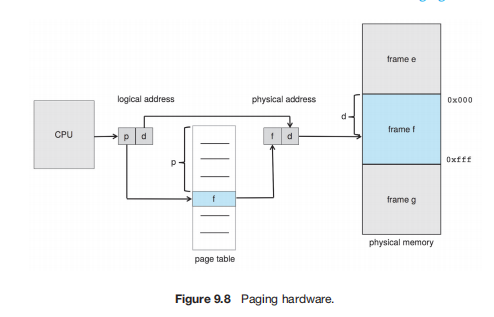
Because it offers numerous advantages, paging in its various forms is used in most operating systems, from those for large servers through those for mobile devices. Paging is implemented through cooperation between the operating system and the computer hardware.

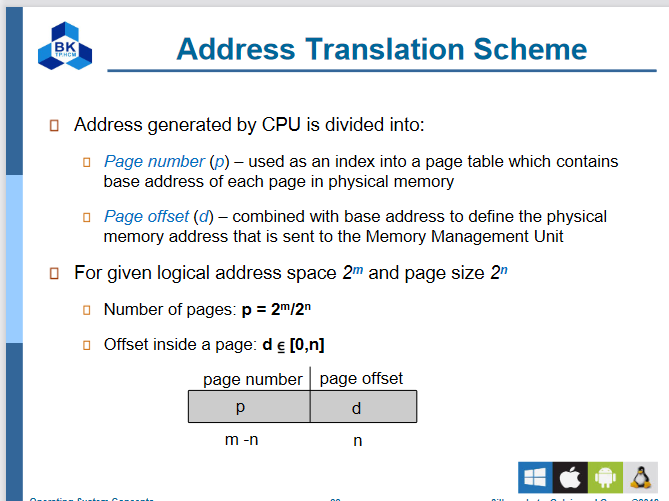
1. The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames

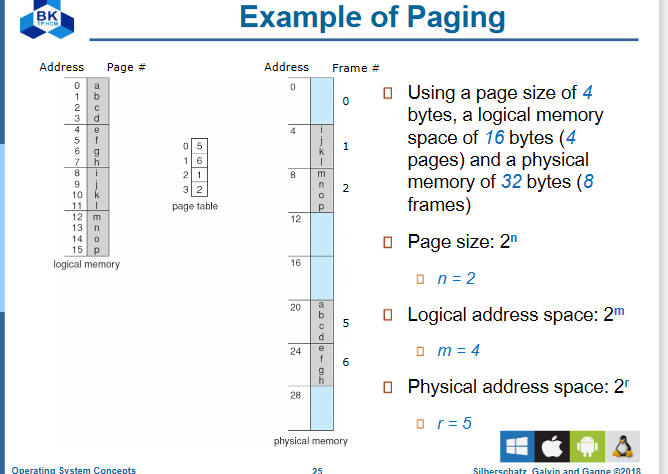
Size is power of 2 (512 bytes <= x <= 16 MB).

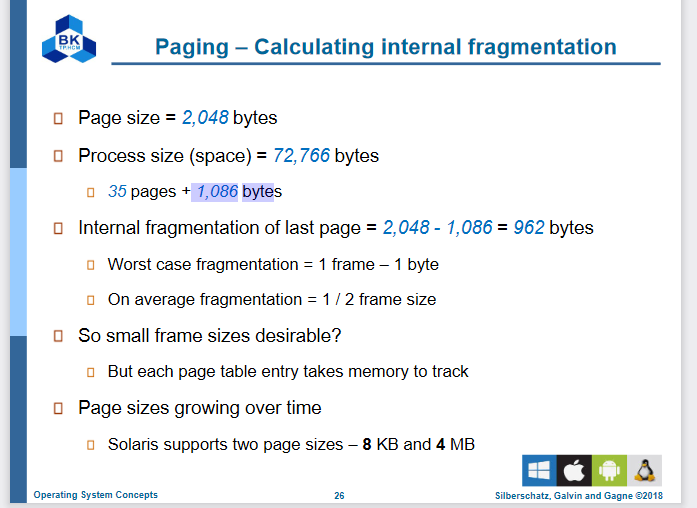
and breaking logical memory into blocks of the same size called pages.

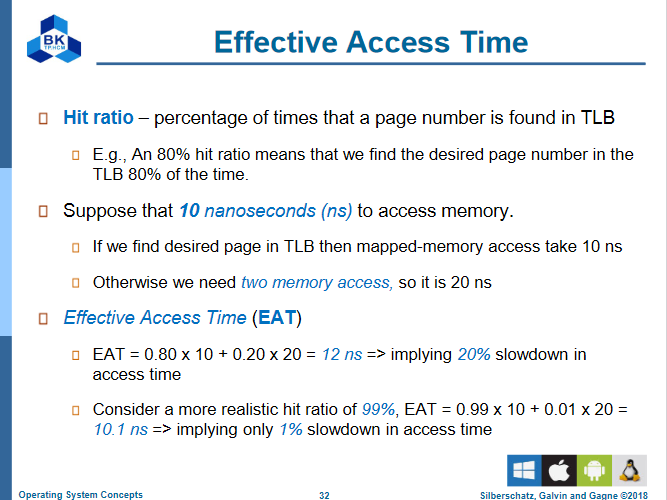
1. The page number is used as an index into a per-process page table



1. 



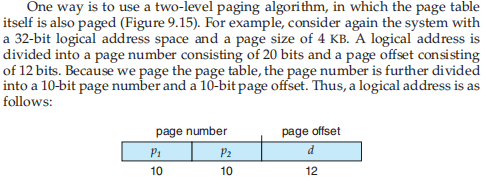


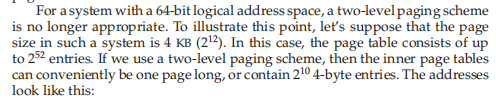


Translation Look-Aside Buffer

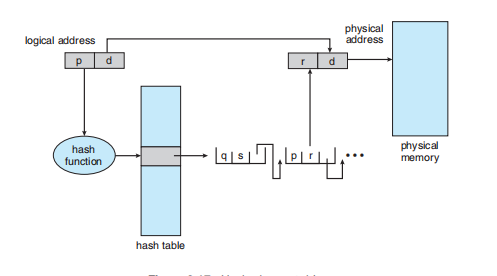
1. One additional bit is generally attached to each entry in the page table: a valid–invalid bit. When this bit is set to valid, the associated page is in the process’s logical address space and is thus a legal (or valid) page. When the bit is set to invalid, the page is not in the process’s logical address space. Illegal addresses are trapped by use of the valid –invalid bit. The operating system sets this bit for each page to allow or disallow access to the page.
2. Structure of the Page Table (Bảng phân trang)

**Hierarchical Paging**

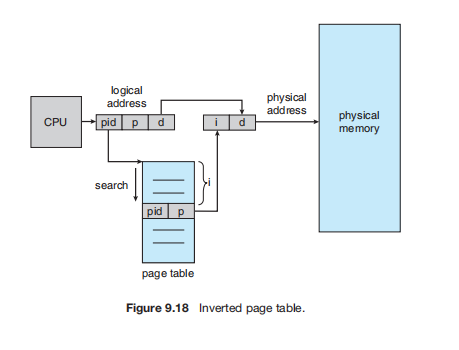


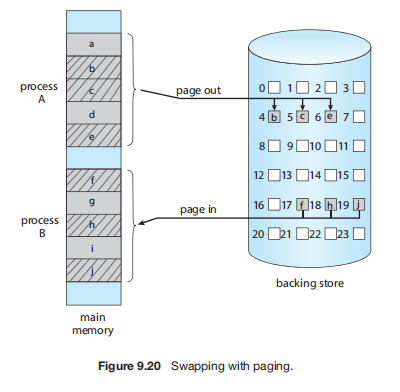


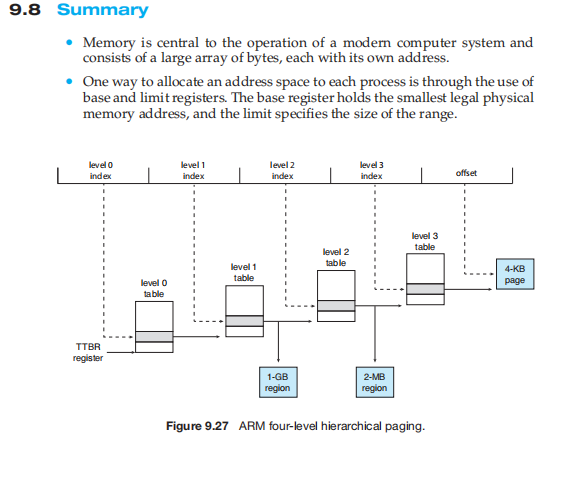
**Hashed Page Tables**

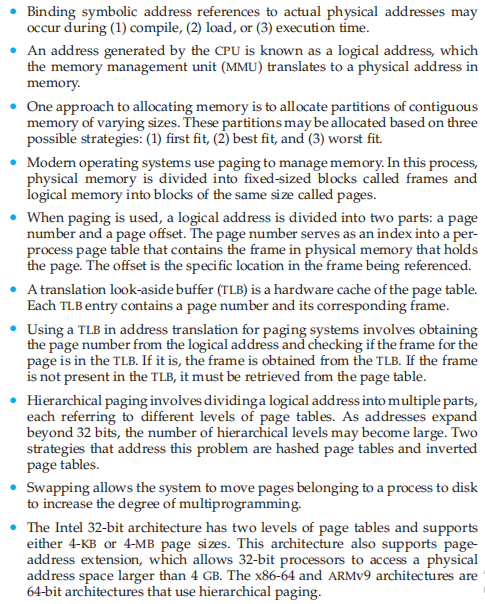


**Inverted Page Tables (Bảng phân trang ngược)**

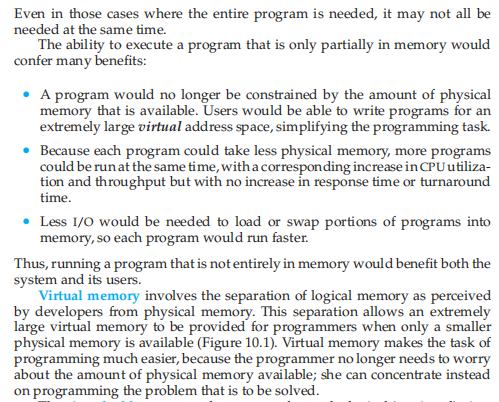


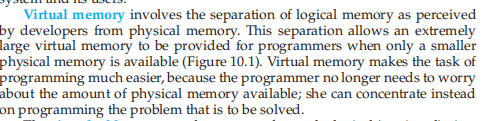


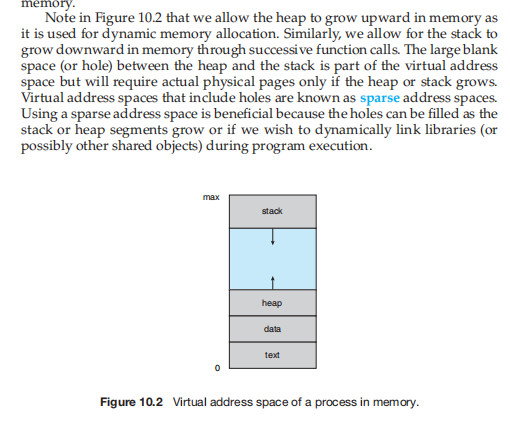




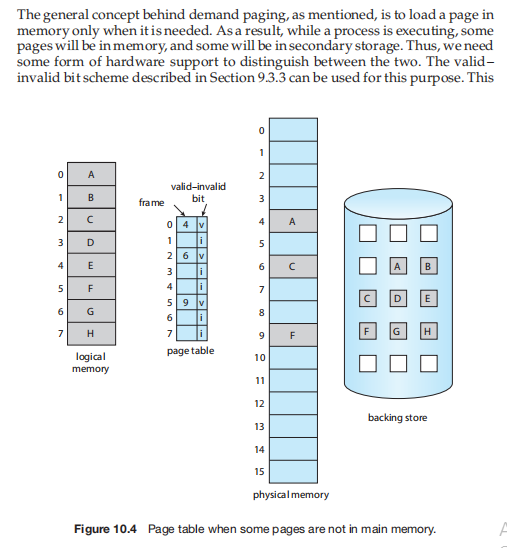
**Chapter 10: Virtual memory**

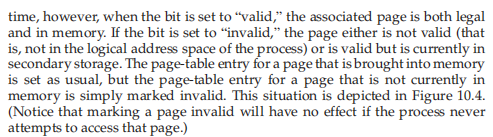




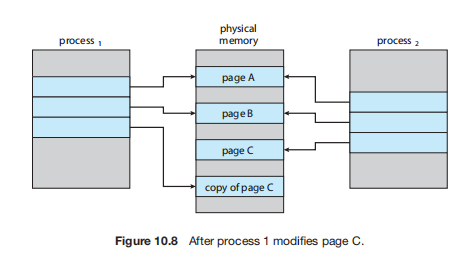


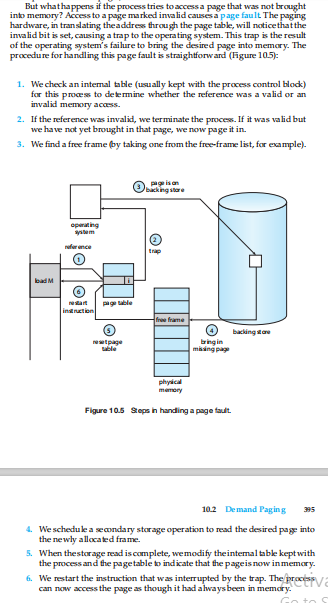
**DEMAND PAGING**



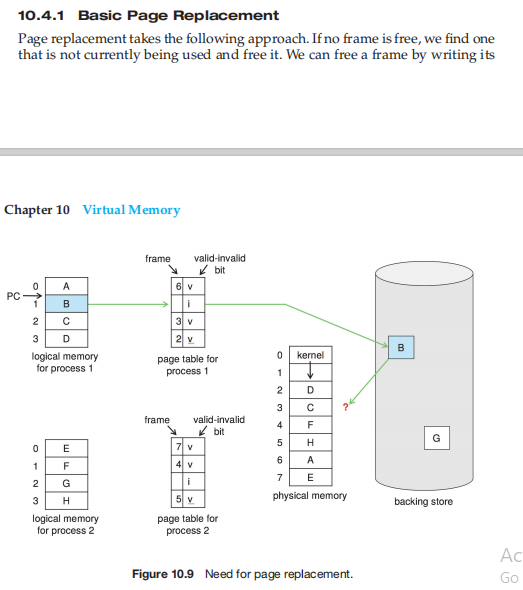


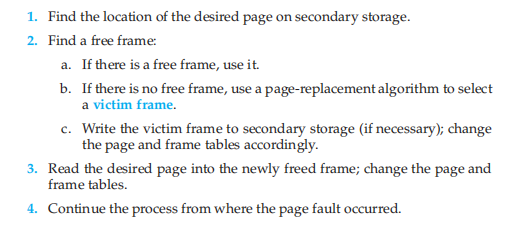
**Copy-on-write**

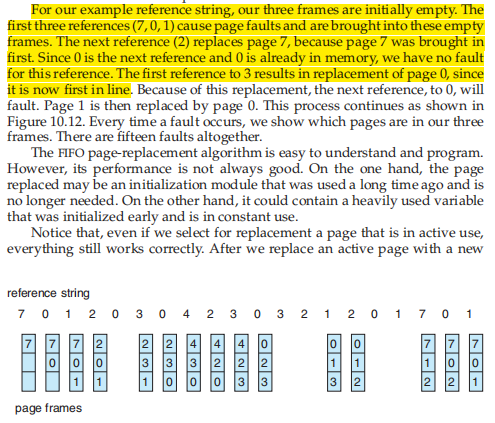




**---------------------------------------------------------------------------------**

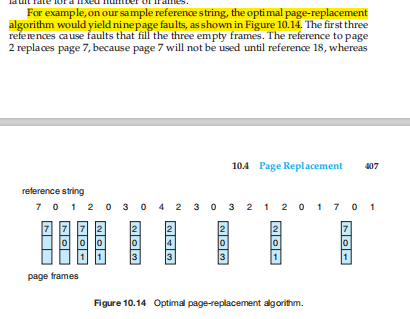






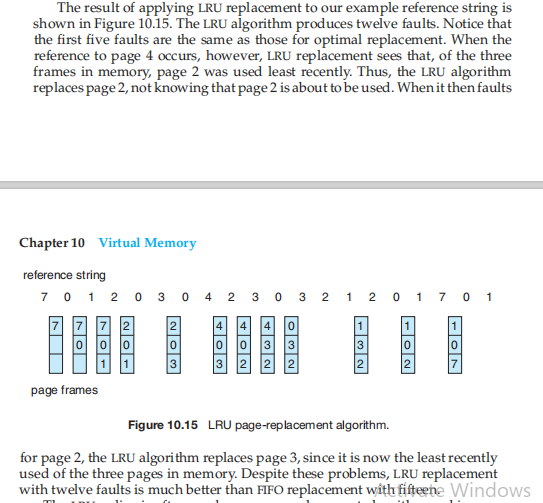
**FIFO**

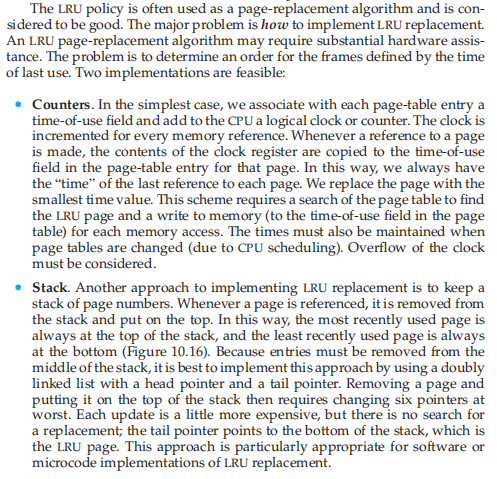
**OPTIMAL**



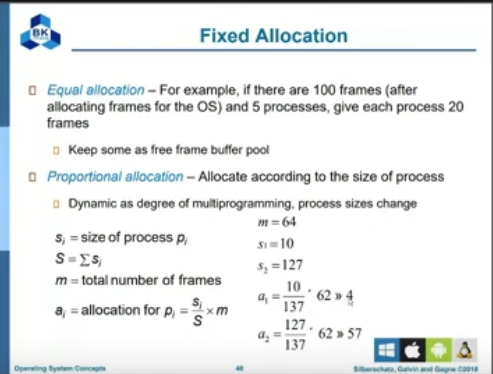
**Nhìn tới thấy cái nào sẽ dùng xa nhất thì thay ra**

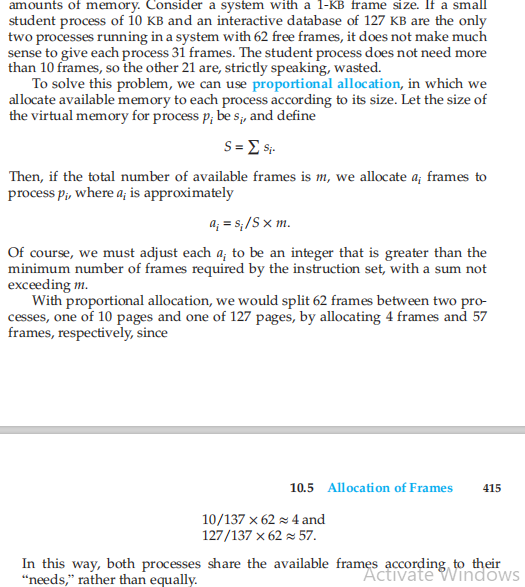
**LRU Page Replacement (Lest recently used)**



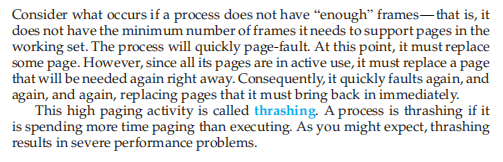


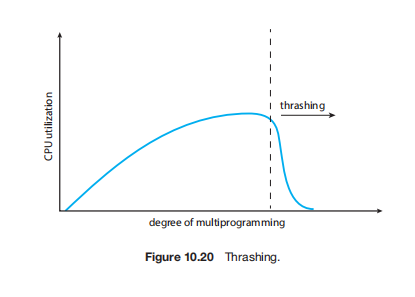
**Allocation of frames**

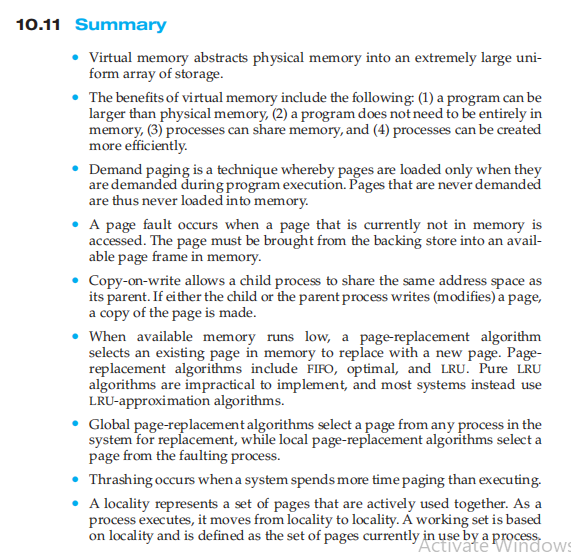


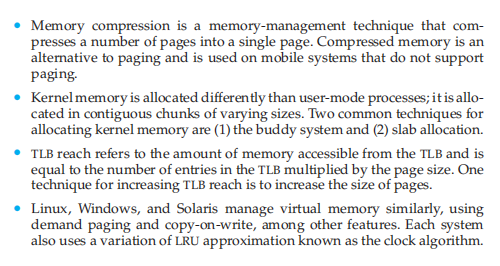


**Thrashing**

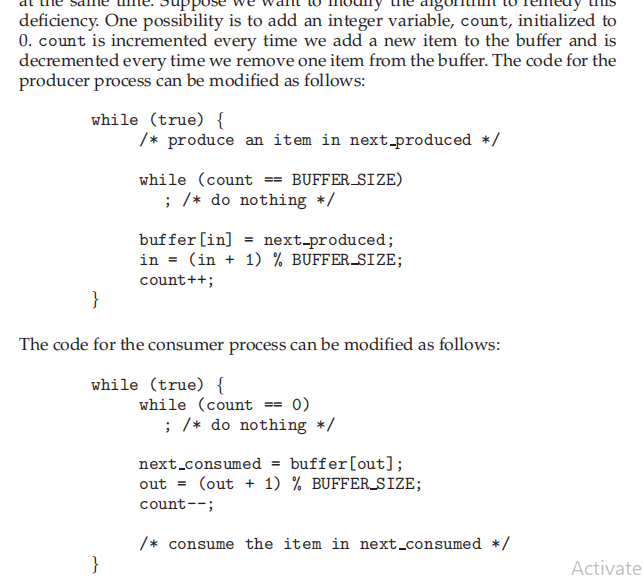


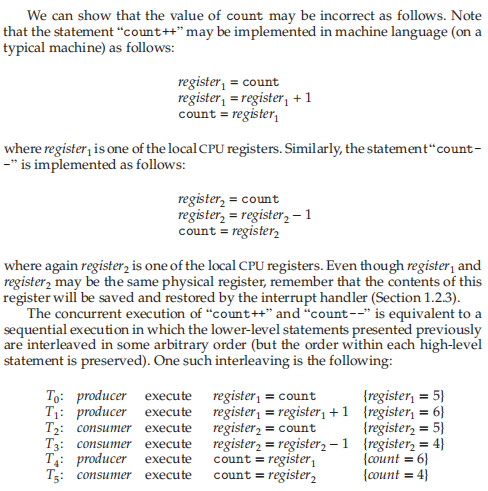


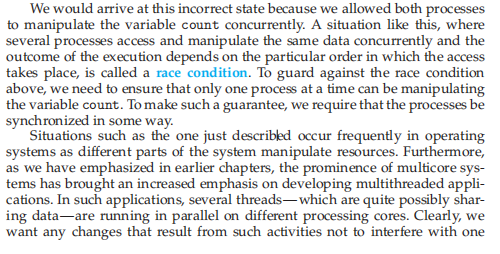
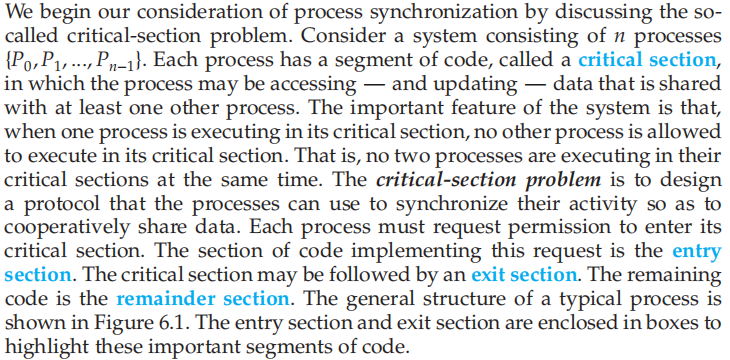


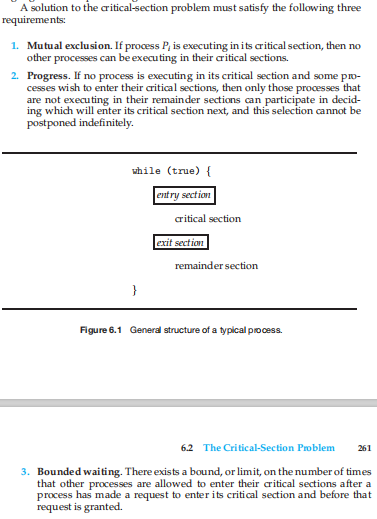


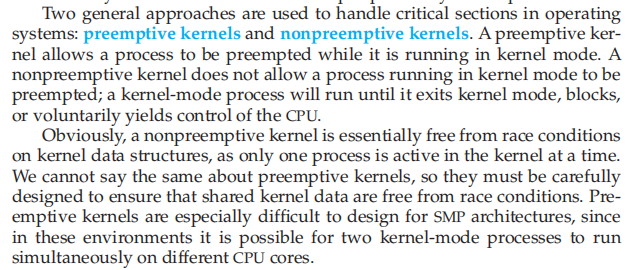
**Chapter 6: Synchronization tools**

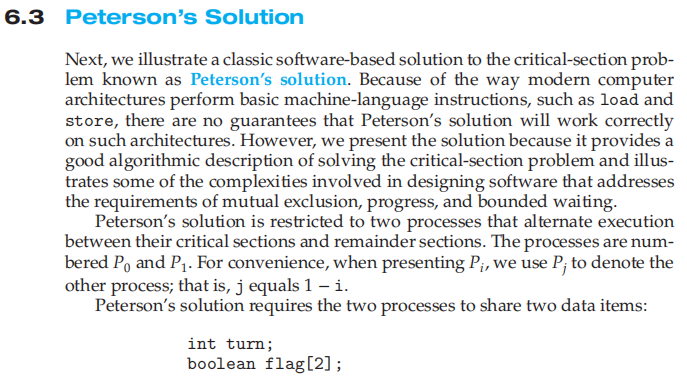
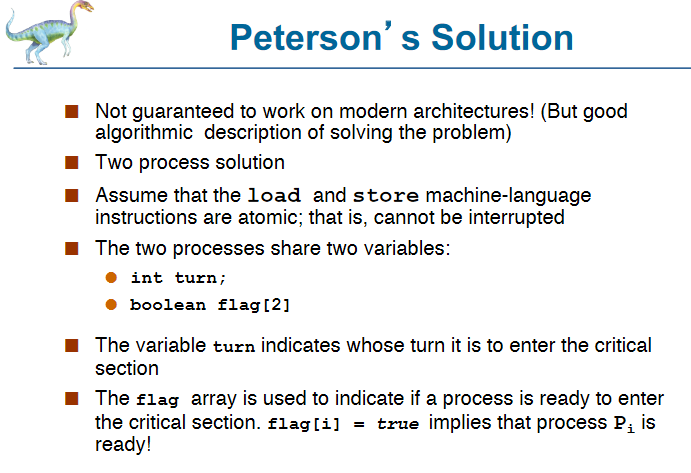


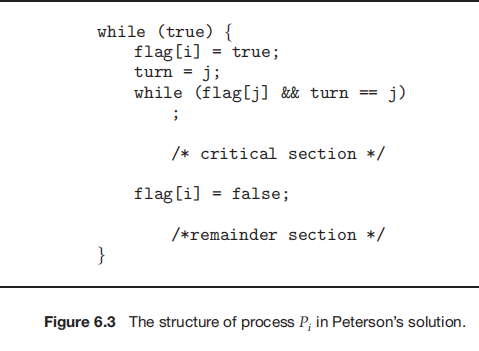


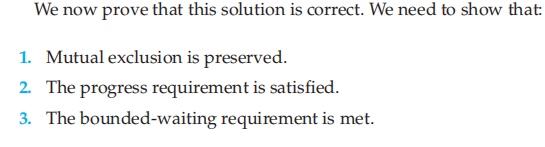


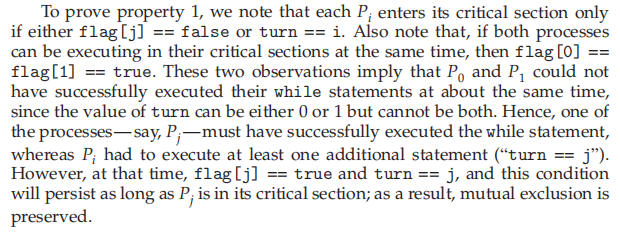


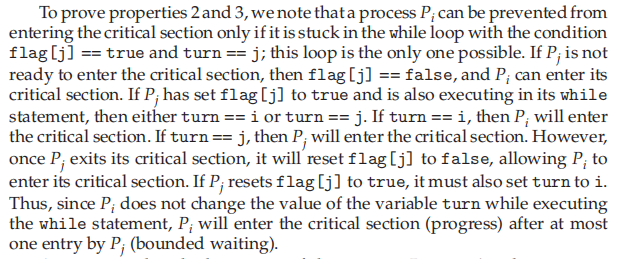












**Chapter 11: Mass-Storage Structure**

