Unit	Details		Lectures
I	**Operating System Overview:** Objectives and Functions, Evolution, Achievements, Modern Operating Systems, Fault tolerance, OS design considerations for multiprocessor and multicore, overview of different operating systems **Processes:** Process Description and Control.		12
II	**Threads, Concurrency:** Mutual Exclusion and Synchronization.		12
III	**Concurrency:** Deadlock and Starvation, **Memory:** Memory Management, Virtual Memory.		12
IV	**Scheduling:** Uniprocessor Scheduling, Multiprocessor and Real-Time Scheduling		12
V	**IO and File Management:** I/O Management and Disk Scheduling, File Management, **Operating System Security.**		12

Concurrency: Deadlock and Starvation

When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.

Statute passed by the Kansas State Legislature, early in the 20th century
—A Treasury of Railroad Folklore,
B. A. Botkin and Alvin F. Harlow

PRINCIPLES OF DEADLOCK

Deadlock can be defined as the permanent blocking of a set of processes that either compete for system resources or communicate with each other.

A set of processes is deadlocked when each process in the set is blocked awaiting an event (typically the freeing up of some requested resource) that can only be triggered by another blocked process in the set.

Deadlock is permanent because none of the events is ever triggered.

Unlike other problems in concurrent process management, there is no efficient solution in the general case.

The Conditions for Deadlock

Three conditions of policy must be present for a deadlock to be possible:

1. Mutual exclusion. Only one process may use a resource at a time. No process may access a resource unit that has been allocated to another process.

2. Hold and wait. A process may hold allocated resources while awaiting assignment of other resources.

3. No preemption. No resource can be forcibly removed from a process holding it.

The first three conditions are necessary but not sufficient for a deadlock to exist. For deadlock to actually take place, a fourth condition is required:

4. Circular wait. A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.

To summarize:

Possibility of Deadlock	Existence of Deadlock
1. Mutual exclusion	1. Mutual exclusion
2. No preemption	2. No preemption
3. Hold and wait	3. Hold and wait
	4. Circular wait

Three general approaches exist for dealing with deadlock.
First, one can **prevent** deadlock by adopting a policy that eliminates one of the conditions (conditions 1 through 4).

Second, one can **avoid** deadlock by making the appropriate dynamic choices based on the current state of resource allocation.

Third, one can attempt to **detect** the presence of deadlock (conditions 1 through 4 hold) and take action to recover.

DEADLOCK PREVENTION

The strategy of deadlock prevention is to design a system in such a way that **the possibility of deadlock is excluded**.

Two classes of deadlock prevention methods.

An **indirect method** of deadlock prevention is to prevent the occurrence of one of the three necessary conditions listed previously (items 1 through 3).

A **direct method** of deadlock prevention is to prevent the occurrence of a circular wait (item 4).

Mutual Exclusion

In general, the first of the four listed conditions cannot be disallowed. If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the OS. Some resources, such as files, may allow multiple accesses for reads but only exclusive access for writes. Even in this case, deadlock can occur if more than one process requires write permission.

Hold and Wait

The hold-and-wait condition can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously. This approach is inefficient in two ways. First, a process may be held up for a long time waiting for all of its resource requests to be filled, when in fact it could have proceeded with only some of the resources. Second, resources allocated to a process may remain unused for a considerable period, during which time they are denied to other processes. Another problem is that a process may not know in advance all of the resources that it will require.

No Preemption

This condition can be prevented in several ways. First, if a process holding certain resources is denied a further request, that process must release its original resources and, if necessary, request them again together with the additional resource.

Alternatively, if a process requests a resource that is currently held by another process, the OS may preempt the second process and require it to release its resources.

This latter scheme would prevent deadlock only if no two processes possessed the same priority.

The circular-wait condition can be prevented by defining a linear ordering of resource types. **If a process has been allocated resources of type R , then it may subsequently request only those resources of types following R in the ordering.**

To see that this strategy works, let us associate an index with each resource type. Then resource Ri precedes Rj in the ordering if i < j . Now suppose that two processes, A and B, are deadlocked because A has acquired Ri and requested Rj ,and B has acquired Rj and requested Ri . This condition is impossible because it implies i < j and j < i .

As with hold-and-wait prevention, circular-wait prevention may be inefficient, slowing down processes and denying resource access unnecessarily.

DEADLOCK AVOIDANCE

An approach to solving the deadlock problem that differs subtly from deadlock prevention is deadlock avoidance.

In deadlock prevention , we **constrain resource requests to prevent at least one of the four conditions of deadlock.** This is either done indirectly, by preventing one of the three necessary policy conditions (mutual exclusion, hold and wait, no preemption), or directly, by preventing circular wait.

This leads to inefficient use of resources and inefficient execution of processes.

Deadlock avoidance , on the other hand, **allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached.**

As such, **avoidance allows more concurrency than prevention**.

With deadlock avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock. **Deadlock avoidance thus requires knowledge of future process resource requests.**

Two approaches to deadlock avoidance:

- Do not start a process if its demands might lead to deadlock.

- **Do not grant an incremental resource request to a process** if this allocation might lead to deadlock.

DEADLOCK DETECTION

Process Initiation Denial : **Claim matrix** technique is used.

Resource Allocation Denial:

The strategy of resource allocation denial, referred to as the **banker's algorithm.**

The **state** of the system reflects the current allocation of resources to processes.

A **safe state** is one in which there is at least one sequence of resource allocations to processes that does not result in a deadlock.

An **unsafe state** is, of course, a state that is not safe.

DEADLOCK DETECTION

Deadlock prevention strategies are very conservative; they solve the problem of deadlock by limiting access to resources and by imposing restrictions on processes.

At the opposite extreme, **deadlock detection strategies do not limit resource access or restrict process actions.** With deadlock detection, requested resources are granted to processes whenever possible. Periodically, the OS performs an algorithm that allows it to detect the circular wait condition described earlier in condition (4).

Recovery

Once deadlock has been detected, some strategy is needed for recovery. The following are possible approaches, listed in order of increasing sophistication:

1. Abort all deadlocked processes. This is, one of the most common, if not the most common, solution adopted in operating systems.

2.**Back up each deadlocked process to some previously defined checkpoint, and restart all processes.** This requires that rollback and restart mechanisms be built in to the system. The risk in this approach is that the original deadlock may recur.

3. **Successively abort deadlocked processes until deadlock no longer exists.** The order in which processes are selected for abortion should be on the basis of some criterion of minimum cost. After each abortion, the detection algorithm must be re-invoked to see whether deadlock still exists.

4. **Successively preempt resources until deadlock no longer exists.** As in (3), a cost-based selection should be used, and re-invocation of the detection algorithm is required after each preemption. A process that has a resource preempted from it must be rolled back to a point prior to its acquisition of that resource.

SUMMARY

Deadlock is the blocking of a set of processes that either compete for system resources or communicate with each other. The blockage is **permanent** unless the OS takes some extraordinary action, such as killing one or more processes or forcing one or more processes to backtrack.

There are **three** general **approaches** to dealing with deadlock: **prevention, detection, and avoidance.** Deadlock prevention guarantees that deadlock will not occur, by assuring that one of the necessary conditions for deadlock is not met. Deadlock detection is needed if the OS is always willing to grant resource requests; periodically, the OS must check for deadlock and take action to break the deadlock. Deadlock avoidance involves the analysis of each new resource request to determine if it could lead to deadlock, and granting it only if deadlock is not possible.

Memory: Memory Management

In a **uniprogramming** system, main memory is divided into **two parts**: one part for the operating system (resident monitor, kernel) and one part for the program currently being executed.

In a **multiprogramming** system, the “user” part of memory must be further subdivided to accommodate multiple processes.

The task of subdivision is carried out dynamically by the operating system and is known as memory management .

Effective memory management is vital in a multiprogramming system. If only a few processes are in memory, then for much of the time all of the processes will be waiting for I/O and the processor will be idle. Thus **memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time.**

Some key terms for our discussion.

Frame

A fixed-length block of main memory.

Page

A fixed-length block of data that resides in secondary memory (such as disk). A page of data may temporarily be copied into a frame of main memory.

Segment

A variable-length block of data that resides in secondary memory. An entire segment may temporarily be copied into an available region of main memory (segmentation) or the segment may be divided into pages which can be individually copied into main memory (combined segmentation and paging).

MEMORY MANAGEMENT REQUIREMENTS

The requirements for memory management intended to satisfy are the following:

- Relocation
- Protection
- Sharing
- Logical organization
- Physical organization

Relocation

In a multiprogramming system, the available main memory is generally shared among a number of processes. Typically, it is not possible for the programmer to know in advance which other programs will be resident in main memory at the time of execution of his or her program. In addition, we would like to be able **to swap active processes in and out of main memory to maximize processor utilization** by providing a large pool of ready processes to execute. Once a program is swapped out to disk, it would be quite limiting to specify that when it is next swapped back in, it must be placed in the same main memory region as before. Instead, **we may need to relocate the process to a different area of memory.**

Thus, we cannot know ahead of time where a program will be placed, and we must allow for the possibility that the program may be moved about in main memory due to swapping. These facts raise some **technical concerns** related to addressing,as illustrated in Figure.

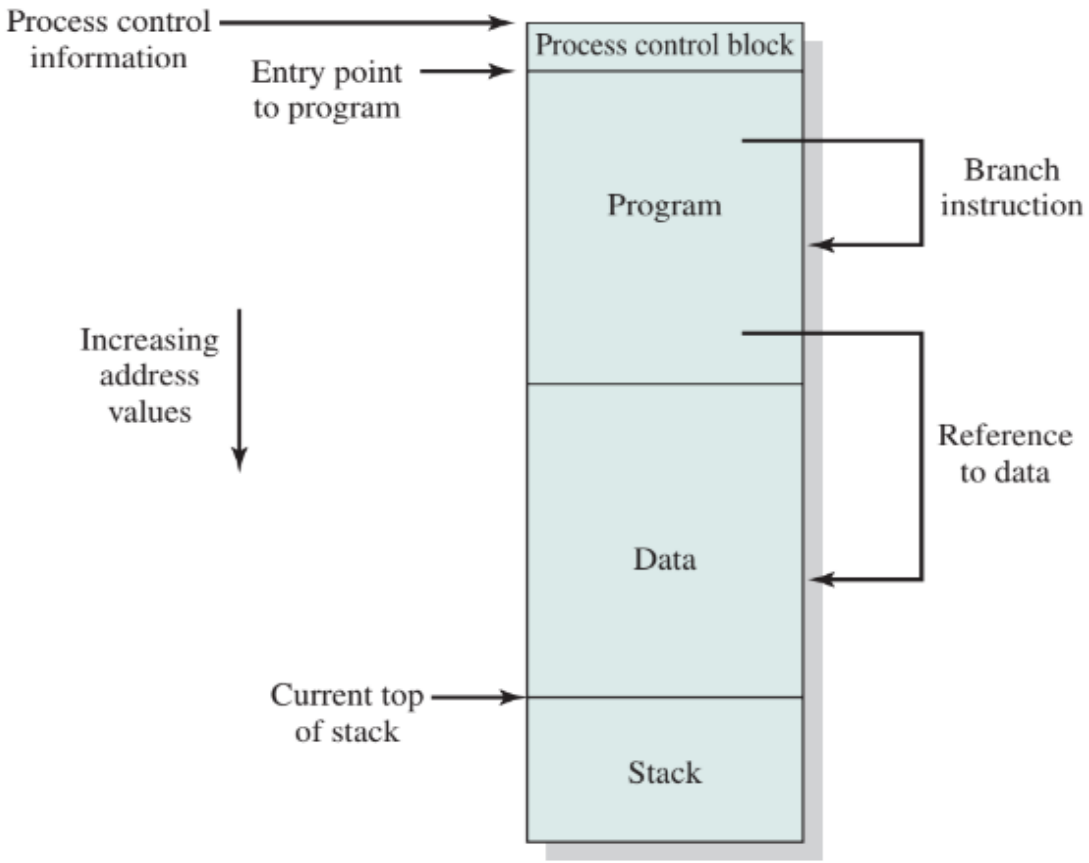


Figure 7.1 Addressing Requirements for a Process

The figure depicts a process image. For simplicity, let us assume that the process image occupies a contiguous region of main memory. Clearly, the operating system will need to know the location of process control information and of the execution stack, as well as the entry point to begin execution of the program for this process. Because the operating system is managing memory and is responsible for bringing this process into main memory, these addresses are easy to come by. In addition, however, the processor must deal with memory references within the program.

Branch instructions contain an address to reference the instruction to be executed next. **Data reference instructions** contain the address of the byte or word of data referenced.

Somehow, the processor hardware and operating system software must be able to translate the memory references found in the code of the program into actual physical memory addresses, reflecting the current location of the program in main memory.

2. Protection

Each process should be protected against unwanted interference by other processes, whether accidental or intentional. Thus, programs in other processes should not be able to reference memory locations in a process for reading or writing purposes without permission.

In one sense, **satisfaction of the relocation requirement increases the difficulty of satisfying the protection requirement.** Because the location of a program in main memory is unpredictable, it is impossible to check absolute addresses at compile time to assure protection. Most programming languages allow the dynamic calculation of addresses at run time. Hence **all memory references generated by a process must be checked at run time to ensure that they refer only to the memory space allocated to that process.**

Normally, a user process cannot access any portion of the operating system, neither program nor data. Again, usually a program in one process cannot branch to an instruction in another process. Without special arrangement, a program in one process cannot access the data area of another process. The processor must be able to abort such instructions at the point of execution.

Note that the memory protection requirement must be satisfied by the processor (hardware) rather than the operating system (software). To accomplish this, the processor hardware must have that capability.

3. Sharing

Any protection mechanism must have the flexibility to allow several processes to access the same portion of main memory.

For example,

1. if a number of processes are executing the same program, it is advantageous to allow each process to access the same copy of the program rather than have its own separate copy.
2. Processes that are cooperating on some task may need to share access to the same data structure.

The memory management system must therefore allow controlled access to shared areas of memory without compromising essential protection.

There are mechanisms used to support relocation support sharing capabilities.

4. Logical Organization

Main memory in a computer system is organized as a linear, or one-dimensional, address space, consisting of a sequence of bytes or words. Secondary memory, at its physical level, is similarly organized.

While this organization closely mirrors the actual machine hardware, it does not correspond to the way in which programs are typically constructed.

Most **programs are organized into modules,** some of which are unmodifiable (read only, execute only) and some of which contain data that may be modified.

If the operating system and computer hardware can effectively deal with user programs and data in the form of modules of some sort, then a number of advantages can be realized:

1. Modules can be written and compiled independently, with all references from one module to another resolved by the system at run time.
2. With modest additional overhead, different degrees of protection (read only, execute only) can be given to different modules.
3. It is possible to introduce mechanisms by which modules can be shared among processes. The advantage of providing sharing on a module level is that this corresponds to the user's way of viewing the problem, and hence it is easy for the user to specify the sharing that is desired.**The tool that most readily satisfies these requirements is segmentation.**

5. Physical Organization

Computer memory is organized into at least two levels, referred to as main memory and secondary memory. Main memory provides fast access at relatively high cost. In addition, main memory is volatile; that is, it does not provide permanent storage. Secondary memory is slower and cheaper than main memory and is usually not volatile. Thus secondary memory of large capacity can be provided for long-term storage of programs and data, while a smaller main memory holds programs and data currently in use. In this two-level scheme, the organization of the flow of information between main and secondary memory is a major system concern. **The task of moving information between the two levels of memory should be a system responsibility. This task is the essence of memory management.**

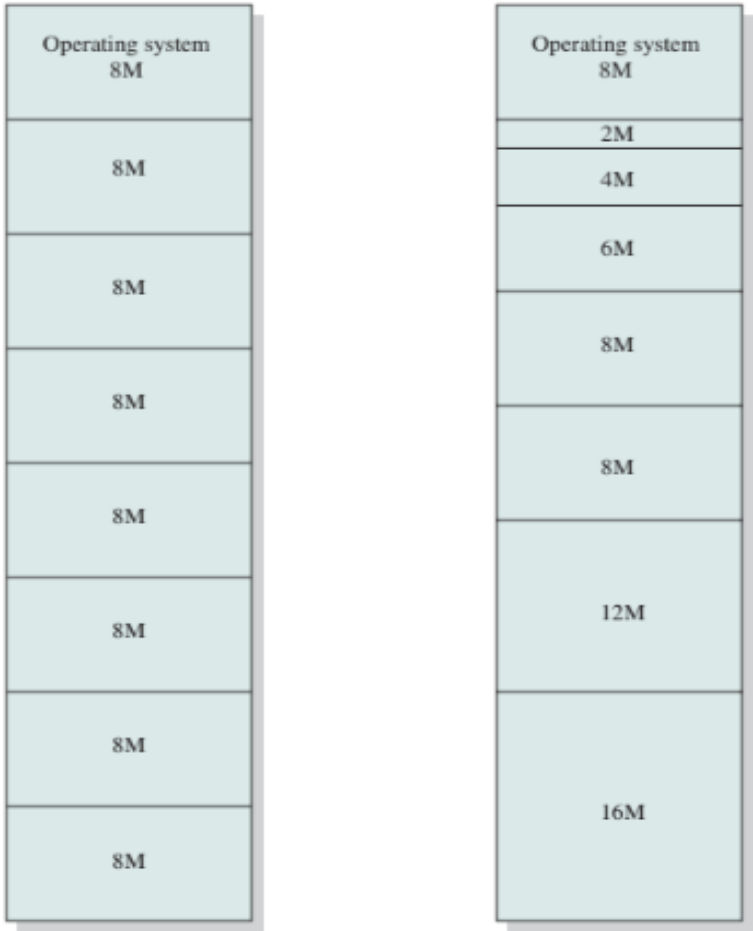
MEMORY PARTITIONING

The principal operation of memory management is to bring processes into main memory for execution by the processor. In almost all modern multiprogramming systems, this involves a sophisticated scheme known as **virtual memory**. Virtual memory is, in turn, based on the use of one or both of two basic techniques: **segmentation and paging.**

Before we can look at these virtual memory techniques, we must prepare the ground by looking at **simpler techniques** that **do not involve** virtual memory. One of these techniques, **partitioning**, has been used in several variations in some now-obsolete operating systems.

Fixed Partitioning

In most schemes for memory management, we can **assume that the OS occupies some fixed portion of main memory and that the rest of main memory is available for use by multiple processes.** The simplest scheme for managing this available memory is to **partition it into regions with fixed boundaries.**



(a) Equal-size partitions (b) Unequal-size partitions

Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

Fixed Partitioning

Main memory is divided into a number of static partitions at system generation time. **A process may be loaded into a partition of equal or greater size.**

Strengths

Simple to implement; little operating system overhead.

Weaknesses

Inefficient use of memory due to internal fragmentation; maximum number of active processes is fixed. **Causing memory utilization extremely inefficient**.

Dynamic Partitioning

Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.

Strengths

No internal fragmentation; more efficient use of main memory.

Weaknesses

Inefficient use of processor due to the need for compaction to counter external fragmentation.

Different types of addresses

When a process image is first created, it is **loaded** into some partition in main memory. Later, the process may be **swapped out**; when it is subsequently **swapped back in**, it may be assigned to a different partition than the last time. The same is true for **fixed as well as dynamic partitioning**.

Furthermore, when **compaction** is used, processes are shifted while they are in main memory. **Thus, the locations (of instructions and data) referenced by a process are not fixed.** They **will change** each time a process is swapped in or shifted. **To solve this problem, a distinction is made among several types of addresses.**

[One technique for overcoming external fragmentation is **compaction** : From time to time, the OS shifts the processes so that they are contiguous and so that all of the free memory is together in one block.]

A **physical address** , **or absolute address**, is an actual location in main memory.

A **logical address** is a reference to a memory location independent of the current assignment of data to memory; a translation must be made to a physical address before the memory access can be achieved. **Logical Address is generated by CPU while a program is running. The logical address is virtual address as it does not exist physically, therefore, it is also known as Virtual Address.** This address is used as a reference to access the physical memory location by CPU. The term **Logical Address Space** is used for the **set** of all logical addresses generated by a program's perspective.

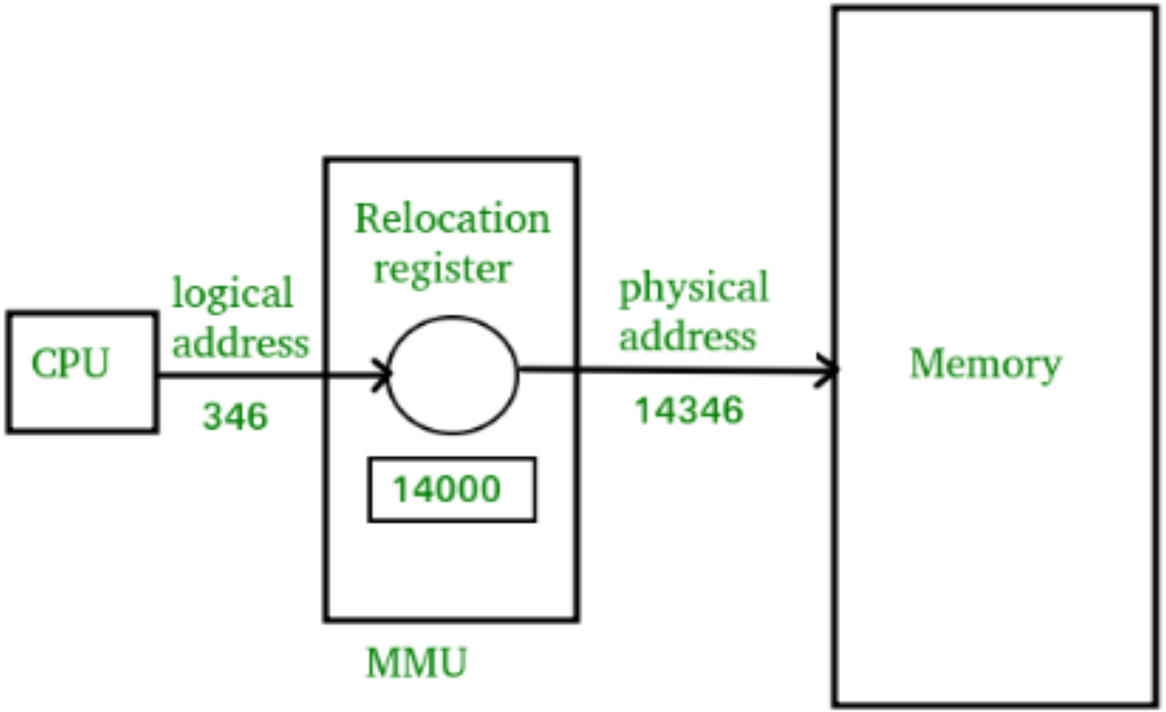
The hardware device called **Memory-Management Unit** is used for mapping logical address to its corresponding physical address.

A **physical address** , **or absolute address**, is an actual location in main memory.

Physical Address identifies a **physical location** of required data in a memory. The user never directly deals with the physical address but can access by its corresponding logical address.

The user program generates the logical address and thinks that the program is running in this logical address but the program needs physical memory for its execution, therefore, **the logical address must be mapped to the physical address by MMU before they are used.** The term Physical Address Space is used for all physical addresses corresponding to the logical addresses in a Logical address space.

Example:


Relocation
register
logical
address
CPU
346
14000
physical
address
14346
Memory
MMU

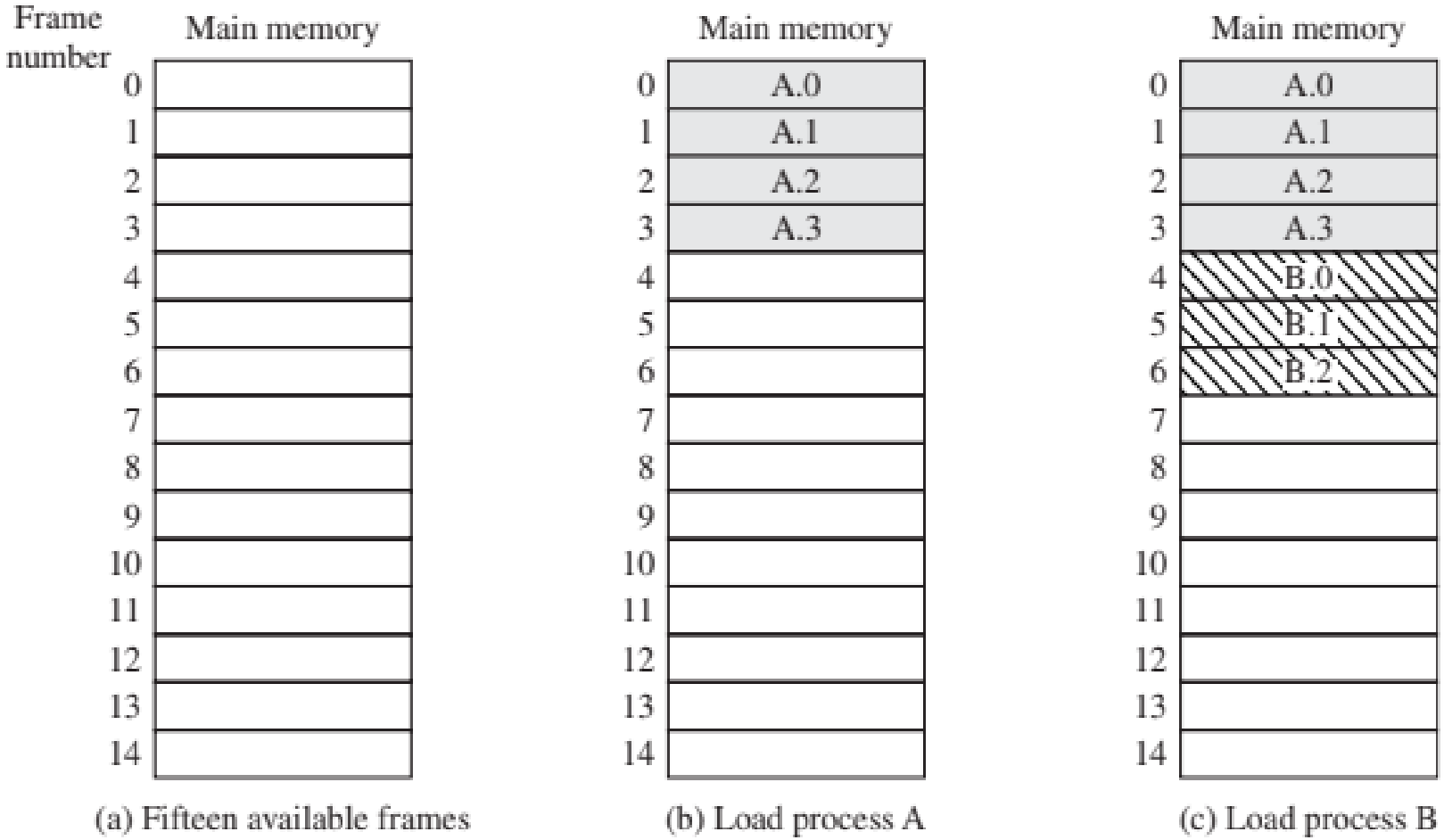
PAGING

Both unequal fixed-size and variable-size partitions are inefficient in the use of memory; the former results in internal fragmentation, the latter in external fragmentation. Suppose, however, that main memory is partitioned into equal fixed-size chunks that are relatively small, and that each process is also divided into small fixed-size chunks of the same size. Then the chunks of a process, known as **pages**, could be assigned to available chunks of memory, known as **frames**, or **page frames.**

The operating system maintains a **page table** for each process. The page table shows the frame location for each page of the process. Within the program, each logical address consists of a page number and an offset within the page.

With paging, the logical-to-physical address translation is still done by processor hardware.

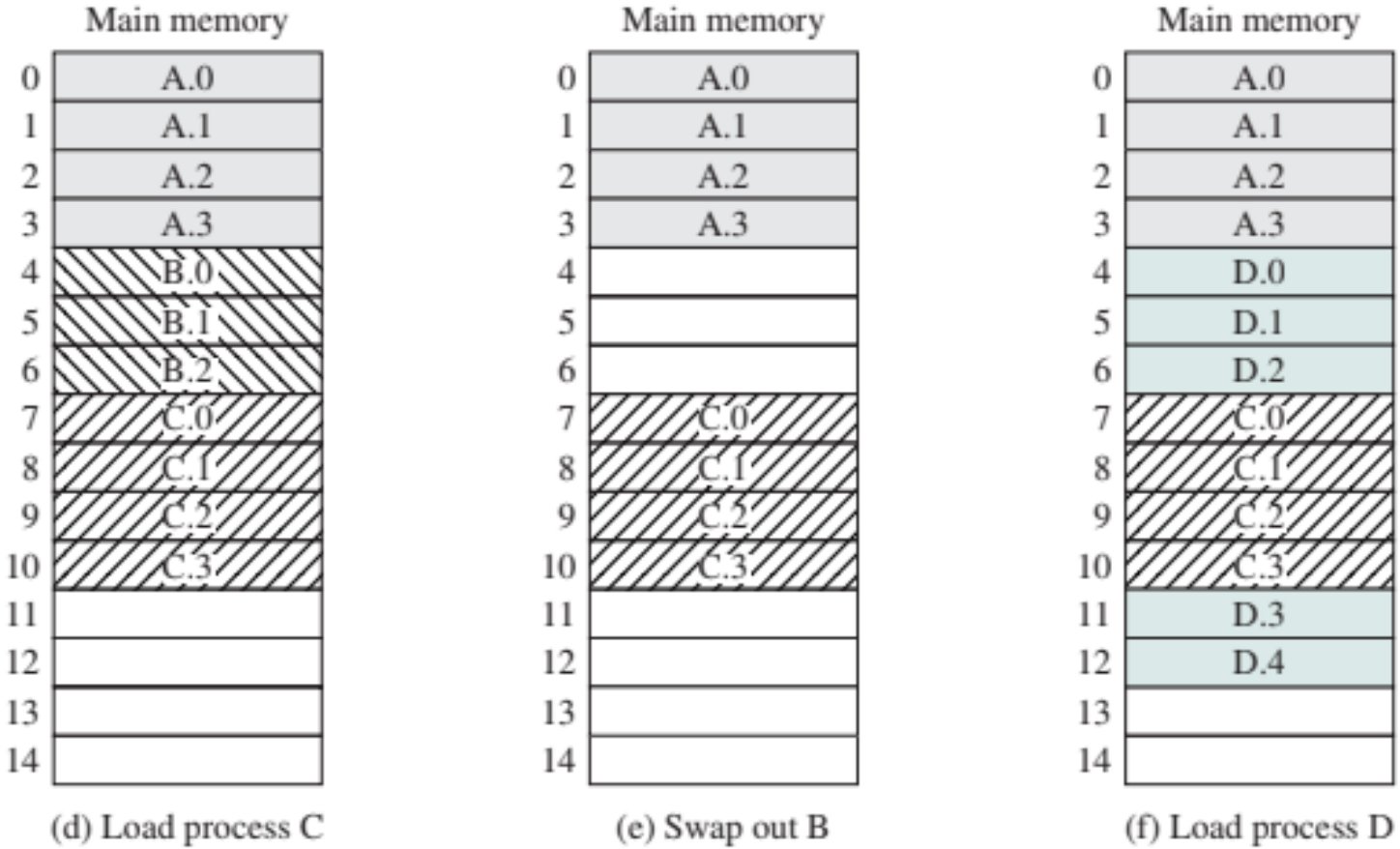
To make this paging scheme convenient, let us dictate that the page size, hence the frame size, must be a power of 2.



(a) Fifteen available frames

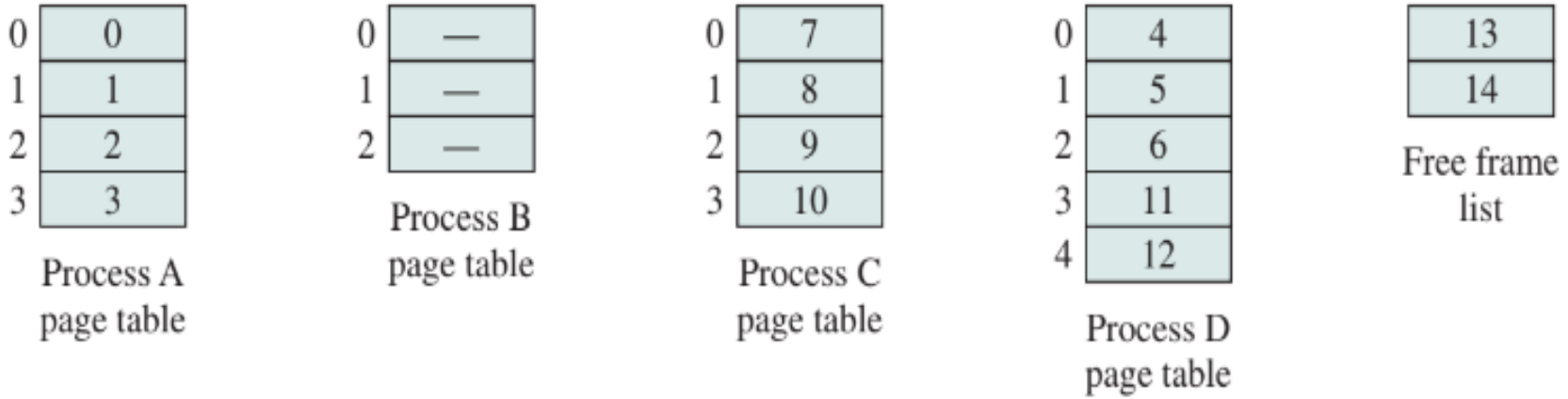
(b) Load process A

(c) Load process B



re 7.9 Assignment of Process to Free Frames

Page table example



Summary

With simple paging, main memory is divided into many small equal-size frames. Each process is divided into frame-size pages. Smaller processes require fewer pages; larger processes require more. When a process is brought in, all of its pages are loaded into available frames, and a page table is set up. This approach solves many of the problems inherent in partitioning.

SEGMENTATION

A user program can be subdivided using segmentation, in which the program and its associated data are divided into a number of segments . It is not required that all segments of all programs be of the same length, although there is a maximum segment length.

Because of the use of unequal-size segments, segmentation is similar to dynamic partitioning. In the absence of an overlay scheme or the use of virtual memory, it would be required that all of a program's segments be loaded into memory for execution.

The difference, compared to dynamic partitioning, is that with segmentation a program may occupy more than one partition, and these partitions need not be contiguous. Segmentation eliminates internal fragmentation but, like dynamic partitioning, it suffers from external fragmentation.

Whereas paging is invisible to the programmer, **segmentation is usually visible and is provided as a convenience for organizing programs and data.** Typically, the programmer or compiler will assign programs and data to different segments. For purposes of modular programming, the program or data may be further broken down into multiple segments. **The principal inconvenience of this service is that the programmer must be aware of the maximum segment size limitation.**

Another consequence of unequal-size segments is that there is no simple relationship between logical addresses and physical addresses. Analogous to paging, a simple segmentation scheme would make use of **a segment table** for each process and a list of free blocks of main memory. **Each segment table entry would have to give the starting address in main memory of the corresponding segment.** The entry should **also provide the length of the segment**, to assure that invalid addresses are not used. When a process enters the Running state, the address of its segment table is loaded into a special register used by the memory management hardware.

SECURITY ISSUES IN MEMORY

Main memory and virtual memory are system resources subject to security threats and for which security countermeasures need to be taken. The most obvious security **requirement is the prevention of unauthorized access to the memory contents of processes.**

If a process has not declared a portion of its memory to be shareable, then no other process should have access to the contents of that portion of memory. If a process declares that a portion of memory may be shared by other designated processes, then the security service of the **OS must ensure that only the designated processes have access.**

Buffer Overflow Attacks Threat

One serious security threat related to memory management remains to be introduced: buffer overflow , also known as a buffer overrun.

buffer overrun: A condition at an interface under which more input can be placed into a buffer or data-holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system.

A buffer overflow can occur as a result of a programming error when a process attempts to store data beyond the limits of a fixed-sized buffer and consequently overwrites adjacent memory locations. These locations could hold other program variables or parameters or program control flow data such as return addresses and pointers to previous stack frames. The buffer could be located on the stack, in the heap, or in the data section of the process. The consequences of this error include corruption of data used by the program, unexpected transfer of control in the program, possibly memory access violations, and very likely eventual program termination.

When done deliberately as part of an attack on a system, the transfer of control could be to code of the attacker’s choosing, resulting in the ability to execute arbitrary code with the privileges of the attacked process. **Buffer overflow attacks are one of the most prevalent and dangerous types of security attacks.**

SUMMARY

One of the most important and complex tasks of an operating system is memory management. Memory management involves treating main memory as a resource to be allocated to and shared among a number of active processes. To use the processor and the I/O facilities efficiently, it is desirable to maintain as many processes in main memory as possible. In addition, it is desirable to free programmers from size restrictions in program development.

The basic tools of memory management are paging and segmentation.

With paging, each process is divided into relatively small, fixed-size pages.

Segmentation provides for the use of pieces of varying size. It is also possible to combine segmentation and paging in a single memory management scheme.

Virtual Memory Terminologies

Virtual memory
A **storage allocation scheme** in which **secondary memory can be addressed as though it were part of main memory.** The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program-generated addresses are translated automatically to the corresponding machine addresses. **The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of secondary memory available and not by the actual number of main storage locations.**

Virtual address The address assigned to a location in virtual memory to allow that location to be accessed as though it were part of main memory.

Virtual Memory Terminologies

Virtual address space The virtual storage assigned to a process.

Address space The range of memory addresses available to a process.

Real address The address of a storage location in main memory.

HARDWARE AND CONTROL STRUCTURES

Comparing simple paging and simple segmentation, on the one hand, with fixed and dynamic partitioning, on the other, we see the foundation for a fundamental breakthrough in memory management.

Two characteristics of paging and segmentation are the keys to this breakthrough:

1. All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a process may be swapped in and out of main memory such that it occupies different regions of main memory at different times during the course of execution.

2. A process may be broken up into a number of pieces (pages or segments) and these pieces need not be contiguously located in main memory during execution. The combination of dynamic run-time address translation and the use of a page or segment table permits this.

Now we come to the breakthrough. **If the preceding two characteristics are present, then it is not necessary that all of the pages or all of the segments of a process be in main memory during execution. If the piece (segment or page) that holds the next instruction to be fetched and the piece that holds the next data location to be accessed are in main memory, then at least for a time execution may proceed.**

How Virtual Memory technique may be accomplished?
We will use the term **piece** to refer to either page or segment, depending on whether paging or segmentation is employed.

Suppose that it is time to bring a new process into memory. The OS begins by bringing in only **one or a few pieces**, to include the initial program piece and the initial data piece to which those instructions refer. **The portion of a process that is actually in main memory at any time is called the resident set of the process.**

1. As the process executes, things proceed smoothly as long as all memory references are to locations that are in the resident set. Using the segment or page table, the processor always is able to determine whether this is so.
2. If the processor encounters a **logical address that is not in main memory**, it generates an interrupt indicating a **memory access fault**. The OS puts the interrupted process in a **blocking state**.
3. For the execution of this process to proceed later, the OS must bring into main memory the piece of the process that contains the logical address that caused the access fault. For this purpose, **the OS issues a disk I/O read request.**
4. After the I/O request has been issued, the OS can dispatch another process to run while the disk I/O is performed.
5. Once the desired piece has been brought into main memory, an I/O interrupt is issued, giving control back to the OS, which places the affected process back into a Ready state.

There are two implications of Virtual Memory strategy.

1. **More processes may be maintained in main memory.** Because we are only going to load some of the pieces of any particular process, there is room for more processes. This leads to more efficient utilization of the processor because it is more likely that at least one of the more numerous processes will be in a Ready state at any particular time.

2. A process may be larger than all of main memory.
One of the most **fundamental restrictions that a programmer must be acutely aware of how much memory is available ?**, in programming **is lifted.** With virtual memory based on paging or segmentation, that job is left to the OS and the hardware. As far as the programmer is concerned, he or she is dealing with a huge memory, the size associated with disk storage. **The OS automatically loads pieces of a process into main memory as required.**

Because a process executes only in main memory, that memory is referred to as **real memory**. But a programmer or user perceives a potentially much larger memory that which is allocated on disk. This latter is referred to as **virtual memory**.

Virtual memory allows for very effective multiprogramming and relieves the user of the unnecessarily tight constraints of main memory. The OS automatically loads pieces of a process into main memory as required.