COP 4600 – Summer 2014

Introduction To Operating Systems

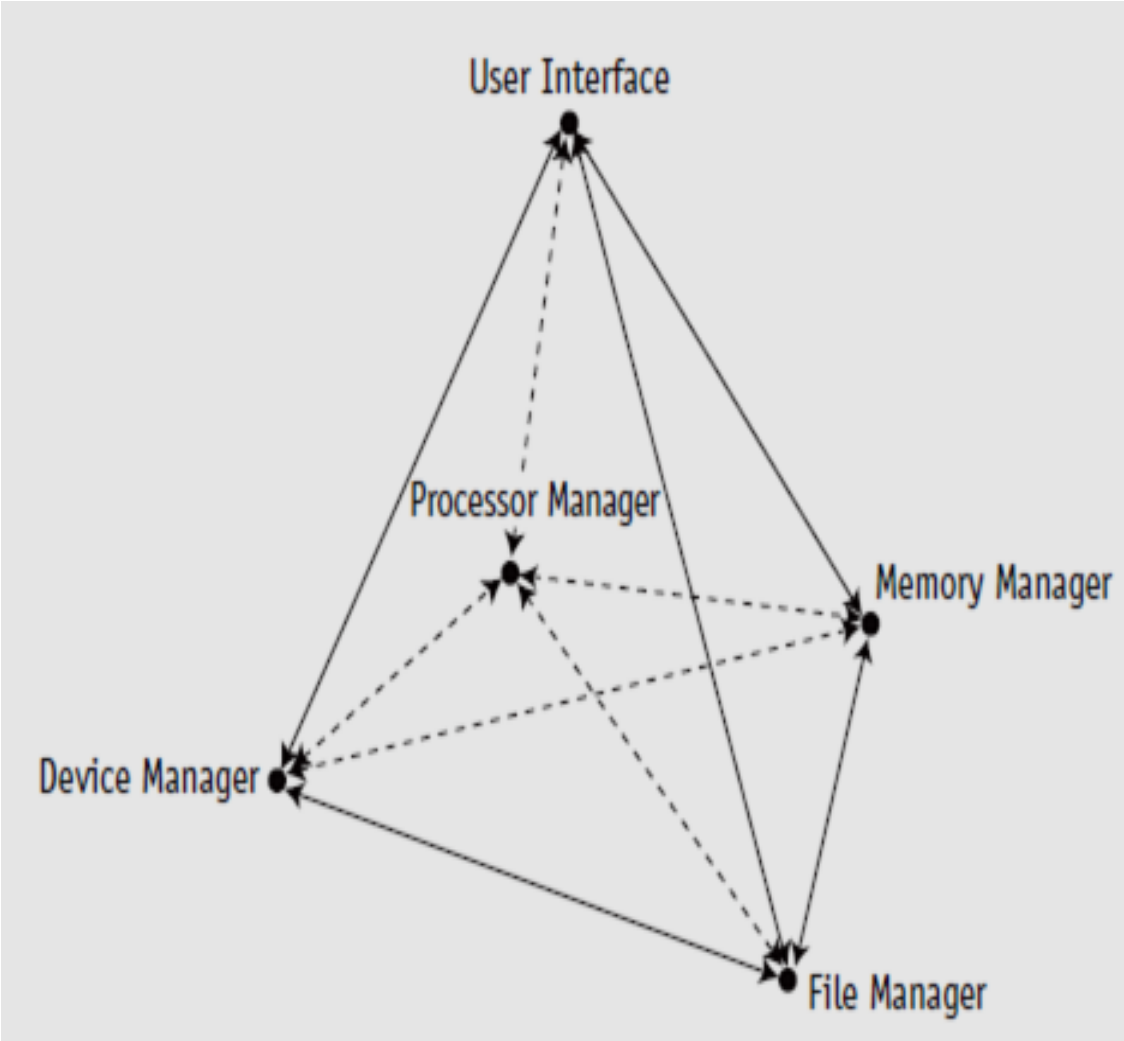
Memory Management – Part 1

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cop4600/sum2014

Department of Electrical Engineering and Computer Science
Computer Science Division
University of Central Florida

Memory Management

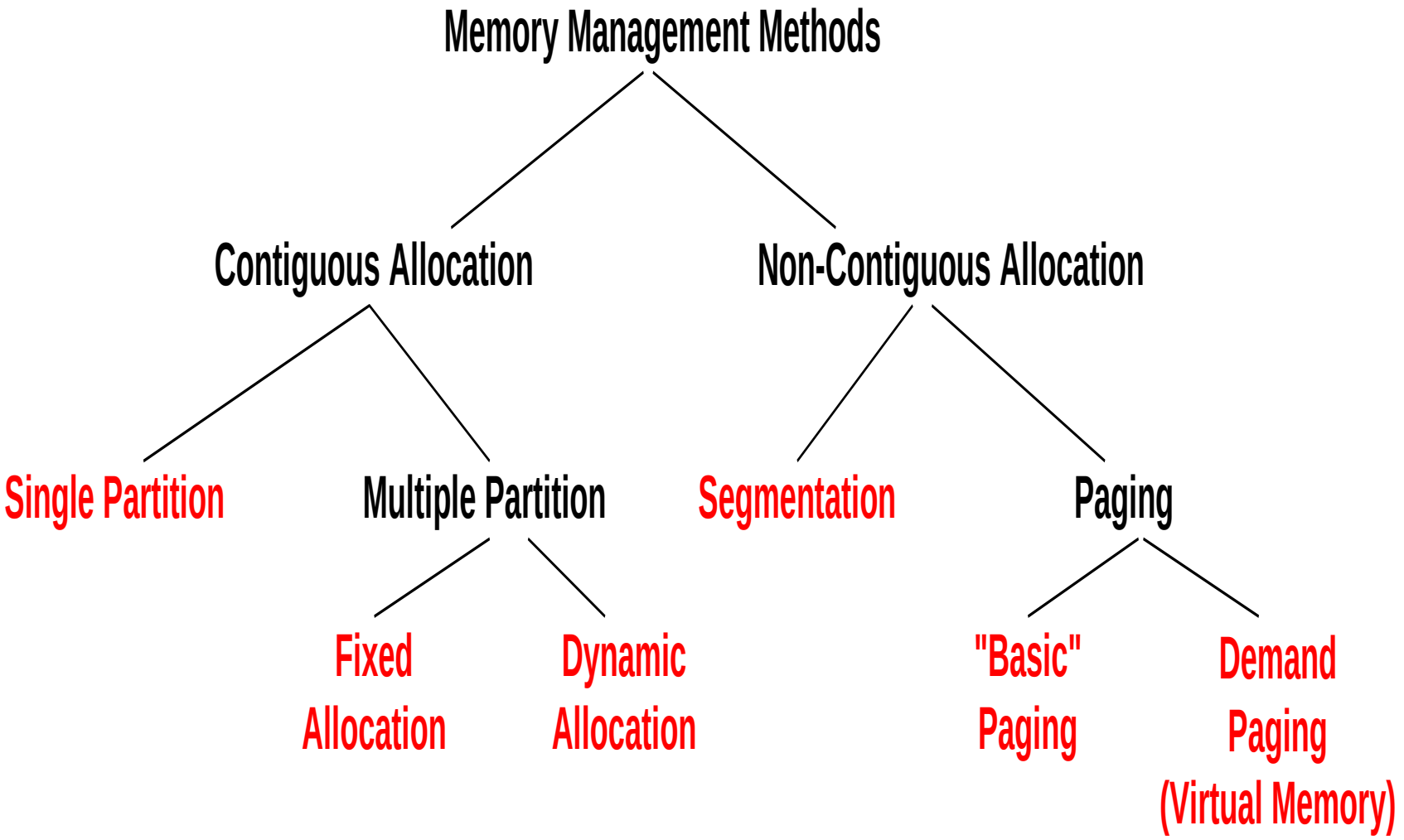
- Previously in the course, we discussed the processor management side of the OS.
- Now we turn our attention to the memory management side of things.



Memory Management

- In a uniprogramming environment, main memory is divided into two parts: one part for the OS and one part for the program currently being executed.
- In a multiprogramming environment, the "user" part of memory must be further subdivided to accommodate multiple processes.
- The task of subdivision is carried out dynamically by the OS and is referred to as memory management.
- Effective memory management is vital in a multiprogramming environment. 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 the available processor time.

Memory Management



Memory Management Requirements

- The memory management component of the operating system must satisfy five basic requirements:

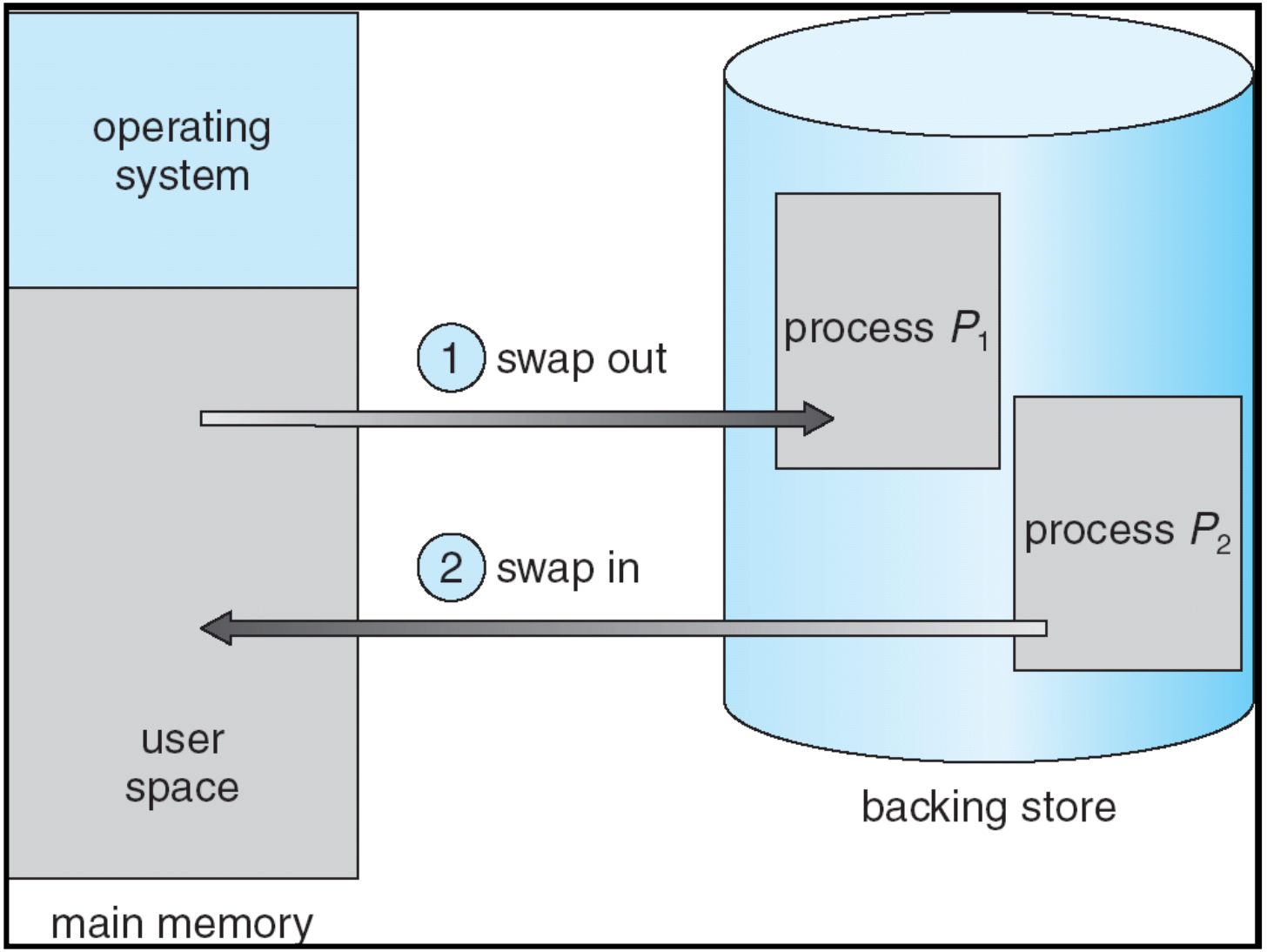
  1) Relocation
  2) Protection
  3) Sharing
  4) Logical organization
  5) Physical organization

- We'll consider each of these requirements separately.

(1) Relocation

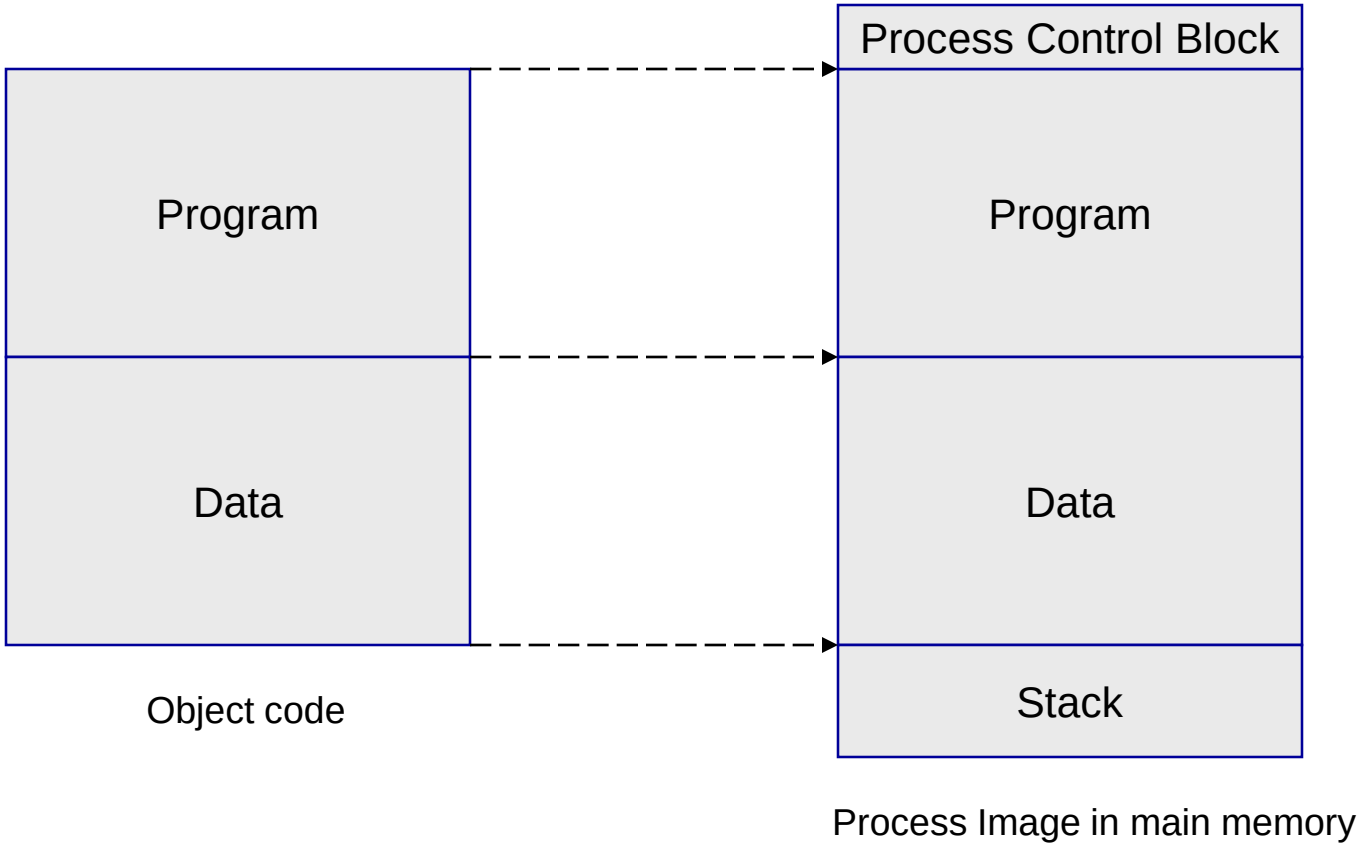
- In a multiprogramming system, the available main memory is generally shared among a number of processes.
- Typically, it is not possible for the programmer to know in advance which other programs will be resident in main memory at the time of execution of their program.
- In addition, we would like 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. (See next page for an illustration of swapping.)
- Once a program has been swapped out to disk, it would be quite limiting to require that it be placed into the same memory region it previously vacated when it is swapped back into the main memory. Rather, we would like to relocate the process to a different area of the memory.

(1) Relocation - Swapping



Aside on Linking and Loading

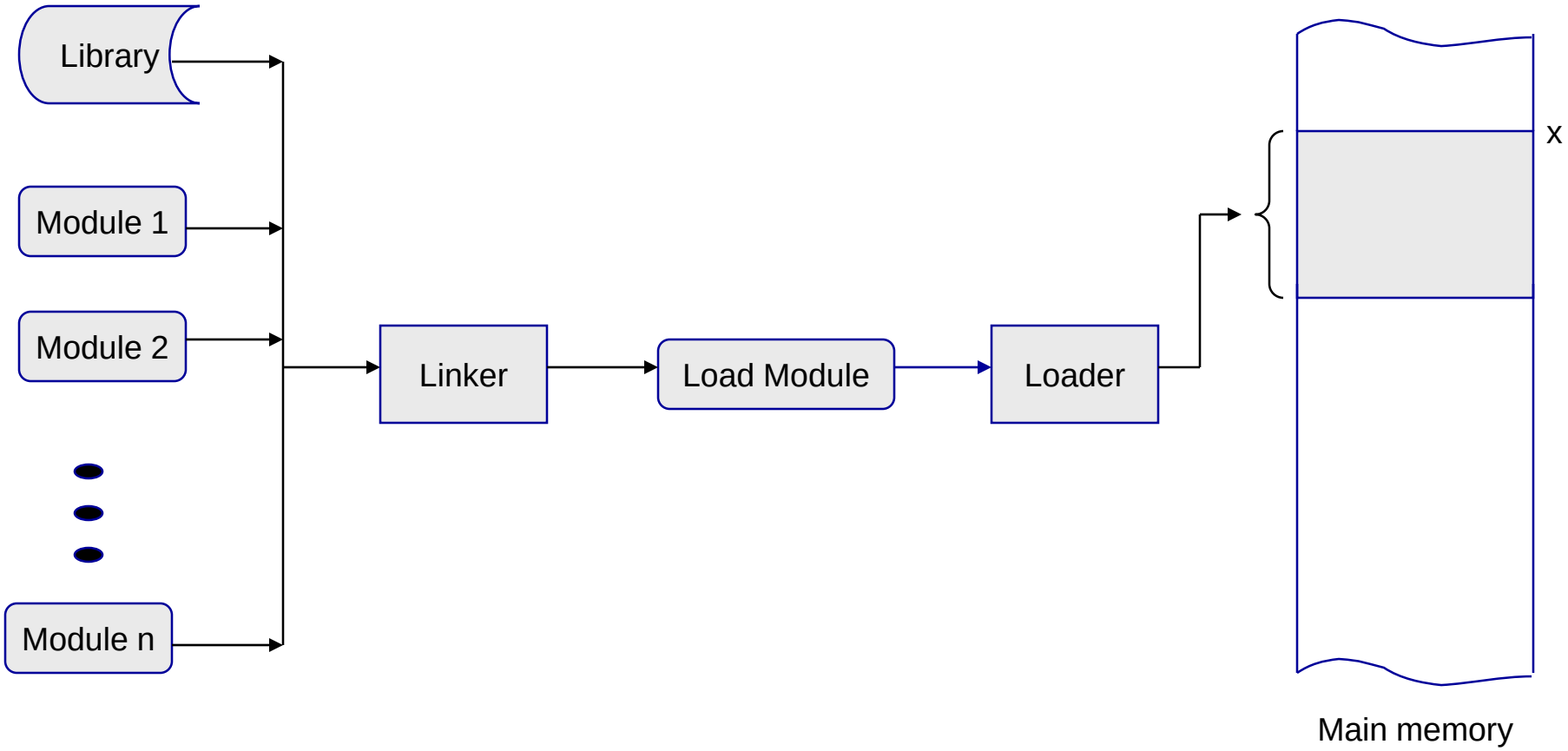
- The first step in the creation of an active process is to load a program into main memory and create a process image.



Aside on Linking and Loading (cont.)

- Typically, applications consist of a number of compiled or assembled modules in object code form.
- These are linked to resolve any references between the modules. At the same time, references to library routines are resolved. The library routines themselves may be incorporated into the program or referenced as shared code that must be supplied by the OS at run time.
- This scenario is illustrated on the next page.

Aside on Linking and Loading (cont.)



Relocation (cont.)

- A physical address, or absolute address, is an actual location in main memory. These are the addresses with which the memory unit works.
- A logical address or virtual 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 addresses are generated by the CPU.
- A relative address is a particular example of a logical address, in which the address is expressed as a location relative to some known point, usually a value in a processor register.
- Programs that employ relative addressing are loaded using dynamic run-time loading. Typically, all of the memory references in the loaded process are relative to the origin of the program. Thus, a hardware mechanism is required for translating relative addresses into physical addresses at the time of execution of the instruction that contains the reference.

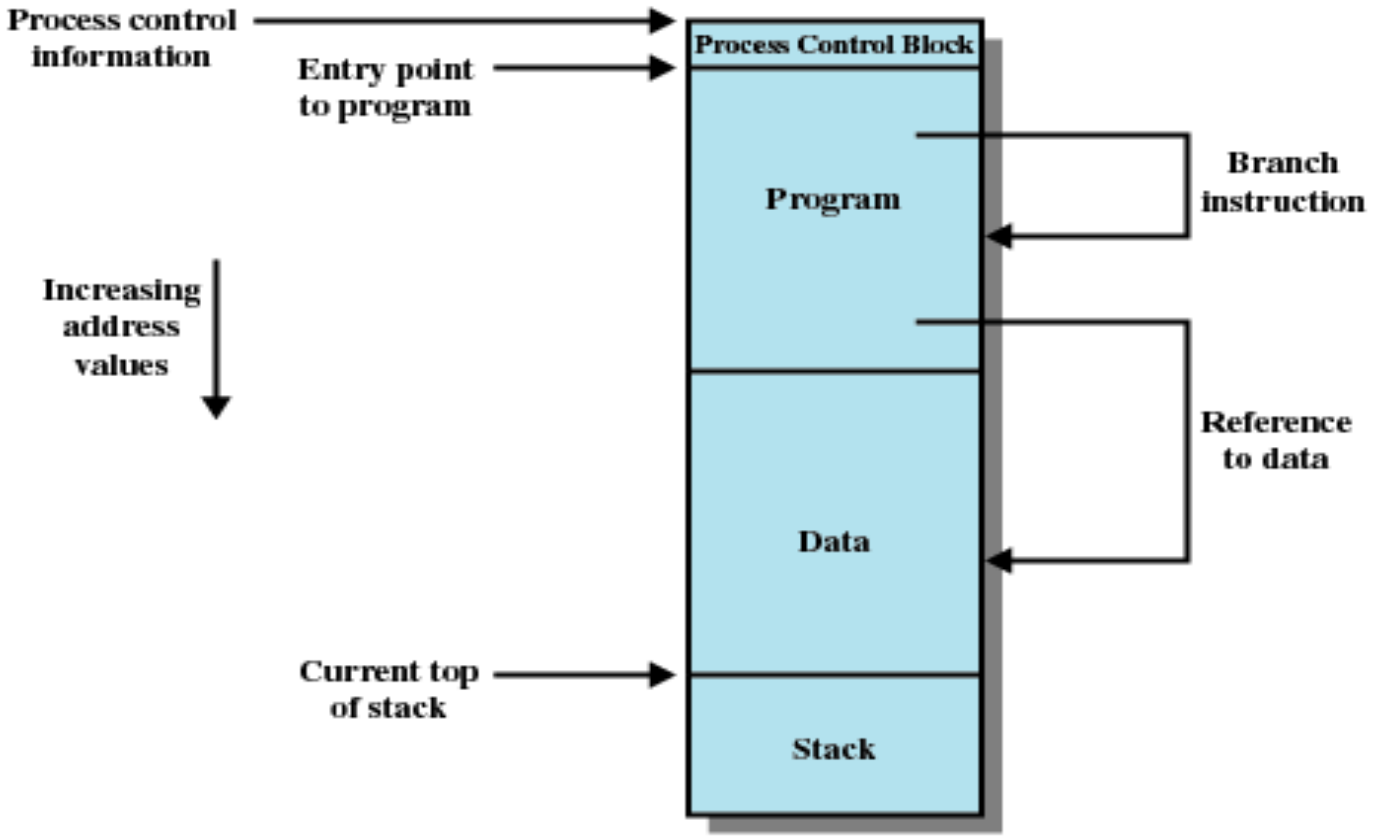
Aside on Linking and Loading (cont.)

Loading

- Looking at the figure on page 10, you'll see that the loader places the load module in main memory starting at location *x*. In loading the program, the addressing requirements of the program (see next page) must be satisfied.
- In general three different approaches can be taken:

1. Absolute loading
2. Relocatable loading
3. Dynamic run-time loading

Aside on Linking and Loading (cont.)



Addressing Requirements for a Process

Aside on Linking and Loading (cont.)

Absolute Loading

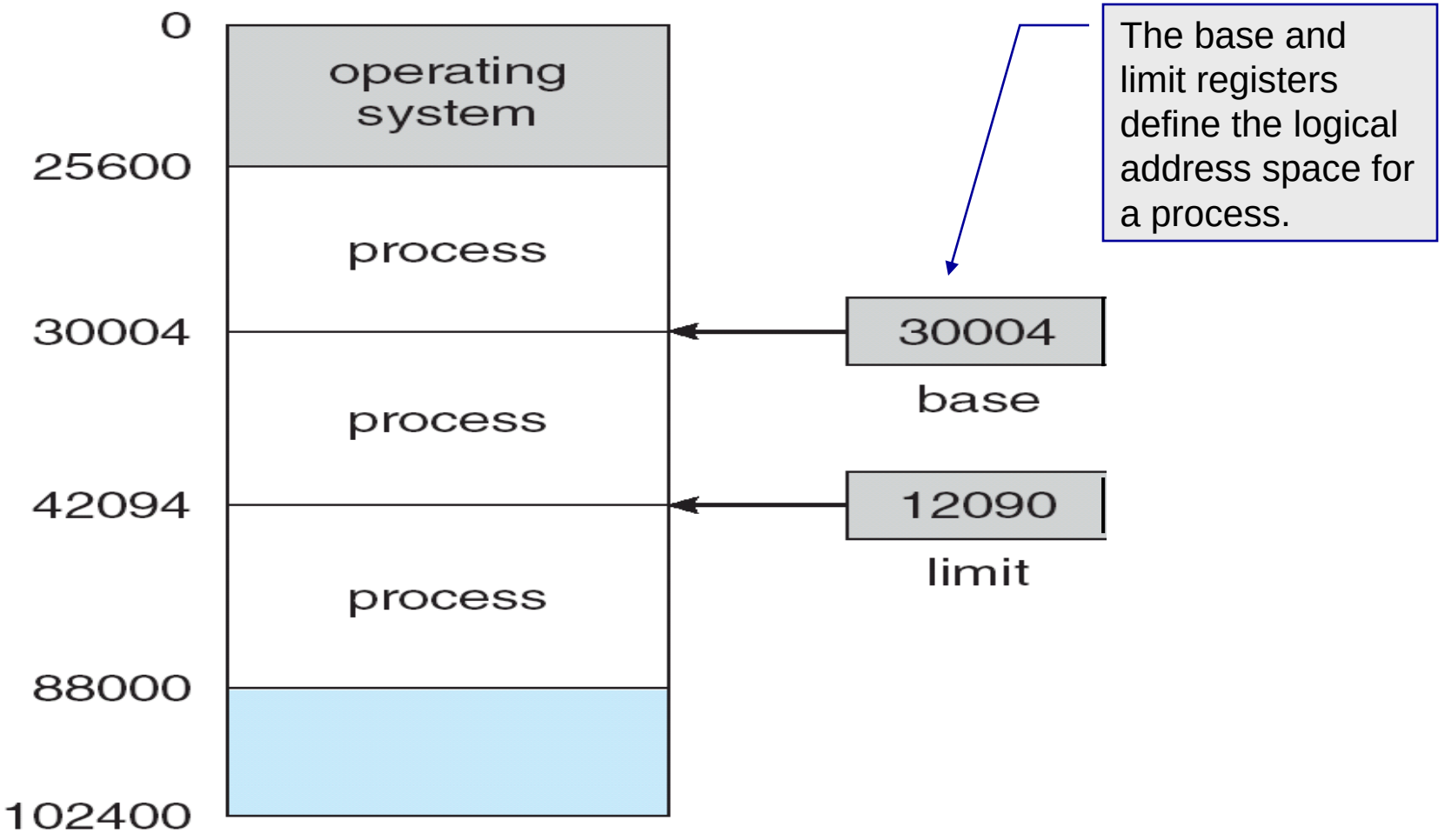
- An absolute loader requires that a given load module always be loaded into the same location in main memory.
- Thus, in the load module presented to the loader, all address references must be specific, or absolute, main memory addresses.
  - For example, if *x* in the figure on page 10 is location 1024, then the first word in a load module destined for that region of memory has address 1024.
- The assignment of specific address values to memory references within a program can be done either by the programmer or at compile or assembly time.

Aside on Linking and Loading (cont.)

Relocatable Loading

- The disadvantage of binding memory references to specific addresses prior to loading is that the resulting load module can only be placed in one region of main memory.
- The relocation requirement implies that we would like to be able to load a module anywhere in main memory.
- To satisfy this requirement, the assembler or compiler produces not absolute addresses but relative addresses.
- The start of the load module is assigned relative address 0 and all memory references within the module are expressed in terms relative to 0.
- The loader now has the simple task of placing the module wherever room is available. If, as in the example, the module is placed beginning at memory location *x*, then the loader must simply add *x* to each memory reference as it loads the module into memory.

Aside on Address Binding (cont.)



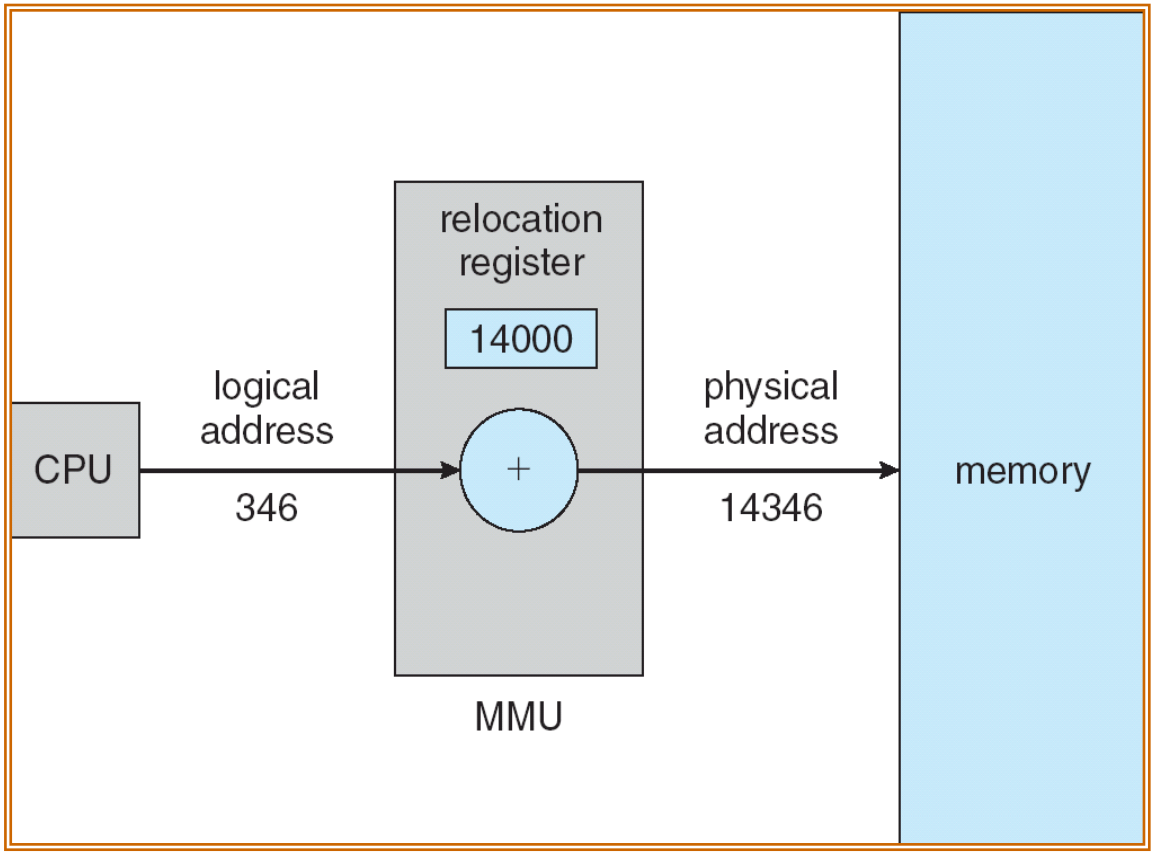
Aside on Linking and Loading (cont.)

Dynamic Run-Time Loading

- Relocatable loaders are common and provide obvious benefits over absolute loaders. However, in a multiprogramming environment, even one that does not depend on virtual memory, the relocating scheme is inadequate.
- Since we may swap a given process in and out of main memory many times during its lifetime, binding relative addresses to absolute addresses at the initial load time will not give us the flexibility needed to relocate the same process image into different locations.
- The alternative is to defer the calculation of an absolute address until it is actually needed at run time.
- For this purpose, the load module is loaded into main memory with all memory references in relative form. It is not until an instruction is executed that the absolute address is calculated.
- To assure that this function does not degrade performance, it is handled by special processor hardware rather than software.

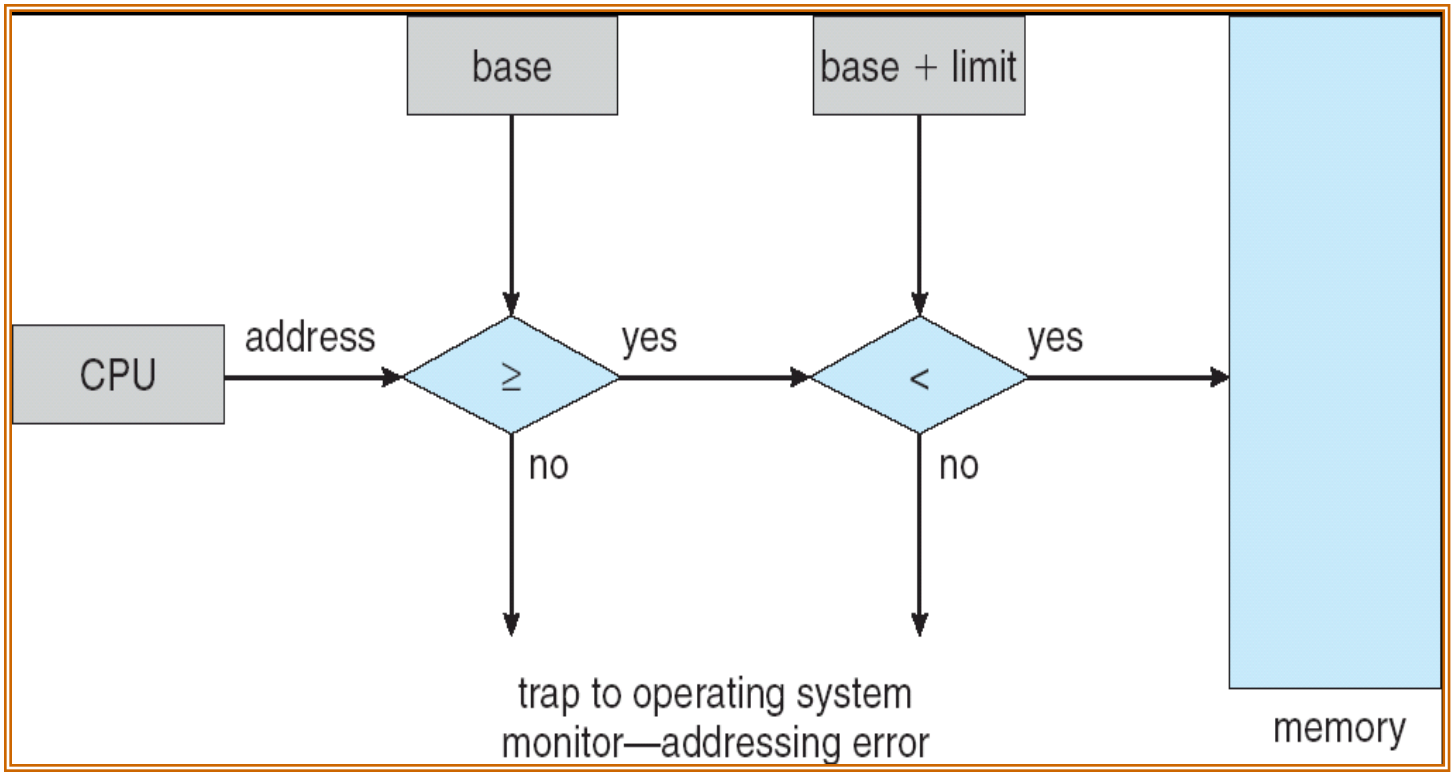
Aside on Linking and Loading (cont.)

Dynamic Run-Time Loading



Aside on Linking and Loading (cont.)

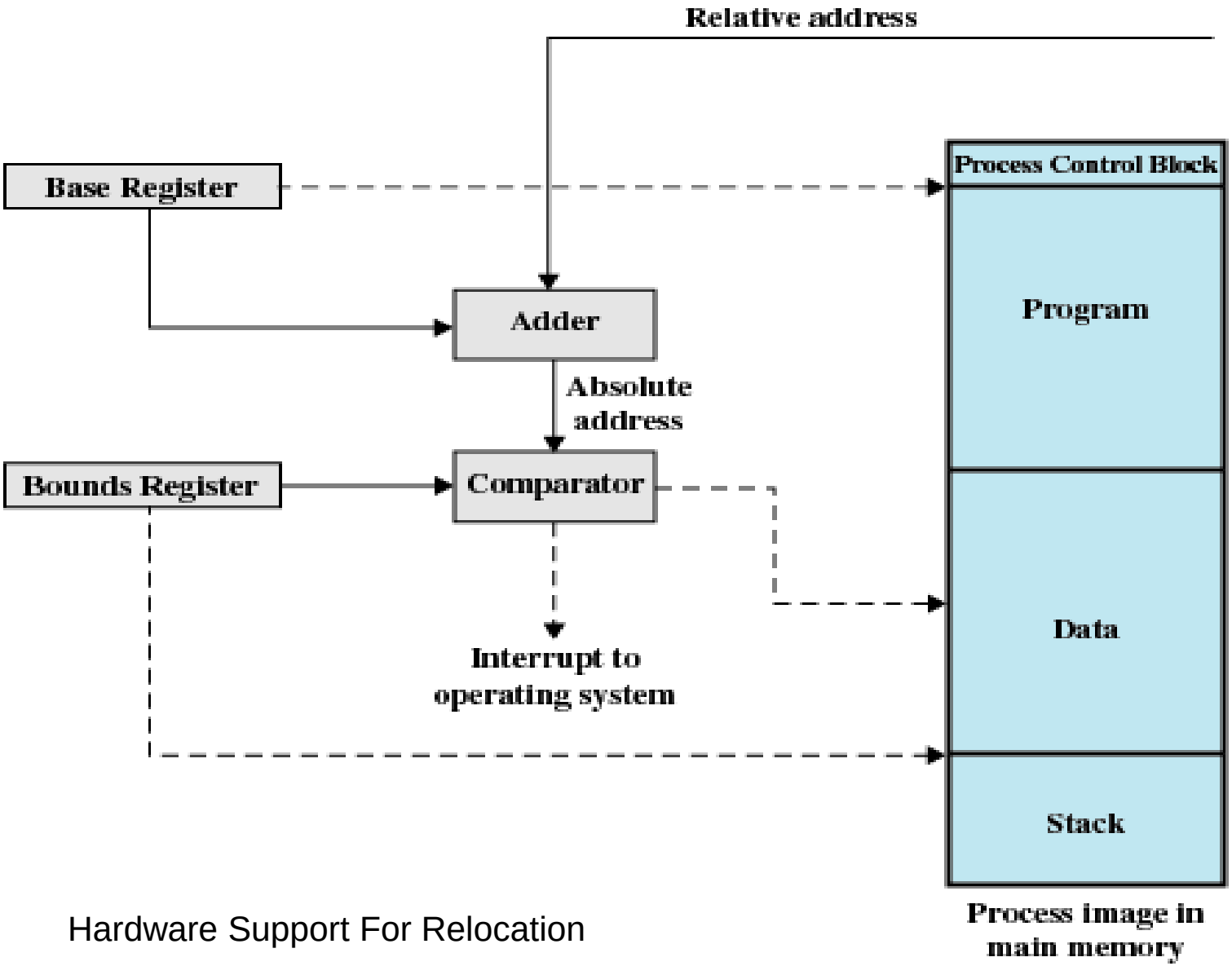
Dynamic Run-Time Loading



Aside on Linking and Loading (cont.)

Memory Management Unit (MMU)

- The MMU is a hardware device that maps logical addresses to physical addresses.
- Base register replaced by relocation register.
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

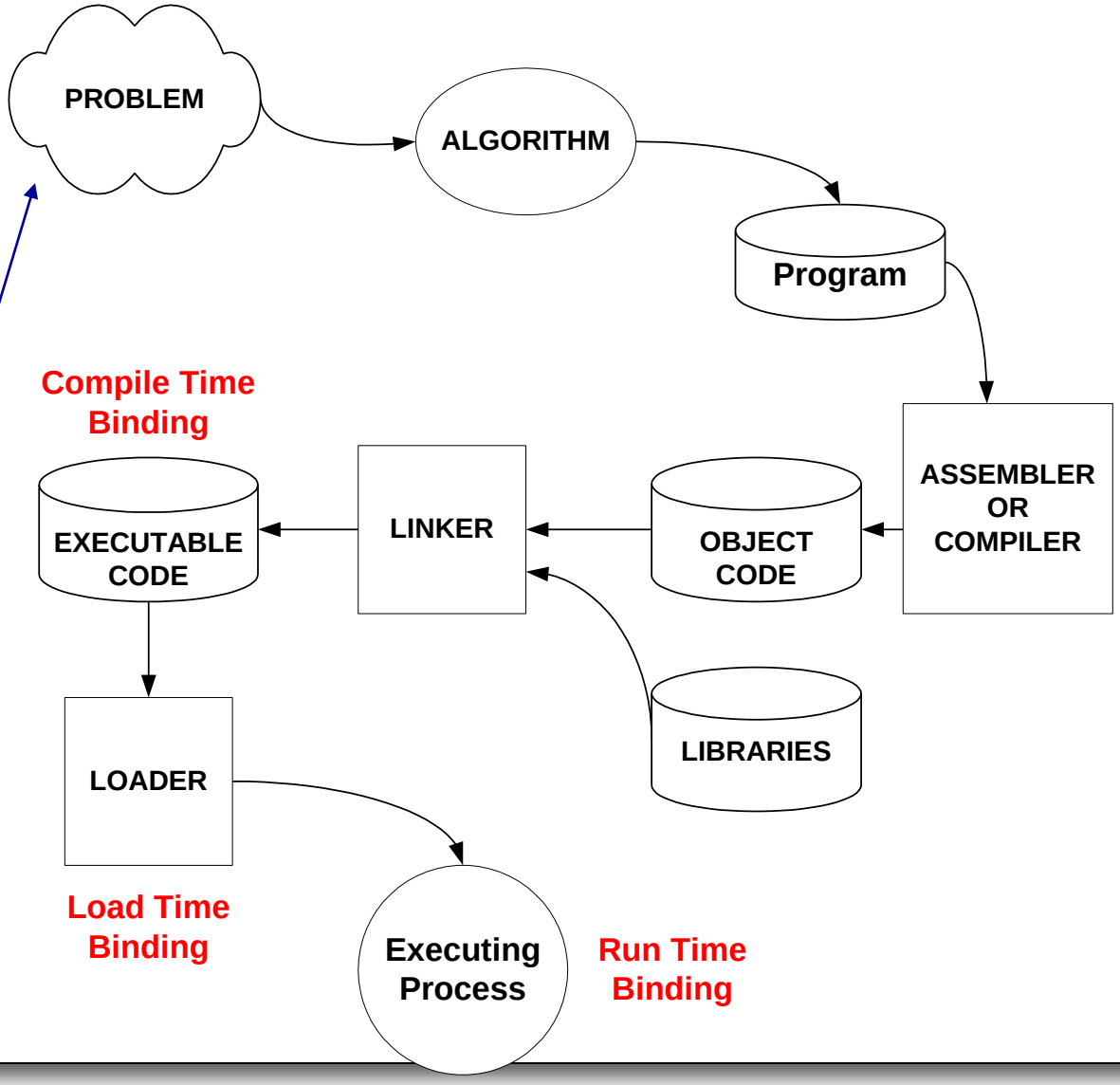


Hardware Support For Relocation

Aside on Address Binding

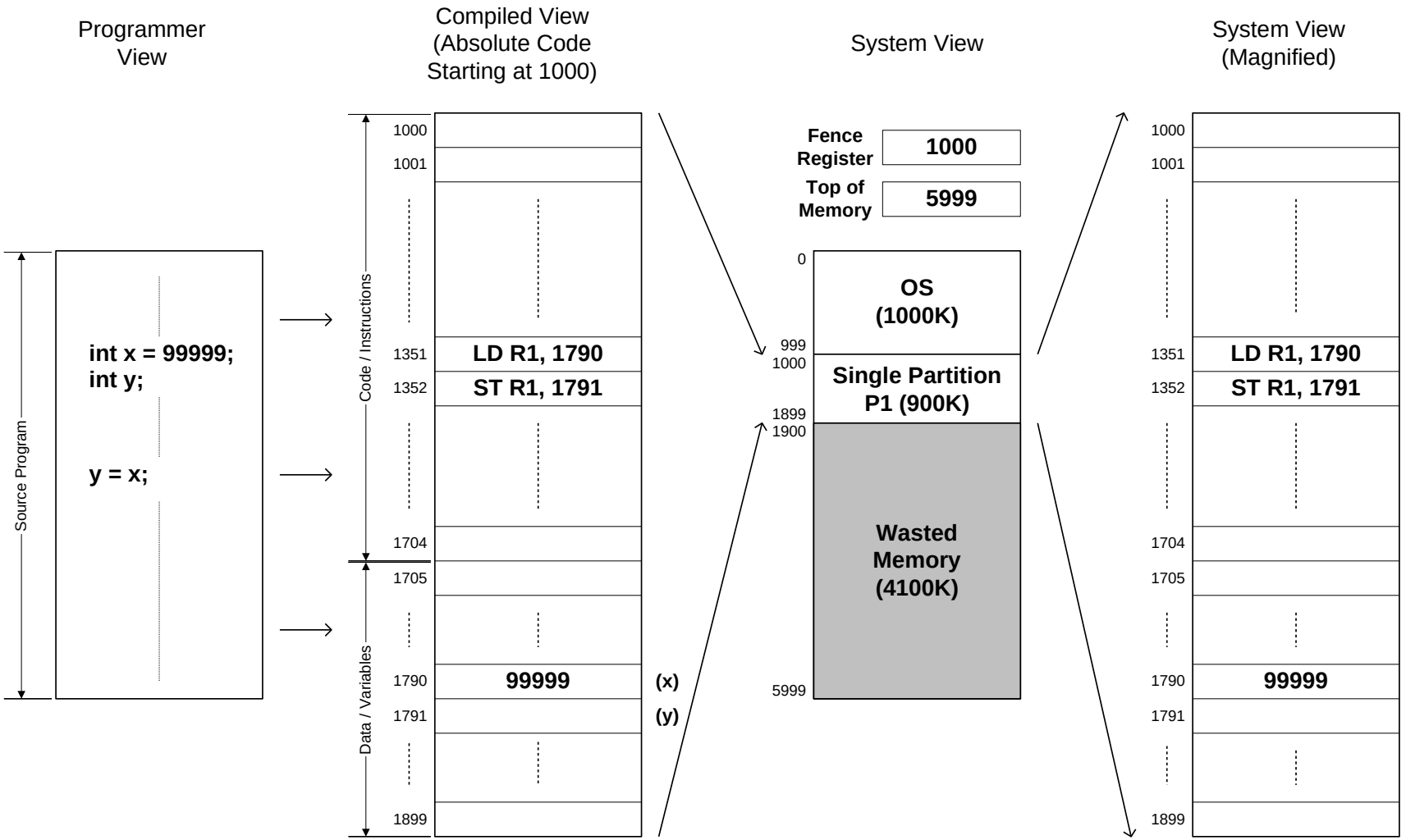
- In order to load a program, instructions and associated data must be mapped or "bound" to specific locations in memory
- Address binding can happen at three different stages.
  - Compile time:
    - If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.
  - Load time:
    - Must generate relocatable code if memory location is not known at compile time.
  - Execution or Run time:
    - Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Special hardware (e.g., MMUs) required for address mapping.

Aside on Address Binding (cont.)

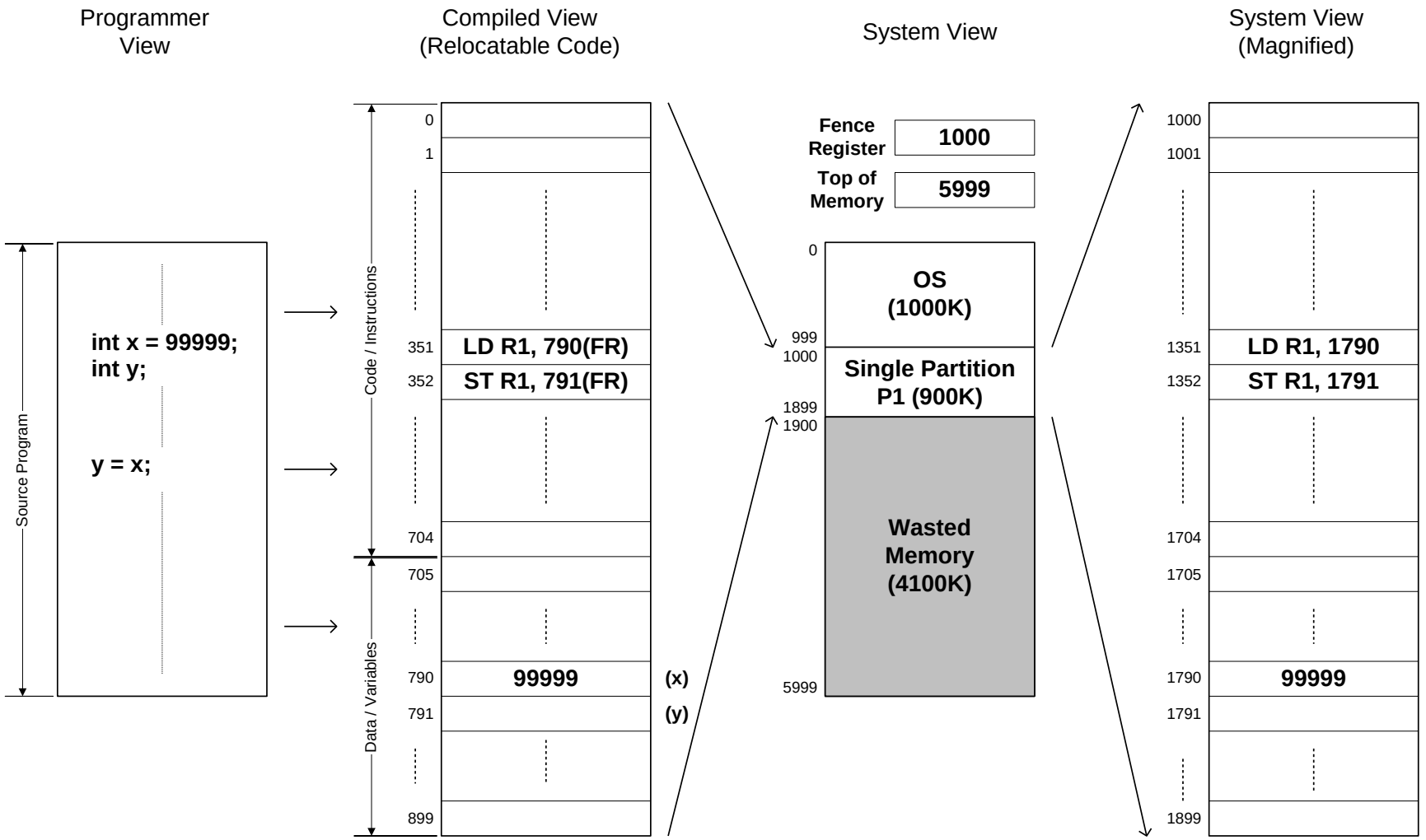


User programs goes through several steps before being executed

SINGLE PROGRAM, COMPILE TIME BINDING



SINGLE PROGRAM, LOAD TIME BINDING



(1) Relocation (cont.)

- Since we cannot know ahead of time where a program will be placed in the memory, and we must allow it to be moved about in the memory due to swapping, some technical concerns related to addressing must be considered.
- The diagram on page 24 shows a process image. For simplicity, assume that the process image occupies a contiguous region of main memory. Obviously, the OS will need to know the location of the PCB and the execution stack, as well as the entry point to begin execution of the program for this process.
- Since the OS is managing the memory and is responsible to loading the process into main memory, these addresses are easy to come by.

(1) Relocation (cont.)

- However, the processor must also deal with memory references within the program.
- Branch instructions contain an address which reference the instruction to be executed next.
- Data reference instructions contain the address of the word or byte of the data referenced.
- Somehow, the processor hardware and the OS software must be able to translate the memory references found in the code of the program into actual physical memory addresses which reflect 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.
- 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, satisfying 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.
- Furthermore, most programming languages allow the dynamic calculation of addresses at run time (e.g., computing an array subscript or a pointer into a data structure).

(2) Protection (cont.)

- Because of this situation, 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.
- Fortunately, as we shall see later, the mechanisms that support relocation also support the protection requirement.
- Normally, a user process cannot access any portion of the OS, 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.

(2) Protection (cont.)

- Note that the memory protection requirement must be satisfied by the processor (hardware) rather than the OS (software).
- This is because the OS cannot anticipate all of the memory references that a program will make. Even if such anticipation were possible, it would be prohibitively time consuming to screen each program in advance for possible memory-reference violations.
- Thus, it is only possible to assess the permissibility of a memory reference (either a data access or a branch) at the time of execution of the instruction making the reference.
- To accomplish this, the processor hardware must have this capability.

(3) Sharing

- Any protection mechanism must have the flexibility to allow several processes to access the same portion of main memory.
  - For example, if a number of processes are executing the same program, it is advantageous to allow each program to access the same copy of the program rather than have its own separate copy.
  - Processes that are cooperating on some task may need to share access to the same data structure.
- The memory management system must allow controlled access to shared areas of the memory without compromising essential protection.
- As before, the mechanisms that are used to support relocation also support sharing capabilities. We'll see evidence of this later.

(4) Logical Organization

- Main memory in a computer system is almost always organized as a linear, or one-dimensional, address space, consisting of a sequence of bytes or words.
- While this organization closely mirrors the actual machine hardware, it does not correspond to the way 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 OS 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:

(4) Logical Organization (cont.)

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 only slight 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 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 mechanism that most readily satisfies these requirements is segmentation, which we will see a bit later.

(5) Physical Organization

- Computer memory is typically organized into a least two levels, referred to as main memory and secondary memory.

Main memory

- Provides fast access at relatively high cost.
- Volatile (not permanent – requires refresh)

Secondary memory

- Slower and cheaper
- Typically not volatile
- Typically massive amounts

(5) Physical Organization (cont.)

- The flow of information between the main and secondary memory is a major system concern in this two-level memory organization scheme.
- While the responsibility for this flow could be assigned to the individual programmer, but this is impractical and undesirable for two reasons:

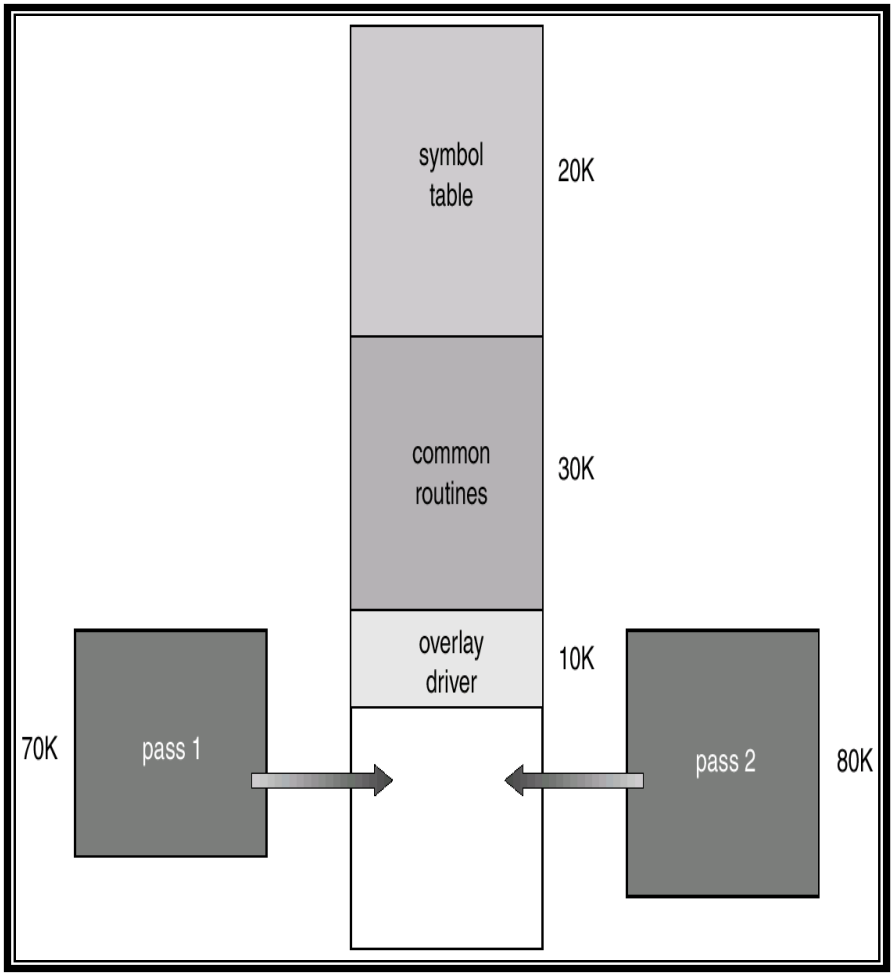
1. The main memory available for a program plus its data may be insufficient. In this case the programmer must use overlays. Overlays waste programmer time. (See next page for overlay detail.)
2. In a multiprogramming environment, the programmer does not know at the time of coding how much space will be available nor where that space will be located.

- Clearly, the task of moving information between the two levels must be a system responsibility. This task is the essence of memory management.

(5) Physical Organization (cont.)

Overlays

- Keep in memory only those instructions and data that are needed at any given time.
- Needed when process is larger than amount of memory allocated to it.
- Examples: 2-Pass Assembler, Multi-Pass Compiler



Memory Partitioning

- The principle 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 known as segmentation and paging.
- Before we examine virtual memory systems we need to look at some simpler techniques (most of which were used in earlier OS) on which virtual memory systems are based. It will make it easier to understand the virtual memory systems, if you first understand the basics of partitioning, simple paging, and simple segmentation.
- The table on the next two pages provides a brief summary of the most common memory management techniques.

Technique	Description	Strengths	Weaknesses
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	Simple to implement. Little OS overhead	Inefficient use of memory due to internal fragmentation. Maximum number of active processes is fixed.
Dynamic Partitioning	Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as the process	No internal fragmentation More efficient use of main memory	Inefficient use of the processor due to the need for compaction to offset external fragmentation.
Simple Paging	Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available, not necessarily contiguous frames.	No external fragmentation	A small amount of internal fragmentation.

Summary of Memory Management Techniques – Part 1

Technique	Description	Strengths	Weaknesses
Simple Segmentation	Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.	No internal fragmentation. Improved memory utilization and reduced overhead compared to dynamic partitioning.	External fragmentation
Virtual-Memory Paging	Same as simple paging, except that it is not necessary to load all of the pages of a process. Non-resident pages that are needed are brought in later automatically.	No external fragmentation. Higher degree of multiprogramming. Large virtual address space.	Overhead of complex memory management
Virtual-Memory Segmentation	Same as simple segmentation, except that it is not necessary to load all of the segments of a process. Non-resident segments that are needed are brought in later automatically.	No internal fragmentation. Higher degree of multiprogramming. Large virtual address space. Protection and sharing support.	Overhead of complex memory management.

Summary of Memory Management Techniques – Part 2

Fixed Partitioning

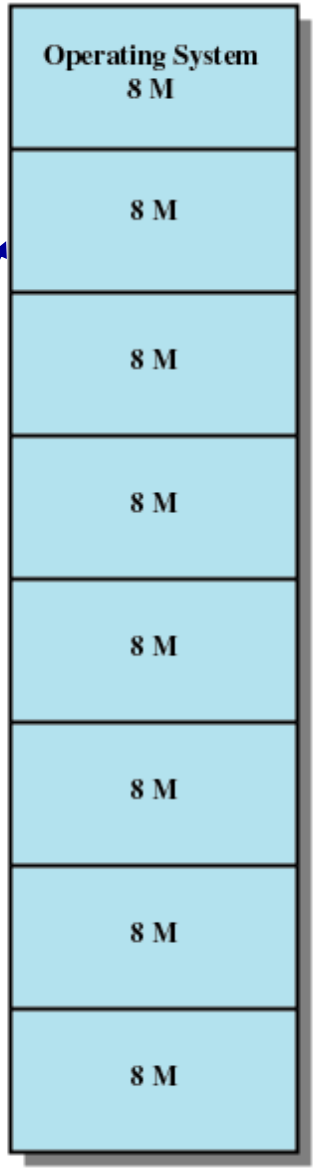
- In most schemes for memory management, we can assume that the OS occupies some fixed portion of main memory and that the rest of main memory is available for use by multiple processes.
- The simplest scheme for managing this available memory is to partition it into regions with fixed boundaries.

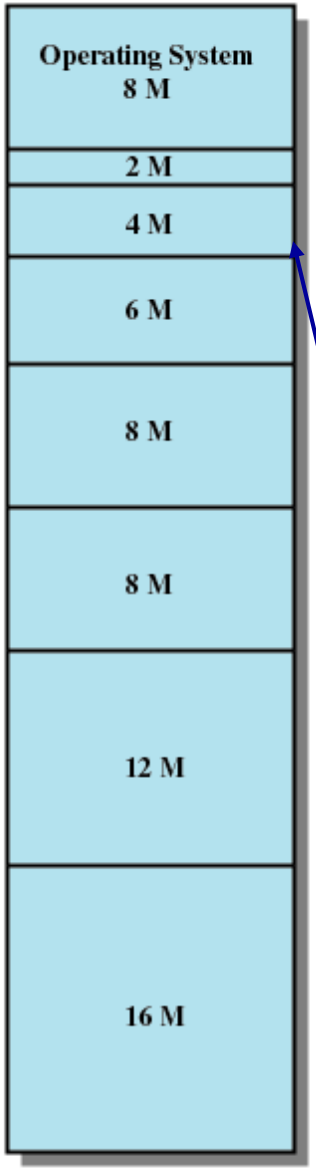
Fixed Partitioning – Partition Sizes

- The figure on the next page illustrates the two alternatives for fixed partitions.
- One possibility is to partition the memory into equal size partitions. This is illustrated by the figure on the left.
- The other possibility, illustrated by the figure on the right, is to partition the memory into varying size partitions.

Fixed Partitioning

Equal-size partitions





Fixed Partitioning

Unequal-size partitions

Fixed Partitioning – Partition Sizes (cont.)

Equal-size partitions

- There are two problems associated with equal-size fixed partitioning:

  1. A program may be too big to fit into a partition. In this case, the programmer must design the program using overlays.
  2. Main memory utilization is extremely inefficient. Any program, no matter how small, occupies an entire partition. In the example shown on the previous page, if we have a program that requires only 1 MB of space, it will occupy an 8 MB partition whenever it is swapped in, rendering 7 MB of wasted space.

- The phenomenon of wasted space internal to a partition, which results whenever the program/data loaded into the partition is smaller than the partition, is known as internal fragmentation.

Fixed Partitioning – Partition Sizes (cont.)

Unequal-size partitions

- The two problems associated with equal-size fixed partitioning can both be lessened, though not solved, by using unequal-sized partitions.:

1. Using the example partitioning shown on page 42, programs as large as 16 MB can be accommodated without using overlays.

2. Again, using the example on page 42, partitions as small as 2 MB exist, so that a 1MB program would result in only 1 MB of wasted space rather than the 7 MB that would be wasted using equal-size partitioning. This results in less internal fragmentation.

Fixed Partitioning – Placement Algorithm

Equal-size partitions

- As long as there is an available partition, a process can be loaded into that partition.
- Because all partitions are of equal size, it does not matter which partition is used.
- If all partitions are occupied with processes that are not ready to run, then one of these processes must be swapped out to make room for a new process.
  - Which one to swap out is a scheduling decision, not a memory management decision.

Fixed Partitioning – Placement Algorithm

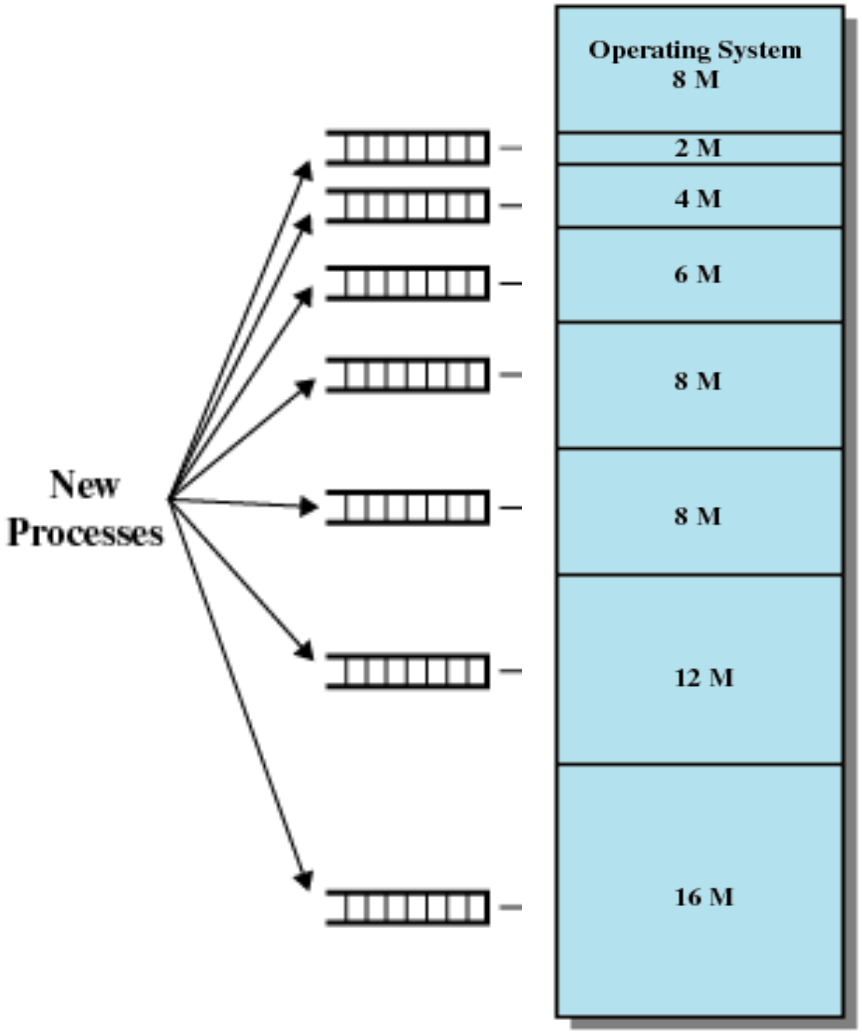
Unequal-size partitions

- With unequal-size partitions, there are two possible ways in which to assign processes to partitions:
- The simplest way is to assign each process to the smallest partition within which it will fit. (See figure (a) on page 48.)
  - In this case a scheduling queue is needed for each partition, to hold swapped out processes destined for that partition.
  - The advantage of this approach is that processes are always assigned in such a way as to minimize internal fragmentation.
- Although this approach seems optimal from the point of view of an individual partition, it is not optimal from the point of view of the system as a whole.
  - For example, consider the case when there are no processes with a size between 12 MB and 16 MB. Then the entire 16 MB partition will remain unused, even though some smaller processes could have been assigned to it.

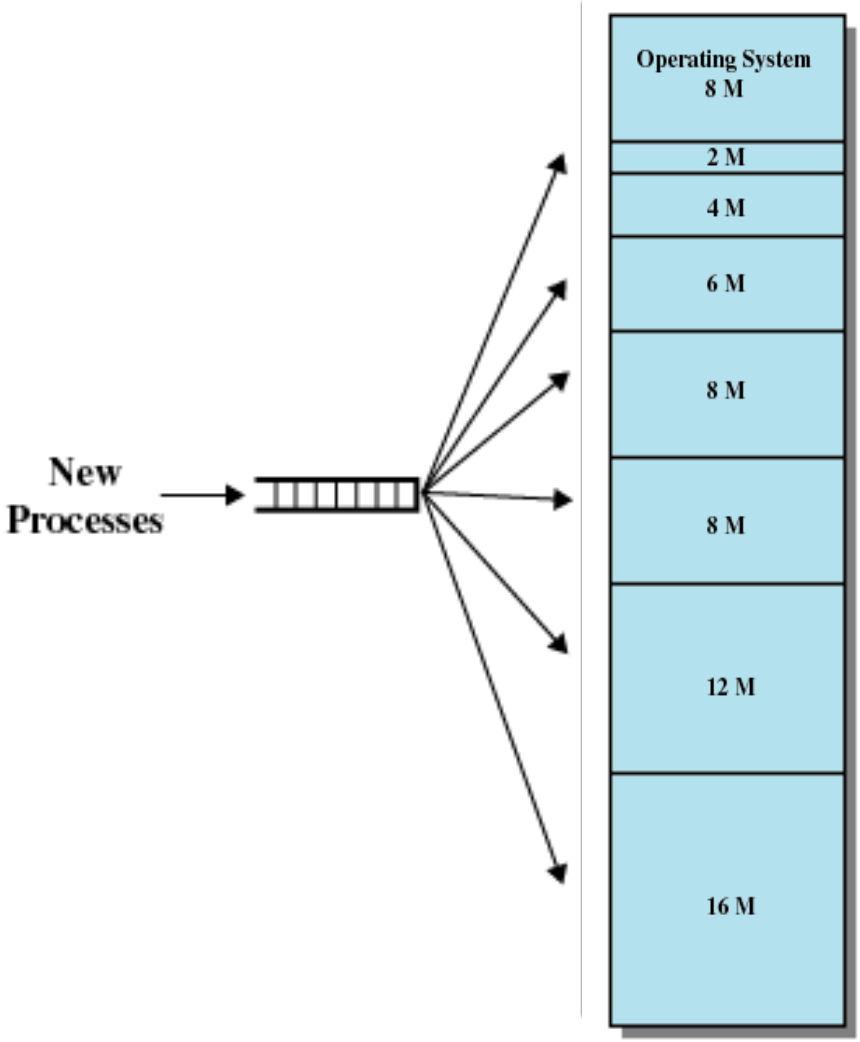
Fixed Partitioning – Placement Algorithm

Unequal-size partitions

- For this reason, a better approach is to use a single queue for all processes. (See figure (b) on page 48.)
  - When it is time to load a process into main memory, the smallest available partition that will hold the process is selected.
  - If all partitions are occupied, then a swapping decision must be made.
  - Preference might be given to swapping out the smallest process from the partition that will hold the incoming process. Although, other factors such as priority and a preference for swapping out blocked processes rather than ready processes, may also be used.



(a) One process queue per partition



(b) Single queue

Fixed Partitioning – Summary

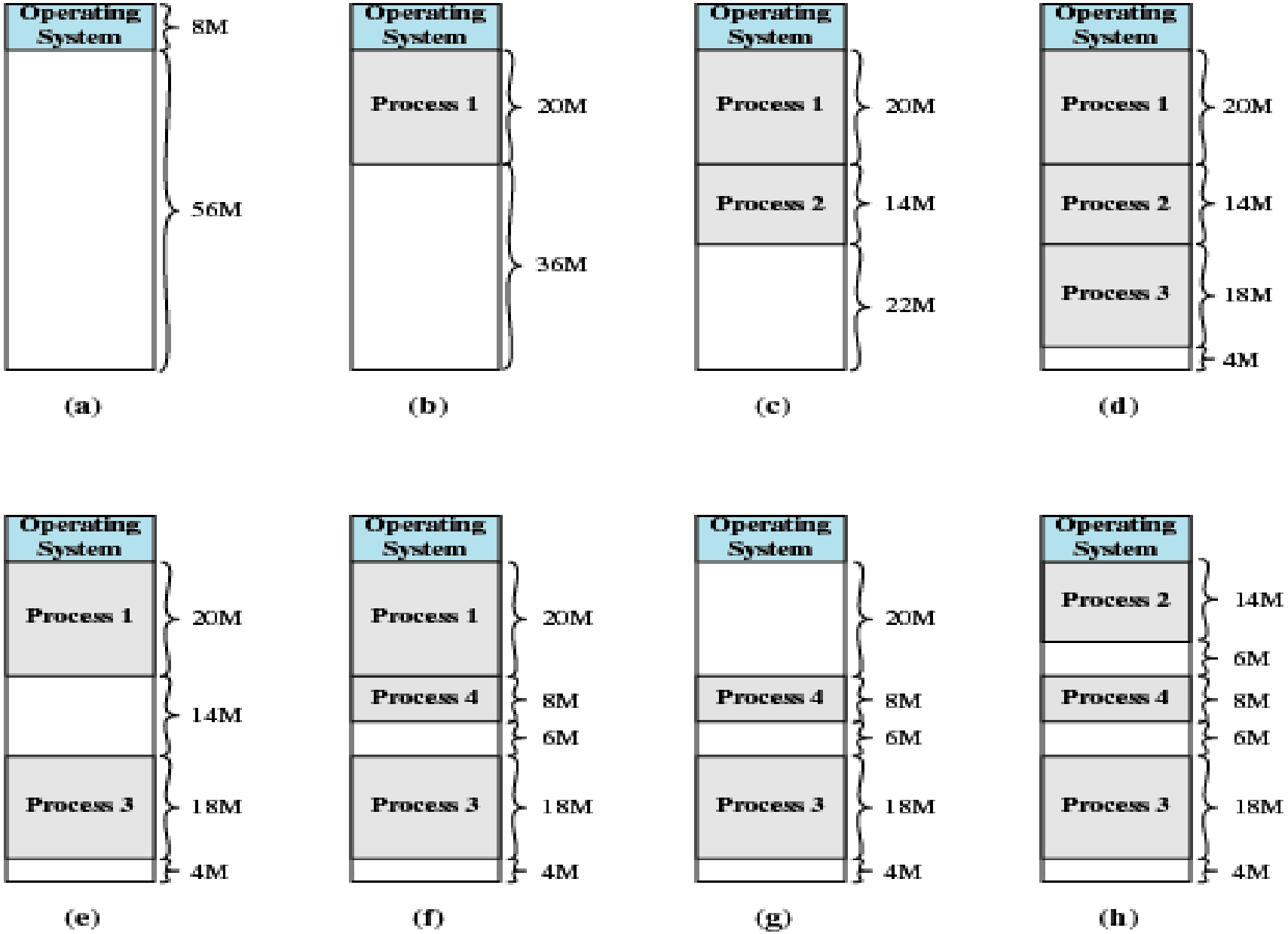
- The use of unequal-size partitions provides a degree of flexibility to the fixed partitioning scheme.
- In addition, fixed partitioning schemes are relatively simple and require minimal OS software and processing overhead.
- However, there are distinct disadvantages to this approach:
  - The number of partitions specified at system generation time limits the number of active (not suspended) processes in the system at any given time.
  - Because partition sizes are preset at system generation time, small jobs will not utilize partition space efficiently causing internal fragmentation.
- The use of fixed partitioning is almost unknown today. One example of a successful OS that utilized this technique was an early IBM mainframe operating system, OS/MFT (Multiprogramming with a Fixed Number of Tasks).

Dynamic Partitioning

- To overcome some of the problems associated with fixed partitioning, an approach known as dynamic partitioning was developed.
- Again, this approach has been replaced by more sophisticated memory management techniques, but it is helpful to understand the basic concepts behind this strategy.
- With dynamic partitioning, the partitions are of variable length and number.
- When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more.

Dynamic Partitioning (cont.)

- The figures on the next page illustrate a dynamic partitioning example, using 64 MB of memory.
- In figure (a), main memory is initially empty, except for the OS.
- In figures (b), (c), and (d), the first three processes are loaded, starting where the OS ends and occupying just enough space for each process. This leaves a "hole" at the end of memory which is too small for a fourth process.
- At some point in time (figure (e)) the OS swaps out process 2, which leaves enough room to load another process, process 4 as shown in figure (f).
- Since process 4 is smaller than process 2, another smaller hole is created.
- Later, a point is reached when none of the processes in main memory are ready, but process 2, in the ready/suspend state, is available. However, since there is insufficient room for process 2, the OS swaps out process 1 (figure (g)) and swaps process 2 back in (figure (h)).


Operating System
8M
56M
(a)
Operating System
Process 1
20M
36M
(b)
Operating System
Process 1
20M
Process 2
14M
22M
(c)
Operating System
Process 1
20M
Process 2
14M
Process 3
18M
4M
(d)
Operating System
Process 1
20M
14M
Process 3
18M
4M
(e)
Operating System
Process 1
20M
Process 4
8M
6M
Process 3
18M
4M
(f)
Operating System
20M
Process 4
8M
6M
Process 3
18M
4M
(g)
Operating System
Process 2
14M
6M
Process 4
8M
6M
Process 3
18M
4M
(h)

Dynamic Partitioning (cont.)

- As the example illustrates, dynamic partitioning starts out well enough, but eventually leads to a situation in which there are a lot of small holes in memory.
- As time goes by, the memory becomes more and more fragmented, and memory utilization declines.
- This phenomenon is known as external fragmentation.
- External fragmentation is the memory that is external to all partitions. This is indirect contrast to the internal fragmentation which was the result of fixed partitioning.

Dynamic Partitioning (cont.)

- One technique for overcoming external fragmentation is compaction.
- With compaction, from time to time, the OS shifts the processes so that they are contiguous and all of the unoccupied (free) memory is placed together in one block.
  - For example, in figure (h) on page 51, compaction would result in a free block of memory 16 MB in size, since the free blocks of size 6 MB, 6 MB, and 4 MB would be coalesced into a single block of free memory.
- The difficulty with compaction is that it is a time consuming process and wasteful of processor time. Note that compaction implies the need for a dynamic relocation capability.

Dynamic Partitioning – Placement Algorithm

- Because memory compaction is time consuming, the OS designer must be clever in deciding how to assign processes to memory (how to plug the holes).
- When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the OS must decide which free block to allocate to the process.
- There are three basic placement algorithms that might be considered:

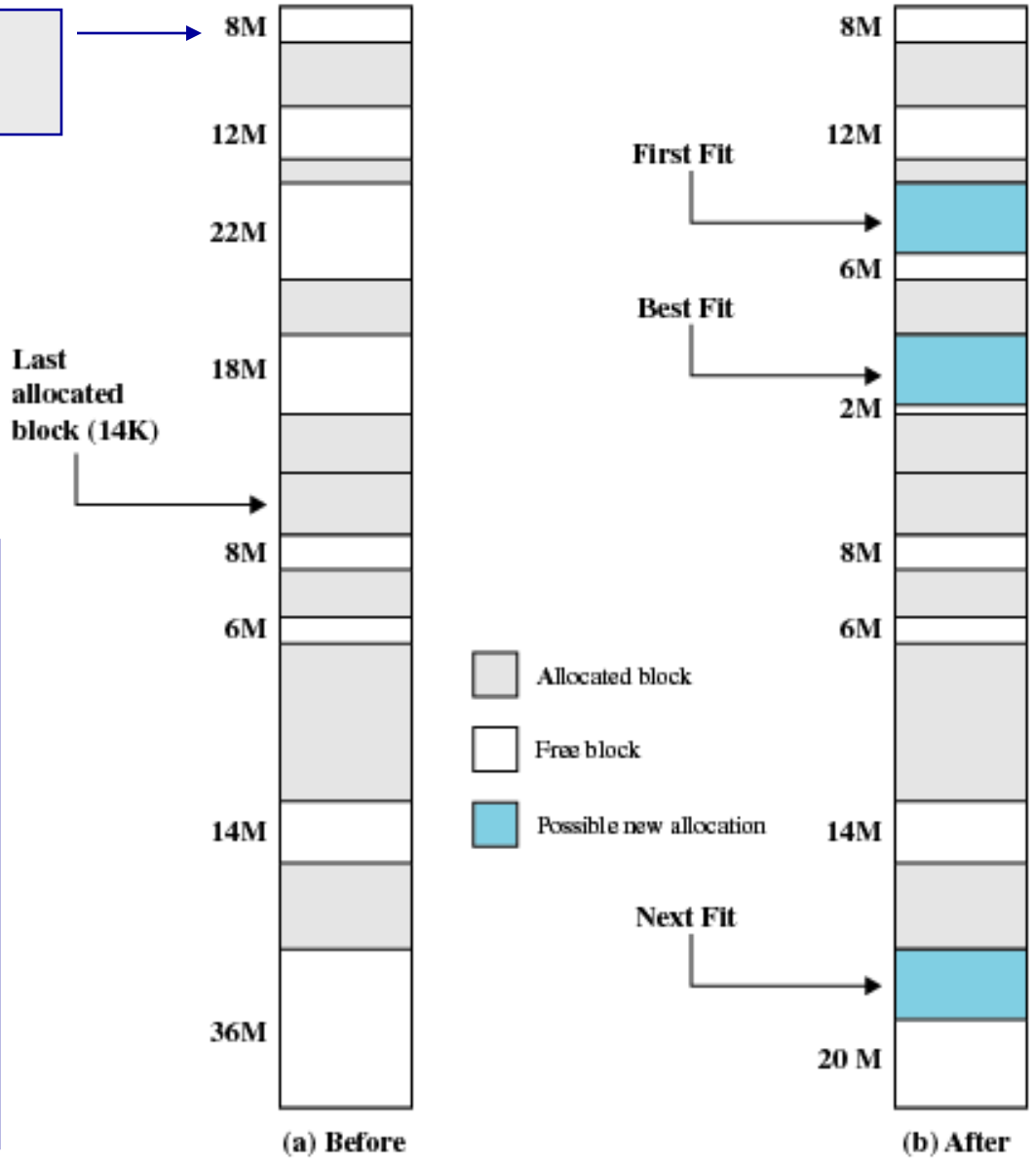
1. Best-fit: chooses the block that is closest in size to the request.
2. First-fit: scans memory from the beginning and chooses the first available block which is large enough to accommodate the request.
3. Next-fit: scans memory from the location of the last placement, and chooses the next available block with sufficient size.

Dynamic Partitioning – Placement Algorithm

- Figure (a) on the next page, shows an example memory configuration after a number of placement and swapping out operations.
- The last block that was used was a 22 MB block from which a 14 MB partition was created.
- Figure (b) illustrates the differences between the best-, first-, and next-fit placement algorithms in satisfying a 16 MB allocation request.
  - Best-fit will search the entire list of available blocks and make use of the 18 MB block, leaving a 2 MB fragment.
  - First-fit results in a 6 MB fragment.
  - Next-fit results in a 20 MB fragment.

Best-fit and first-fit start searching from here.

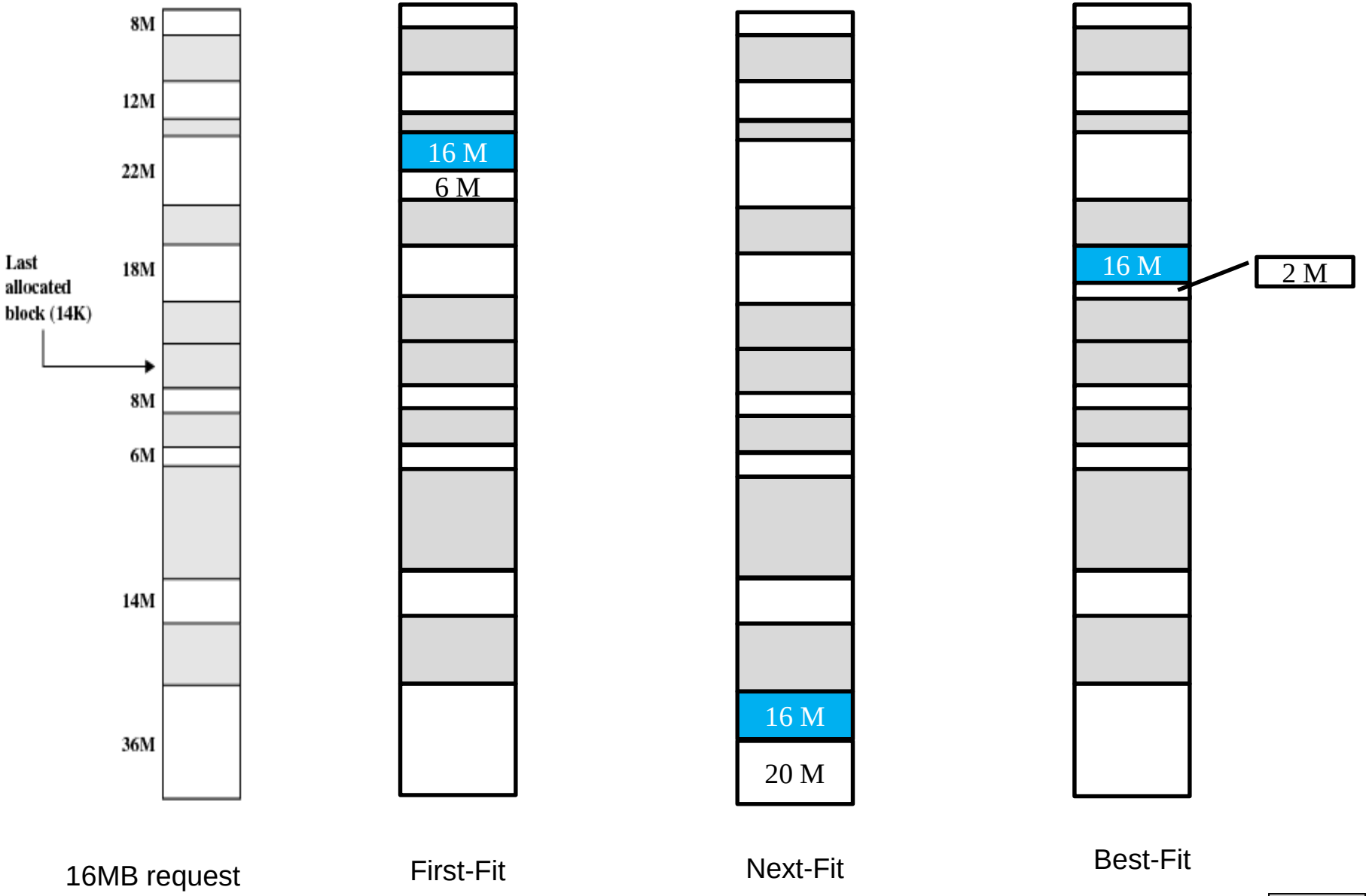
Next-fit starts searching from here



Best-fit considers blocks:
8MB, 12 MB, 22 MB, 18 MB, 8MB, 6 MB, 14 MB, and 20 MB
Best-fit partitions 18 MB block leaving a 2 MB fragment.

First-fit considers blocks:
8 MB, 12 MB, 22 MB
First-fit partitions 22 MB block, Leaving a 6 MB fragment.

Next-fit considers blocks:
8 MB, 6 MB, 14 MB, 36 MB
Next-fit partitions 36 MB block leaving a 20 MB fragment.


8M
12M
22M
18M
Last allocated block (14K)
8M
6M
14M
36M
16 M
6 M
16 M
20 M
16 M
2 M
16MB request
First-Fit
Next-Fit
Best-Fit

Dynamic Partitioning – Placement Algorithm

- Which of the placement algorithms is best depends on the exact sequence of process swapping that occurs and the size of those processes. However, some generalizations can be made:
  - The first-fit algorithm is not only the simplest but is usually the best and fastest as well.
  - The next-fit algorithm tends to produce slightly worse results than the first-fit.
  - The next-fit algorithm will more frequently lead to an allocation from a free block at the end of memory. The result is that the largest block of free memory, which usually appears at the end of the memory space, is quickly broken up into small fragments. Thus, compaction may be required more frequently under the next-fit protocol.
  - On the other hand, the first-fit scheme tends to litter the front-end of the memory space with small free partitions that need to be searched on each subsequent first-fit pass.
  - Despite its name, best-fit usually performs the worst. Since this algorithm looks for the smallest block which satisfies the request, the remaining fragment is as small as possible. The result is many small blocks too small to satisfy any request. Thus, compaction is required even more frequently.

Dynamic Partitioning – Summary

- Dynamic partitioning is rarely utilized in modern OS. One example of a successful OS that utilized this technique was an early IBM mainframe operating system, OS/MVT (Multiprogramming with a Variable Number of Tasks).