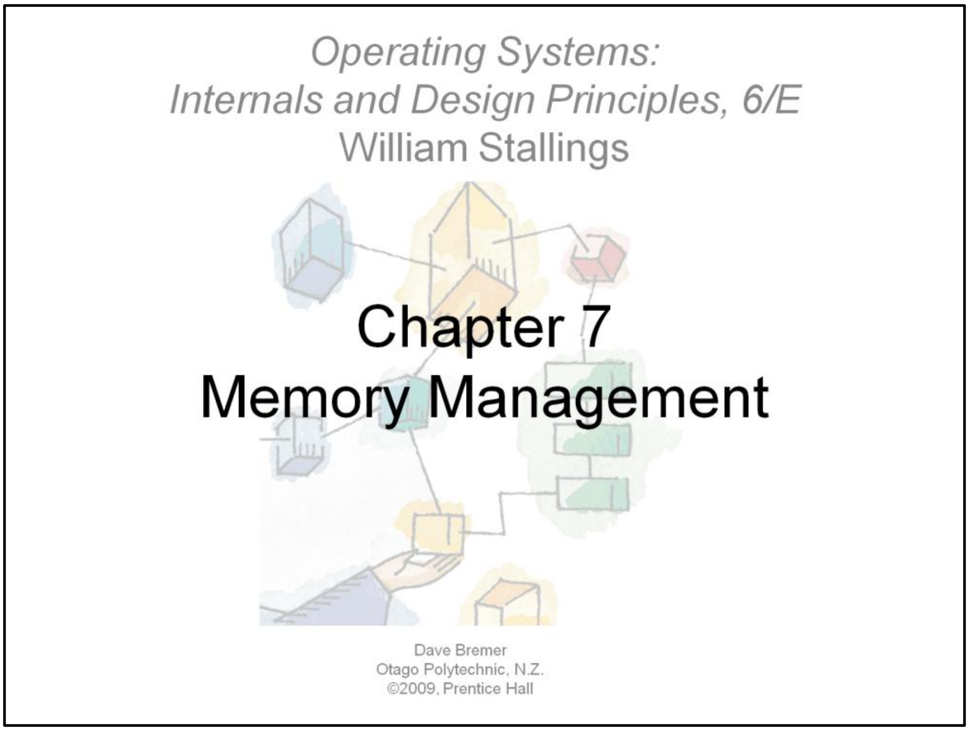
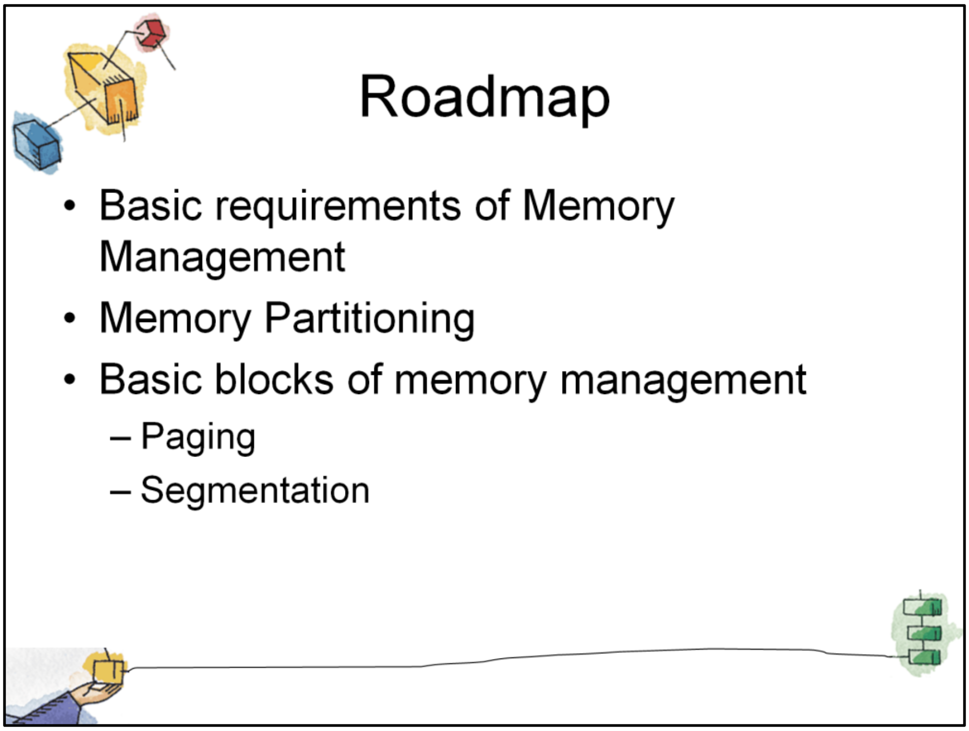


*Operating Systems:*
*Internals and Design Principles, 6/E*
William Stallings

# Chapter 7
# Memory Management

Dave Bremer
Otago Polytechnic, N.Z.
©2009, Prentice Hall

These slides are intended to help a teacher develop a presentation. This powerpoint covers the entire chapter and includes too many slides for a single delivery. Professors are encouraged to adapt this presentation in ways which are best suited for their students and environment.

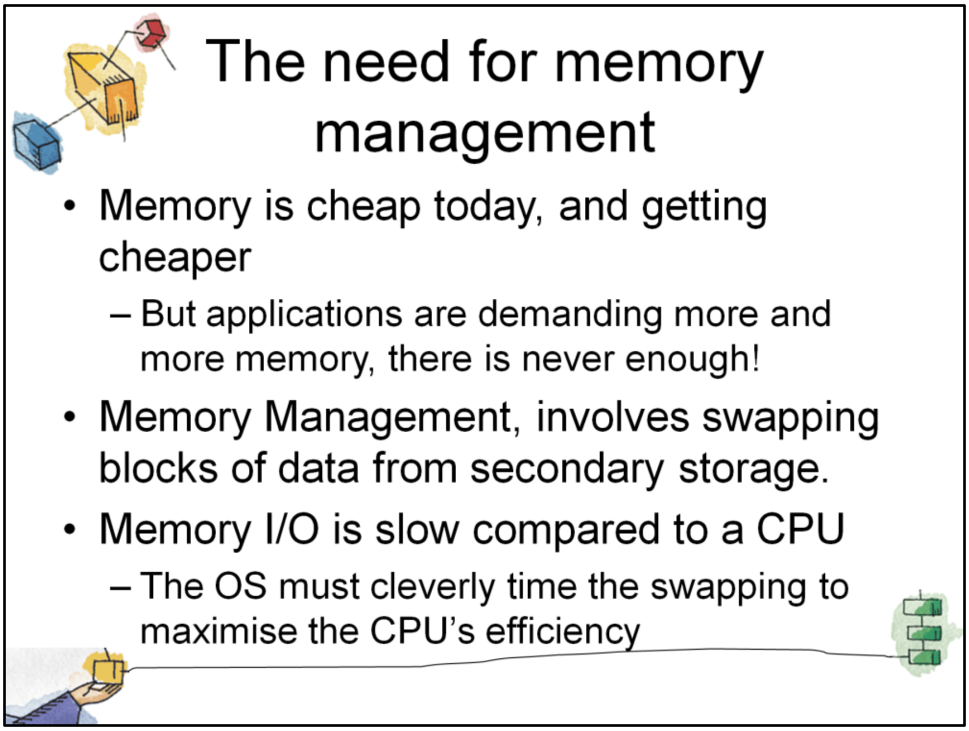


# Roadmap

- Basic requirements of Memory Management
- Memory Partitioning
- Basic blocks of memory management
  - Paging
  - Segmentation

Overview of points covered in this chapter

Point out that memory partitioning isn’t used much except for special cases such as kernel memory management



# The need for memory management

- Memory is cheap today, and getting cheaper
  - But applications are demanding more and more memory, there is never enough!
- Memory Management, involves swapping blocks of data from secondary storage.
- Memory I/O is slow compared to a CPU
  - The OS must cleverly time the swapping to maximise the CPU's efficiency

Introduce by pointing out that in a uniprogramming system, main memory is divided into two parts:

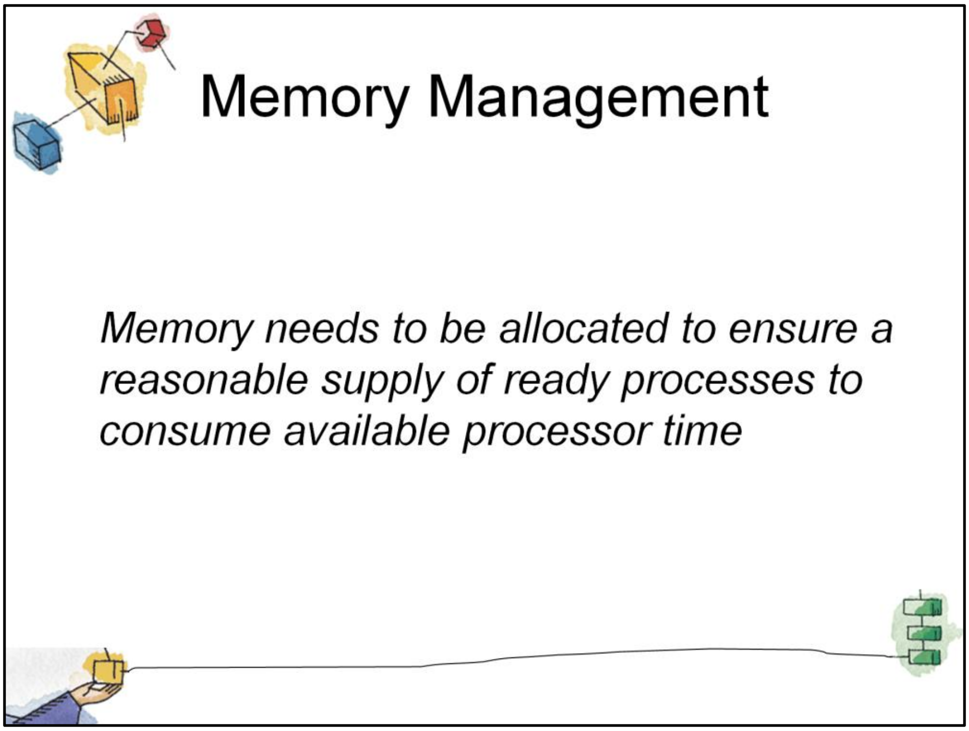
- one part for the operating system (resident monitor, kernel) and
- one part for the program currently being executed.

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

Emphasise that 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

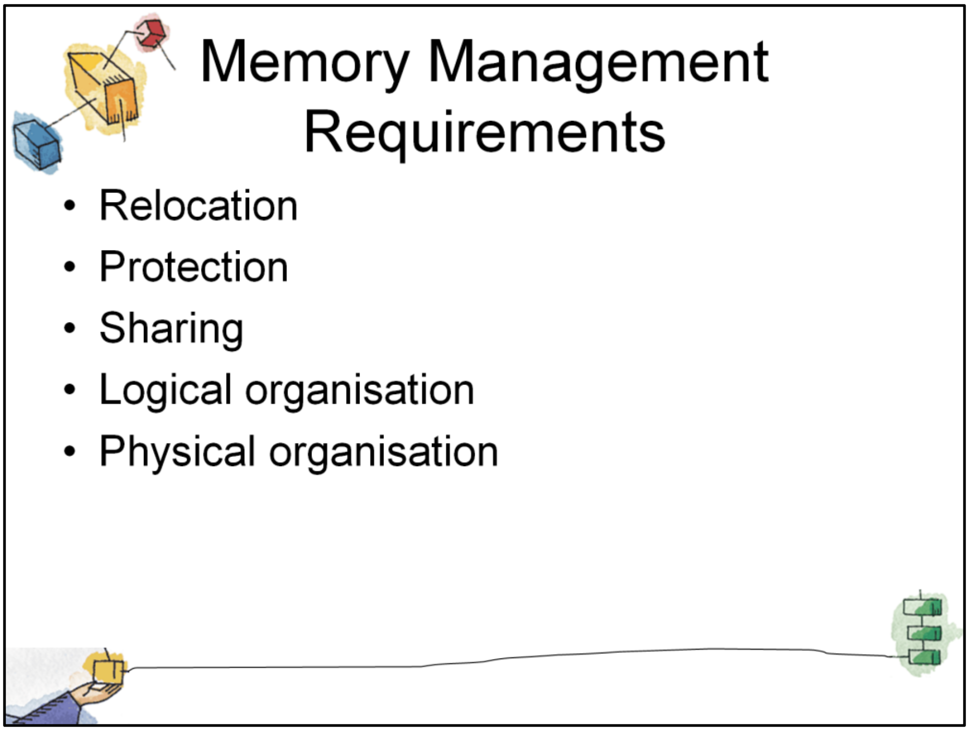
for I/O and the processor will be idle.

Thus memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time.



# Memory Management

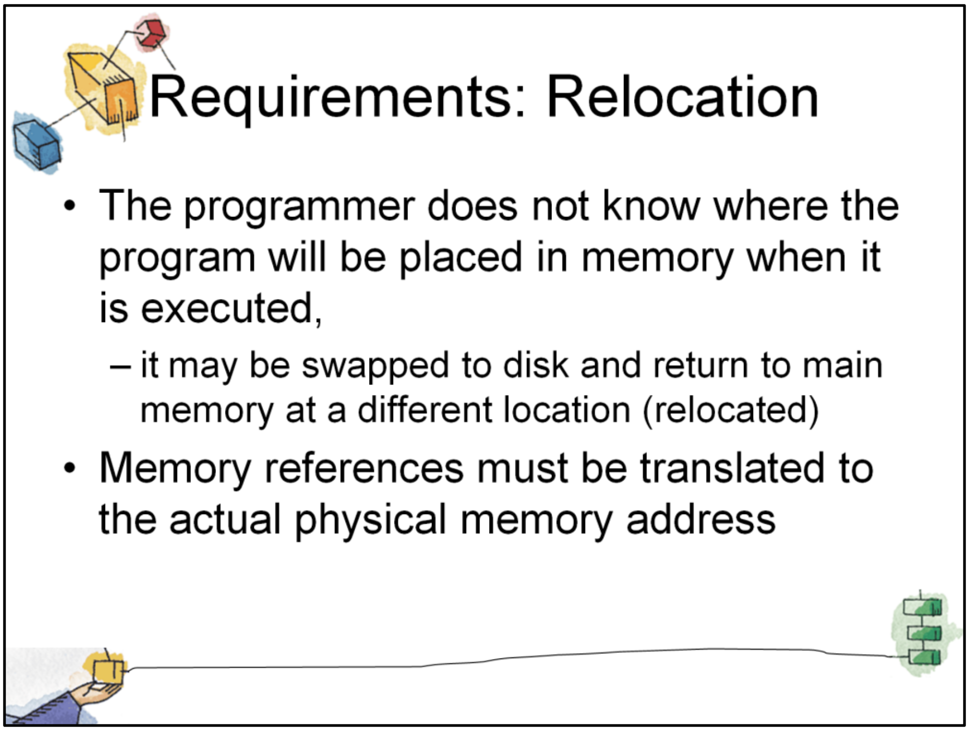
*Memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time*



# Memory Management Requirements

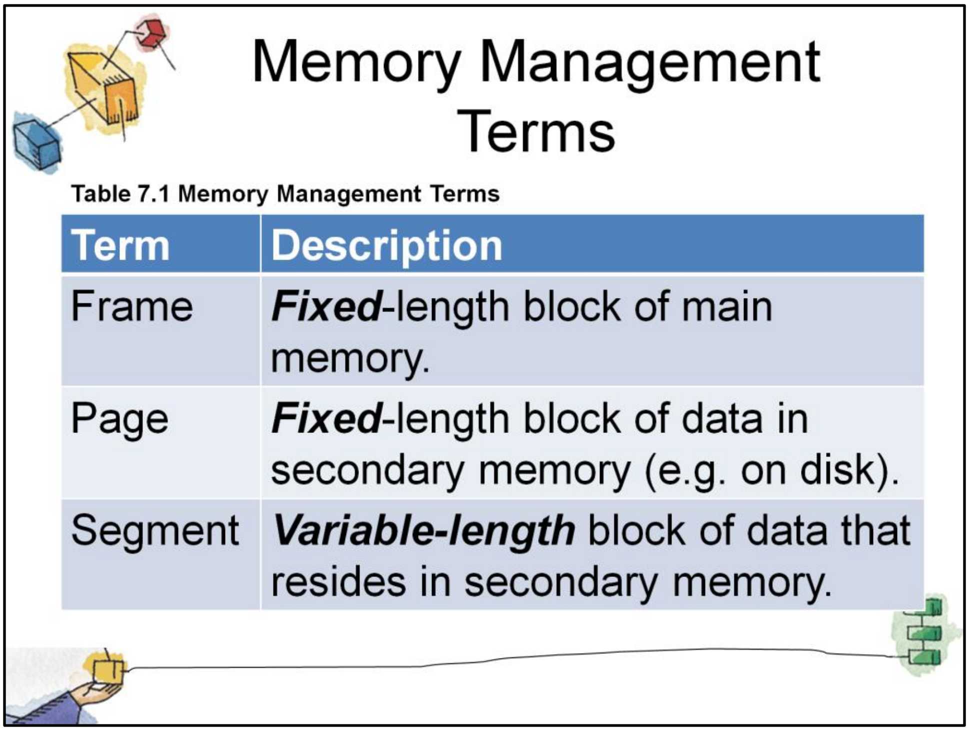
- Relocation
- Protection
- Sharing
- Logical organisation
- Physical organisation

The following slides expand on these topics.



# Requirements: Relocation

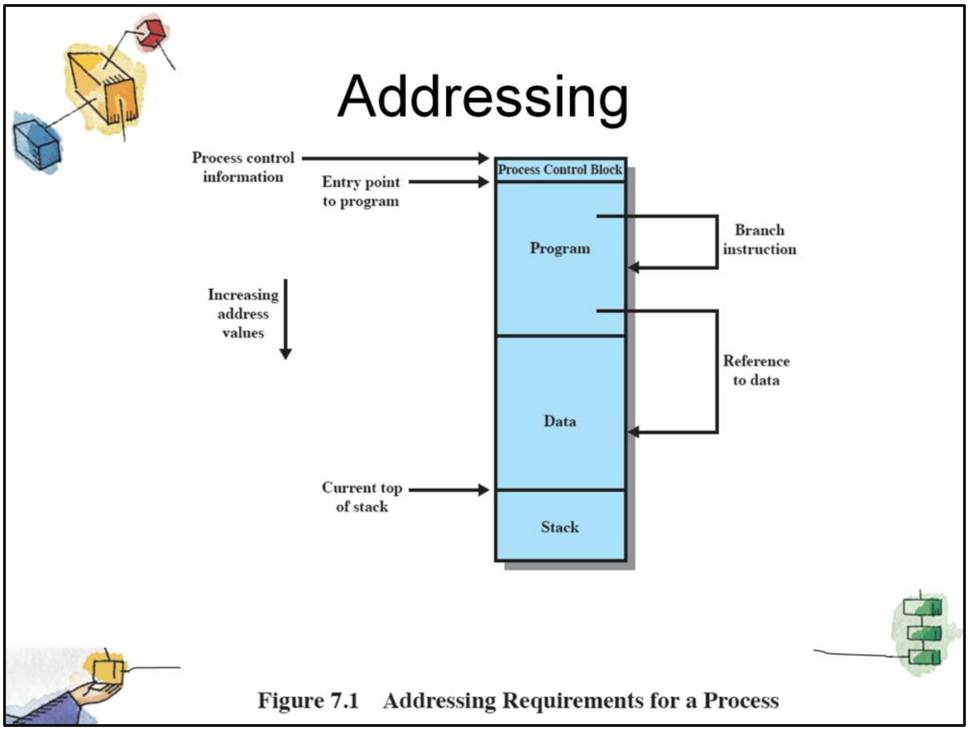
- The programmer does not know where the program will be placed in memory when it is executed,
  - it may be swapped to disk and return to main memory at a different location (relocated)
- Memory references must be translated to the actual physical memory address



# Memory Management Terms

Table 7.1 Memory Management Terms

| Term | Description |
|---|---|
| Frame | ***Fixed***-length block of main memory. |
| Page | ***Fixed***-length block of data in secondary memory (e.g. on disk). |
| Segment | ***Variable-length*** block of data that resides in secondary memory. |



# Addressing

Figure 7.1 Addressing Requirements for a Process

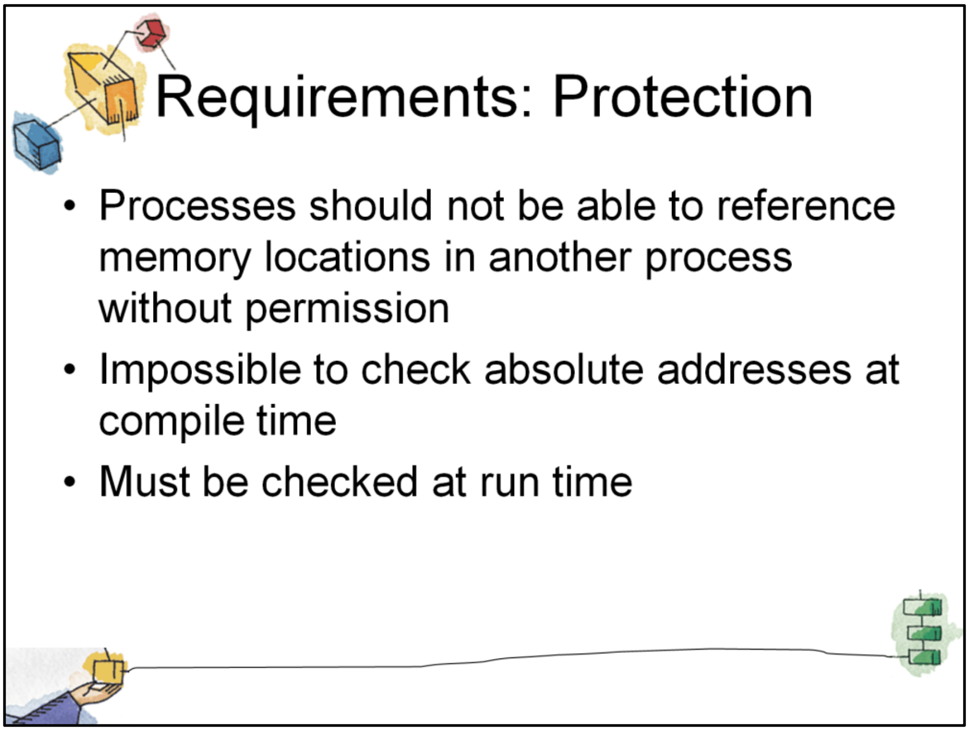
This figure depicts a process image. Talk the students through this diagram

Assume that the process image occupies a contiguous region of main memory.

The OS needs to know the location of:

- process control information
- the execution stack,
- the entry point to begin execution of the program for this process.

Because the operating system knows this information because it is managing memory and is responsible for bringing this process into main memory. 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.



# Requirements: Protection

- Processes should not be able to reference memory locations in another process without permission
- Impossible to check absolute addresses at compile time
- Must be checked at run time

Normally, a user process cannot access any portion of the operating system, neither program nor data.

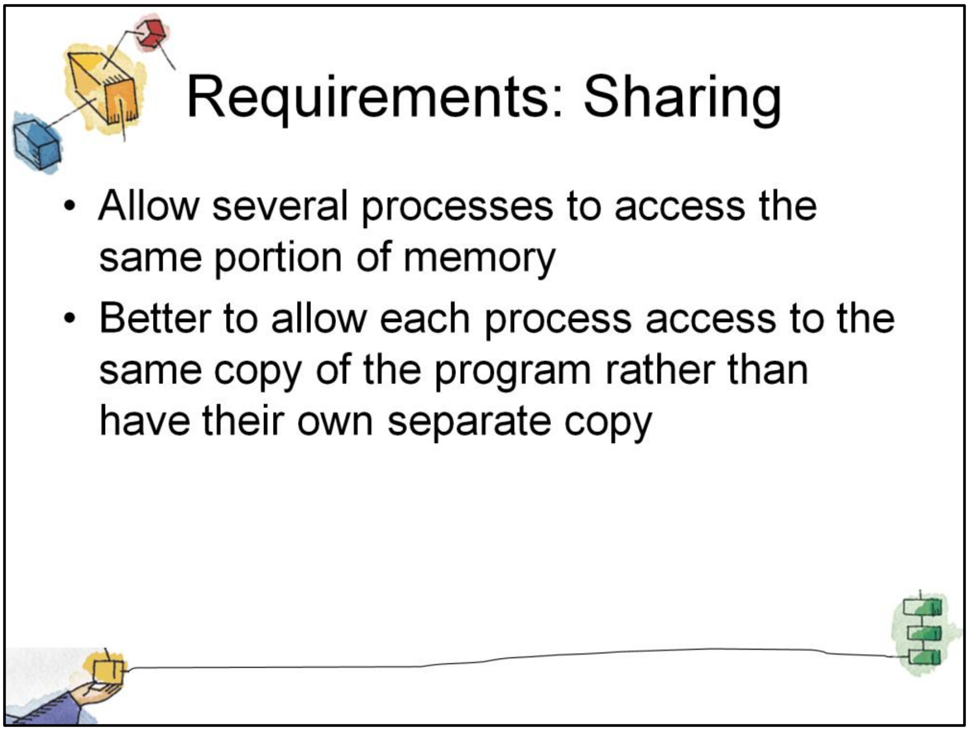
Usually a program in one process cannot branch to an instruction in another process or access the data area of another process. The processor must be able to abort such instructions at the point of execution.

Note that the memory protection requirement must be satisfied by the processor (hardware) rather than the operating system (software) because the OS cannot anticipate all of the memory references that a program will make. It is only possible to assess the permissibility of a memory reference at the time of execution.

Consider asking the students “why” to point 1 & 2.

Why is it a Bad Thing for one process to be able to read, or even write, to memory occupied by a different process?

Why is it impossible to check absolute addresses at compile time (hint: see relocation)

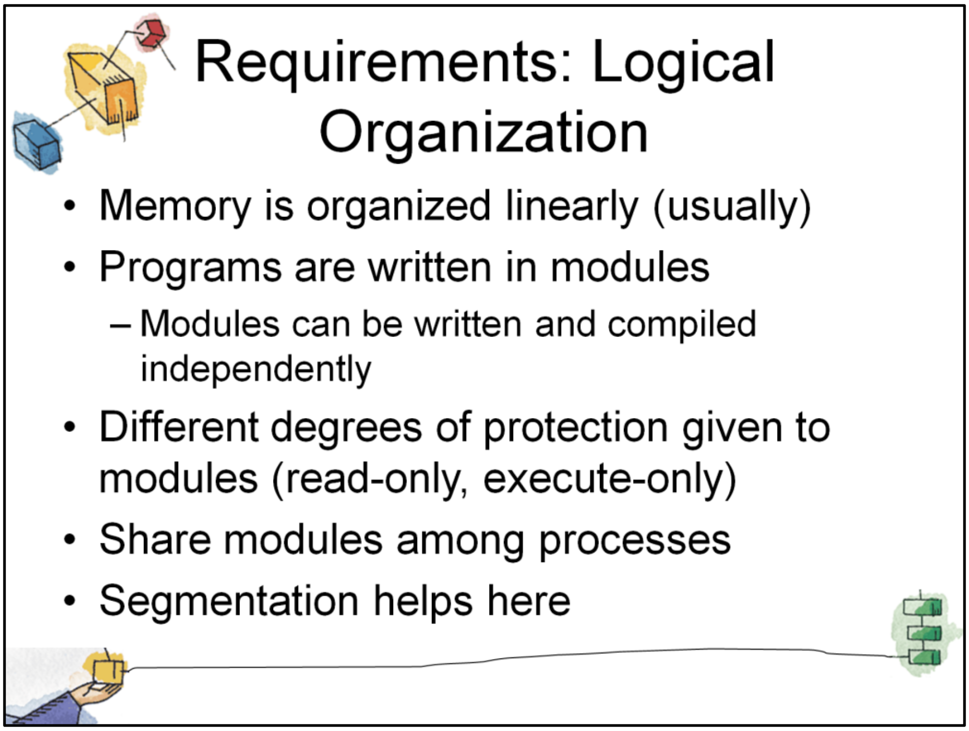


# Requirements: Sharing

- Allow several processes to access the same portion of memory
- Better to allow each process access to the same copy of the program rather than have their own separate copy

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

Processes that are cooperating on some task may need to share access to the same data structure.



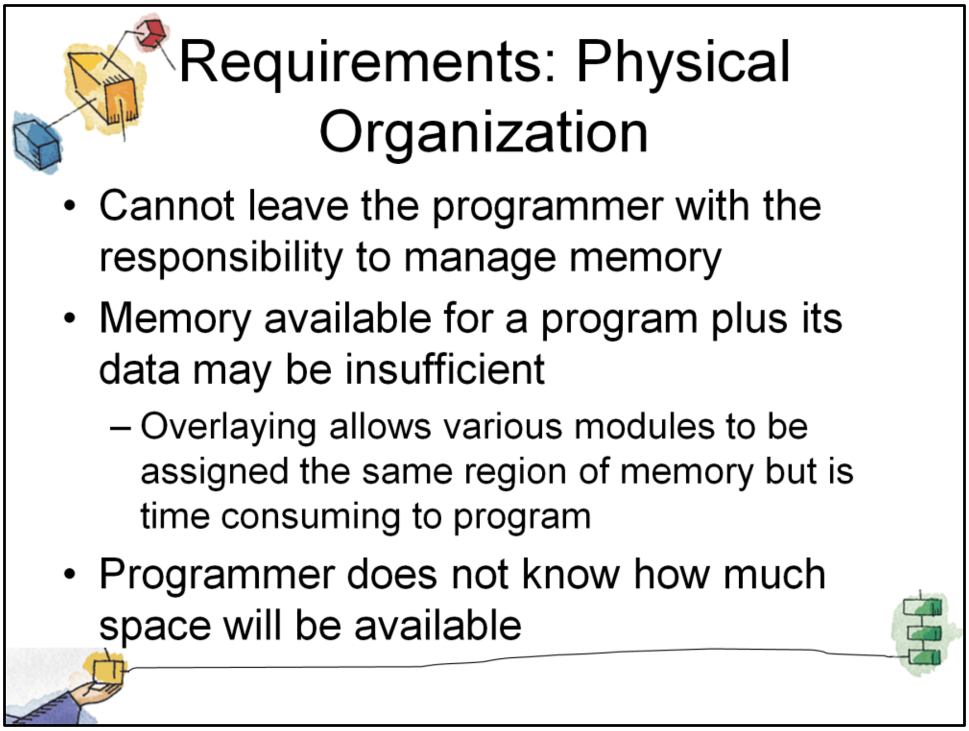
## Requirements: Logical Organization

- Memory is organized linearly (usually)
- Programs are written in modules
  - Modules can be written and compiled independently
- Different degrees of protection given to modules (read-only, execute-only)
- Share modules among processes
- Segmentation helps here

Main memory is usually 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.

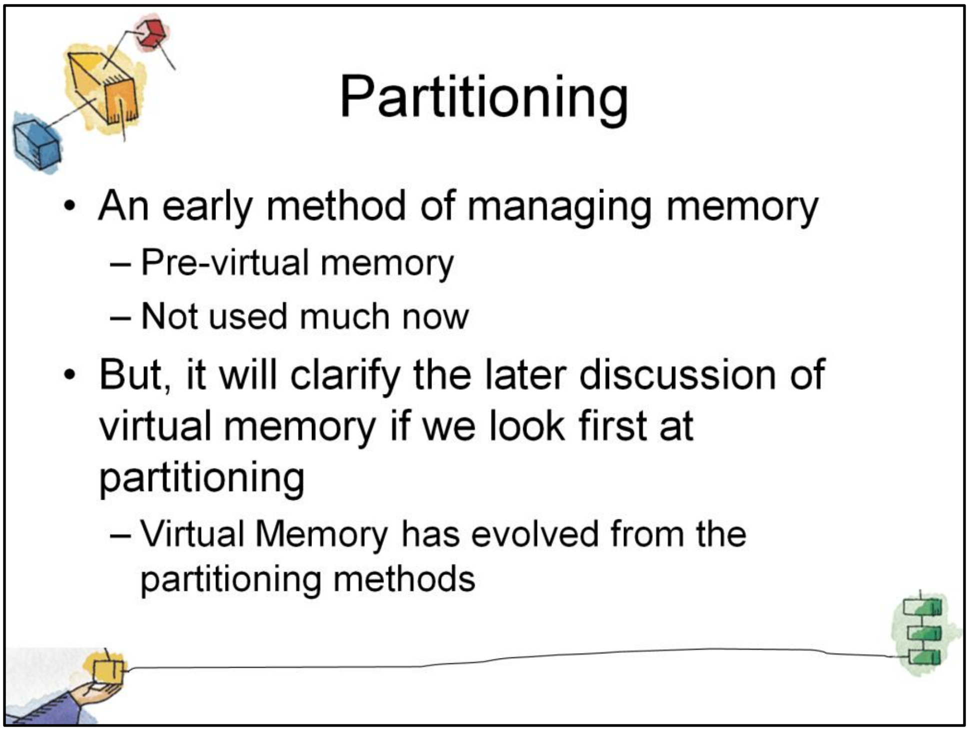
This does not correspond to the way in which programs are typically constructed. Most programs are organized into modules. If the operating system and computer hardware can effectively deal with user programs and data in the form of modules of some sort, then a number of advantages can be realized



# Requirements: Physical Organization

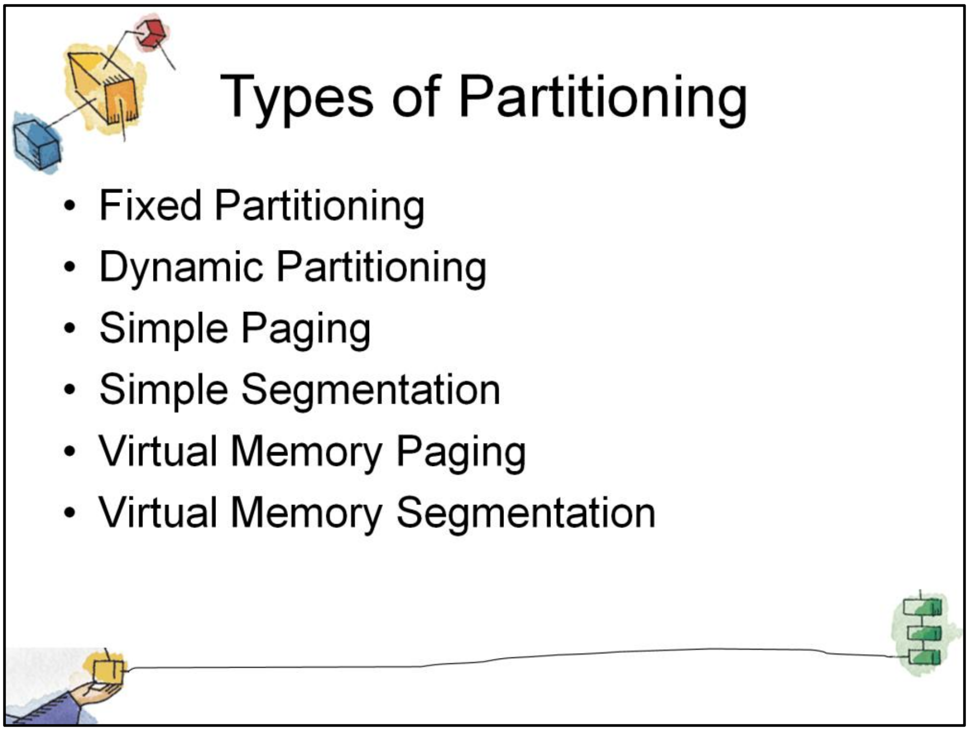
- Cannot leave the programmer with the responsibility to manage memory
- Memory available for a program plus its data may be insufficient
  - Overlaying allows various modules to be assigned the same region of memory but is time consuming to program
- Programmer does not know how much space will be available

Because of this, it is clear that the task of moving information between the two levels of memory should be a system responsibility. This task is the essence of memory management.



# Partitioning

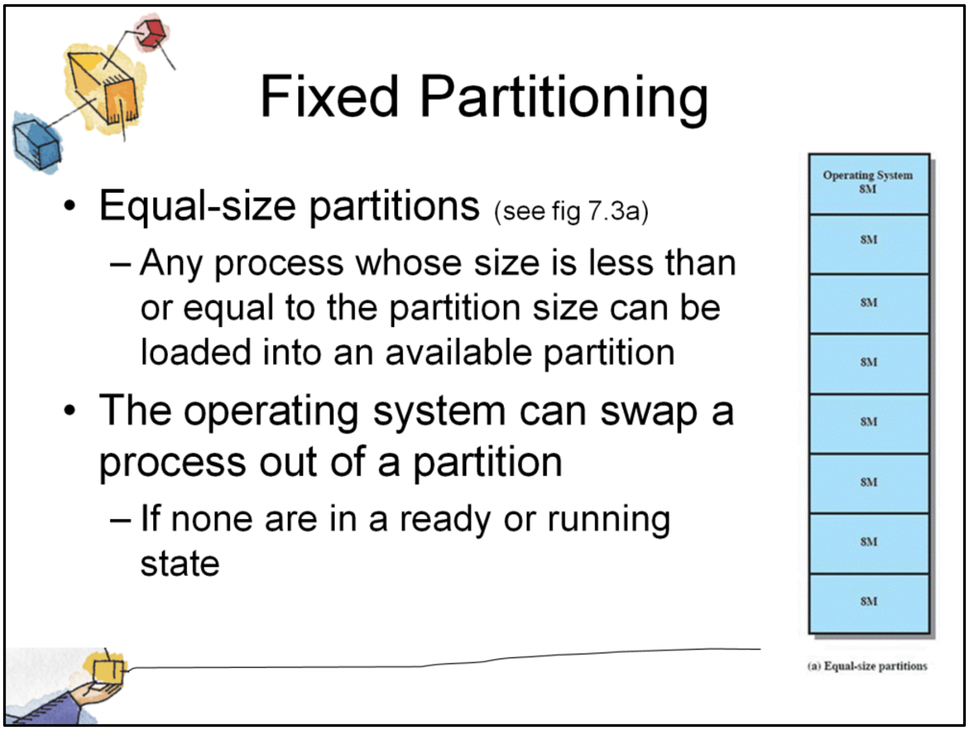
- An early method of managing memory
  - Pre-virtual memory
  - Not used much now
- But, it will clarify the later discussion of virtual memory if we look first at partitioning
  - Virtual Memory has evolved from the partitioning methods



# Types of Partitioning

- Fixed Partitioning
- Dynamic Partitioning
- Simple Paging
- Simple Segmentation
- Virtual Memory Paging
- Virtual Memory Segmentation

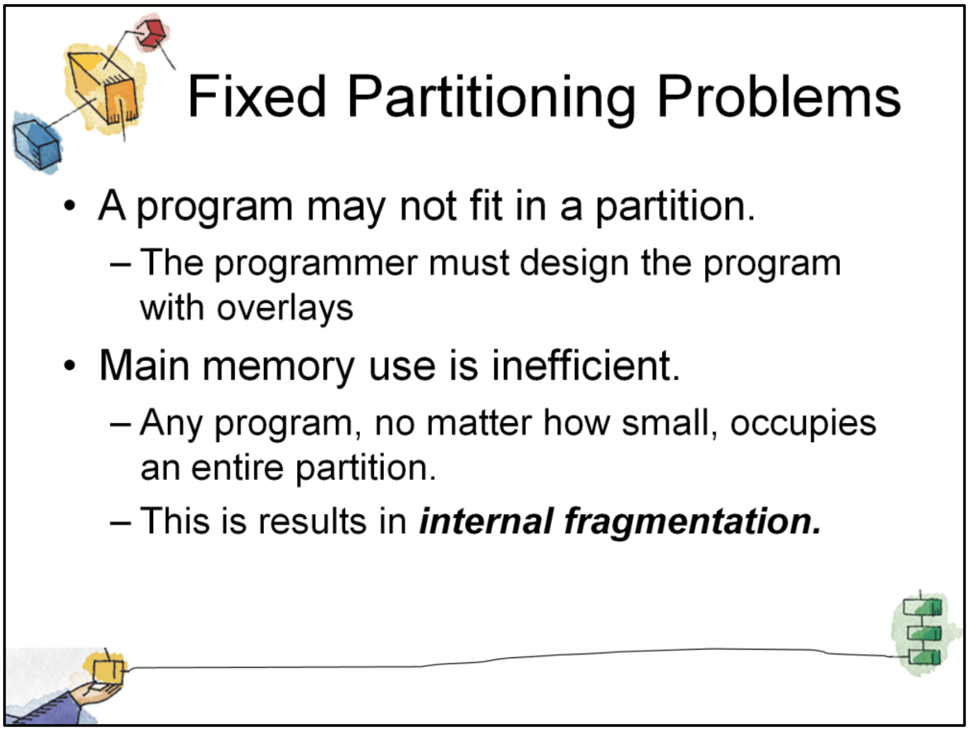
Don’t dwell on this slide – it is just an indication on the various approaches which will be covered in further detail in other slides



# Fixed Partitioning

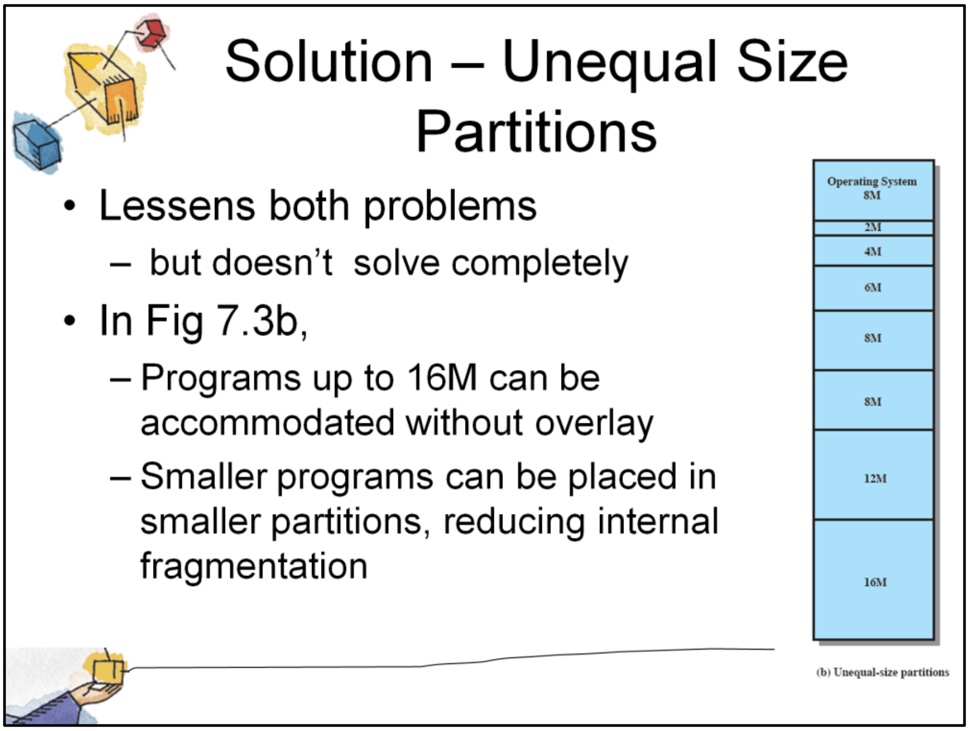
- Equal-size partitions (see fig 7.3a)
  - Any process whose size is less than or equal to the partition size can be loaded into an available partition
- The operating system can swap a process out of a partition
  - If none are in a ready or running state

(a) Equal-size partitions



# Fixed Partitioning Problems

- A program may not fit in a partition.
  - The programmer must design the program with overlays
- Main memory use is inefficient.
  - Any program, no matter how small, occupies an entire partition.
  - This is results in ***internal fragmentation.***

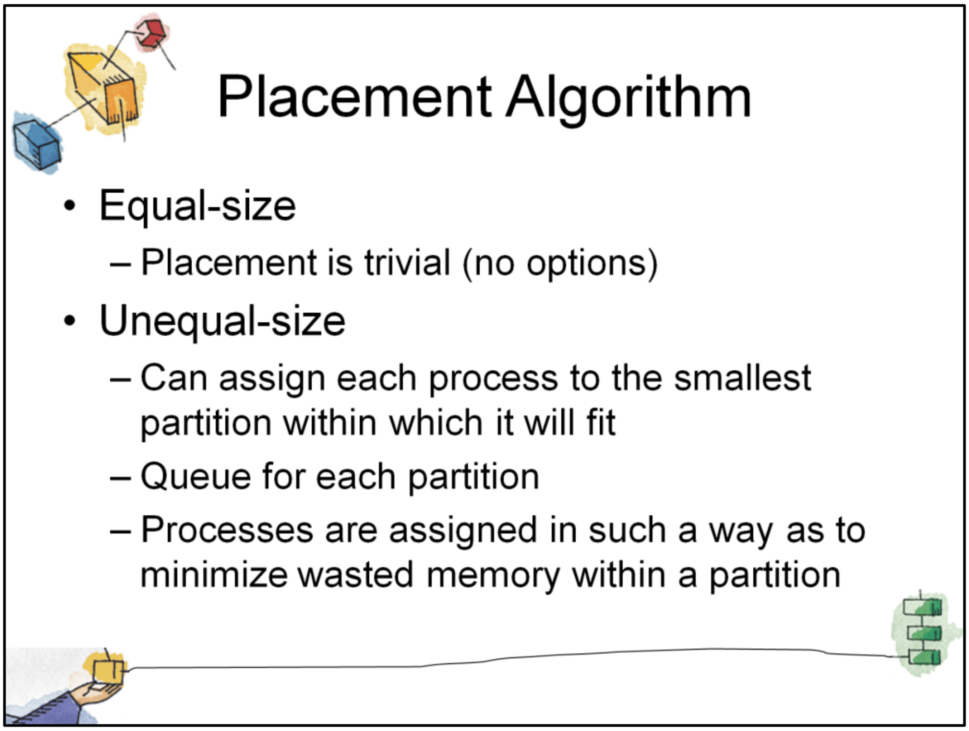


# Solution – Unequal Size Partitions

- Lessens both problems
  - but doesn't  solve completely
- In Fig 7.3b,
  - Programs up to 16M can be accommodated without overlay
  - Smaller programs can be placed in smaller partitions, reducing internal fragmentation

Operating System
8M
2M
4M
6M
8M
8M
12M
16M

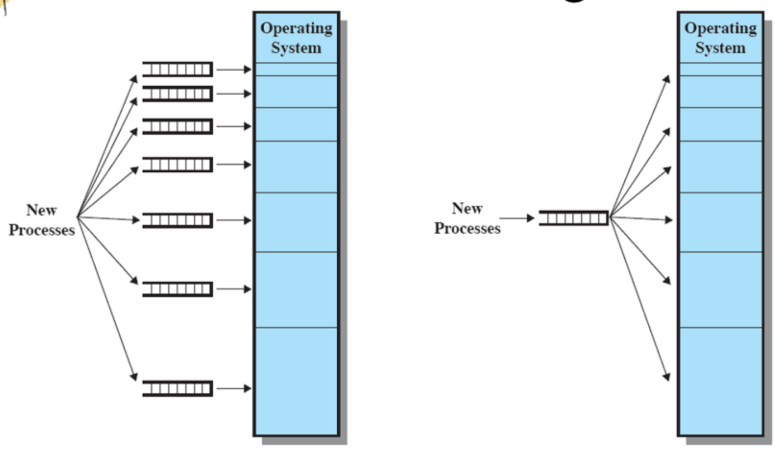
(b) Unequal-size partitions



# Placement Algorithm

- Equal-size
  - Placement is trivial (no options)
- Unequal-size
  - Can assign each process to the smallest partition within which it will fit
  - Queue for each partition
  - Processes are assigned in such a way as to minimize wasted memory within a partition

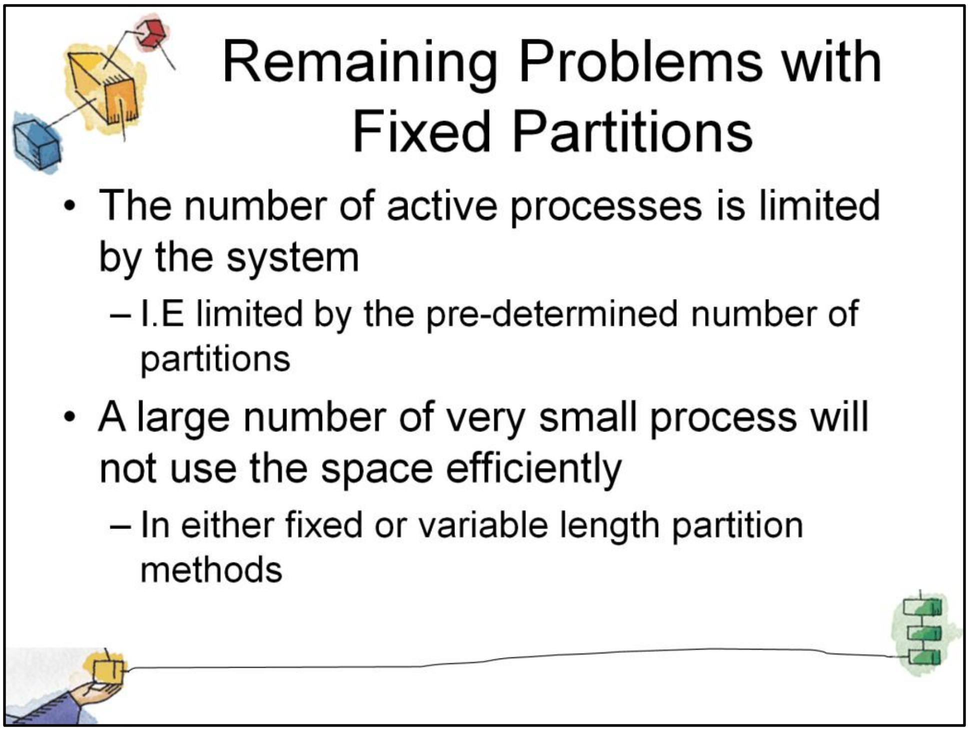
# Fixed Partitioning



(a) One process queue per partition

(b) Single queue

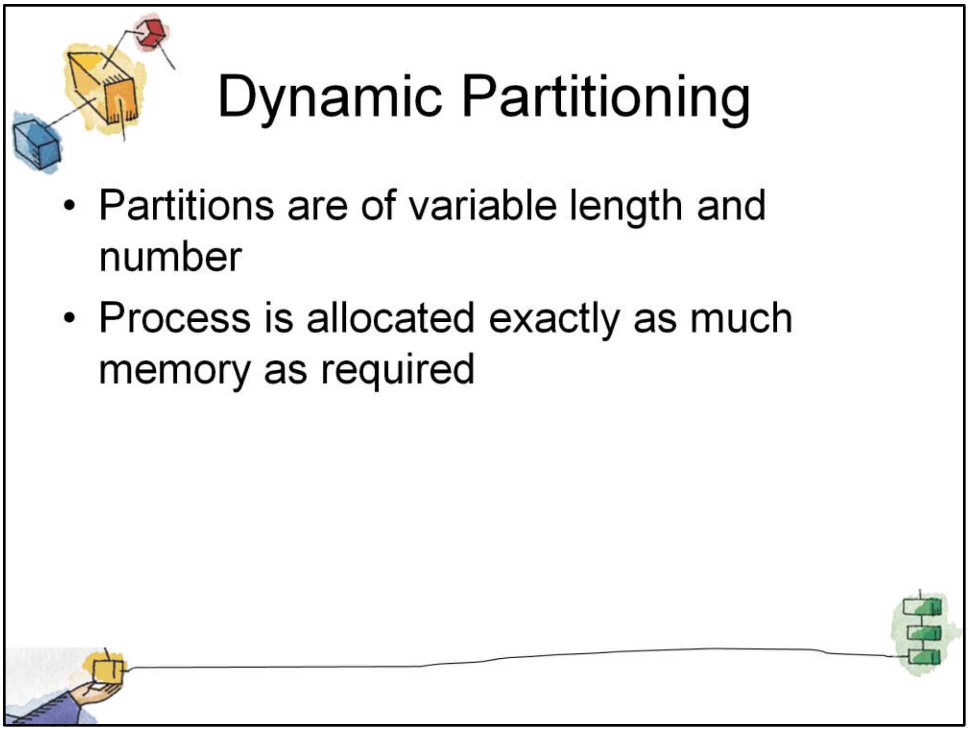
**Figure 7.3 Memory Assignment for Fixed Partitioning**



# Remaining Problems with Fixed Partitions

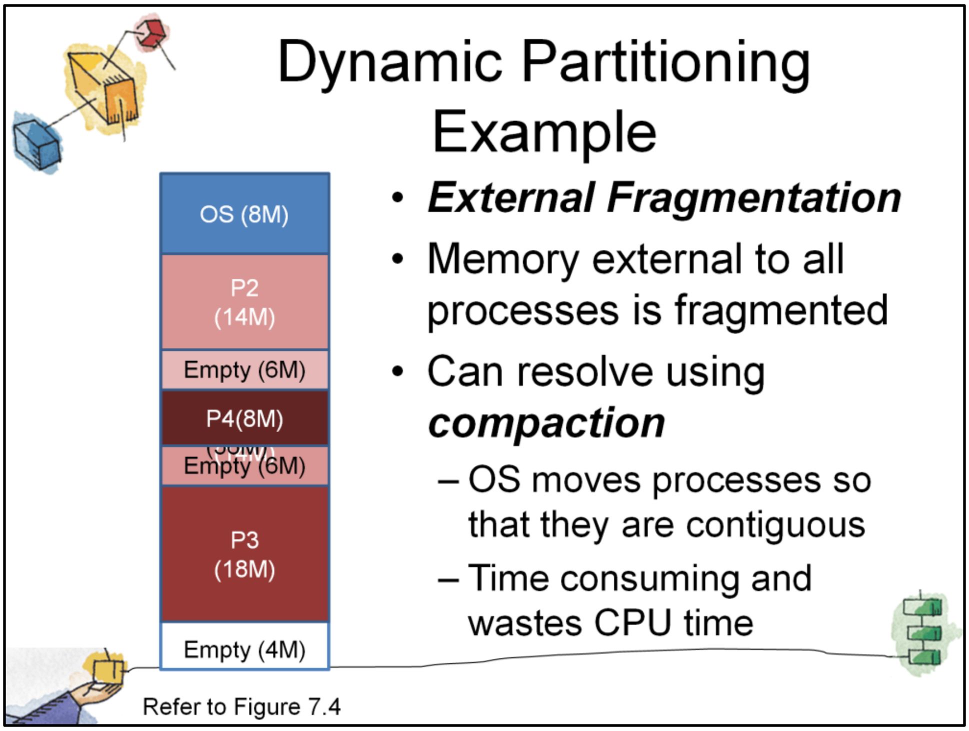
- The number of active processes is limited by the system
  - I.E limited by the pre-determined number of partitions
- A large number of very small process will not use the space efficiently
  - In either fixed or variable length partition methods

Finish by mentioning tat fixed partitioning is almost unknown today



# Dynamic Partitioning

- Partitions are of variable length and number
- Process is allocated exactly as much memory as required



# Dynamic Partitioning Example

OS (8M)

P2
(14M)

Empty (6M)

P4(8M)

Empty (6M)

P3
(18M)

Empty (4M)

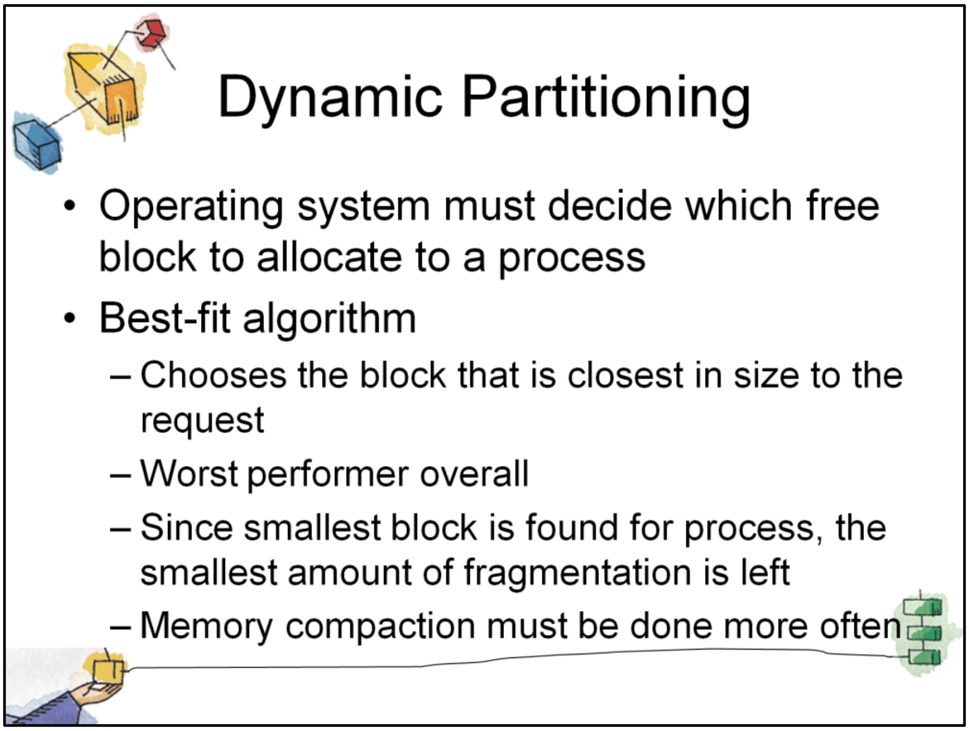
- ***External Fragmentation***
- Memory external to all processes is fragmented
- Can resolve using ***compaction***
  - OS moves processes so that they are contiguous
  - Time consuming and wastes CPU time

Refer to Figure 7.4

Animated slide

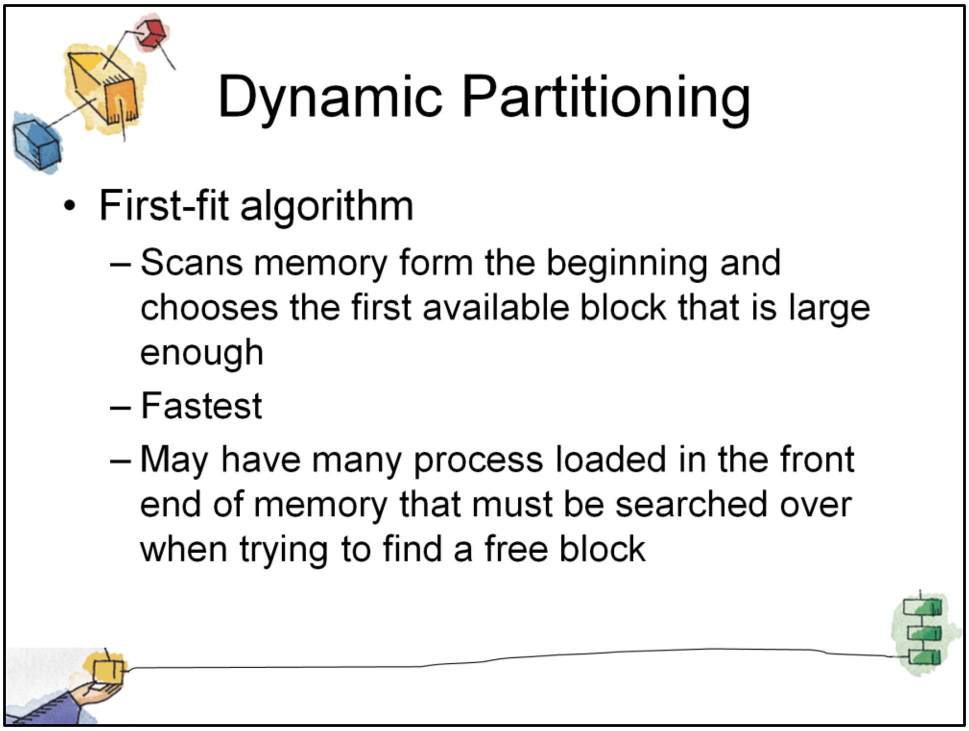
Imagine a system with 64M RAM

1. Initially, main memory is empty, except for the operating system
2. Three processes are loaded in – leaving a ‘hole’ too small for any further process
3. At some point, none of the processes in memory is ready. The operating system swaps out process 2,
4. Which leaves sufficient room to load a new process, process 4 – but that creates another hole
5. Later, a point is reached at which none of the processes in main memory is ready, but process 2, in the Ready-Suspend state, is available. Because there is insufficient room in memory for process 2, the operating system swaps process 1 out and swaps process 2 back in leaving yet another hole
6. Explain External Fragmentation and compaction – mention that compaction implies the capability of dynamic relocation



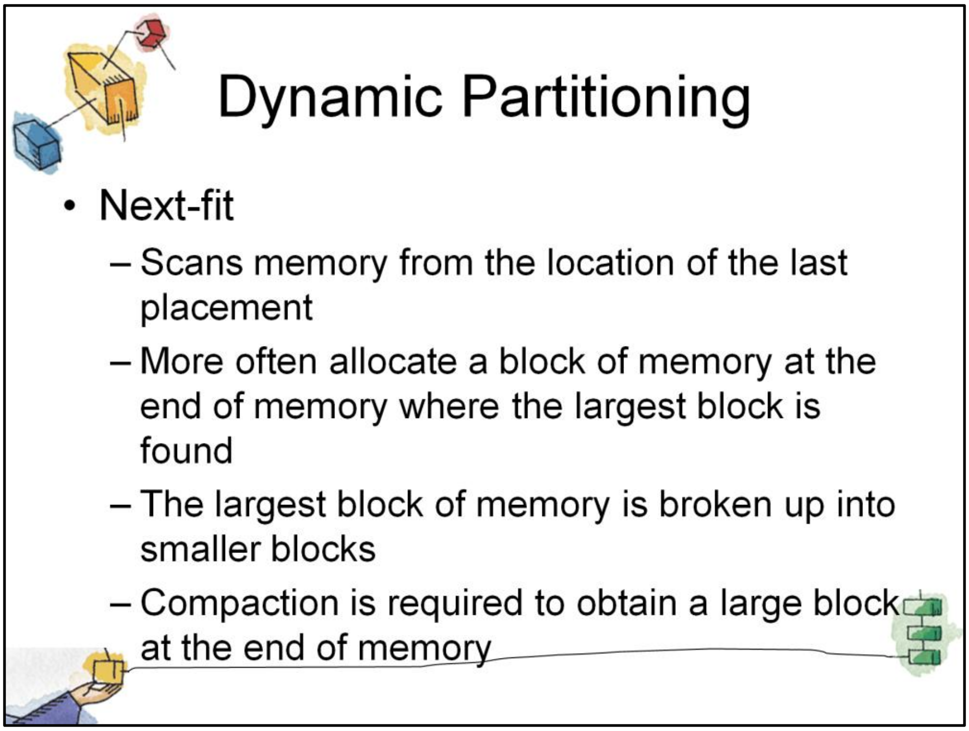
# Dynamic Partitioning

- Operating system must decide which free block to allocate to a process
- Best-fit algorithm
  - Chooses the block that is closest in size to the request
  - Worst performer overall
  - Since smallest block is found for process, the smallest amount of fragmentation is left
  - Memory compaction must be done more often



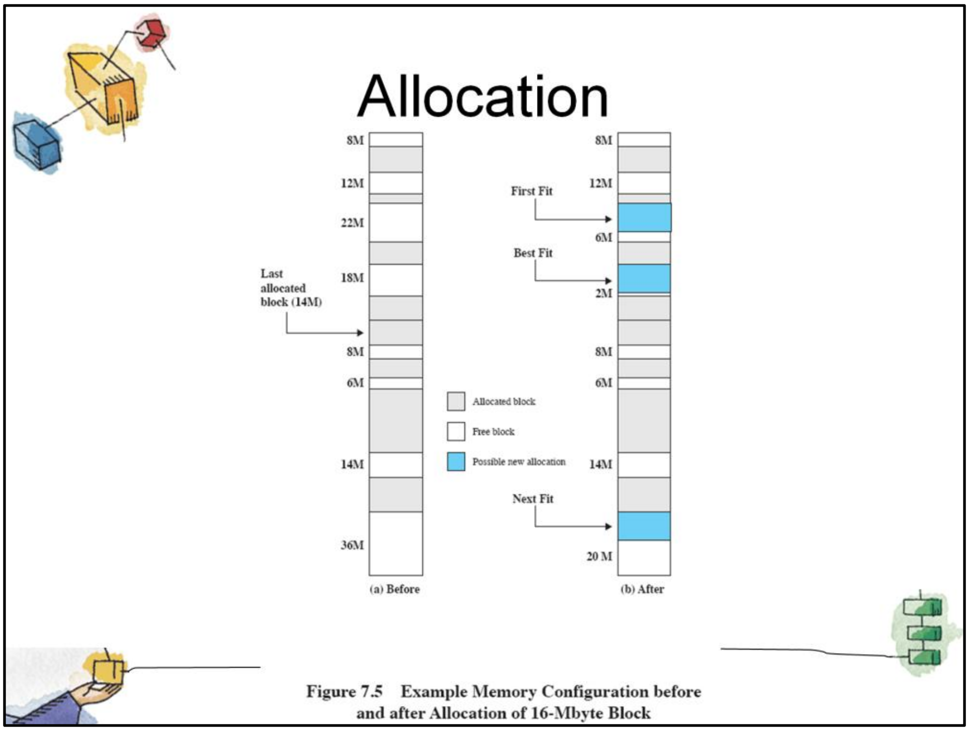
# Dynamic Partitioning

- First-fit algorithm
  - Scans memory form the beginning and chooses the first available block that is large enough
  - Fastest
  - May have many process loaded in the front end of memory that must be searched over when trying to find a free block



## Dynamic Partitioning

- Next-fit
  - Scans memory from the location of the last placement
  - More often allocate a block of memory at the end of memory where the largest block is found
  - The largest block of memory is broken up into smaller blocks
  - Compaction is required to obtain a large block at the end of memory



# Allocation

**Figure 7.5** Example Memory Configuration before and after Allocation of 16-Mbyte Block

Slide shows Fig 7.5 - an example memory configuration after a number of placement and swapping-out operations.

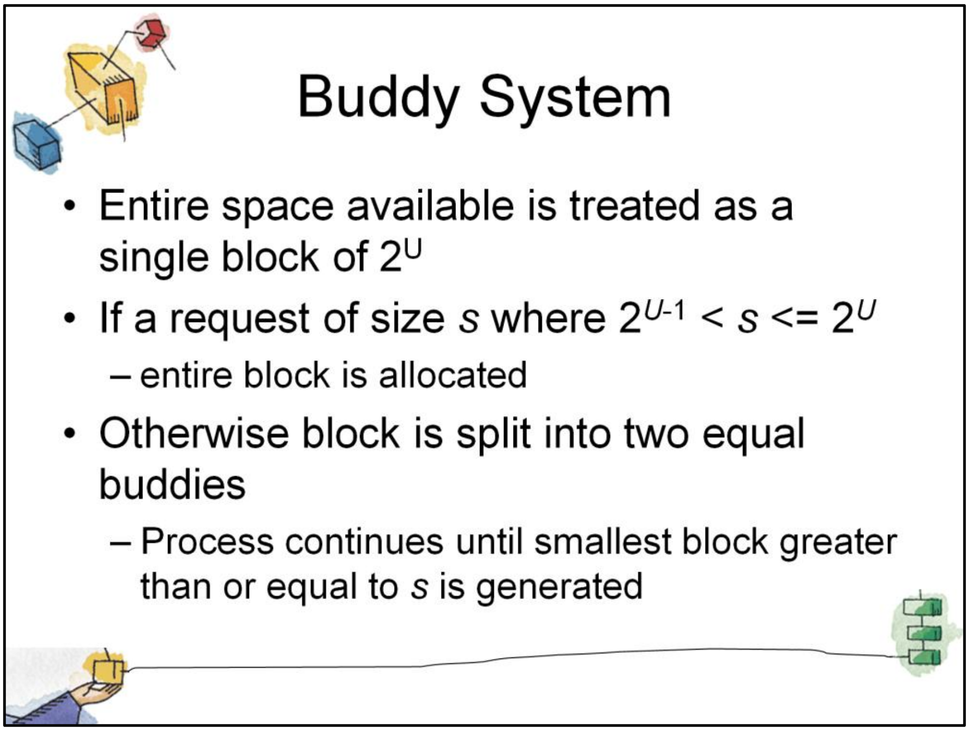
•The last block that was used was a 22-Mbyte block from which a 14-Mbyte partition was created.

•Figure 7.5b shows the difference between the best, first, and next-fit placement algorithms in satisfying a 16-Mbyte allocation request.

•**Best-fit** will search the entire list of available blocks and make use of the 18-Mbyte block, leaving a 2-Mbyte fragment.

•**First-fit** results in a 6-Mbyte fragment, and

•**Next-fit** results in a 20-Mbyte fragment.



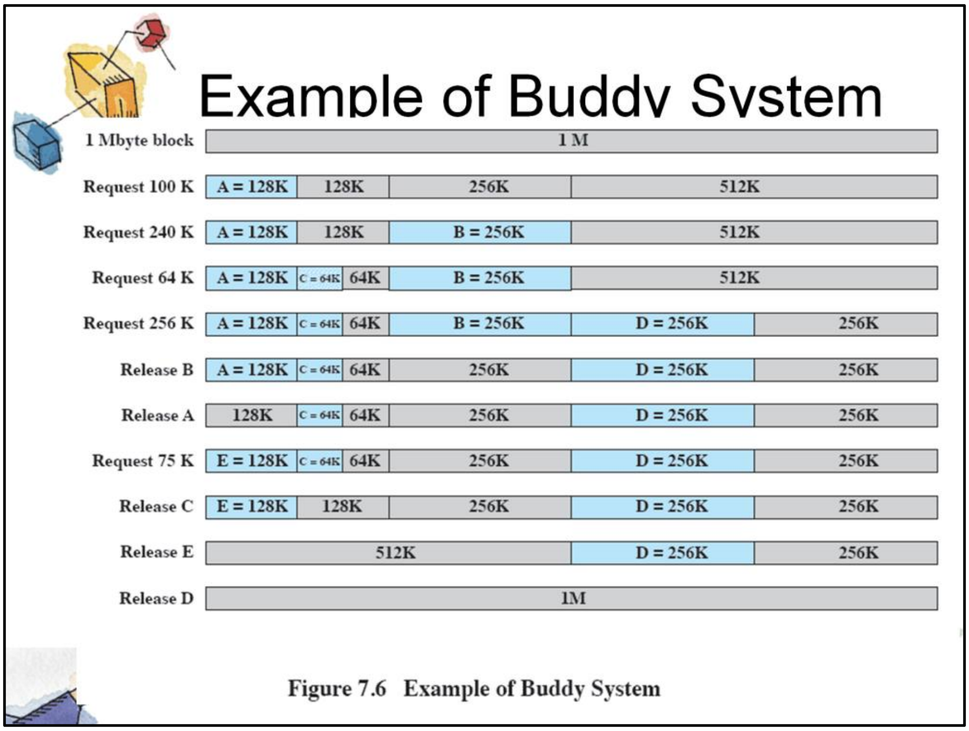
# Buddy System

- Entire space available is treated as a single block of $2^U$
- If a request of size $s$ where $2^{U-1} < s <= 2^U$
  - entire block is allocated
- Otherwise block is split into two equal buddies
  - Process continues until smallest block greater than or equal to $s$ is generated

In a fixed partitioning scheme limits the number of active processes and may use space inefficiently if there is a poor match between available partition sizes and process sizes.

A dynamic partitioning scheme is more complex to maintain and includes the overhead of compaction.

An interesting compromise is the buddy system.



# Example of Buddy System

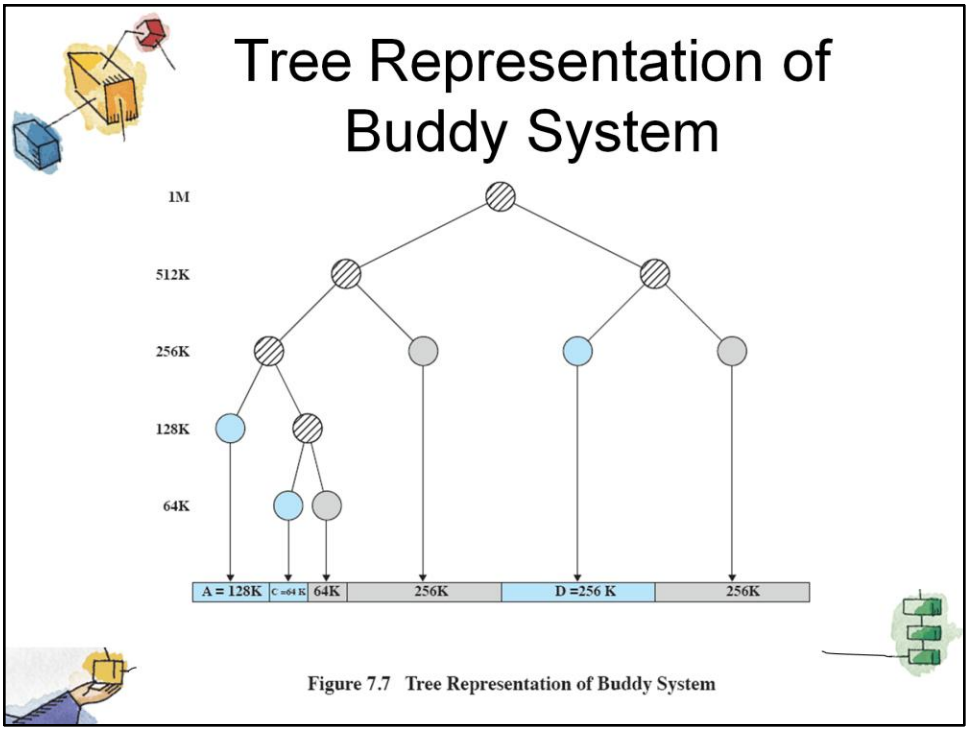
| | | | | | | |
|---|---|---|---|---|---|---|
| 1 Mbyte block | 1 M | | | | | |
| Request 100 K | A = 128K | 128K | | 256K | 512K | |
| Request 240 K | A = 128K | 128K | | B = 256K | 512K | |
| Request 64 K | A = 128K | C = 64K | 64K | B = 256K | 512K | |
| Request 256 K | A = 128K | C = 64K | 64K | B = 256K | D = 256K | 256K |
| Release B | A = 128K | C = 64K | 64K | 256K | D = 256K | 256K |
| Release A | 128K | C = 64K | 64K | 256K | D = 256K | 256K |
| Request 75 K | E = 128K | C = 64K | 64K | 256K | D = 256K | 256K |
| Release C | E = 128K | 128K | | 256K | D = 256K | 256K |
| Release E | 512K | | | | D = 256K | 256K |
| Release D | 1M | | | | | |

Figure 7.6 Example of Buddy System

Figure 7.6 gives an example using a 1-Mbyte initial block.

The first request,A, is for 100 Kbytes, for which a 128K block is needed.

- The initial block is divided into two 512K buddies.
- The first of these is divided into two 256K buddies,
- and the first of these is divided into two 128K buddies,
- one of which is allocated to A.
- The next request,B, requires a 256K block. Such a block is already available and is allocated.
- The process continues with splitting and coalescing occurring as needed.
- Note that when E is released,two 128K buddies are coalesced into a 256K block, which is immediately coalesced with its buddy



# Tree Representation of Buddy System

Figure 7.7 Tree Representation of Buddy System

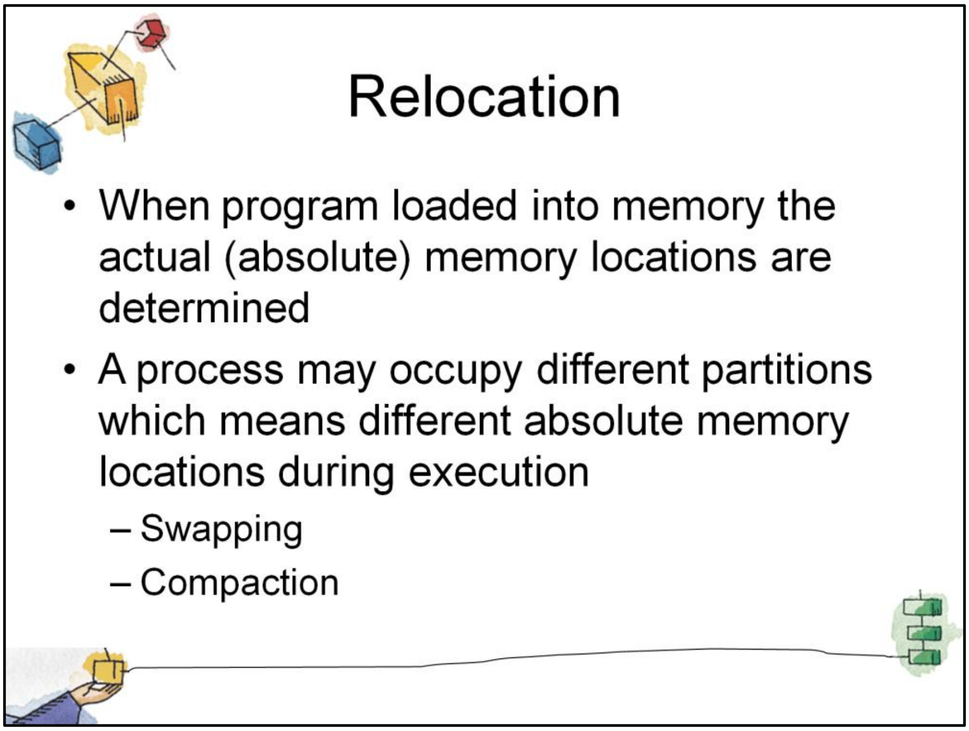
Figure 7.7 shows a binary tree representation of the buddy allocation immediately after the Release B request.

The leaf nodes represent the current partitioning the memory.

If two buddies are leaf nodes, **then at least one must be allocated;**

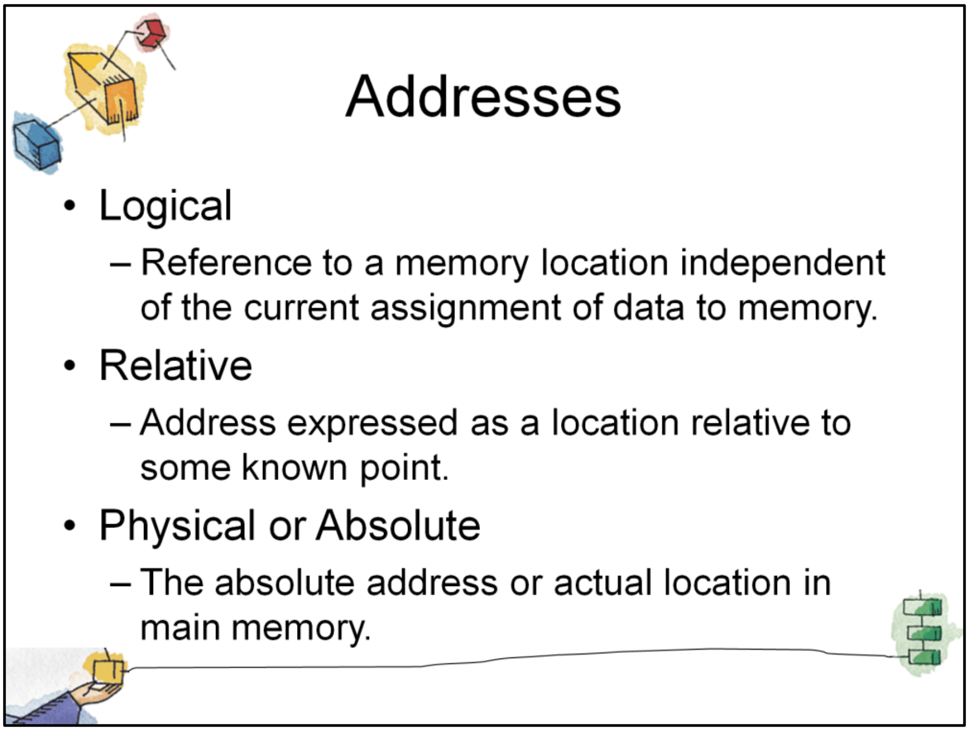
otherwise they would be coalesced into a larger block.

- The buddy system is a reasonable compromise to overcome the disadvantages of both the fixed and variable partitioning schemes,
- But in contemporary operating systems, virtual memory based on paging and segmentation is superior.
- However, the buddy system has found application in parallel systems as an efficient means of allocation and release for parallel programs. A modified form of the buddy system is used for UNIX kernel memory allocation (described in Chapter 8).



# Relocation

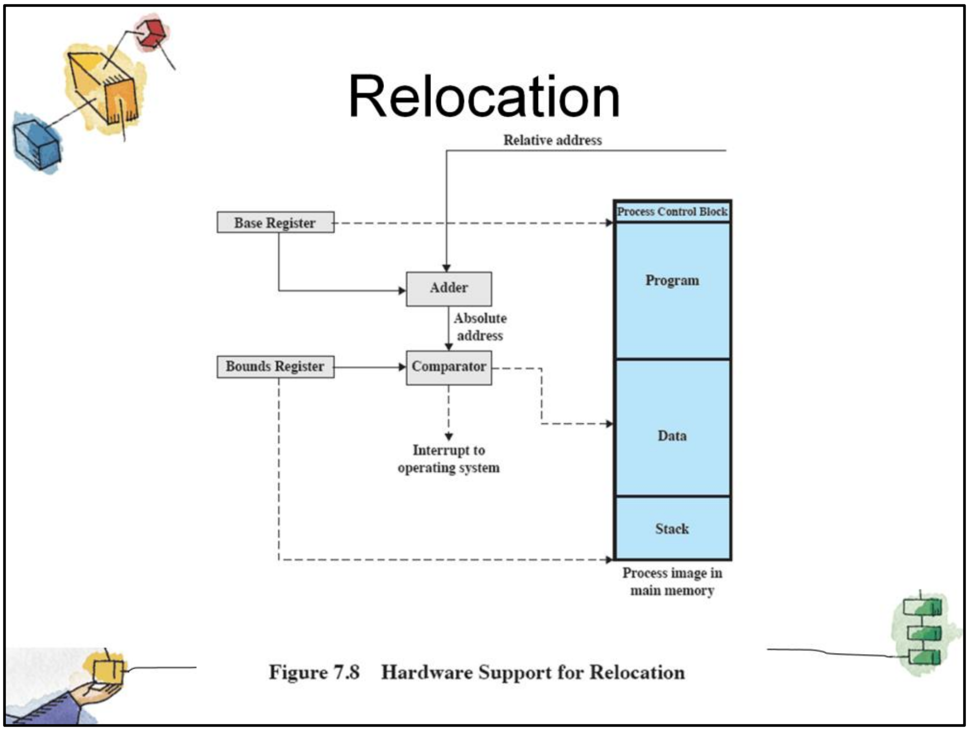
- When program loaded into memory the actual (absolute) memory locations are determined
- A process may occupy different partitions which means different absolute memory locations during execution
  - Swapping
  - Compaction



# Addresses

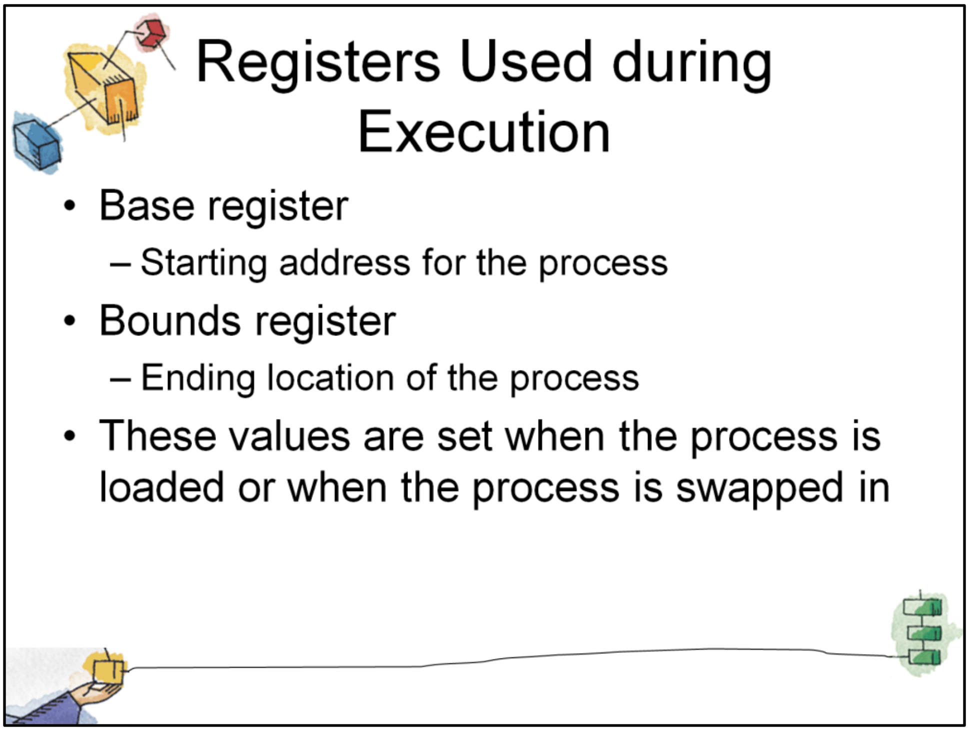
- Logical
  - Reference to a memory location independent of the current assignment of data to memory.
- Relative
  - Address expressed as a location relative to some known point.
- Physical or Absolute
  - The absolute address or actual location in main memory.

A translation must be made from both Logical and Relative addresses to arrive at the Absolute address



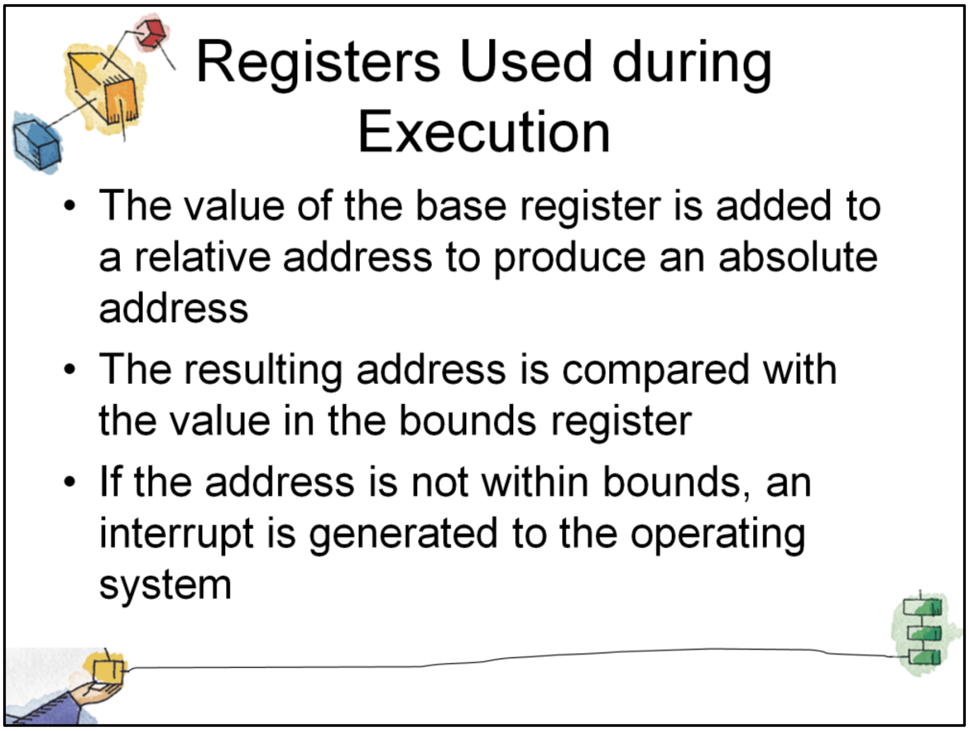
# Relocation

Figure 7.8 Hardware Support for Relocation



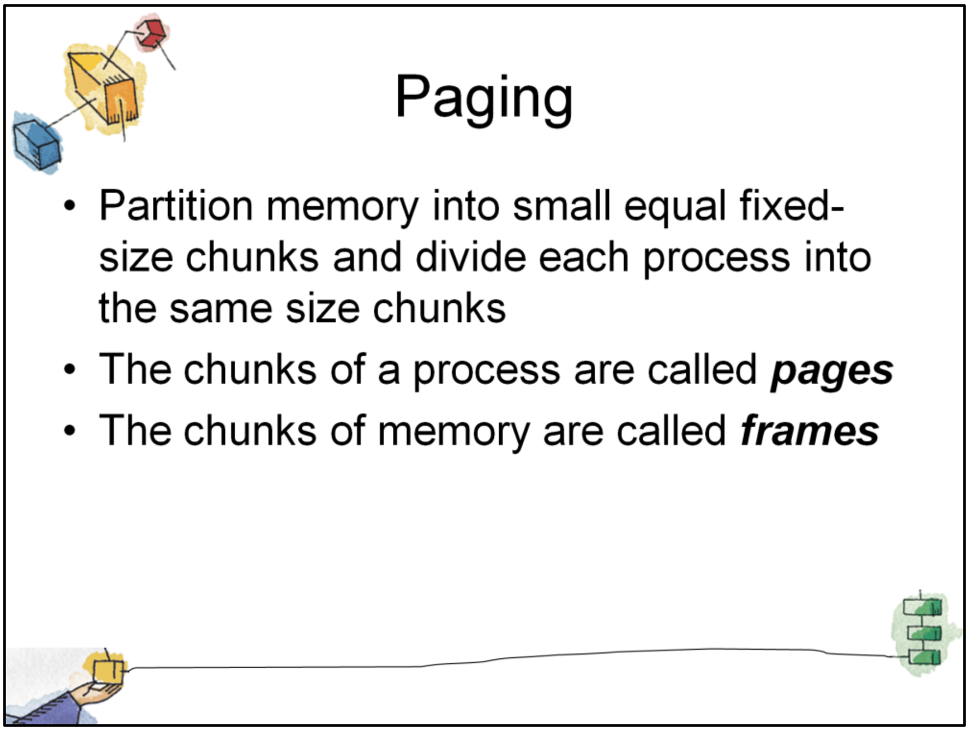
# Registers Used during Execution

- Base register
  - Starting address for the process
- Bounds register
  - Ending location of the process
- These values are set when the process is loaded or when the process is swapped in



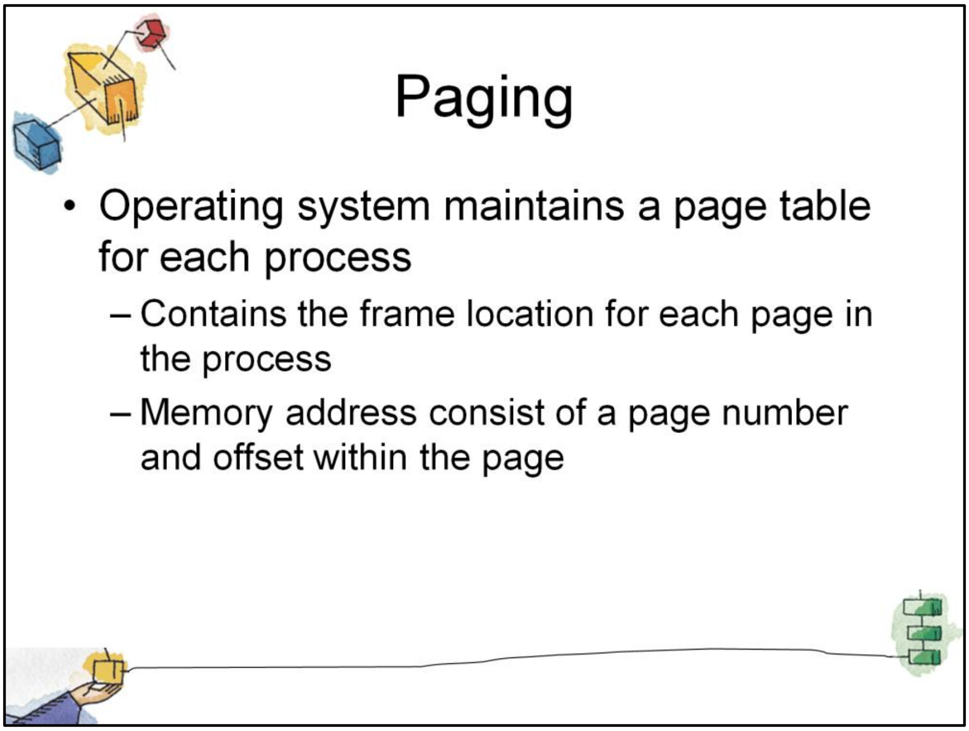
# Registers Used during Execution

- The value of the base register is added to a relative address to produce an absolute address
- The resulting address is compared with the value in the bounds register
- If the address is not within bounds, an interrupt is generated to the operating system



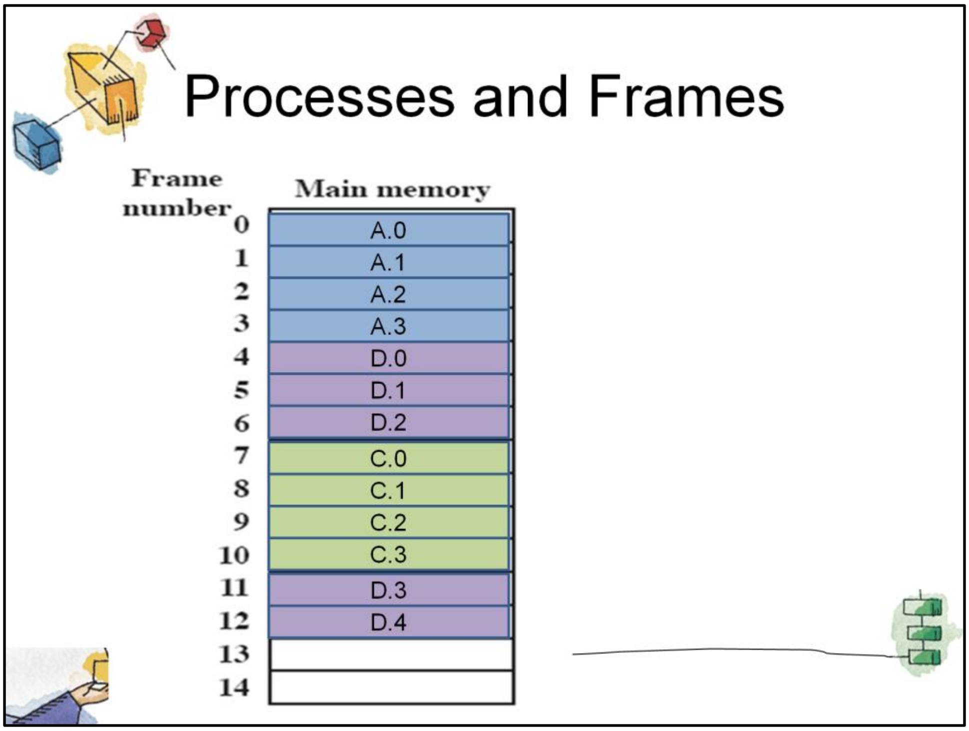
# Paging

- Partition memory into small equal fixed-size chunks and divide each process into the same size chunks
- The chunks of a process are called ***pages***
- The chunks of memory are called ***frames***



# Paging

- Operating system maintains a page table for each process
  - Contains the frame location for each page in the process
  - Memory address consist of a page number and offset within the page

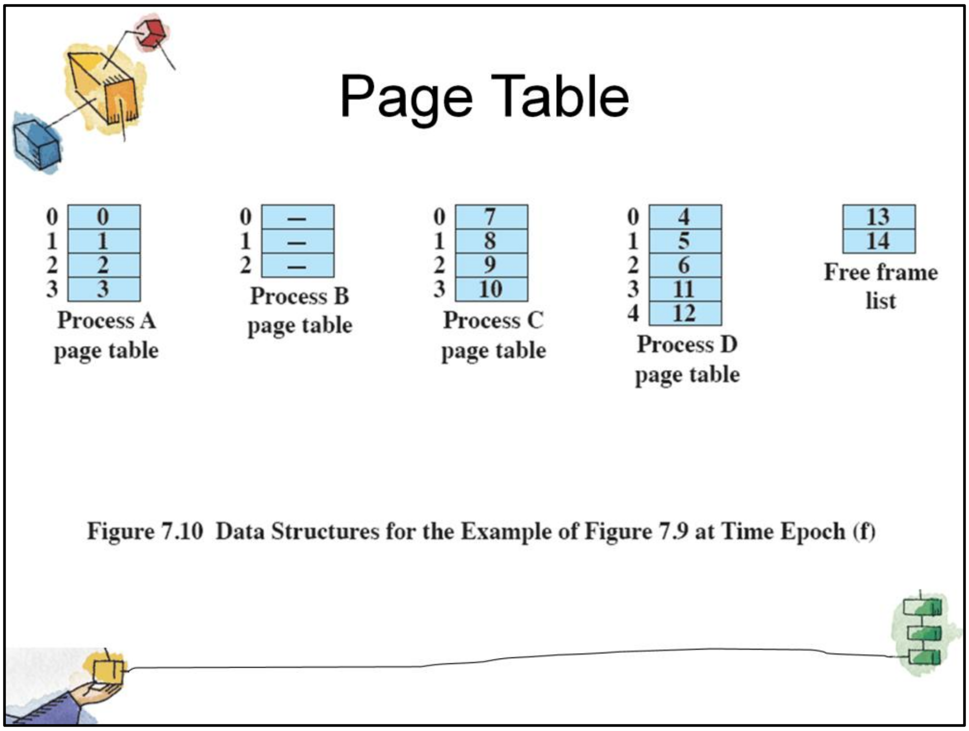


# Processes and Frames

| Frame number | Main memory |
|---|---|
| 0 | A.0 |
| 1 | A.1 |
| 2 | A.2 |
| 3 | A.3 |
| 4 | D.0 |
| 5 | D.1 |
| 6 | D.2 |
| 7 | C.0 |
| 8 | C.1 |
| 9 | C.2 |
| 10 | C.3 |
| 11 | D.3 |
| 12 | D.4 |
| 13 | |
| 14 | |

Animated slide

1. System with a number of frames allocated
2. Process A, stored on disk, consists of four pages. When it comes time to load this process, the operating system finds four free frames and loads the four pages of process A into the four frames.
3. Process B, consisting of three pages, and process C, consisting of four pages, are subsequently loaded.
4. Then process B is suspended and is swapped out of main memory.
5. Later, all of the processes in main memory are blocked, and the operating system needs to bring in a new process, process D, which consists of five pages. The Operating System loads the pages into the available frames and updates the ***page table***



# Page Table

| | |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |

Process A page table

| | |
|---|---|
| 0 | — |
| 1 | — |
| 2 | — |

Process B page table

| | |
|---|---|
| 0 | 7 |
| 1 | 8 |
| 2 | 9 |
| 3 | 10 |

Process C page table

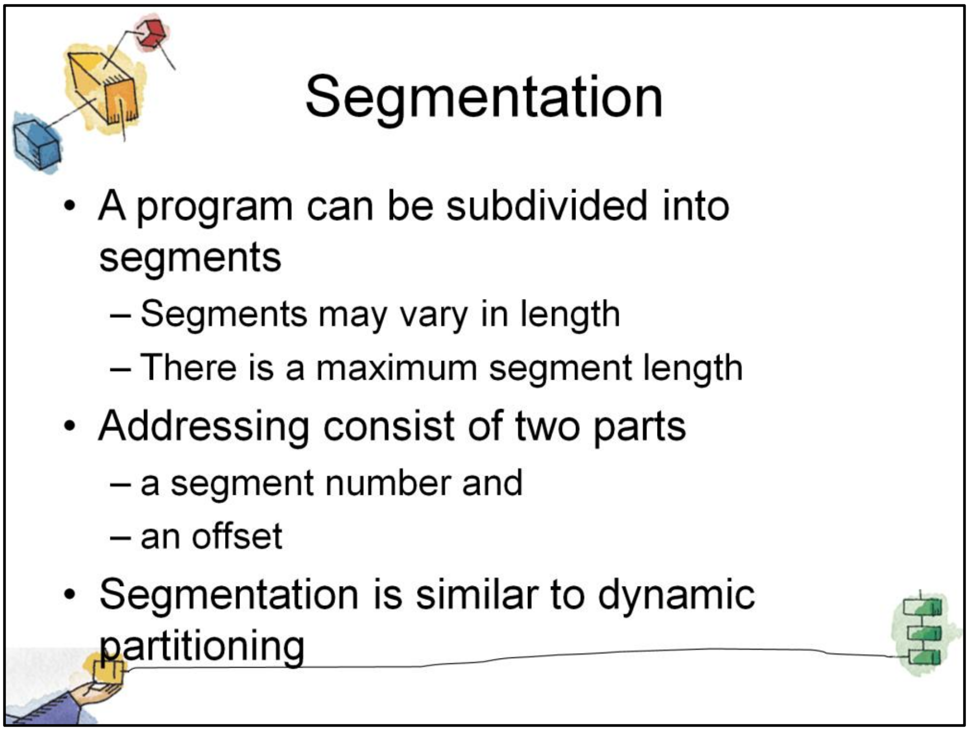
| | |
|---|---|
| 0 | 4 |
| 1 | 5 |
| 2 | 6 |
| 3 | 11 |
| 4 | 12 |

Process D page table

| |
|---|
| 13 |
| 14 |

Free frame list

Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)



## Segmentation

- A program can be subdivided into segments
  - Segments may vary in length
  - There is a maximum segment length
- Addressing consist of two parts
  - a segment number and
  - an offset
- Segmentation is similar to dynamic partitioning

The difference with 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 suffers from external fragmentation (as does dynamic partitioning)

However, because a process is broken up into a number of smaller pieces, the external fragmentation should be less.

A 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 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.

# Logical Addresses

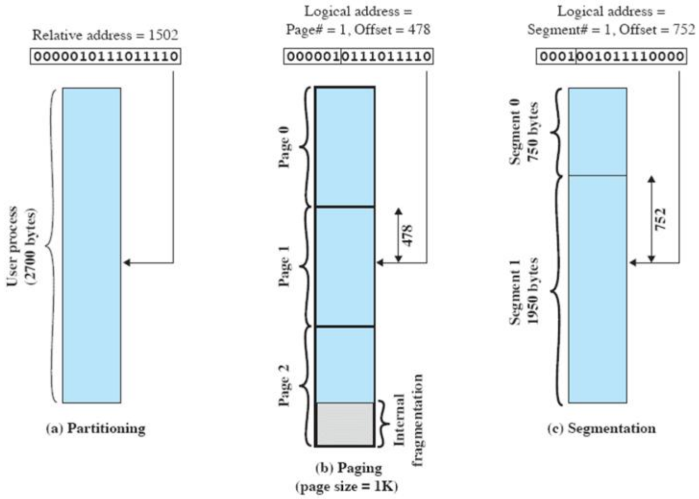


Figure 7.11 Logical Addresses

In this example, 16-bit addresses are used, and the page size is 1K =1024 bytes.
The relative address 1502, in binary form, is 0000010111011110.
With a page size of 1K, an offset field of 10 bits is needed, leaving 6 bits for the page number.
Thus a program can consist of a maximum of $2^6$ =64 pages of 1K bytes each.

As Figure 7.11b shows, relative address 1502 corresponds to

- an offset of 478 (0111011110) on page 1 (000001),
- which yields the same 16-bit number, 0000010111011110.

Consider an address of n + m bits, where the leftmost n bits are the segment number and the rightmost m bits are the offset.
In the example on the slide

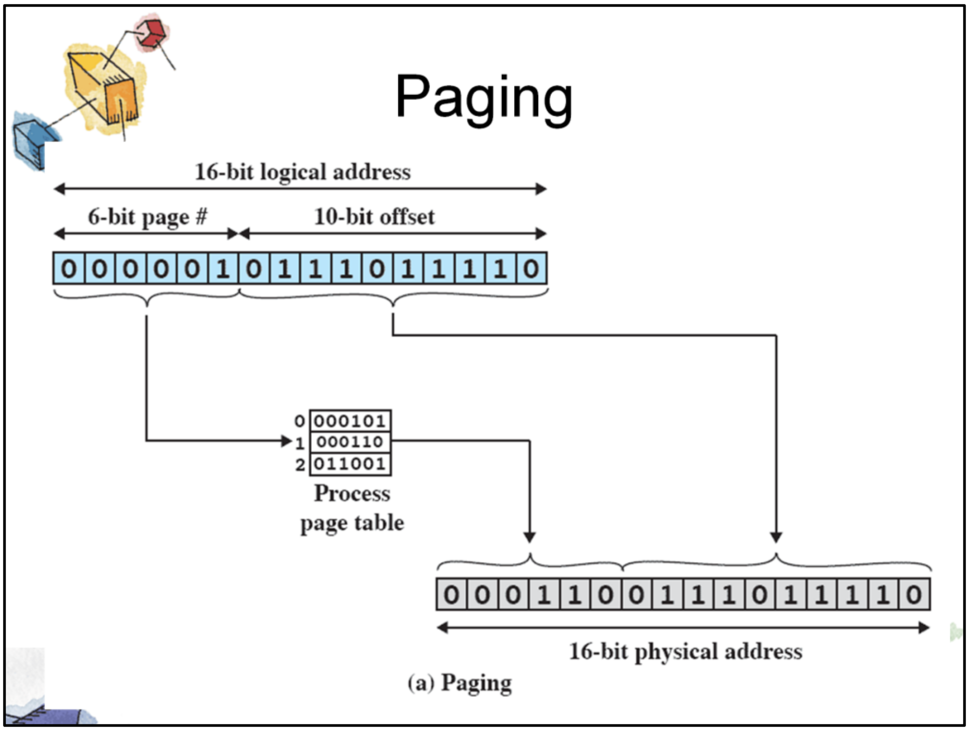
- n = 4 and
- m =12.

Thus the maximum segment size is $2^{12}$ = 4096.

The following steps are needed for address translation:

- Extract the segment number as the leftmost n bits of the logical address.
- Use the segment number as an index into the process segment table to find the starting physical address of the segment.

Compare the offset, expressed in the rightmost m bits, to the length of the segment. If the offset is greater than or equal to the length, the address is invalid.
The desired physical address is the sum of the starting physical address of the segment plus the offset.



## Paging

16-bit logical address

6-bit page #

10-bit offset

0 0 0 0 0 1 0 1 1 1 0 1 1 1 1 0

0 000101
1 000110
2 011001

Process
page table

0 0 0 1 1 0 0 1 1 1 0 1 1 1 1 0

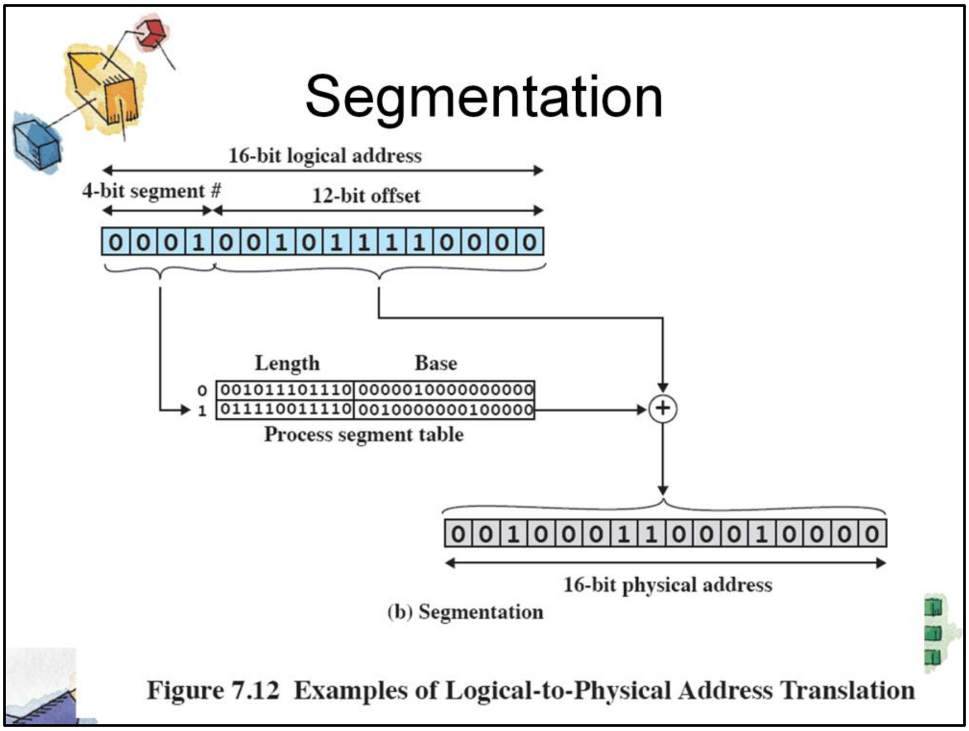
16-bit physical address

(a) Paging

In our example, we have the logical address 0000010111011110,
which is page number 1, offset 478.

Suppose that this page is residing in main memory frame 6 = binary 000110.

Then the physical address is frame number 6, offset 478 = 0001100111011110



# Segmentation

| | Length | Base |
|---|---|---|
| 0 | 001011101110 | 0000010000000000 |
| 1 | 011110011110 | 0010000000100000 |

(b) Segmentation

Figure 7.12 Examples of Logical-to-Physical Address Translation

In our example, we have the logical address 0001001011110000, which is segment number 1, offset 752.

Suppose that this segment is residing in main memory starting at physical address 0010000000100000.

Then the physical address is 0010000000100000 + 001011110000 = 0010001100010000