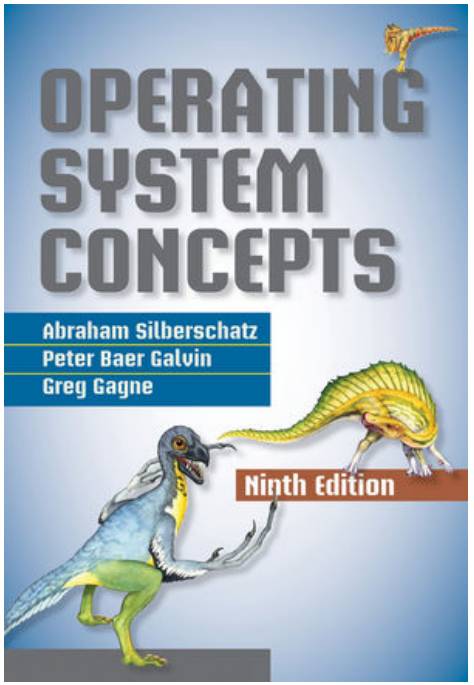
6 - Main Memory

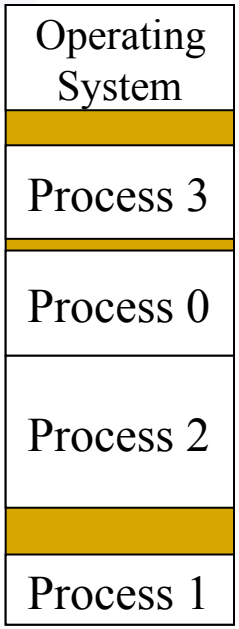
EECE 315 (101)
ECE – UBC
2013 W2



Acknowledgement: This set of slides is partly based on the PPTs provided by the Wiley's companion website (including textbook images, when not explicitly mentioned/referenced).

Lecture Outline

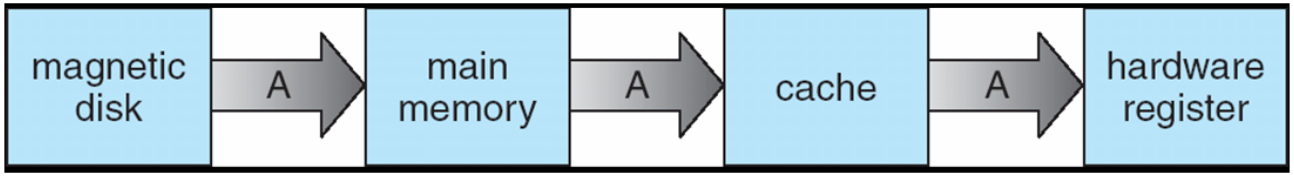
- **Background**
- Swapping
- Contiguous Memory Allocation
  - Fragmentation
- Paging
  - Structure of the Page Table
- Segmentation



Background

- A program must be brought into memory (from disk) and placed within a process for it to be run

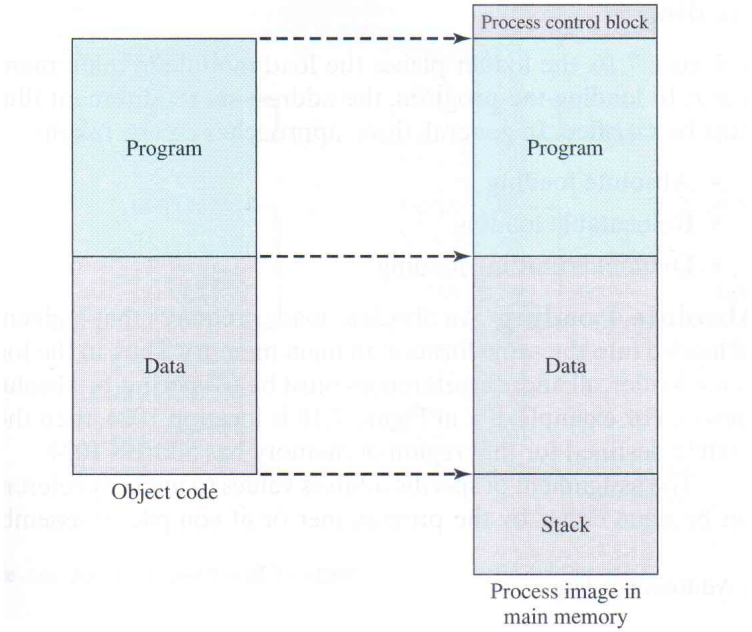
- Main memory and registers are the only storage the CPU can access directly
  - Register access takes one CPU clock (or less)
  - Main memory can take many cycles (the CPU may have to stall)
  - Cache sits between main memory and CPU registers

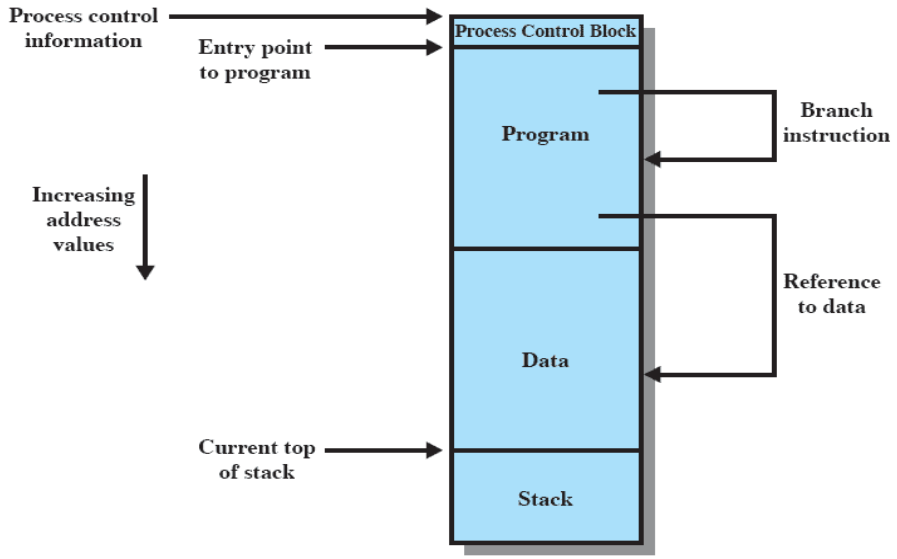


- The OS, executing in kernel mode, is given unrestricted access to both OS and users' memory

Background (cont)

- We must make sure that each process has a separate memory space
  - we need the ability to determine the range of legal addresses that the process may access,
  - and to make sure the process can access only these legal addresses
- A simplified view of a process image in main memory (compare this with Slide 3-5)

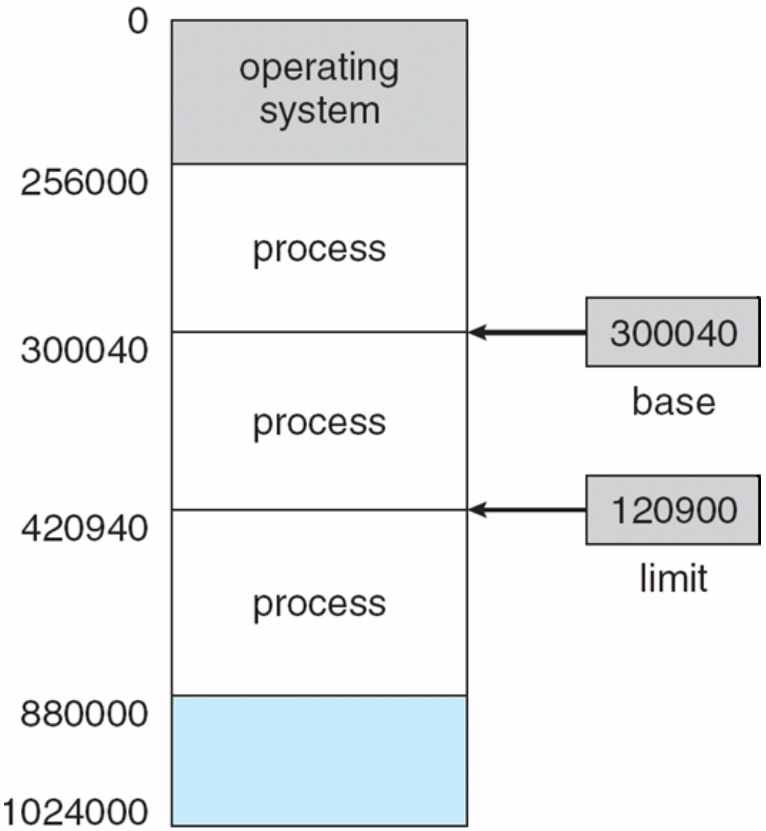




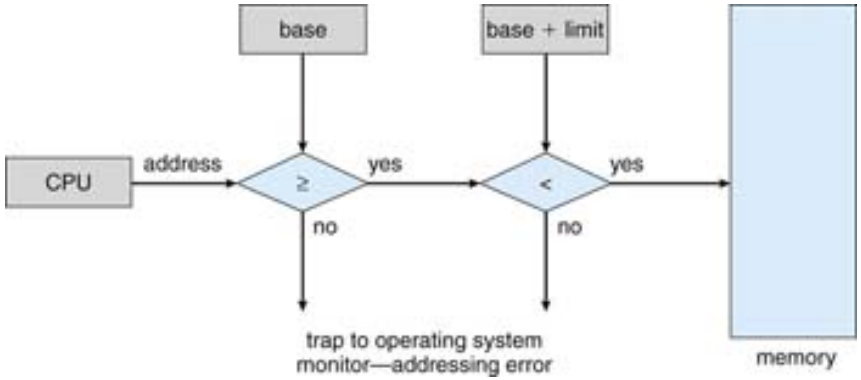
Source: Stallings

Background (cont)

- A pair of base and limit registers define the address space

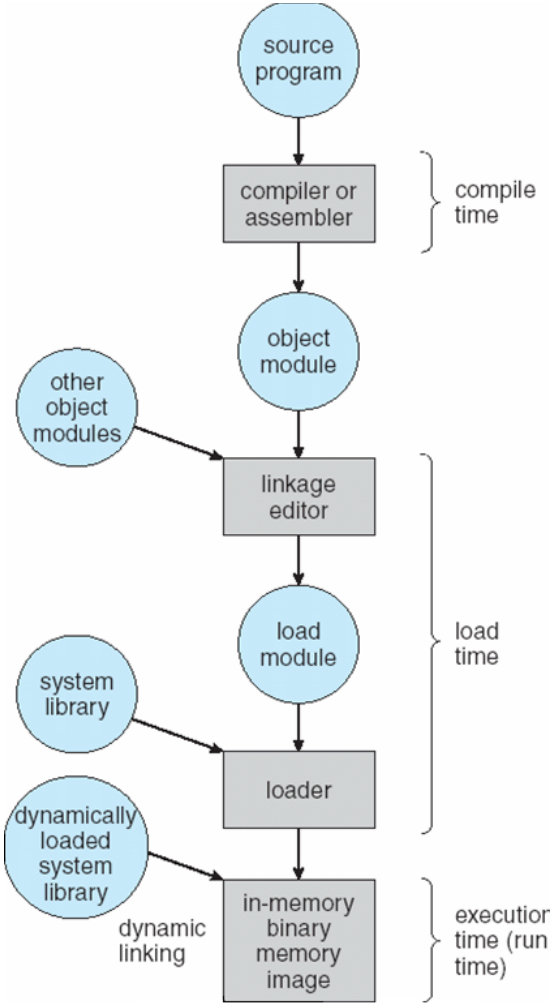


- Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode to ensure correct operation



Binding of Instructions and Data to Memory

- Most systems allow a user process to reside in any part of the physical memory, (e.g. it doesn't need to start at 0)
  - This approach affects the addresses that the user program can use
  - Addresses in the source code are generally symbolic
    - e.g. *counter*
  - A compiler will typically bind these symbolic addresses to relocatable addresses
    - e.g. "14 bytes from the beginning of this module"
  - The linkage editor or loader may bind the relocatable addresses to absolute addresses

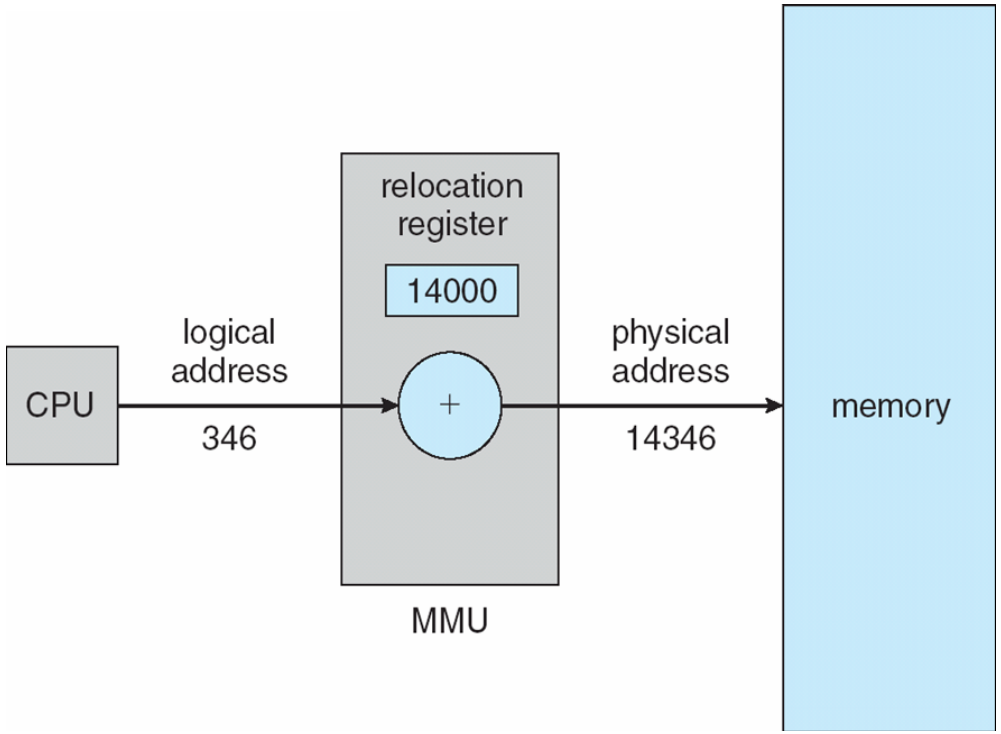


Logical Versus Physical Address Space

- Logical address: an address generated by the CPU is referred to as a logical address (also referred to as virtual address)
- Physical address: an address seen by the memory unit (i.e. the one loaded into the memory-address register of the memory) is referred to as a physical address.
  - The compile-time and load-time address-binding methods generate identical logical and physical addresses.
  - However, logical (virtual) and physical addresses differ in the execution-time address-binding scheme
- The set of all logical addresses generated by a program is a logical address space; the set of all physical addresses corresponding to these logical addresses is a physical address space.
- The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU)

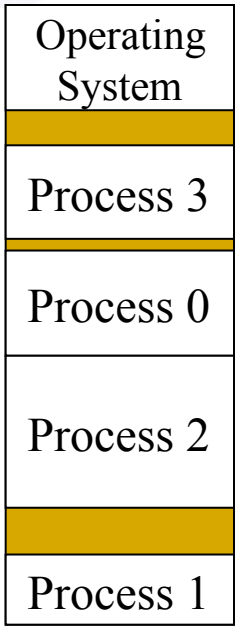
Memory-Management Unit (MMU)

- MMU can use many different methods to accomplish the mapping
- We first illustrate a simple MMU scheme:
  - The base register is now called a relocation register
  - The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - The user program deals with *logical* addresses; it never sees the *real* physical addresses
- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management



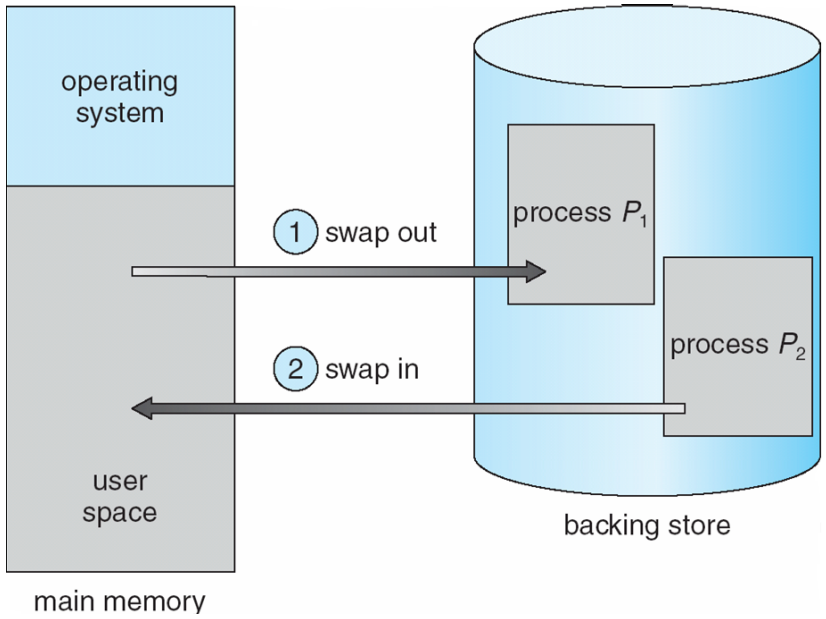
Lecture Outline

- Background
- **Swapping**
- Contiguous Memory Allocation
  - Fragmentation
- Paging
  - Structure of the Page Table
- Segmentation
- Examples: Intel and ARM processors



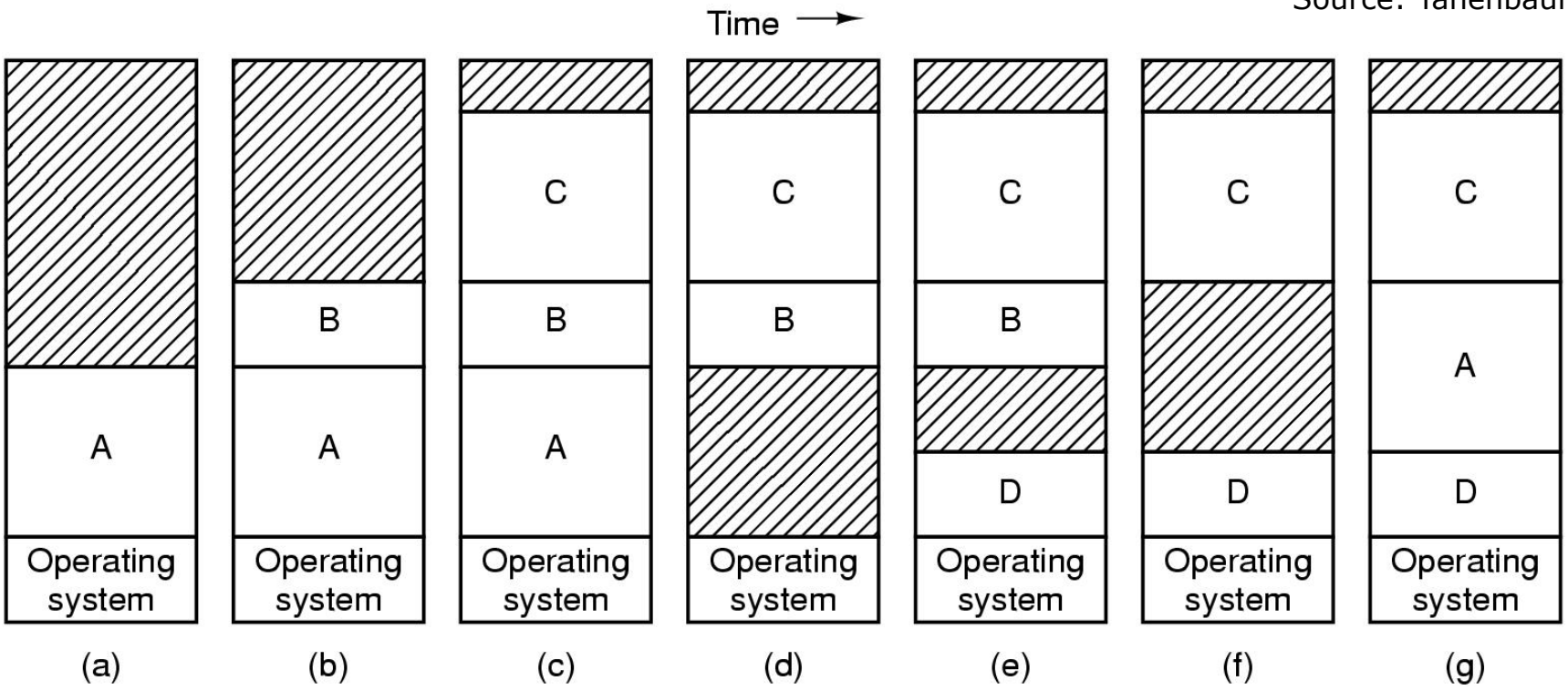
Swapping

- A process must be in the memory to be executed
  - A process, however, can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- The system maintains a **ready queue** consisting of all ready-to-run processes whose memory images are in memory or on the backing store
  - whenever the CPU scheduler decides to execute a process, it call the dispatcher,
  - if the dispatcher determines that the process is not in memory and there is not enough free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process.



Swapping Example

Source: Tanenbaum



a) only A in memory

b & c) B and C are created or swapped in

d) A is swapped out

e & F) D comes in and B goes out

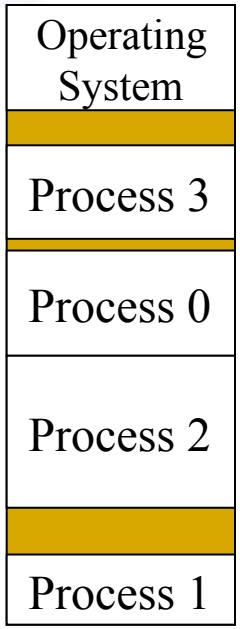
g) A is swapped in again

Swapping (cont)

- **Backing store** is a fast disk that is large enough to accommodate copies of all memory images for all users
  - it must provide direct access to these memory images
- **Roll out, roll in** is a swapping variant used for priority-based scheduling algorithms;
  - a lower-priority process is swapped out, so that a higher-priority process can be loaded and executed
- The context-switch time in such a swapping system is fairly high
  - a major part of swap time is transfer time
  - the total transfer time is directly proportional to the amount of memory swapped
  - Swapping is constrained by other factors as well (e.g. pending I/O)
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

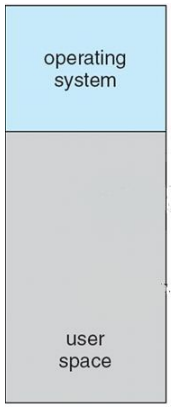
Lecture Outline

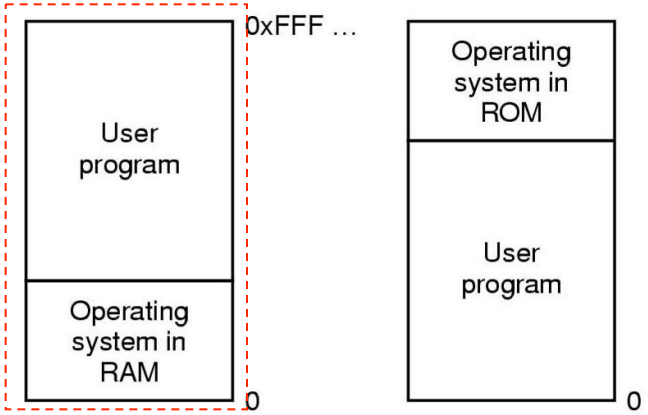
- Background
- Swapping
- **Contiguous Memory Allocation**
  - Fragmentation
- Paging
  - Structure of the Page Table
- Segmentation



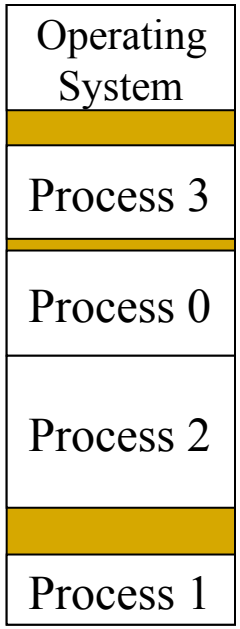
Contiguous Allocation

- The main memory must accommodate both the OS and the various user processes
- The main memory is usually divided into two partitions:
  - resident operating system, usually held in low memory with interrupt vector (equally it might be in high memory)
  - user processes then held in high memory





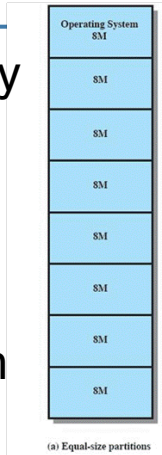
Source: Tanenbaum

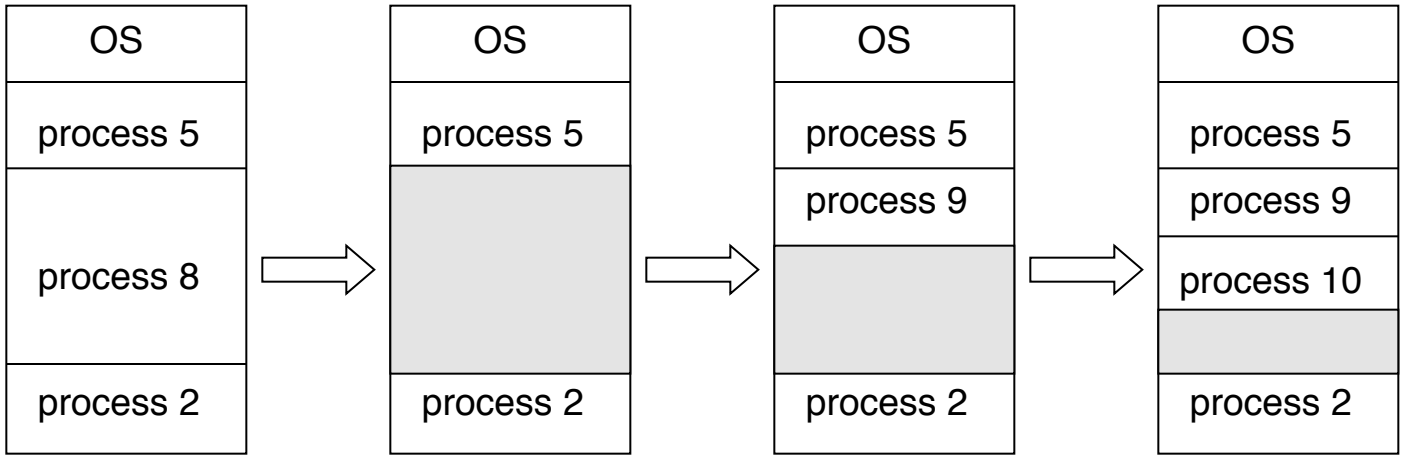


- In contiguous memory allocation, each process is contained in a single contiguous section of memory.

Contiguous Allocation (Cont)

- One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions.
  - each partition may contain exactly one process
  - was used in IBM OS/360 (no longer is used)
- In the variable-partition scheme, the OS keeps a table indicating which parts of memory are available and which are occupied
  - a hole is a block of available memory. So holes of various sizes may be scattered throughout the memory
  - when a process arrives and is to be allocated space, it is allocated memory from a hole large enough to accommodate it





Dynamic Storage-Allocation Problem

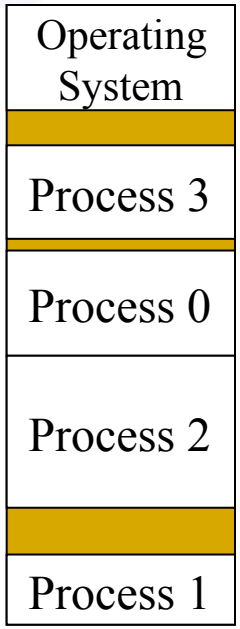
- Dynamic storage-allocation problem concerns how to satisfy a request of size *n* from a list of free holes
- The following three solutions are the ones most commonly used:
  - **First-fit**: Allocate the *first* hole that is big enough
  - **Best-fit**: Allocate the *smallest* hole that is big enough
    - we must search the entire list, unless ordered by size
    - this strategy produces the smallest leftover hole
  - **Worst-fit**: Allocate the *largest* hole
    - we must also search the entire list
    - this strategy produces the largest leftover hole (which might be more useful that the smaller leftover hole from a best-fit approach)
- Simulations have shown that both first-fit and best-fit are better than worst-fit in terms of speed and storage utilization
- Neither first-fit nor best-fit is better than the other, but first-fit is generally faster

Fragmentation

- As processes are loaded and removed from memory, the free memory space is broken into little pieces.
  - External Fragmentation exists when the total memory space is enough to satisfy a request, but it is not contiguous
  - Internal Fragmentation: when the allocated memory (e.g. using fixed-size blocks) is larger than the requested memory, the difference is internal fragmentation, i.e. unused memory that is internal to a partition
- One possible method to reduce external fragmentation is **compaction**
  - we shuffle memory contents to place all free memory together in one large block
  - compaction is possible *only* if the relocation is dynamic, and is done at execution time

Lecture Outline

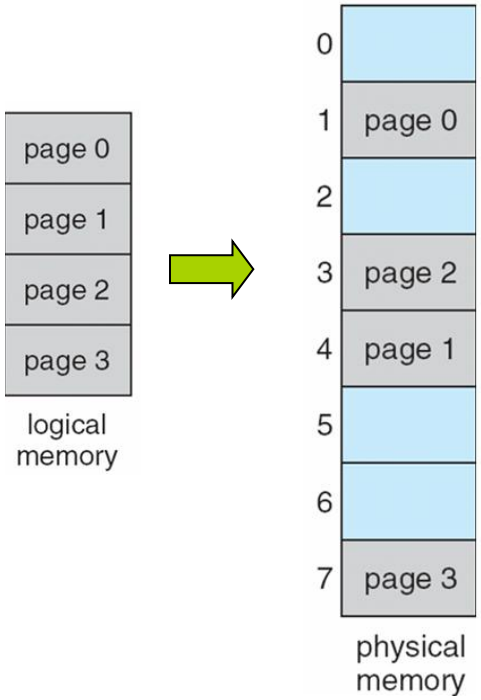
- Background
- Swapping
- Contiguous Memory Allocation
  - Fragmentation
- **Paging**
  - Structure of the Page Table
- Segmentation
- Examples: Intel and ARM processors



Paging

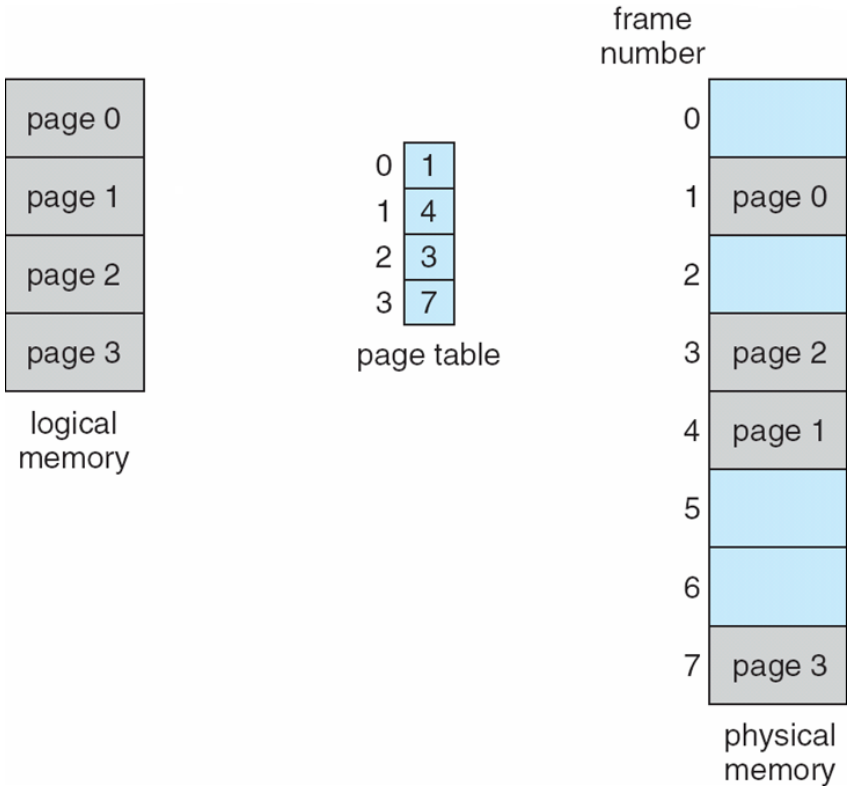
- Another solution to the external-fragmentation is to permit the logical address space to be noncontiguous.
  - Two complementary techniques achieve this solutions: Paging and segmentation

- Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous
  - Paging avoids external fragmentation and the need for compaction
  - It also solves the considerable problem of fitting memory chunks of varying sized onto the backing store (when swapped out)



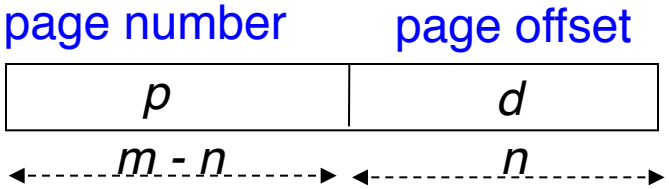
Paging (cont)

- The basic method for implementing paging involves
  - breaking the physical memory into fixed-sized blocks called **frames**
  - and breaking the logical memory into blocks of same size called **pages**
    - the page and frame size is defined by the hardware
    - the page size is a power of 2, between 512 bytes and 16 MB
- To run a program of size *k* pages, we need to find *k* free frames



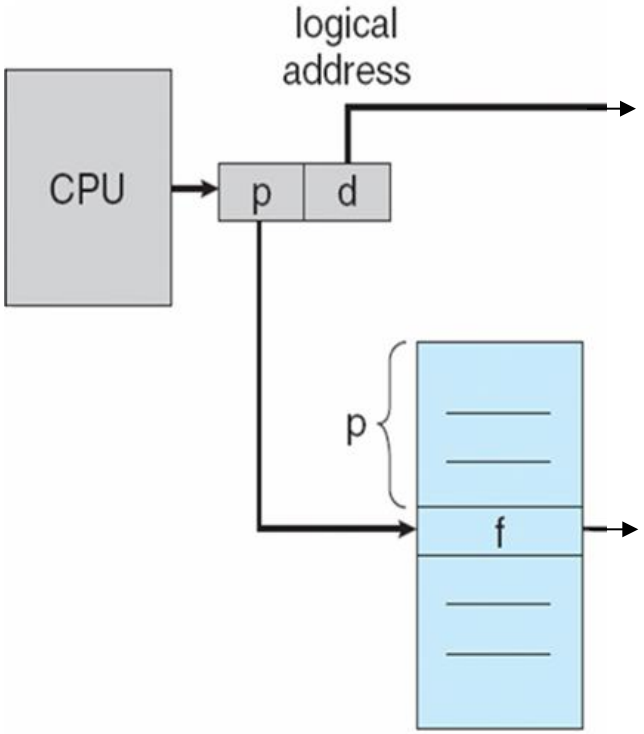
Address Translation Scheme

- Every address generated by the CPU is divided into two parts
  - assume a given logical address space 2^m *and page size* 2^n



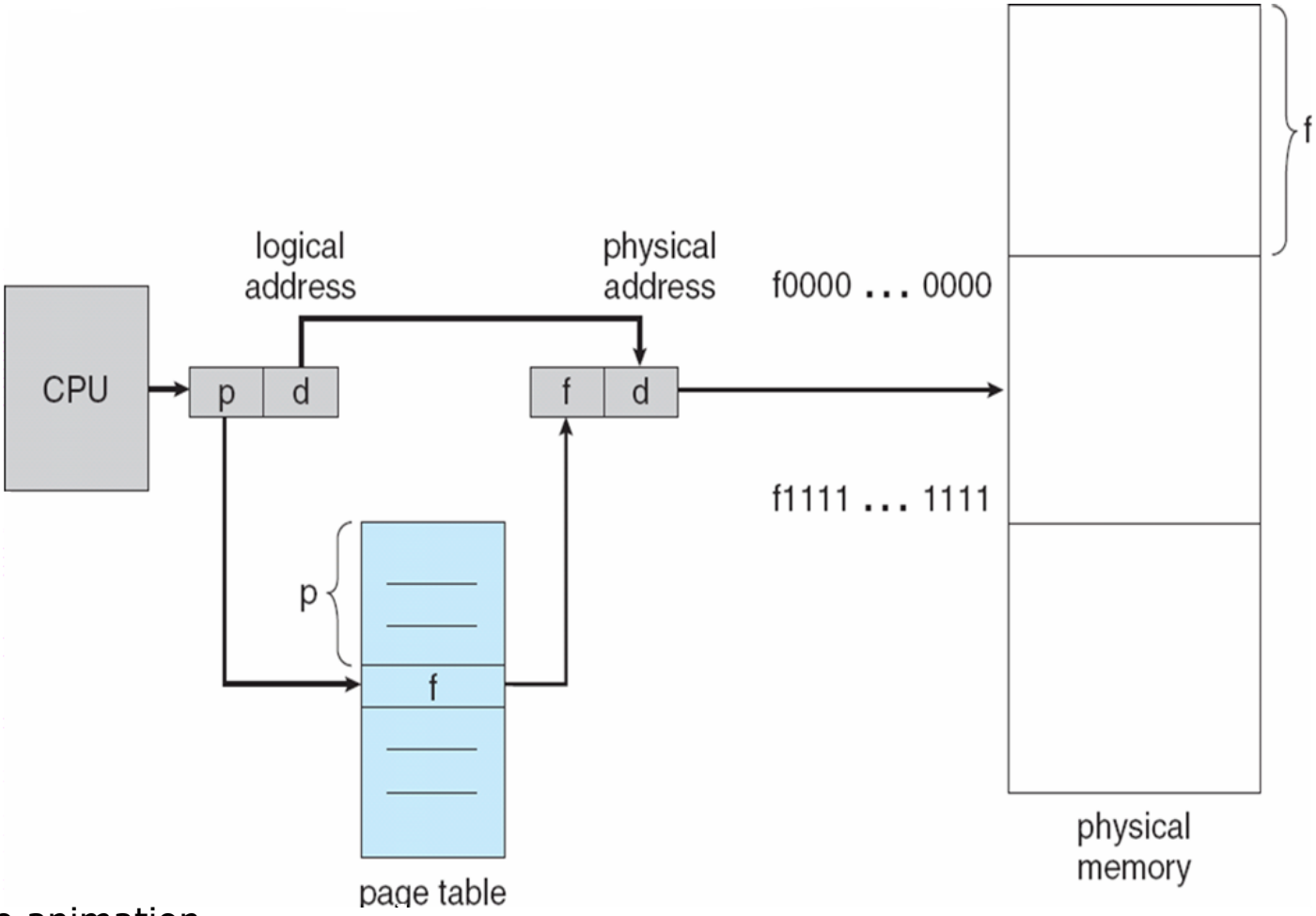
- **Page number (*p*)** is used as an index into a *page table* which contains the base address of each page in physical memory

- **Page offset (d)** is combined with the above base address to define the physical memory address that is sent to the memory unit



Paging Hardware

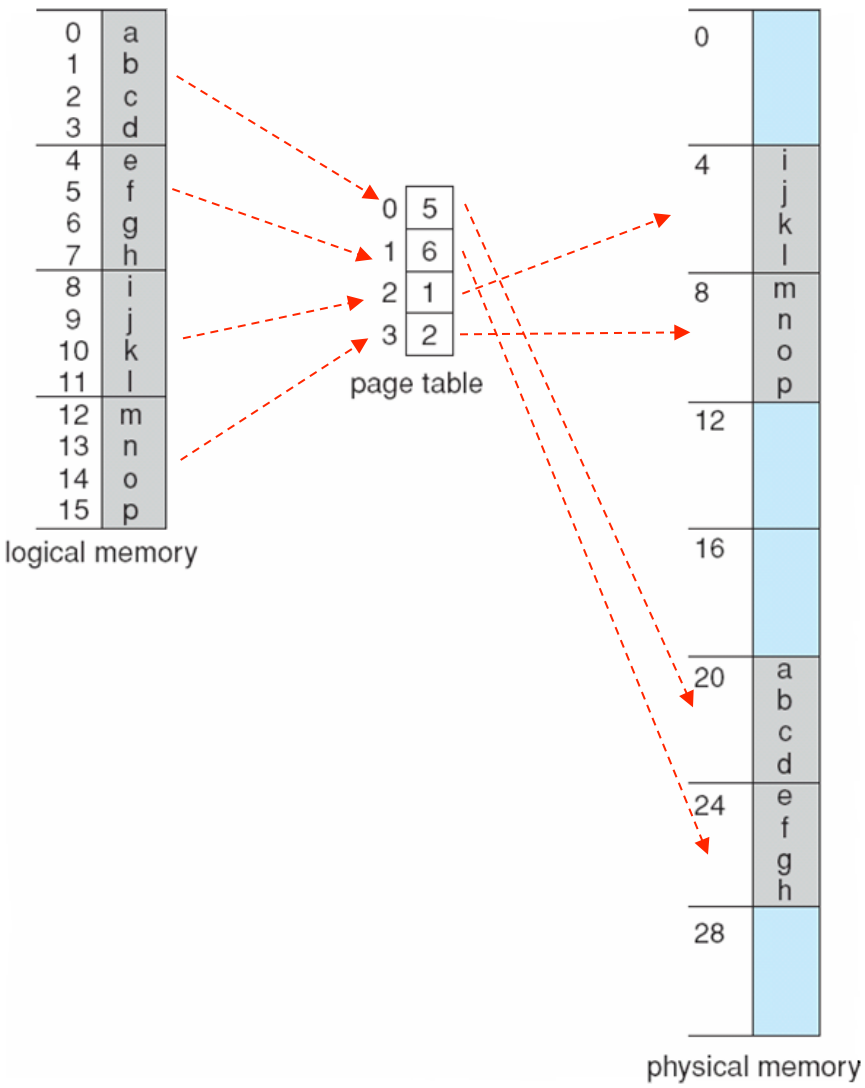
- The hardware support for paging in illustrated below:



= See animation

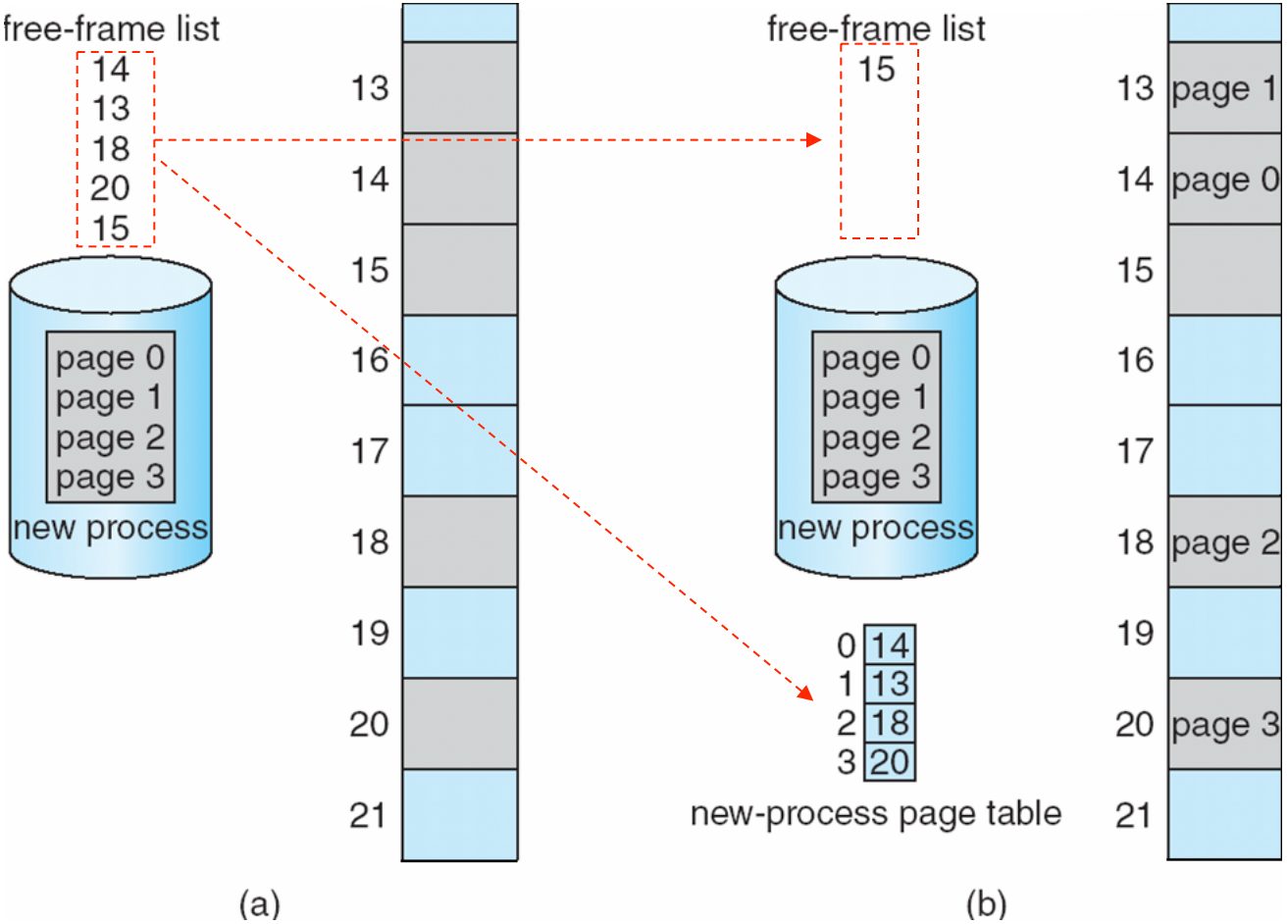
Paging Example

- As an example, consider the memory in the following figure
  - assume a 16-byte logical address (m=4), a 4-byte pages (n=2) and a 32-byte memory
  - We show how the user's view of the memory can be mapped into physical memory
- As shown in the example, paging is a form of dynamic relocation.
  - Every logical address is bound by the paging hardware to some physical address.
- By using paging, we have no external fragmentation, but we may have some internal fragmentation



Free Frames

- The OS must manage the allocation details of physical memory. This info is generally kept in a data structure called a frame table.



Before allocation

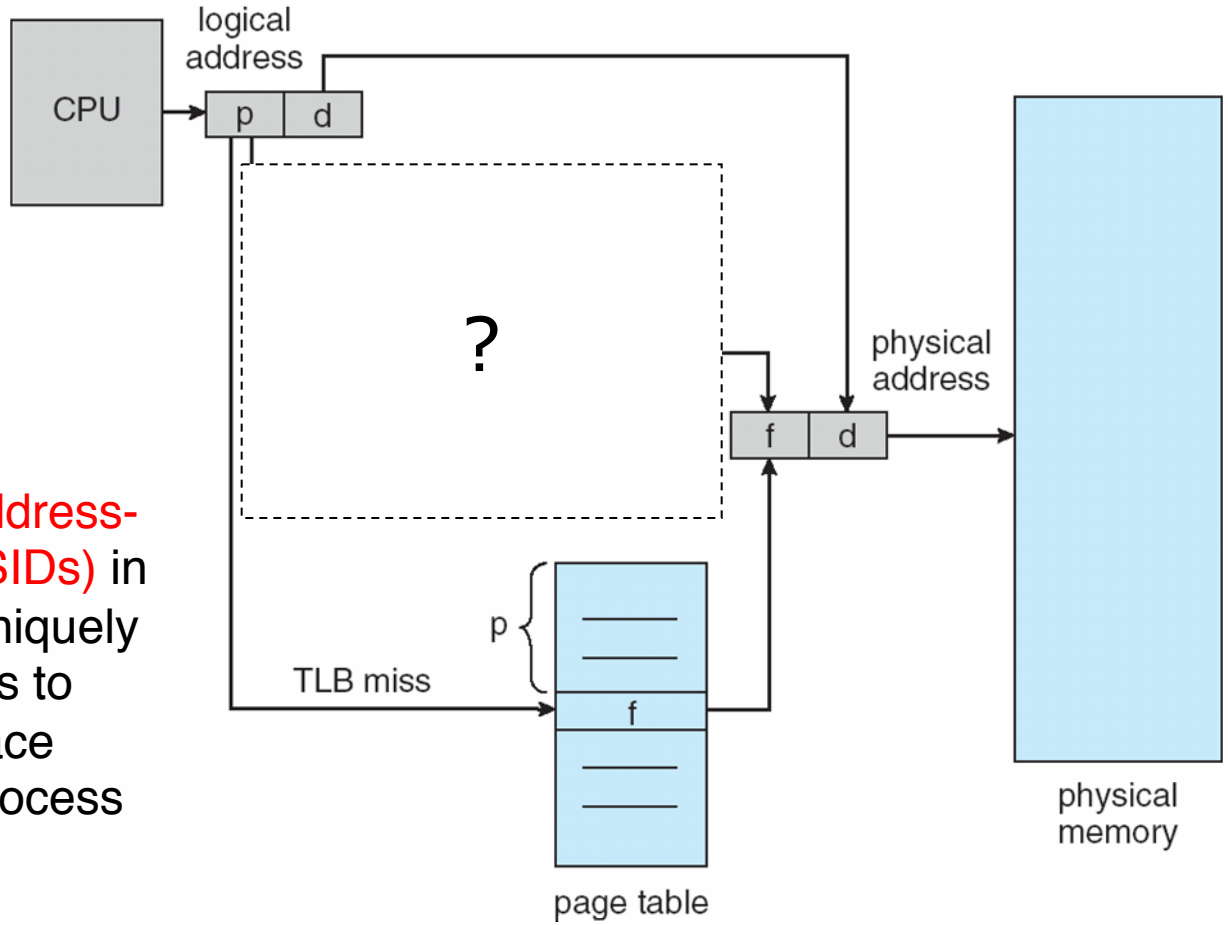
After allocation

Implementation of Page Table

- Each OS has its own methods for storing page tables
- The hardware implementation of the page table can be done is several ways:
  - In the simplest form, by using a set of dedicated registers
    - this is only feasible if the page table is reasonably small
  - Most contemporary computers allow the page table to be very large
    - so the page table is kept in main memory
    - Page-table base register (PTBR) points to the page table
    - Page-table length register (PRLR) indicates size of the page table
      - used for protection
    - In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
    - A solution to the above two memory access problem is to use a special fast-lookup hardware cache called translation look-aside buffers (TLBs) (also known as associative memory)

Paging Hardware With TLB

- TLB miss: the page number is not in the TLB
- TLB hit



- Some TLBs store address-space identifiers (ASIDs) in each TLB entry to uniquely identify each process to provide address-space protection for that process

Effective Access Time

- The percentage of times that a particular page number is found in the TLB is hit ratio
- As an example, an 80-percent hit ratio means that we find the desired page number in the TLB, 80 percent of the time
  - assuming a 20 ns to search the TLB and 100 ns to access memory, then the access takes 120 ns, when the page is in the TLB;
  - if we fail to find the page number in TLB (20 ns), then we must first access memory for the page table and frame number (100 ns) and then access the desired byte in memory (100 ns) for a total of 220 ns;
  - to find the effective memory-access time, we weight the case by its probability

$$\text{effective access time} = 0.8 \times 120 + 0.2 \times 220 = 140 \text{ ns}$$

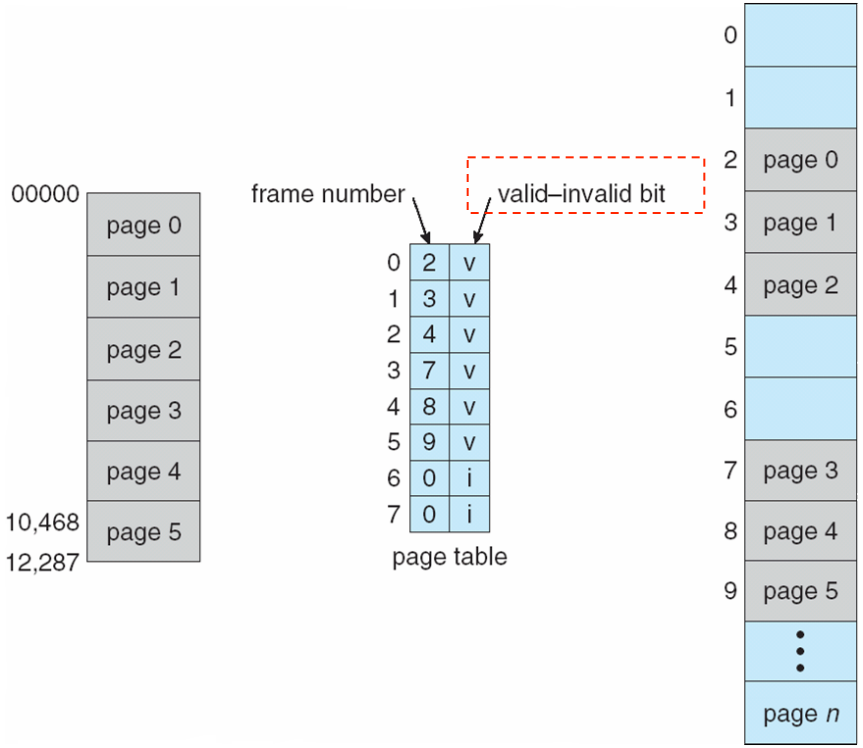
    - That is, with a 80 percent hit rate, we will see an access time of 140ns on average, compared with 120ns (with 100 percent hit rate) and 200ns (when no TLB is used).

Memory Protection

- In a paged environment, memory protection can be implemented by associating protection bits
  - one bit can define a page to be read-write or read-only
  - another bit, valid-invalid bit attached to each entry in the page table:

- "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page

- "invalid" indicates that the page is not in the process' logical address space
  - so illegal addresses can be trapped

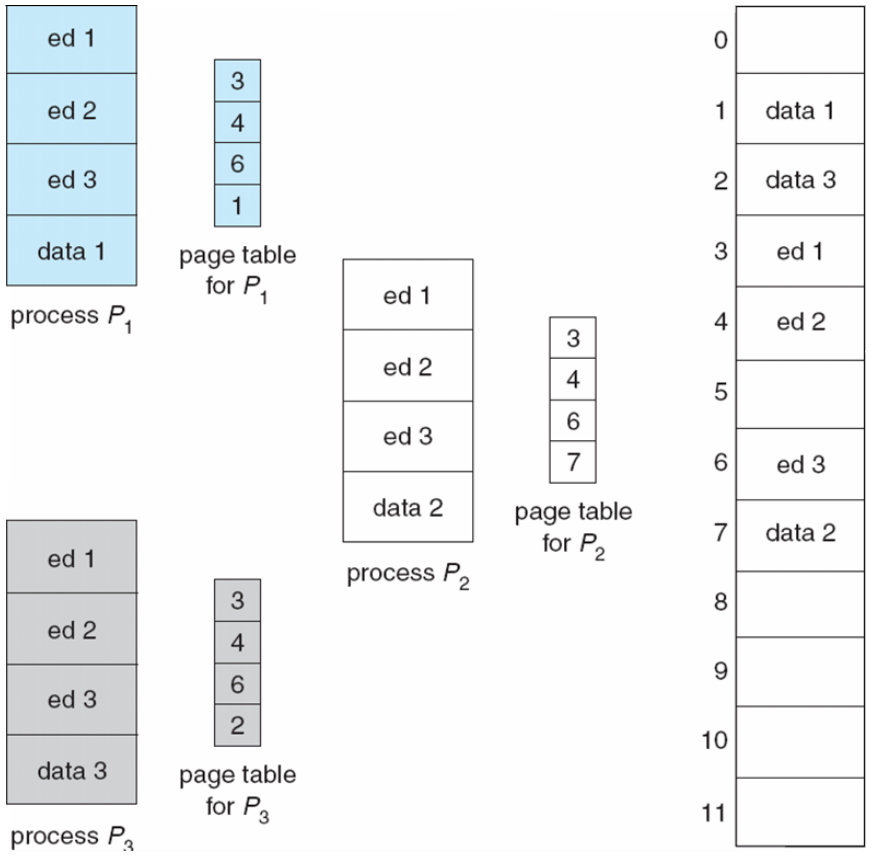


Shared Pages

- An advantage of paging is the possibility of sharing common code

- For the shared code
  - one copy of read-only (reentrant or pure) code is shared among processes (i.e., text editors, compilers, window systems)
  - shared code must appear in the same location in the logical address space of all processes

- For the private code and data
  - each process keeps a separate copy of the code and data
  - the pages for the private code and data can appear anywhere in the logical address space

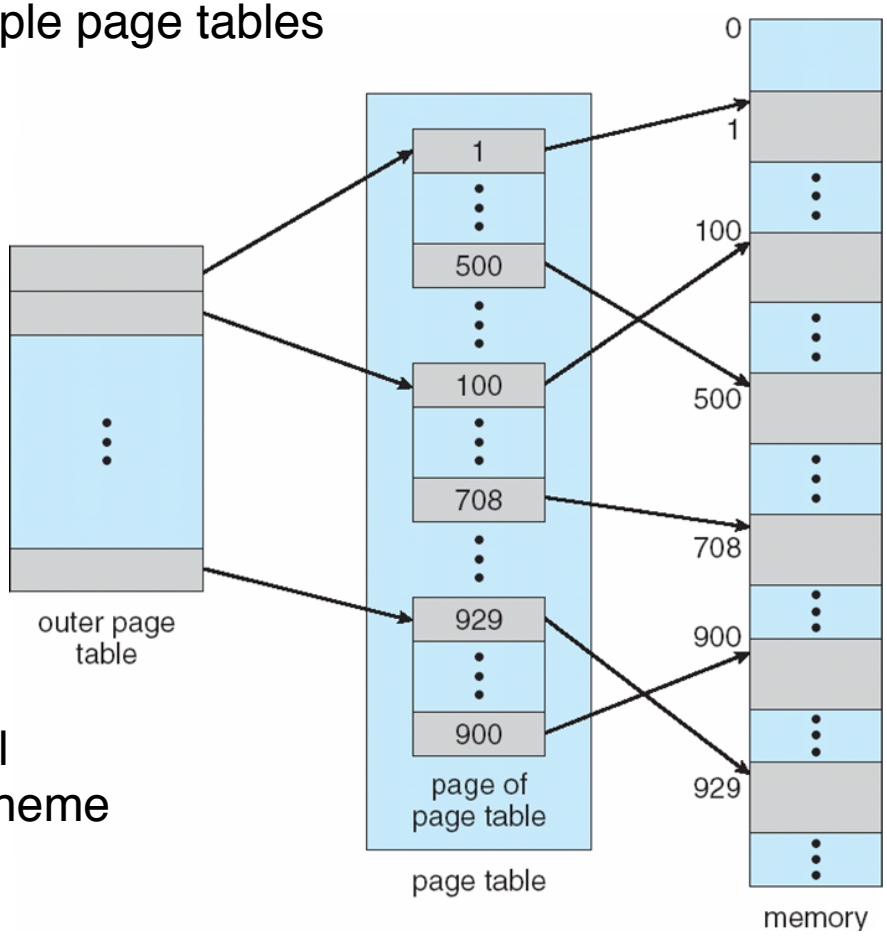


Structure of the Page Table

- There are three common techniques for structuring the page table
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables

Hierarchical Page Tables

- Most modern computer systems support a large logical address space (2^{32} and 2^{64}).
  - Since the page table itself becomes very large, the logical address space is broken up into multiple page tables
- A simple technique is a two-level page table:



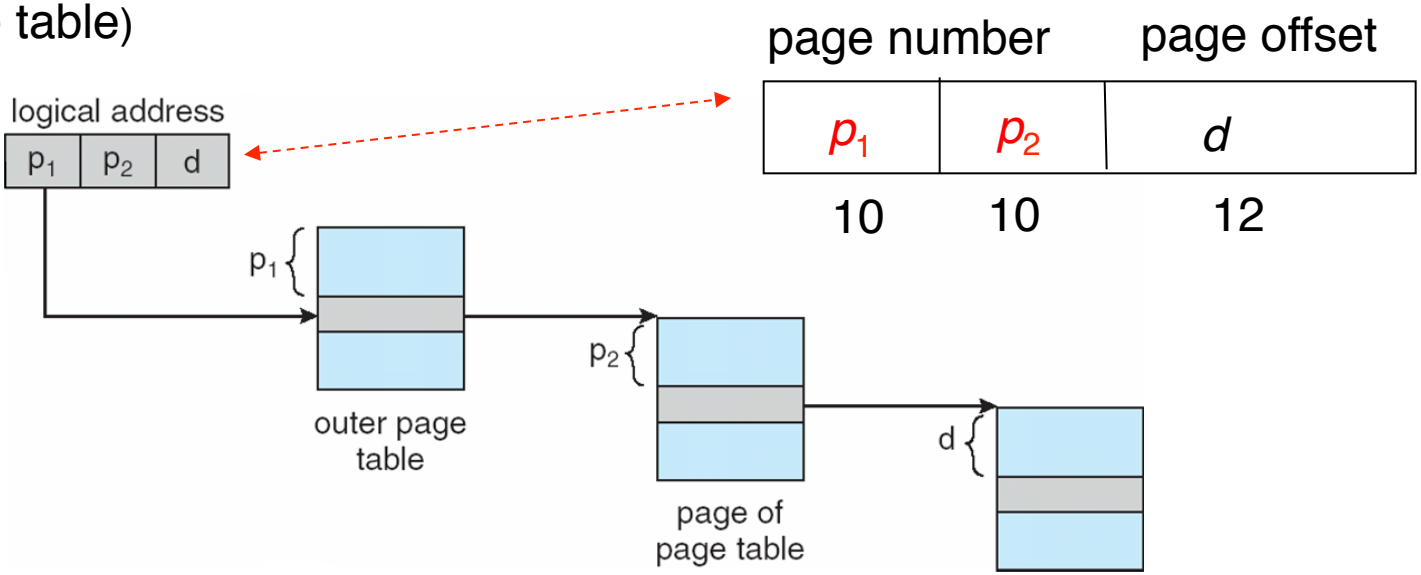
- For a system with a 64-bit logical address, a three-level paging scheme may be used

Two-Level Paging Example

- A logical address on a 32-bit machine with 4KB page size is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits

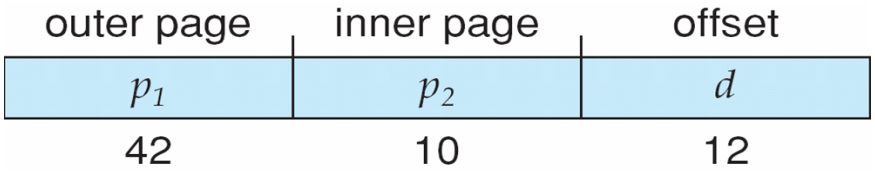
  $i.e.: \; 2^{32} / 2^{12} \approx 1 \text{ million enteries}$

  if each entry is $4B \Rightarrow 4$ MB for page table
- One simple solution to the problem is to divide the page table into smaller pieces. Since the page table is paged, the page number is further divided into:
  - a 10-bit page number (p_1, an index into the outer page table)
  - a 10-bit page offset (p_2, the displacement within the page of the outer page table)



Three-Level Paging Example

- For a 64-bit address space and two-level paging, the address would look like:

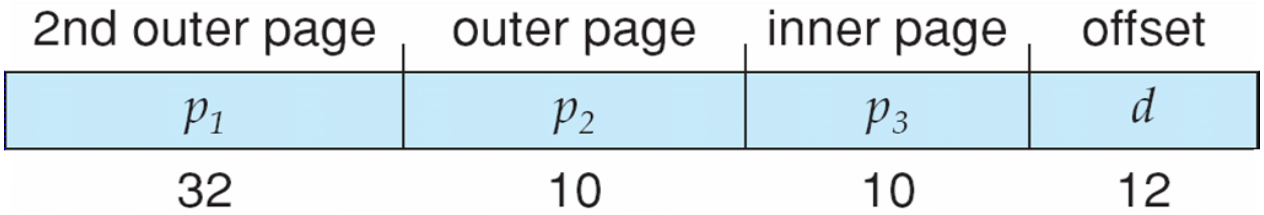


2^{42} outer page table enteries

4K page size (2^{12})

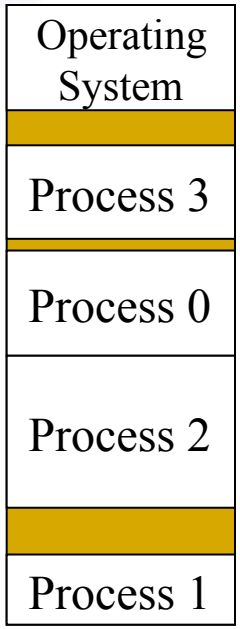
2^{10} inner page table enteries

- If we use three-level paging, then:



Lecture Outline

- Background
- Swapping
- Contiguous Memory Allocation
  - Fragmentation
- Paging
  - Structure of the Page Table
- **Segmentation**
- Examples: Intel and ARM processors

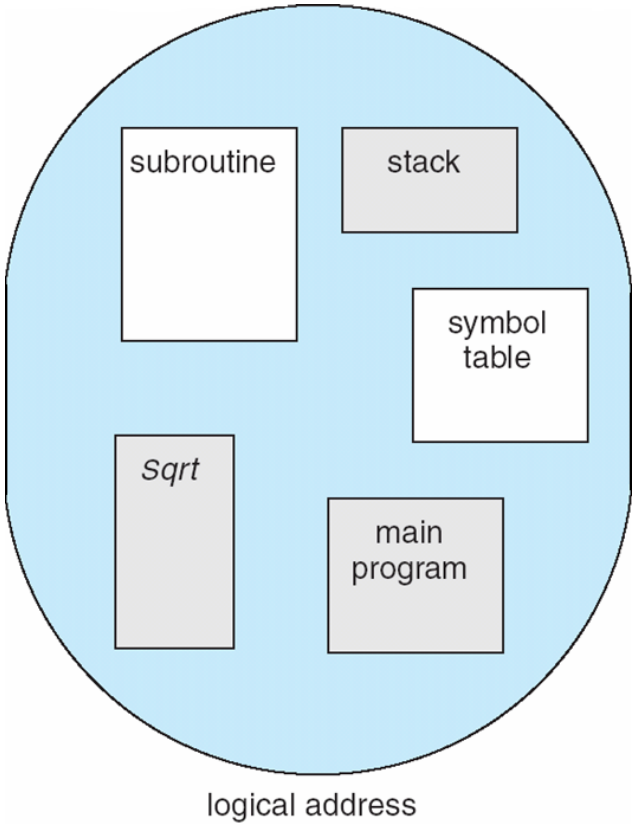


Segmentation

- Usually users prefer to view memory as a collection of variable-sized segments, not a linear array of bytes

- A program is a collection of segments
  - A segment is a logical unit
    - such as: main program, function, method, object, array, stack, …

- e.g., a C compiler may create separate segments for the following:
  - the code
  - global variables
  - the heap (from which memory is allocated)
  - stacks used by each thread
  - the standard C library



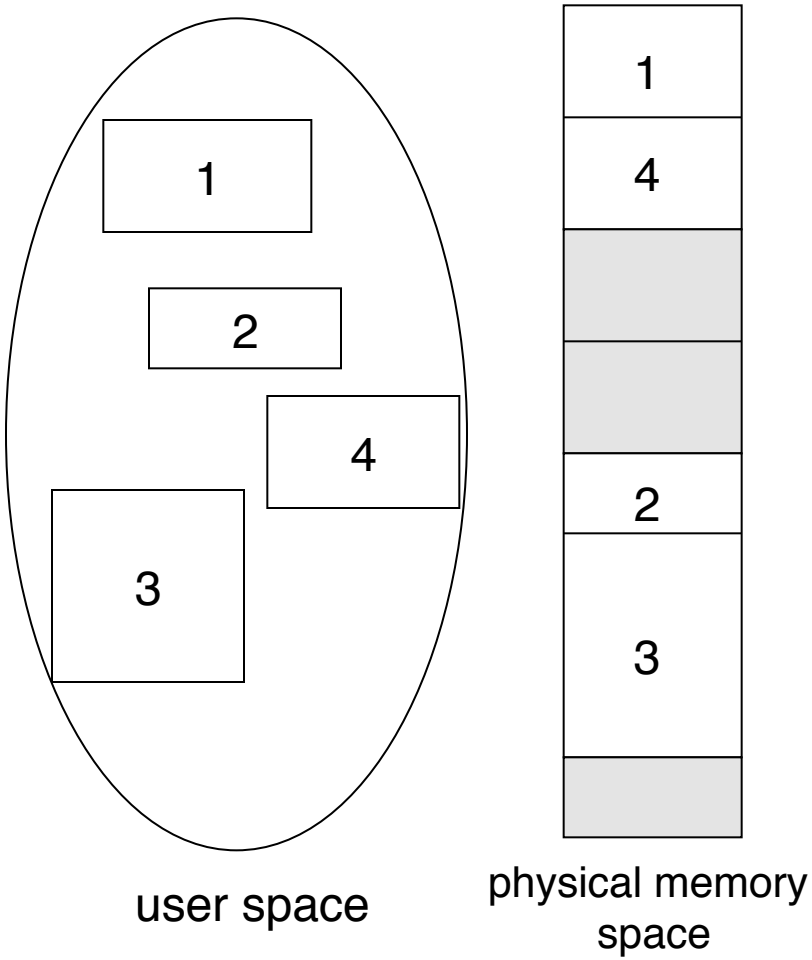
User's View of a Program

Logical View of Segmentation

- Segmentation is a memory-management scheme that supports this user view of memory

- A logical address space is a collection of segments
- Each segment has a name (a #) and a length

- The addresses specify both
  - the segment name
  - and the offset within the segment



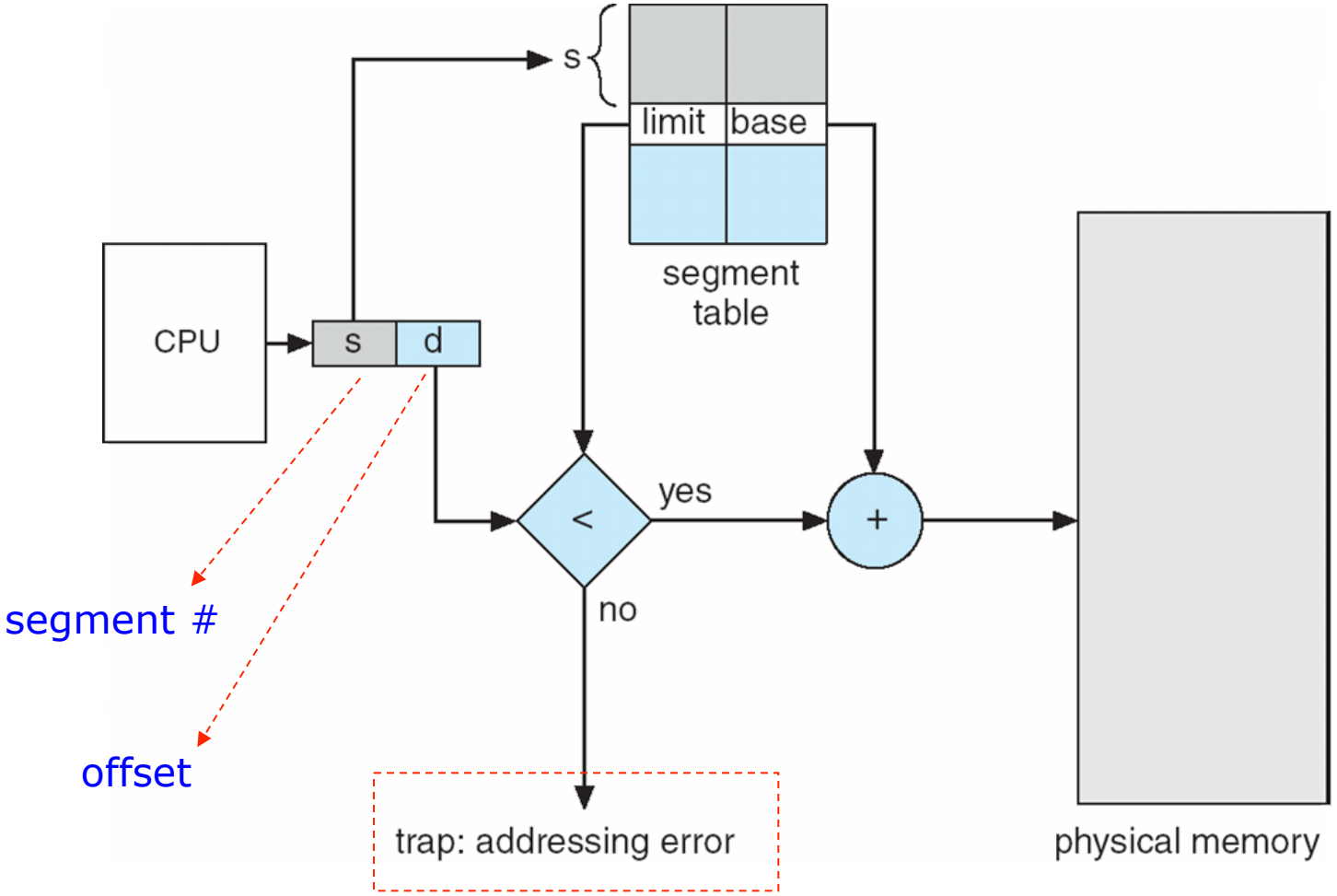
Segmentation Architecture

- A logical address consists of a two-tuple:

  <segment-number, offset>

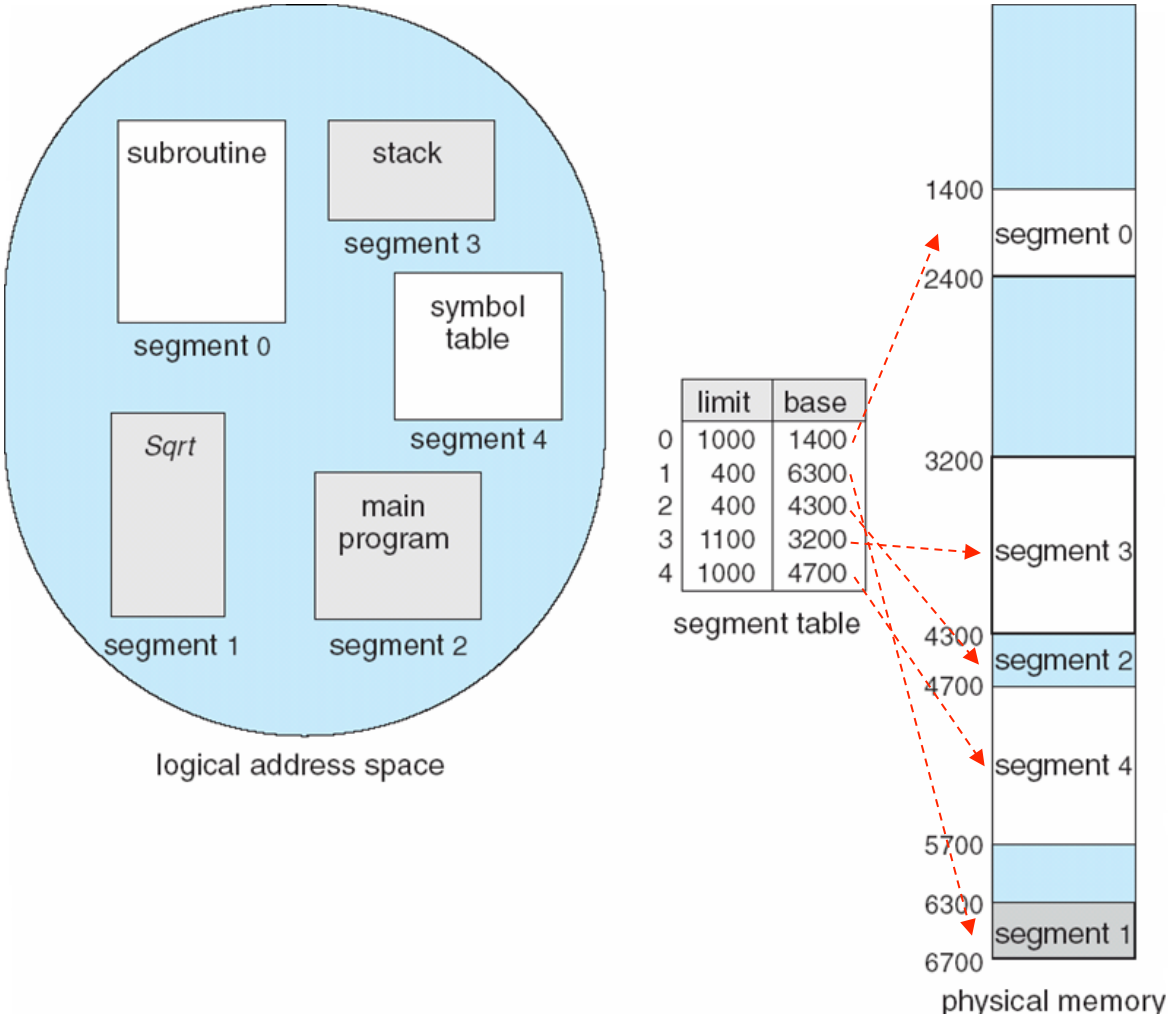
- Although the user now refer to objects in the program by a two-dimensional address, the actual physical memory is of course one-dimensional,
  - so memory map is needed
  - Segment table is used to map the two-dimensional physical addresses; each entry in the table has:
    - a base segment contains the starting physical address where the segments reside in memory
    - a limit segment specifies the length of the segment

Segmentation Hardware



Example of Segmentation

- Consider an example in which we have five segments numbered from 0 to 4
- The segments are stored in memory as shown:



= See animation

Lecture Outline

- Background
- Swapping
- Contiguous Memory Allocation
  - Fragmentation
- Paging
  - Structure of the Page Table
- Segmentation