OPERATING SYSTEMS

Memory Management

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Main Memory:

Hardware and control structures, OS support, Address translation

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

Background

- What is a memory?
  - Memory consists of a large array of bytes, each with its own address.
- Execution of an instruction,
  - Fetch an Instruction from memory, Decode the instruction, operands are fetched(from memory or registers)
  - After the instruction is executed, results are stored back.
- The memory unit(MU) sees the stream of addresses.
- Memory unit does not know how these addresses are generated.
- We will learn how the addresses are generated by the running program.

OPERATING SYSTEMS

Basic Hardware

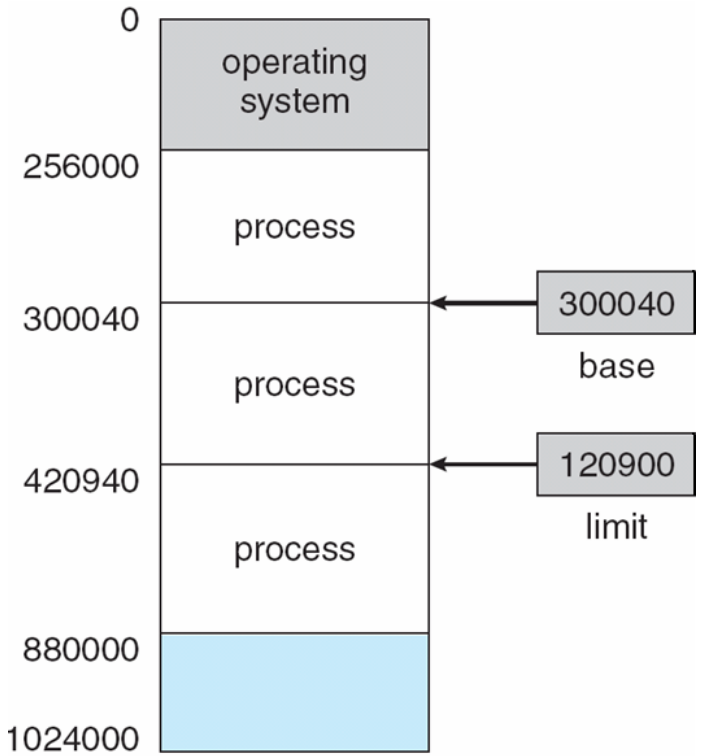
- Program must be brought (from disk) into memory and placed within a ready queue it to be run
- Main memory and registers are only storage CPU can access directly
- The data required for CPU must be made available in the registers.
  - Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
  - Speeds up memory access without any OS control
- Protection of memory required to ensure correct operation

Basic Hardware

- Protection of OS from its access by user processes is needed
- On multiuser systems user processes must be protected from each other.
- This protection must be provided by the hardware because the operating system doesn’t usually intervene between the CPU and its memory accesses.
- Several hardware protection methods will be discussed.
- Protection by using two registers, usually a base and a limit.

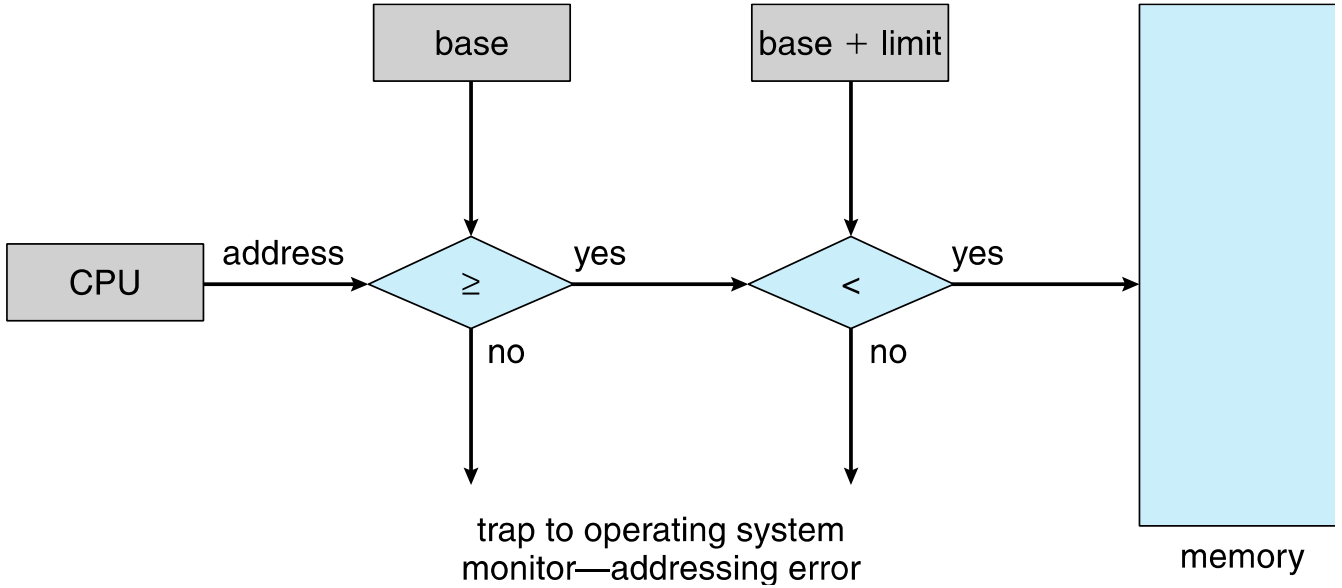
Basic Hardware

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



- prevents a user program from modifying the code or data structures of either the operating system or other users.

Basic Hardware

- The base and limit registers can be loaded only by the operating system.
  - Using special privileged instruction.
  - privileged instructions can be executed only in kernel mode.
  - OS runs in the kernel mode. Only OS can change the register values.
  - This prevents other user programs to modify the register values.

OPERATING SYSTEMS

Address Binding

- Programs on disk as binary executable and are brought to main memory for execution
- Processes may be moved between the disk and memory during execution.
- What are the steps involved in the program execution?
- Most systems allow a user process to reside in any part of the physical memory.
- The address space of the computer may start at 00000 but the first address of the user process need not be 00000
- Addresses in the source program are generally symbolic (Ex. variable count).

Address Binding

- A compiler typically binds these symbolic addresses to relocatable addresses
  - “14 bytes from the beginning of this module”
- The linkage editor or loader in turn binds the relocatable addresses to absolute addresses
  - Ex. 74014
- Each binding is a mapping from one address space to another.

Memory-Management Unit (Cont.)

Base-register scheme for address mapping

- The base register also referred to as a relocation register.
- The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory .
- Example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000;
- An access to location 346 is mapped to location 14346.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

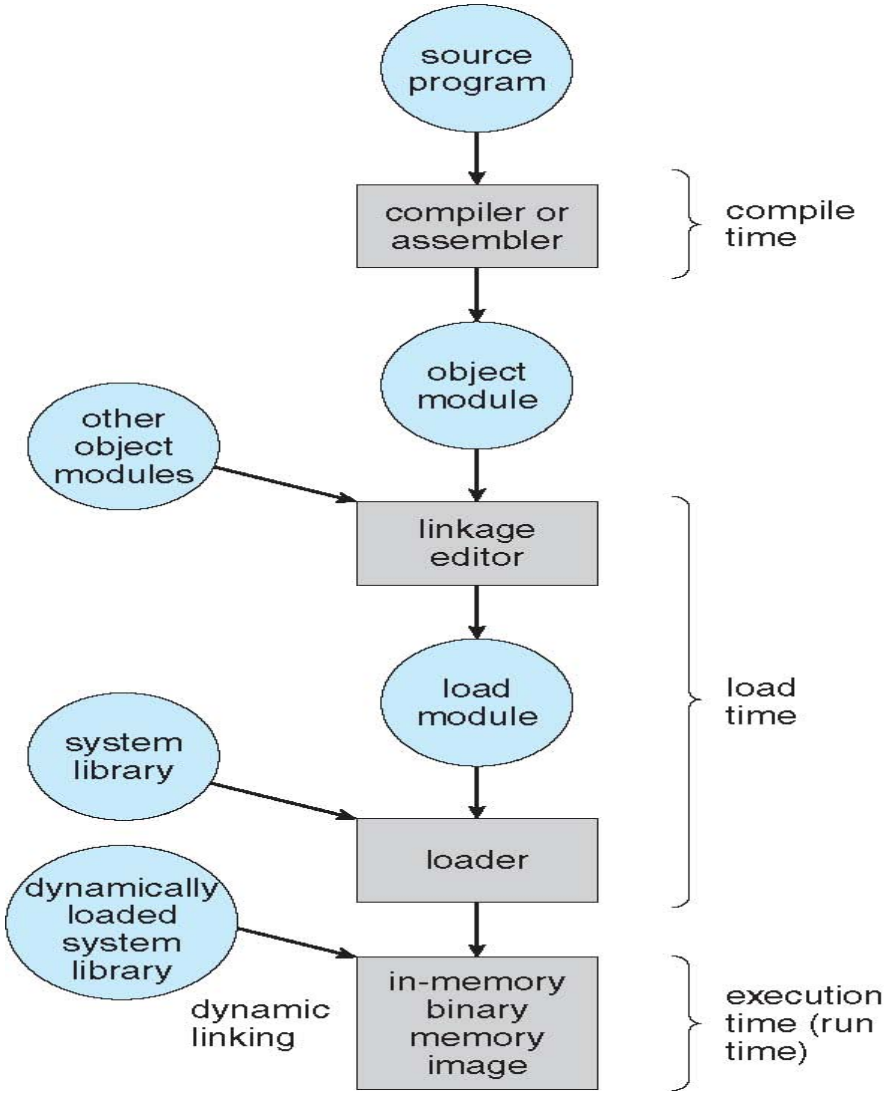
Memory-Management Unit (Cont.)

The binding of instructions and data to memory addresses

- **Compile time**. If you know at compile time where the process will reside in memory, then **absolute code** can be generated.
  - If the start address changes, it is necessary to recompile once again
  - The MS-DOS .COM-format programs are bound at compile time.
- **Load time**. If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code**.
  - final binding is delayed until load time.
  - If the starting address changes, we need only reload the user code to incorporate this changed value.
- **Execution time**. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.
  - Special hardware must be available for this scheme to work

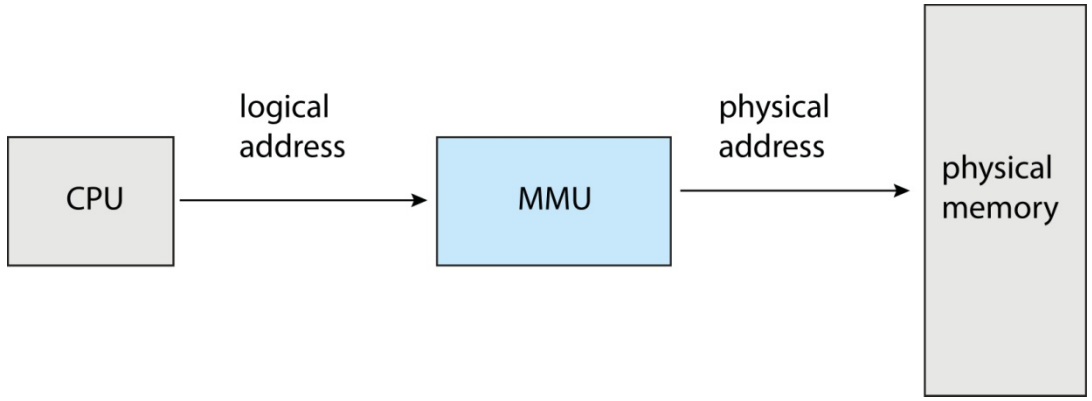
OPERATING SYSTEMS

Multistep Processing of a User Program



Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address



- Many methods possible to accomplish this mapping, will be discussed in the next few lectures.

OPERATING SYSTEMS

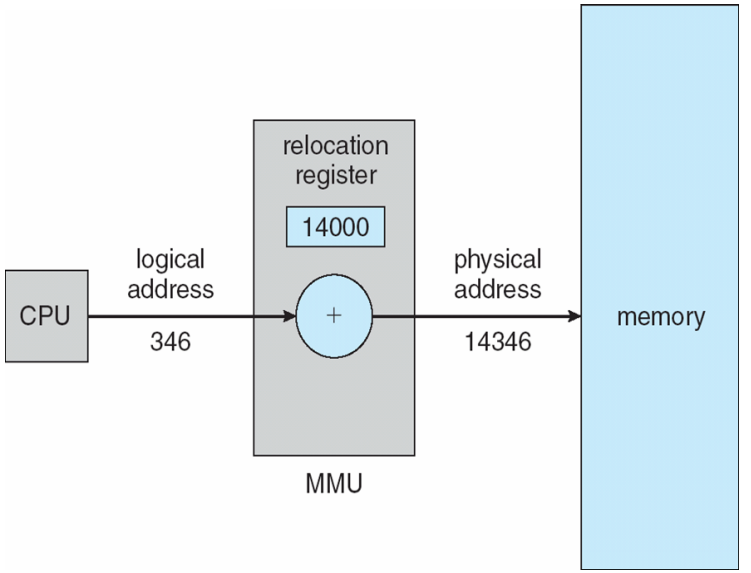
Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

OPERATING SYSTEMS

Dynamic relocation using a relocation register

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading



OPERATING SYSTEMS

Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- **Dynamic linking** –linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- Other programs linked before the new library was installed will continue using the older library
- This System also known as **shared libraries**
- Consider applicability to patching system libraries

Static and Dynamic Linking

- A program whose necessary library functions are embedded directly in the program's executable binary file is ***statically*** linked to its libraries
- The main disadvantage of static linkage is that every program generated must contain copies of exactly the same common system library functions
- *Dynamic* linking is more efficient in terms of both physical memory and disk-space usage because it loads the system libraries into memory only once

Demo

- $cc fork.c
- size a.out
- $cc -static fork.c
- size a.out

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Memory Management

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Swapping, Memory Allocation , Fragmentation

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

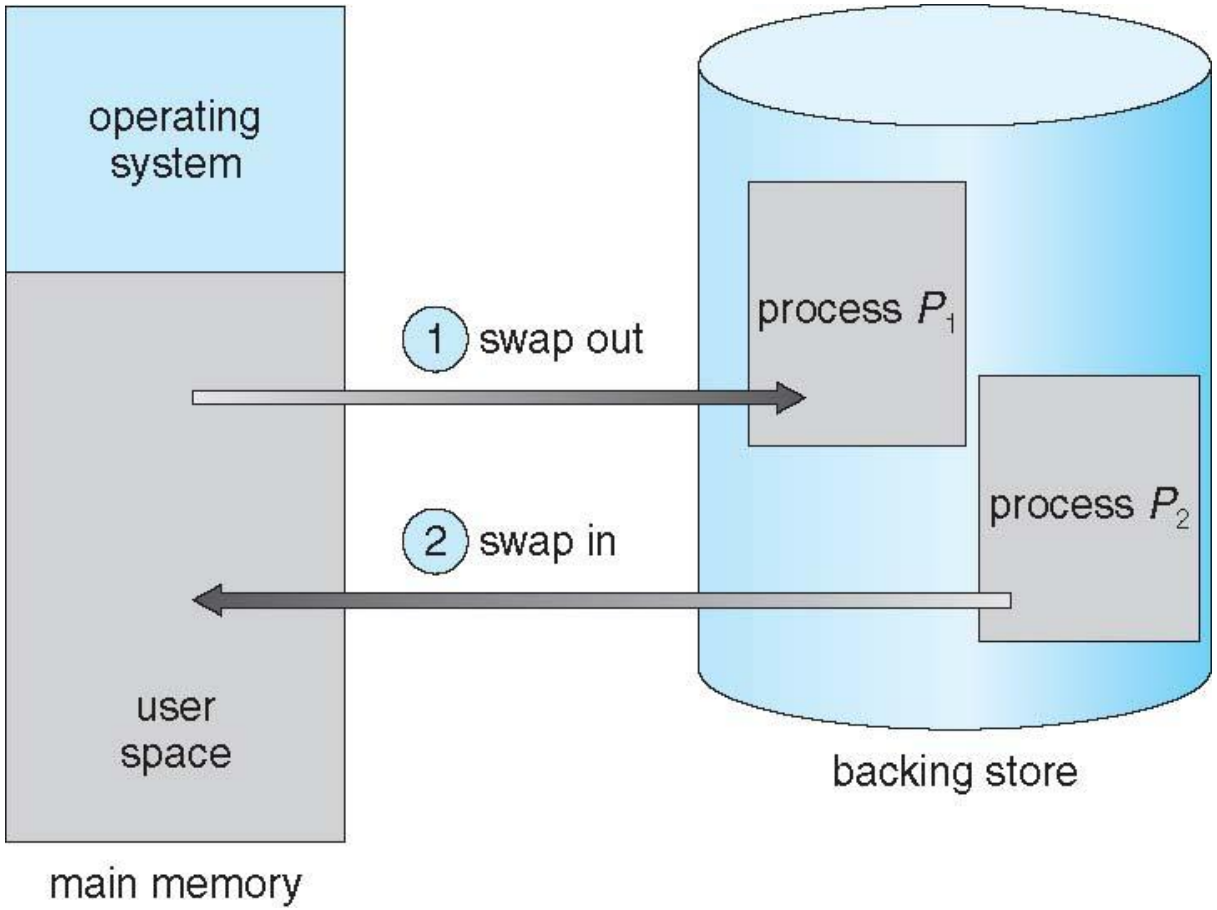
OPERATING SYSTEMS

Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

OPERATING SYSTEMS

Schematic View of Swapping



OPERATING SYSTEMS

Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- Let us consider a user process with size 100MB
- hard disk transfer rate of 50MB/sec
  - Swap out time=100MB/50MB per sec= 2 sec
  - Swap in in time is same as swap out time
  - Total context switch swapping component time = 4 seconds =swap in time + swap out time
- What will be the swap time for process with size 3GB? – 60 secs
- Is it right to swap complete process?
- Can be reduced if size of memory to be swapped is reduced – by knowing how much memory really being used
  - System calls to inform OS of memory use via request_memory() and release_memory()

Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to process memory space
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - But modified version common
    - Swap only when free memory extremely low

Swapping on Mobile Systems

- Not typically supported
  - Flash memory is used rather than hard disks
    - Storage is a constraint for swapping
    - Limited number of writes are tolerated by flash memory
    - Poor throughput between flash memory and CPU on mobile platform
- Instead of swapping, other methods to free memory if memory is low
  - ***iOS*** asks apps to voluntarily relinquish allocated memory
  - Read-only data are removed system and are loaded if needed
  - Data that have been modified (such as the stack) are never removed.
  - Applications that fail to free up sufficient memory may be terminated by the operating system.

Swapping on Mobile Systems

Android OS

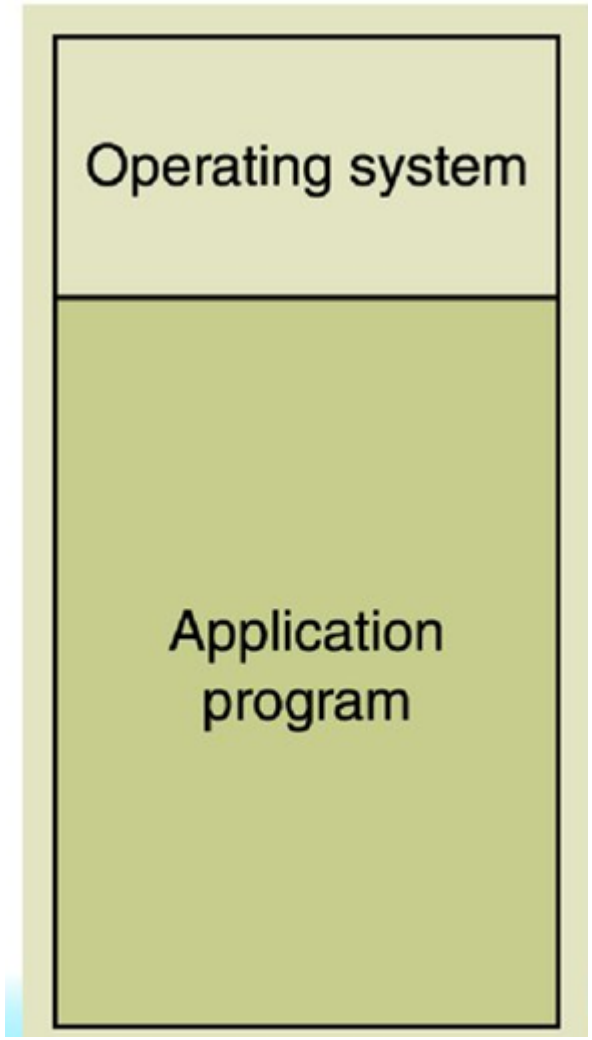
- Does not support swapping
- ***Android*** terminates apps if low free memory, but first writes **application state** to flash for fast restart
- Both OSes support paging which will be discussed later

Contiguous Allocation

- Main memory must accommodate both OS and user processes
- Memory needs to be allocated efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single section of memory that is contiguous to the section containing the next process.

OPERATING SYSTEMS

Contiguous Allocation



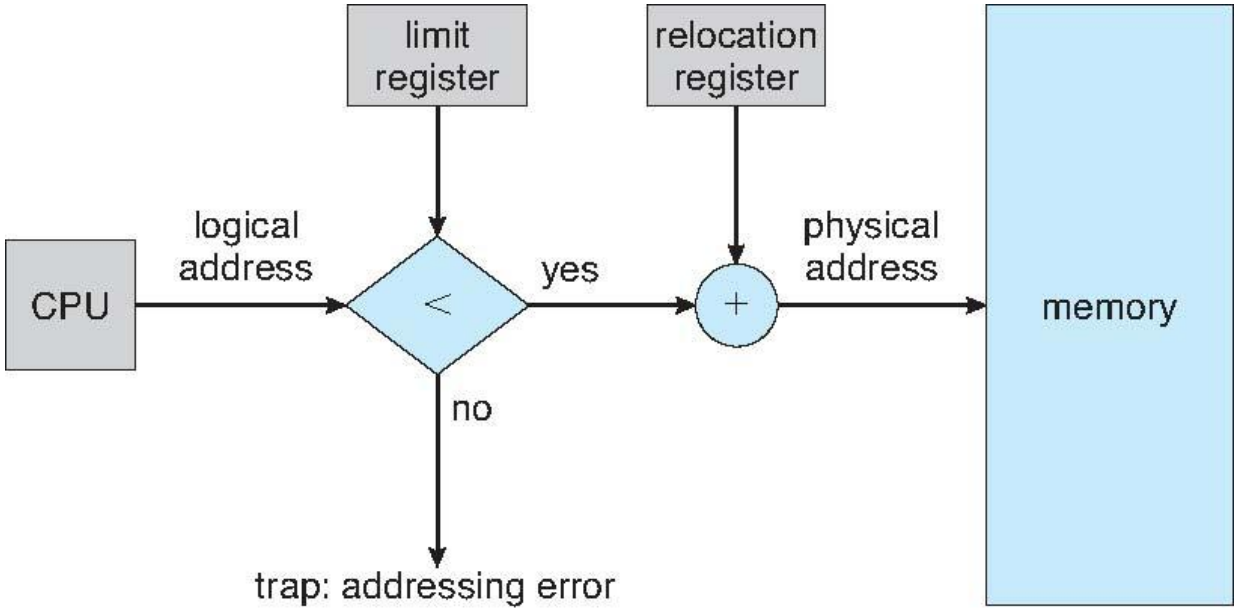
Lower address

higher address

Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest **physical** address
  - Limit register contains range of **logical** addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** (it comes and goes as needed) and kernel/OS changing size during program execution.

Hardware Support for Relocation and Limit Registers



When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch.

Memory Protection

- Every address generated by a CPU is checked against these registers, so possible to protect both the operating system and the other users' programs and data from being modified by this running process .
- The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically.
  - This flexibility is desirable in many situations. For example, the operating system contains **code** and **buffer space** for **device drivers**. If a device driver is not currently in use, it makes little sense to keep it in memory; instead, it can be loaded into memory only when it is needed. Likewise, when the device driver is no longer needed, it can be removed and its memory allocated for other needs.

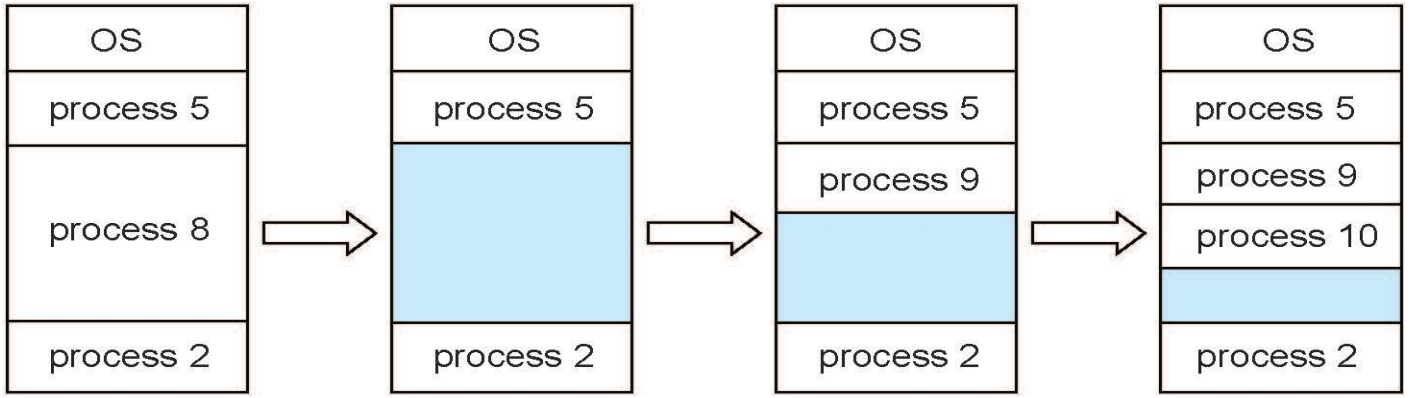
Multiple-partition allocation

- Multiple-partition allocation
  - Divide memory into several **fixed-size partitions**.
    - Each partition may contain exactly one process
    - Degree of multiprogramming is bounded by number of partitions.
    - when a partition is free, a process is selected from the input queue and is loaded into the free partition
    - used by the IBM OS/360 operating system,but is no longer in use
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
    - This scheme keeps a table indicating which parts of memory are available/occupied
  - **Hole** – block of available memory; holes of various size are scattered throughout memory

OPERATING SYSTEMS

Multiple-partition allocation (Cont.)

- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
  a) allocated partitions b) free partitions (hole)
- Considering the size of process as,
- Process 5=100MB,Processes 8=400MB, Process 2=200MB,Process 9=200MB, process 10=100MB, can the process 11=200MB be accommodated after process 5 terminates?



Partitions

- **Advantages**

1. No Internal Fragmentation
2. No restriction on Degree of Multiprogramming
3. No Limitation on the size of the process

- **Disadvantages**
  - Causes External Fragmentation
  - Difficult to implement

Dynamic Storage-Allocation Problem and Strategies

How to satisfy a request of size ***n*** from a list of free holes?

- **First-fit**: Allocate the ***first*** hole that is big enough (generally faster)
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

OPERATING SYSTEMS

Allocations strategies: Examples

Example: Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)? Rank the algorithms in terms of how efficiently they use memory.

Soln:

Partitions	First-Fit
M1=300 KB	P1
185KB	
M2=600 KB	P2
100KB	
M3=350 KB	P4
150KB	
M4=200 KB	
M5=750 KB	P3
392KB	P5
17KB	
M6=125 KB	

Partitions	Best Fit
M1=300 KB	
M2=600 KB	P2
100KB	
M3=350 KB	
M4=200 KB	P4
M5=750 KB	P3
392KB	P5
17KB	
M6=125 KB	P1
10KB	

Partitions	Worst-Fit
M1=300 KB	
M2=600 KB	P3
242KB	
M3=350 KB	P4
150KB	
M4=200 KB	
M5=750 KB	P1
635KB	P2
135KB	
M6=125 KB	

P5 must wait

Processes and their sizes

P1=115 KB P2=500 KB

P3=358 KB P4=200 KB

P5=375 KB

Fragmentation

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
  - First fit statistical analysis reveals that given *N* allocated blocks (66% for example), another 0.5 *N* blocks (33% for example) lost to fragmentation
  - 1/3 of memory may be unusable -> **50-percent rule**
    - Unusable memory = (0.5N)/(N+0.5N)= 1/3

Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Memory Management

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Segmentation

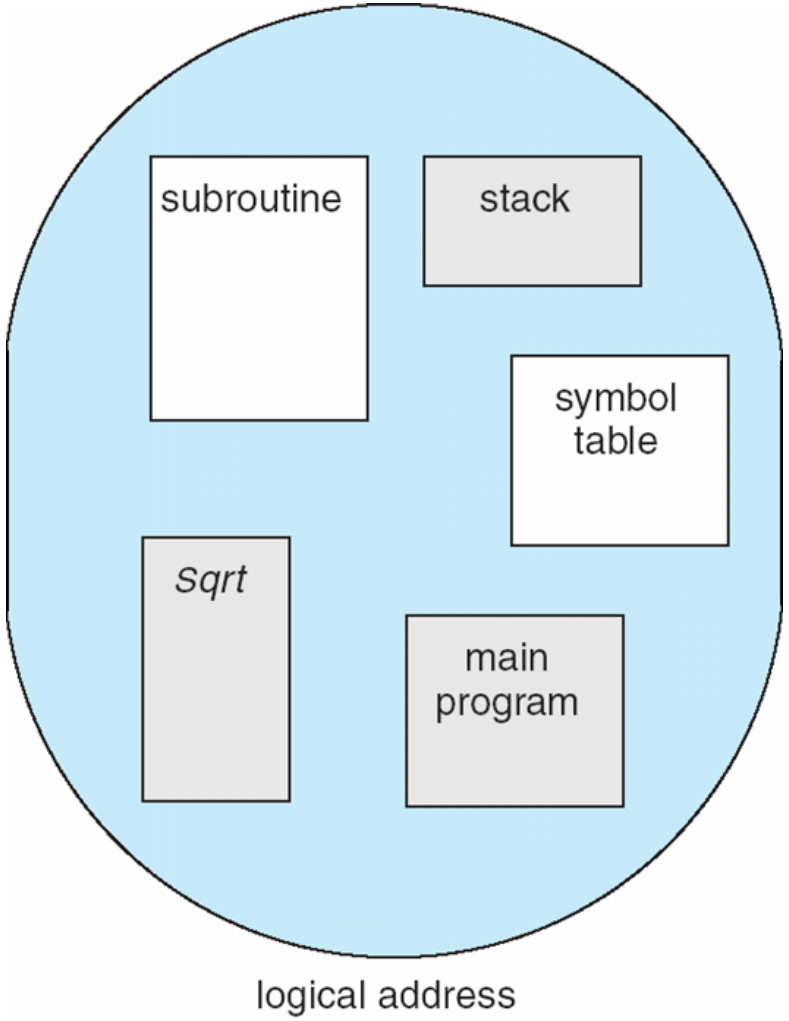
Suresh Jamadagni
Department of Computer Science

Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:
    - main program
    - procedure
    - function
    - method
    - object
    - local variables, global variables
    - common block
    - stack
    - symbol table
    - arrays

OPERATING SYSTEMS

User's View of a Program

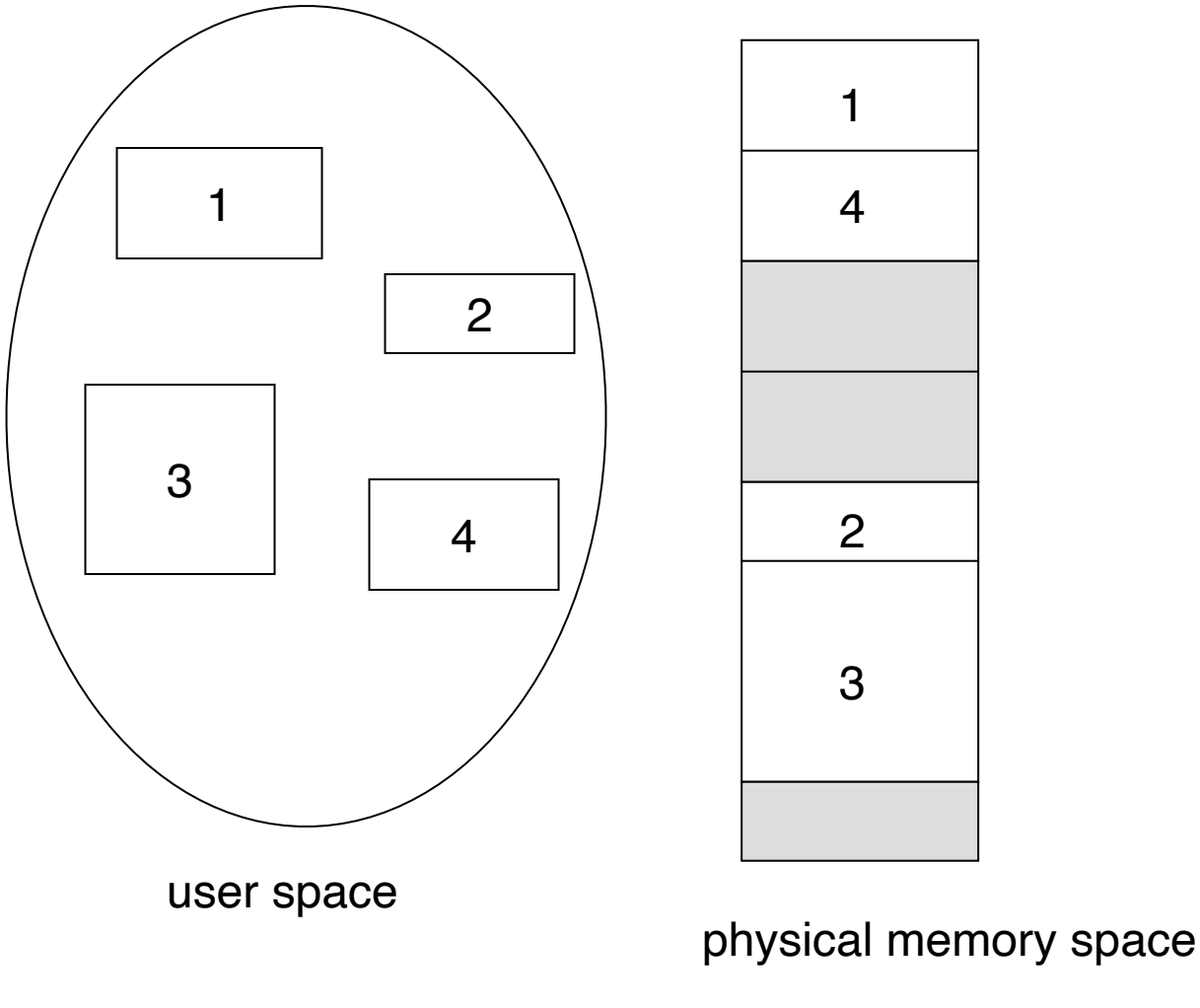


➢Segmentation is a memory-management scheme that supports this programmer view of memory

➢Each segment has a name and a length.

➢Addresses specify segment name + offset within the segment

Logical View of Segmentation



Segmentation Basics

- Segmentation is a technique for breaking memory up into logical pieces
- Each “piece” is a grouping of related information
  - data segments for each process
  - code segments for each process
  - data segments for the OS
  - etc.
- Segmentation permits the physical address space of a process to be non-contiguous.
- Like paging, use virtual addresses and use disk to make memory look bigger than it really is
- Segmentation can be implemented with or without paging

OPERATING SYSTEMS

Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number s, offset d>,

- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **Segment base** – contains the starting physical address where the segments reside in memory
  - **Segment limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**

Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem

OPERATING SYSTEMS

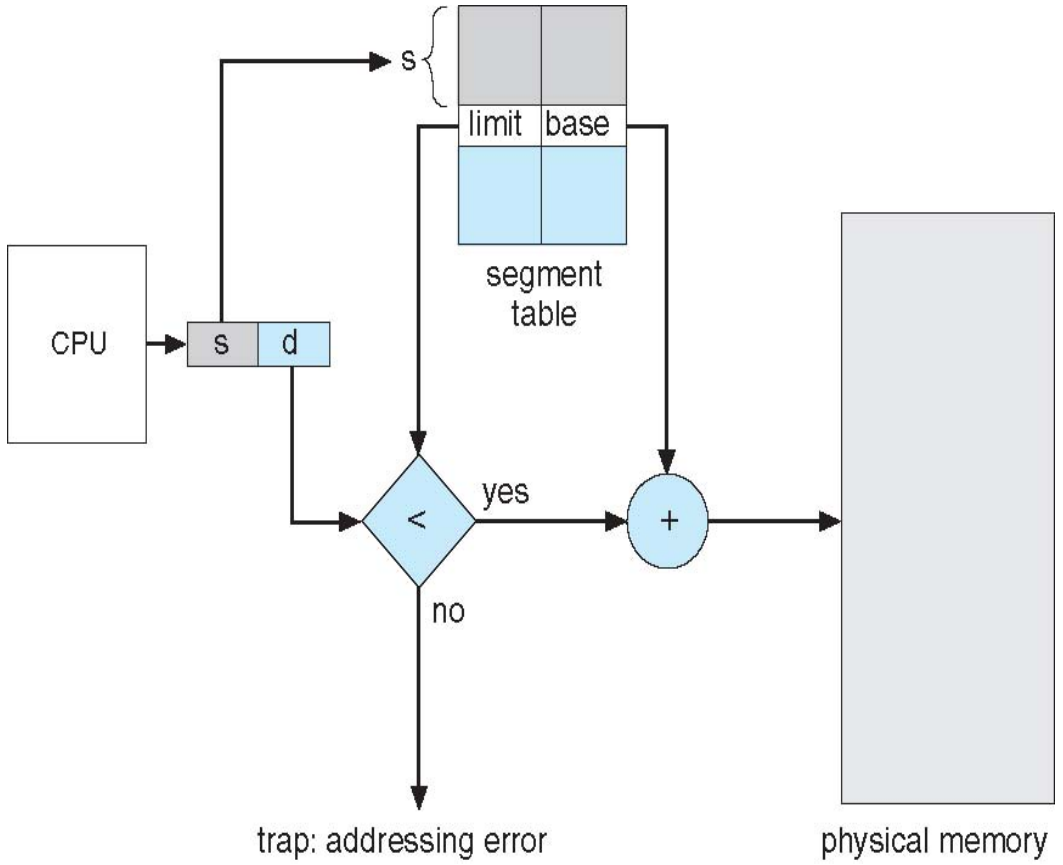
Segmentation Hardware

- Sounds very similar to paging
- Big difference – segments can be variable in size
- As with paging, to be effective hardware must be used to translate logical address
- Most systems provide segment registers
- If a reference isn't found in one of the segment registers
  - trap to operating system
  - OS does lookup in segment table and loads new segment descriptor into the register
  - return control to the user and resume

OPERATING SYSTEMS

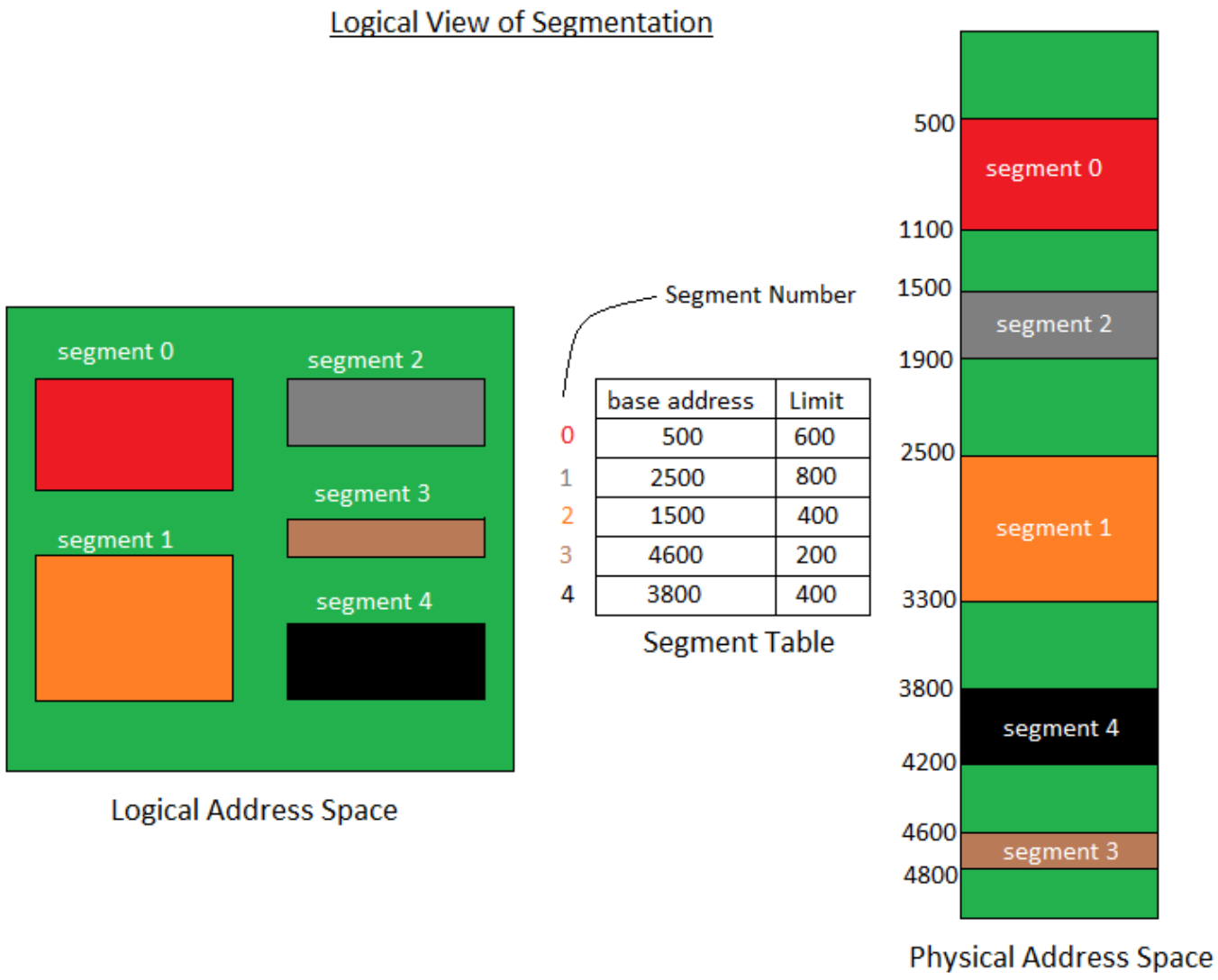
Segmentation Hardware (Cont.)

- A logical address consists of two parts: a segment number, s, and an offset into that segment, d.
- The segment number is used as an index to the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment).
- When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base–limit register pairs.



OPERATING SYSTEMS

Segmentation Example



Segmentation Example (Cont.)

- We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown.
- The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit).
- For example, segment 2 is 400 bytes long and begins at location 1500.
  - Thus, a reference to byte 53 of segment 2 is mapped onto location 1500 + 53 = 1553.
- A reference to segment 3, byte 852, is mapped to 4600 (the base of segment 3) + 852 = 5452.
- A reference to byte 722 of segment 0 would result in a trap to the operating system, as this segment is only 600 bytes long.

Segmentation Comments

- In each segment table entry, we have both the starting address and length of the segment; the segment can thus dynamically grow or shrink as needed.
- But variable length segments introduce external fragmentation and are more difficult to swap in and out.
- It is natural to provide protection and sharing at the segment level since segments are visible to the programmer (pages are not).
- Useful protection bits in segment table entry:
  - read-only/read-write bit
  - Kernel/User bit

Segmentation Issues

- Entire segment is either in memory or on disk
- Variable sized segments leads to external fragmentation in memory
- Must find a space big enough to place segment into
- May need to swap out some segments to bring a new segment in.

Segmentation: Examples

- Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

a. 0,430

b. 1,10

c. 2,500

d. 3,400

e. 4,112

Segmentation: Examples

❑ Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

a. 0,430
b. 1,10
c. 2,500
d. 3,400
e. 4,112

Solution:

a. 0, 430 →219 + 430 = 649
b. 1, 10 →2300 + 10 = 2310
c. 2, 500 Illegal address since size of segment 2 is 100 and the offset in logical address is 500.
d. 3, 400 →1327 + 400 = 1727
e. 4, 112 Illegal address since size of segment 4 is 96 and the offset in logical address is 112.

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Memory Management

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Paging

Suresh Jamadagni
Department of Computer Science

Paging

- Segmentation permits Physical address space of a process can be **noncontiguous**
- Paging is another memory-management scheme that offers this advantage.
  - **Avoids external fragmentation** and need for compaction
  - Solves the problem of fitting memory chunks of varying sizes onto the backing store.
- The backing store also has the fragmentation problems

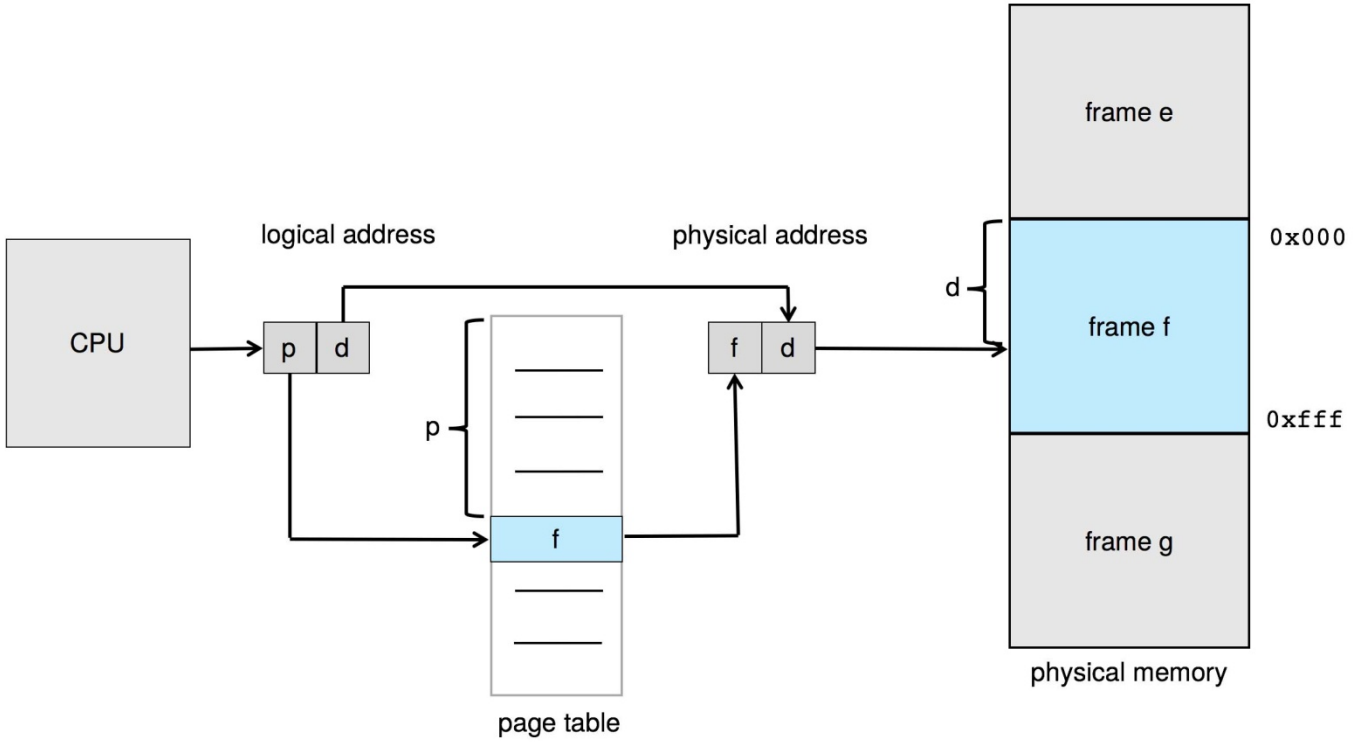
Paging: Basic Method

- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- **Page size and frame size are defined by the hardware**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation
- Following command displays the page size supported in the system,

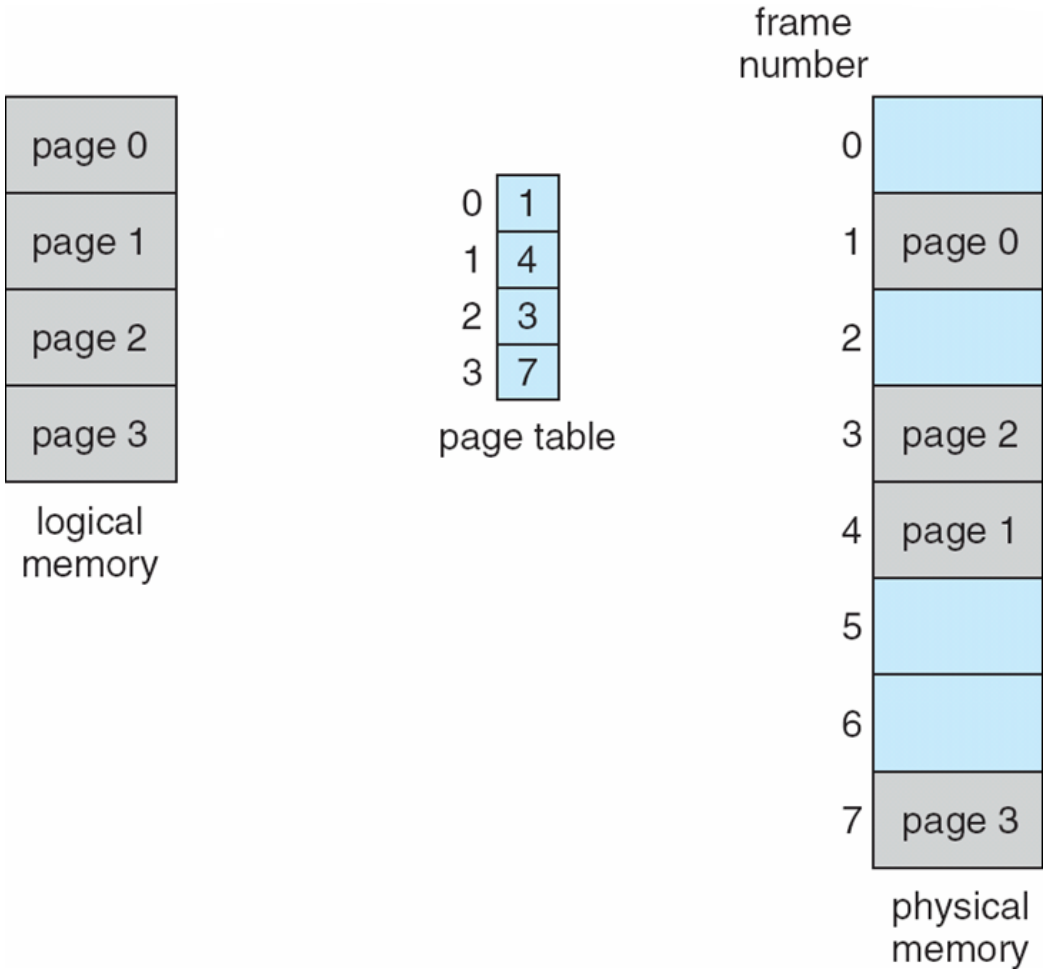
```
getconf   PAGESIZE
```

OPERATING SYSTEMS

Paging Hardware



Paging Model of Logical and Physical Memory



OPERATING SYSTEMS

Address Translation Scheme

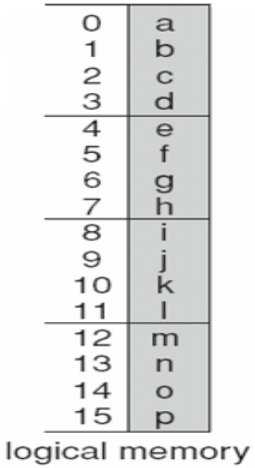
- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit

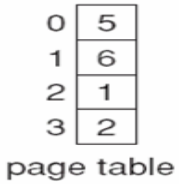
  - Page size defined by hardware.
  - Page size varies between 512 bytes and 1 GB per page
  - The logical address space is 2^m and page size 2^n
  - The high-order m − n bits of a logical address designate the page number, and the n low-order bits designate the page offset

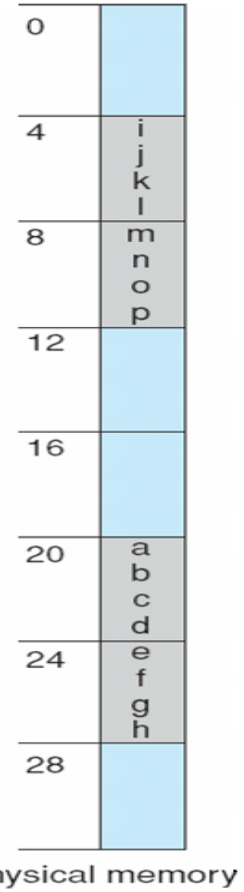
OPERATING SYSTEMS

Paging Example

- Logical address: n = 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)
- Logical address space = 2^m
- Page size = 2^n bytes







- No external fragmentation, we may have internal fragmentation

OPERATING SYSTEMS

Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - **Internal fragmentation of 2,048 - 1,086 = 962 bytes**
  - Worst case fragmentation = **1 frame – 1 byte**
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
- Programmer's view and physical memory now very different
- By implementation the user process can only access its own memory

Paging (Cont.)

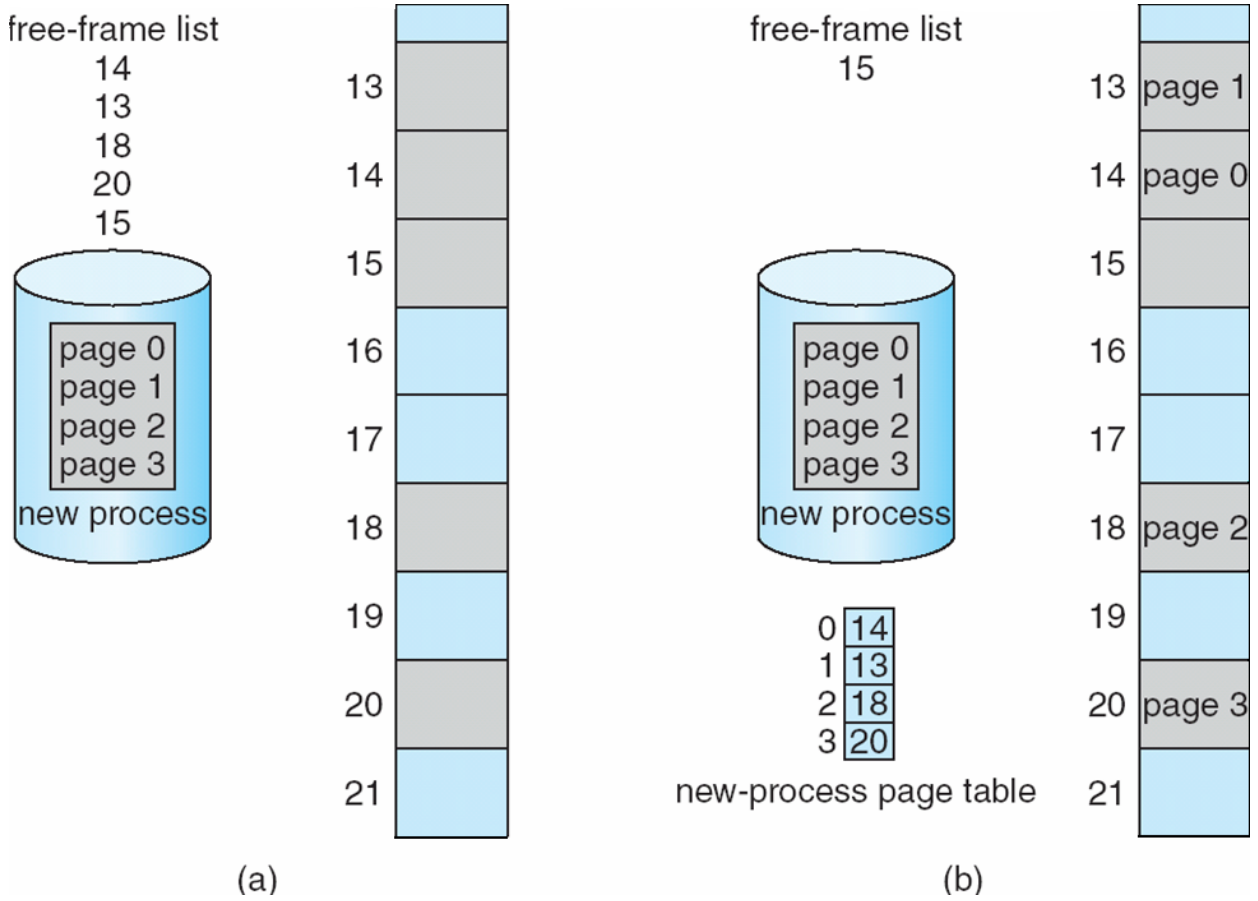
- Worst case, a process would need *n* pages plus 1 byte.
  - needs to be allocated *n* + 1 frames, resulting in internal fragmentation of almost an entire frame.
- If the page size is smaller, an overhead is involved with each page table entry
- The overhead can be reduced by increasing the page size
- Page size are 4KB, 8KB
- Some CPU and kernel support multiple page sizes.
- Researchers are now developing support for variable on-the-fly page size.

Paging (Cont.)

- On a 32-bit CPU, each page-table entry is 4 bytes long, but that size can vary as well.
- A 32-bit entry can point to one of 2^{32} physical page frames.
- If frame size is 4 KB (2^{12}), then a system with 4-byte entries can address 2^{44} bytes (or 16 TB) of physical memory.
- If other information is kept in the page-table entries, it reduces number of bits available to address page frames.
- A system with 32-bit page-table entries may address less physical memory than the possible maximum.

OPERATING SYSTEMS

Free Frames



Before allocation

After allocation

Paging (Cont.)

- OS manages physical memory, keeps track of which frames are free or allocated using **frame table**.
- OS must be aware that user processes operate in user space, and all logical addresses must be mapped to produce physical addresses.
- **The operating system maintains a copy of the page table for each process**.
- It is used to translate logical addresses to physical addresses
- It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU.
- Paging therefore increases the context-switch time.
- How to reduce the context-switch time?
  - Using TLB

THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Memory Management

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Structure of the page table

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

Hardware Support for Page Table

- The hardware implementation of page table can be done by using dedicated registers.
- But the usage of register for the page table is satisfactory only if page table is small.
- If page table contain large number of entries then we can use TLB(translation Look-aside buffer), a special, small, fast look up hardware cache.

OPERATING SYSTEMS

Hardware Support for Page Table

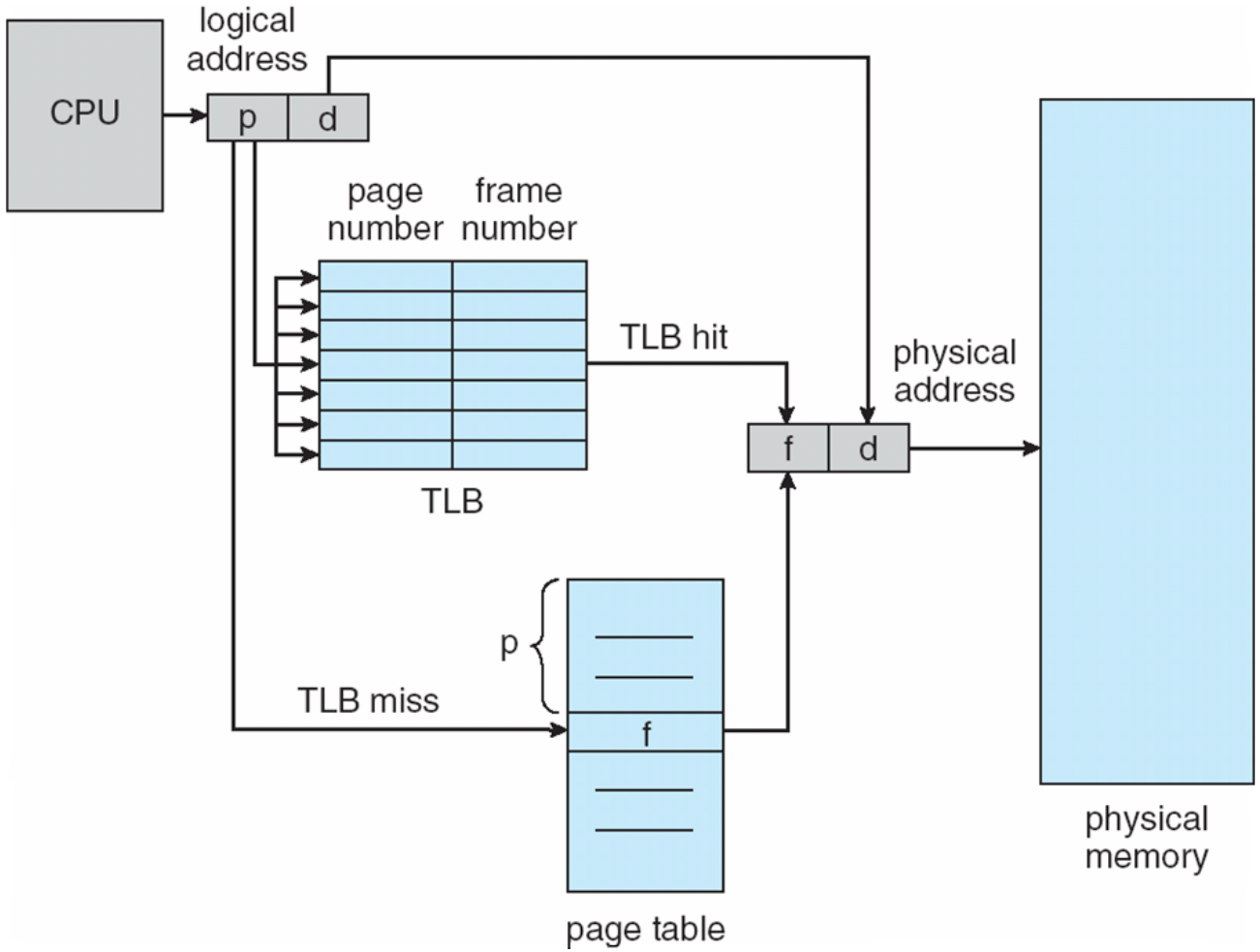
- Page table is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PTLR**) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)

Translation Look-Aside Buffer

- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry
- This identifier is stored with each TLB entry to distinguish between entries loaded for different processes.
- Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

OPERATING SYSTEMS

Paging Hardware With TLB



OPERATING SYSTEMS

Effective Access Time

- Hit ratio – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
  - If we find the desired page in TLB then a mapped-memory access take 10 ns
  - Otherwise we need two memory access so it is 20 ns i.e. if we fail to find the page number in the TLB then we must first access memory for the page table and frame number (10 ns) and then access the desired byte in memory (10 nanoseconds), for a total of 20 ns (assuming that a page table lookup takes only one memory access).
- **Effective Access Time** (**EAT**)

  EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds

  implying 20% slowdown in access time
- Consider a more realistic hit ratio of 99%,

  EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns

  implying only 1% slowdown in access time.

Translation Look-Aside Buffer

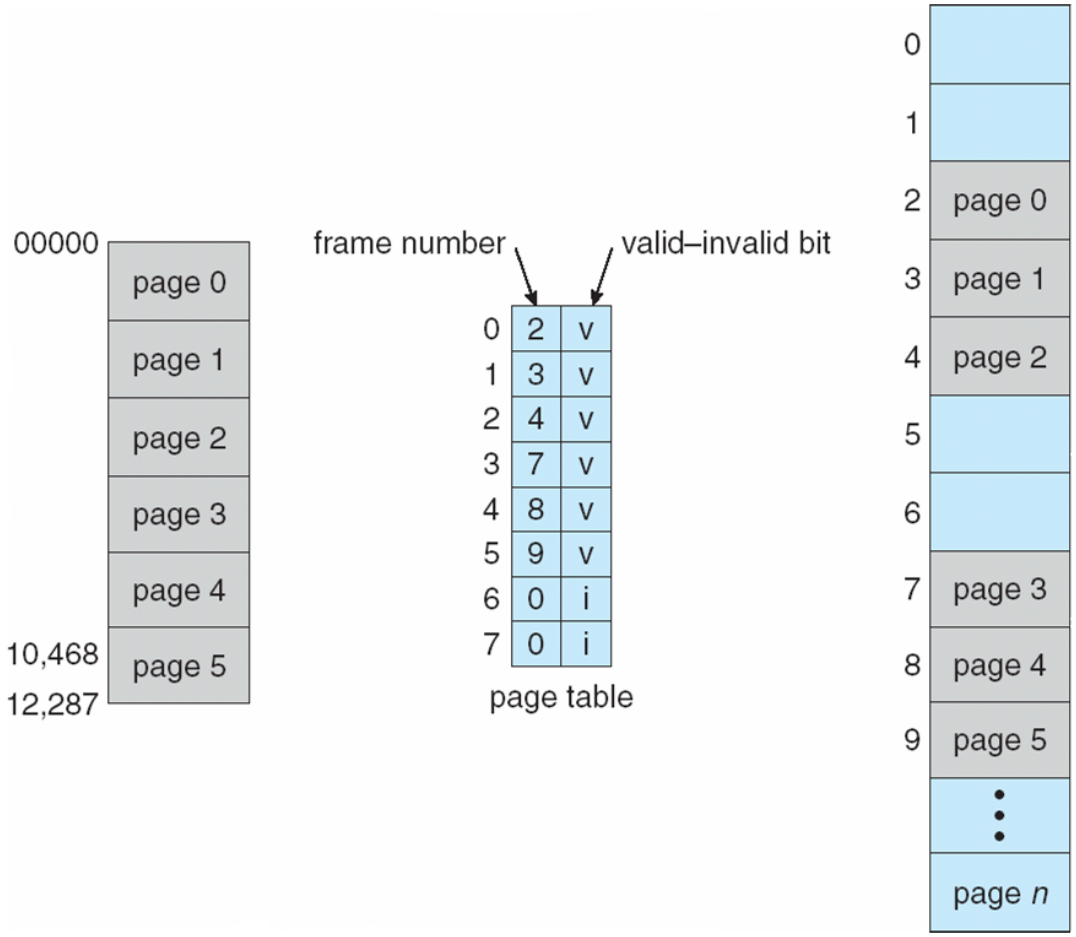
How TLB is different from CPU Cache

- TLB is about ‘speeding up address translation for Virtual/logical memory’ so that page-table needn’t to be accessed for every address.
- CPU Cache is about ‘speeding up main memory access latency’ so that RAM isn’t accessed always by CPU.
- TLB operation comes at the time of address translation by MMU while CPU cache operation comes at the time of memory access by CPU.

Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **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
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

Valid (v) or Invalid (i) Bit In A Page Table



OPERATING SYSTEMS

Shared Pages

- An advantage of paging is the possibility of sharing common code.
  - important in a time-sharing environment
- Consider a system that supports 40 users, each of whom executes a text editor.
- If the text editor consists of 150 KB of code and 50 KB of data space.
- we need 8,000 KB(150 KBX40+50KB X40) to support the 40 users.
- Can this memory space usage be reduced?
  - YES
- If the code is shared, then a total space required will be 2,150 KB

Shared Pages

Shared code

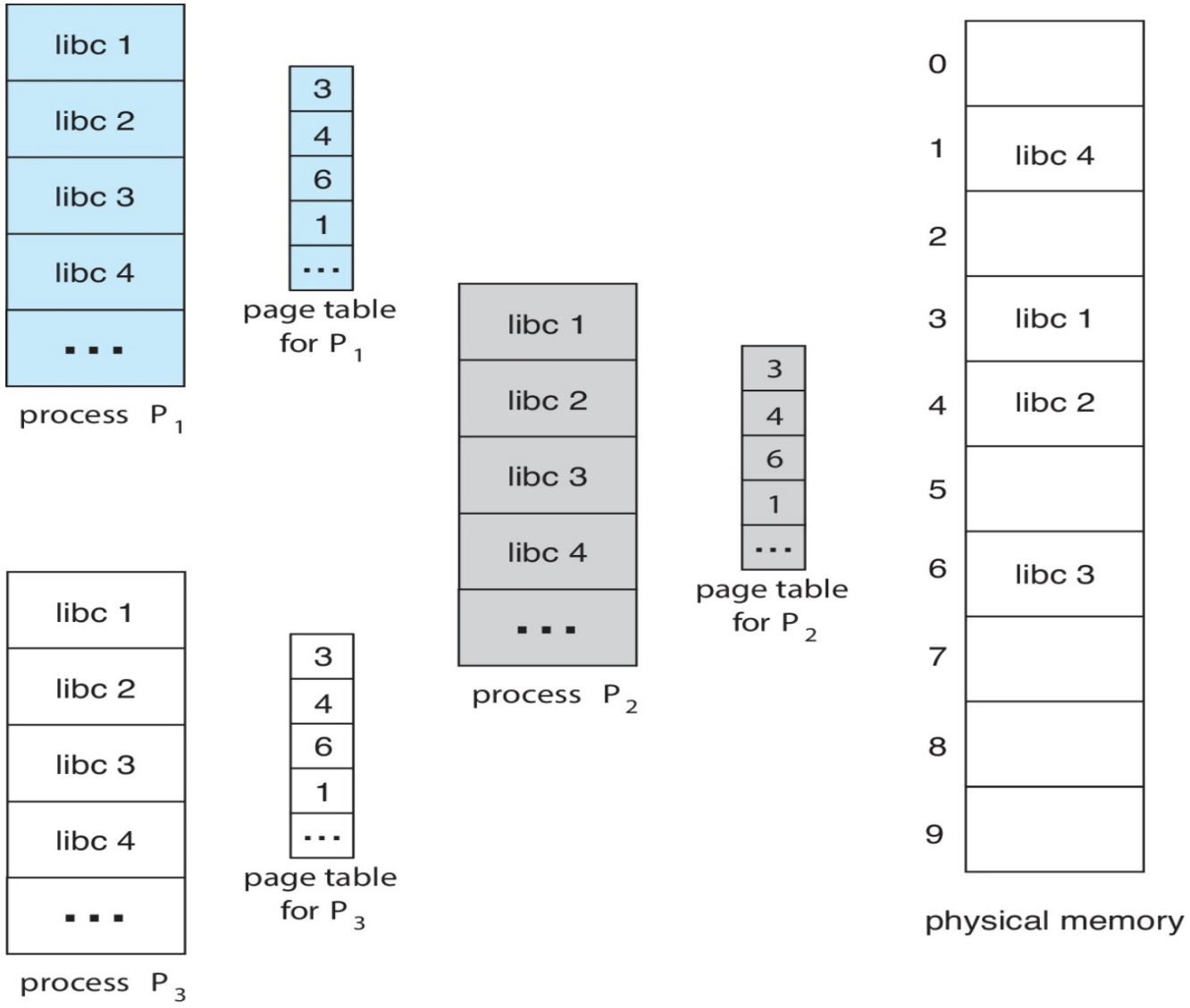
- One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
- Reentrant (multi-instance) code is a reusable routine that multiple programs can invoke, interrupt, and reinvoke simultaneously.
- Reentrant code is non-self-modifying code: it never changes during execution.
- Similar to multiple threads sharing the same process space
- Also useful for interprocess communication if sharing of read-write pages is allowed

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

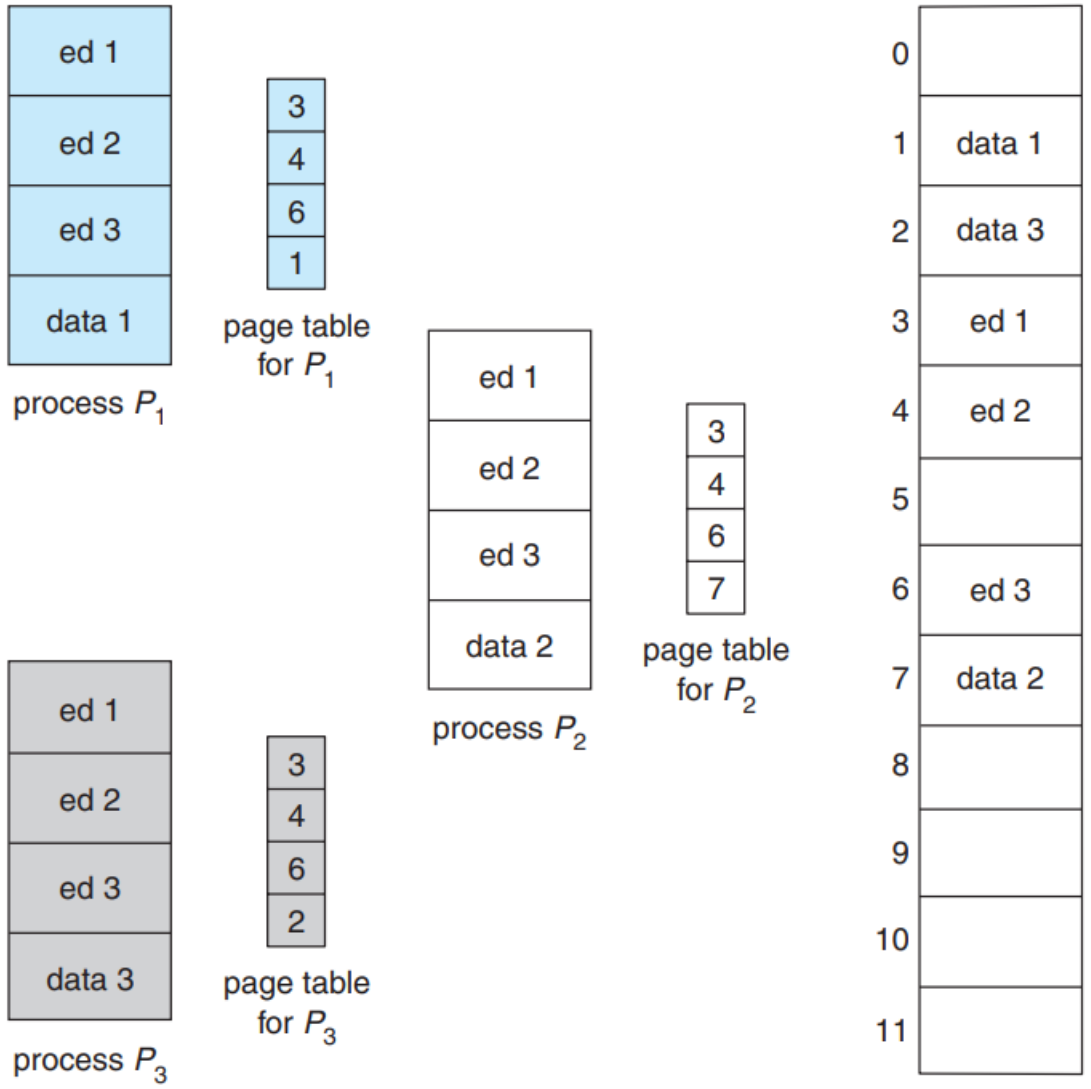
OPERATING SYSTEMS

Shared Pages Example



OPERATING SYSTEMS

Shared Pages Example

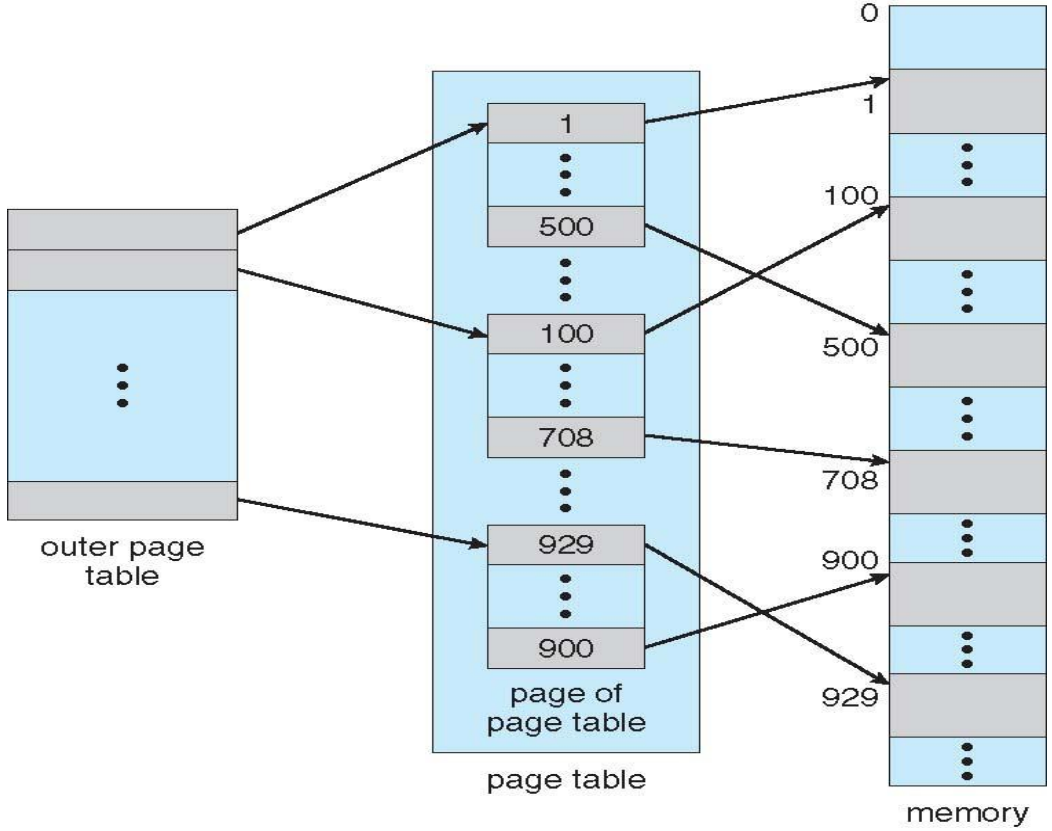


Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes -> 4 MB of physical memory for page table alone
    - That amount of memory used to cost a lot
    - Don’t want to allocate that contiguously in main memory
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

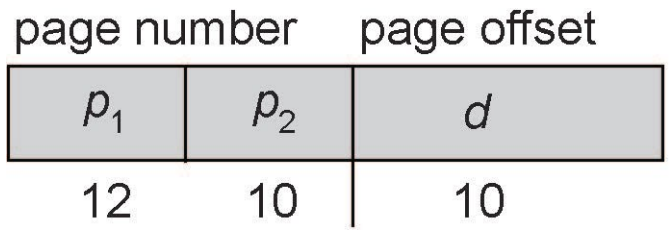
Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table



Two-Level Paging Example

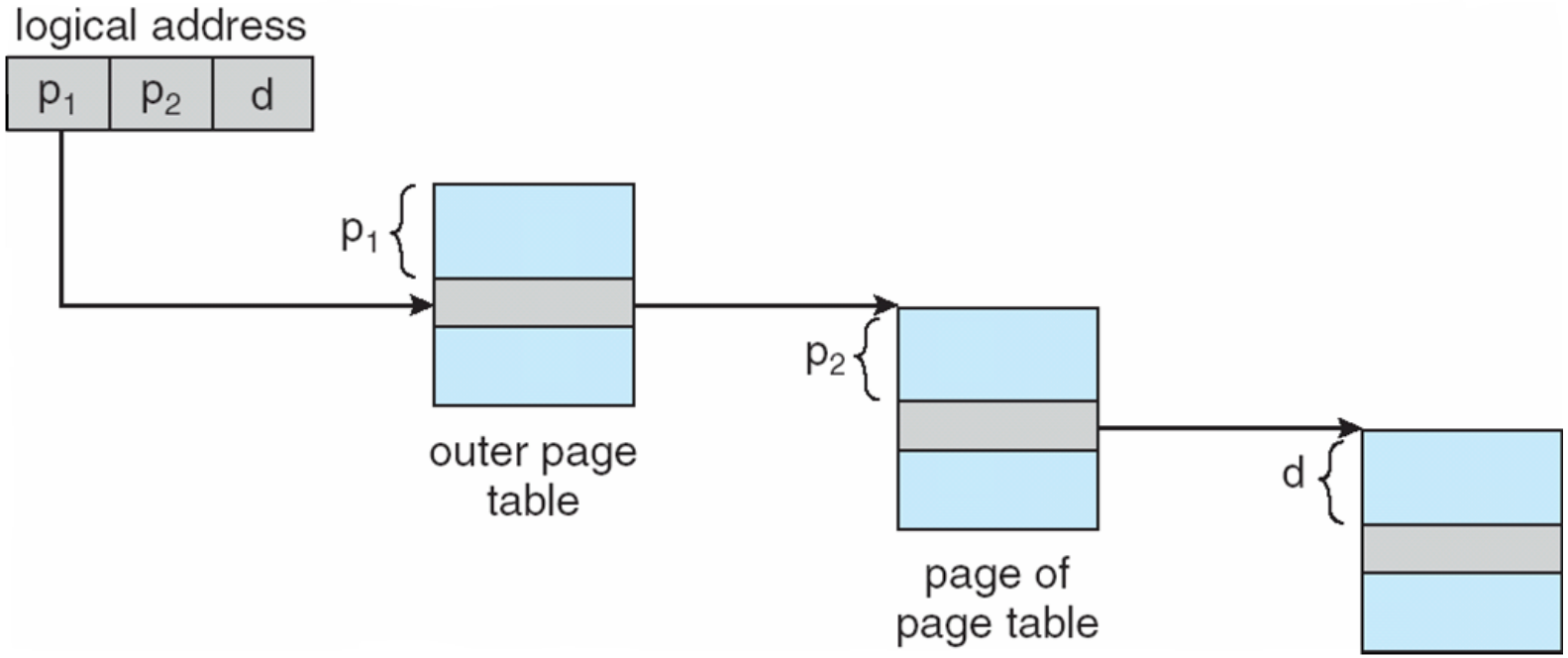
- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:



where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table

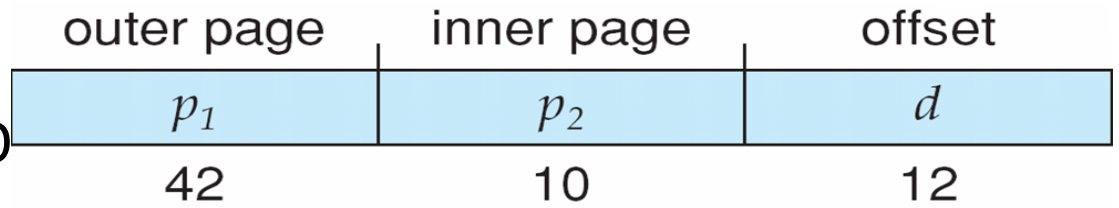
Address-Translation Scheme

- This scheme is known as **forward-mapped page table as** address translation works from the outer page table inward.



64-bit Logical Address Space

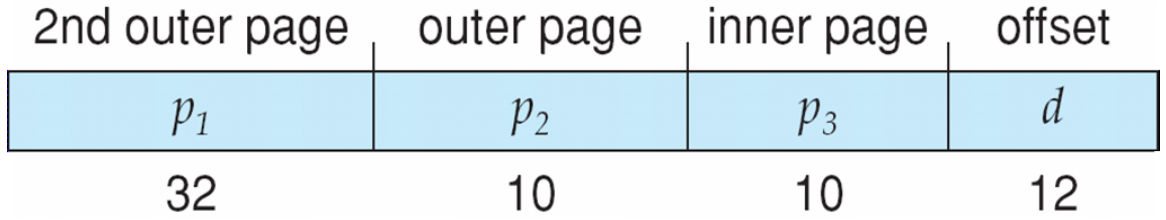
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



  - O
  - One solution is to add a 2nd outer page table

Three-level Paging Scheme

- We can divide the outer page table in various ways.
- For example, we can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (2^{10} entries, or 2^{12} bytes).
- In this case, a 64-bit address space is still daunting:



The outer page table is still 2^{34} bytes (16 GB) in size. And possibly 4 memory access to get to one physical memory location

- The next step would be a four-level paging scheme, where the second-level outer page table itself is also paged, and so forth.
- The 64-bit UltraSPARC would require seven levels of paging—a prohibitive number of memory accesses— to translate each logical address. So, for 64-bit architectures, hierarchical page tables are generally considered inappropriate.

Examples

Consider that the size of main memory (physical memory) is 512 MB and frame size of 4KB. Calculate

a.Page offset(d)

b.Bits to address page number/Frame number

c.What will be the total number of frames

d.Number of level of paging

Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number (VPN) is hashed into a page table
  - Each entry in the page table contains a linked list of elements that hash to the same location (to handle collisions)
- Each element contains three fields:

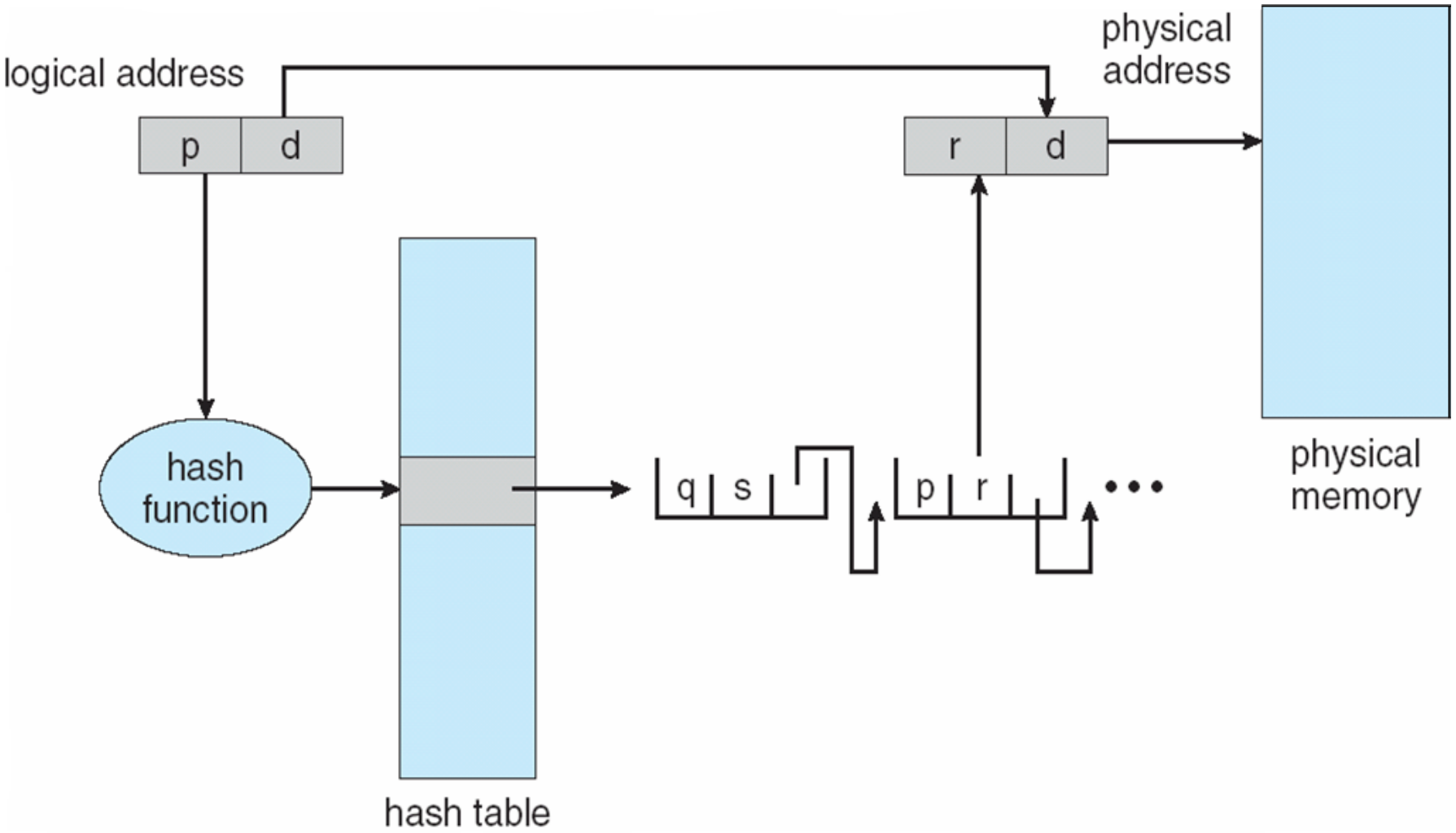
  (1) the virtual page number

  (2) the value of the mapped page frame

  (3) a pointer to the next element
- Virtual page numbers are compared (with field 1) in this chain searching for a match
  - If a match is found, the corresponding physical frame (field 2) is extracted
  - If there is no match, subsequent entries in the linked list are searched for a matching VPN

OPERATING SYSTEMS

Hashed Page Table



OPERATING SYSTEMS

Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

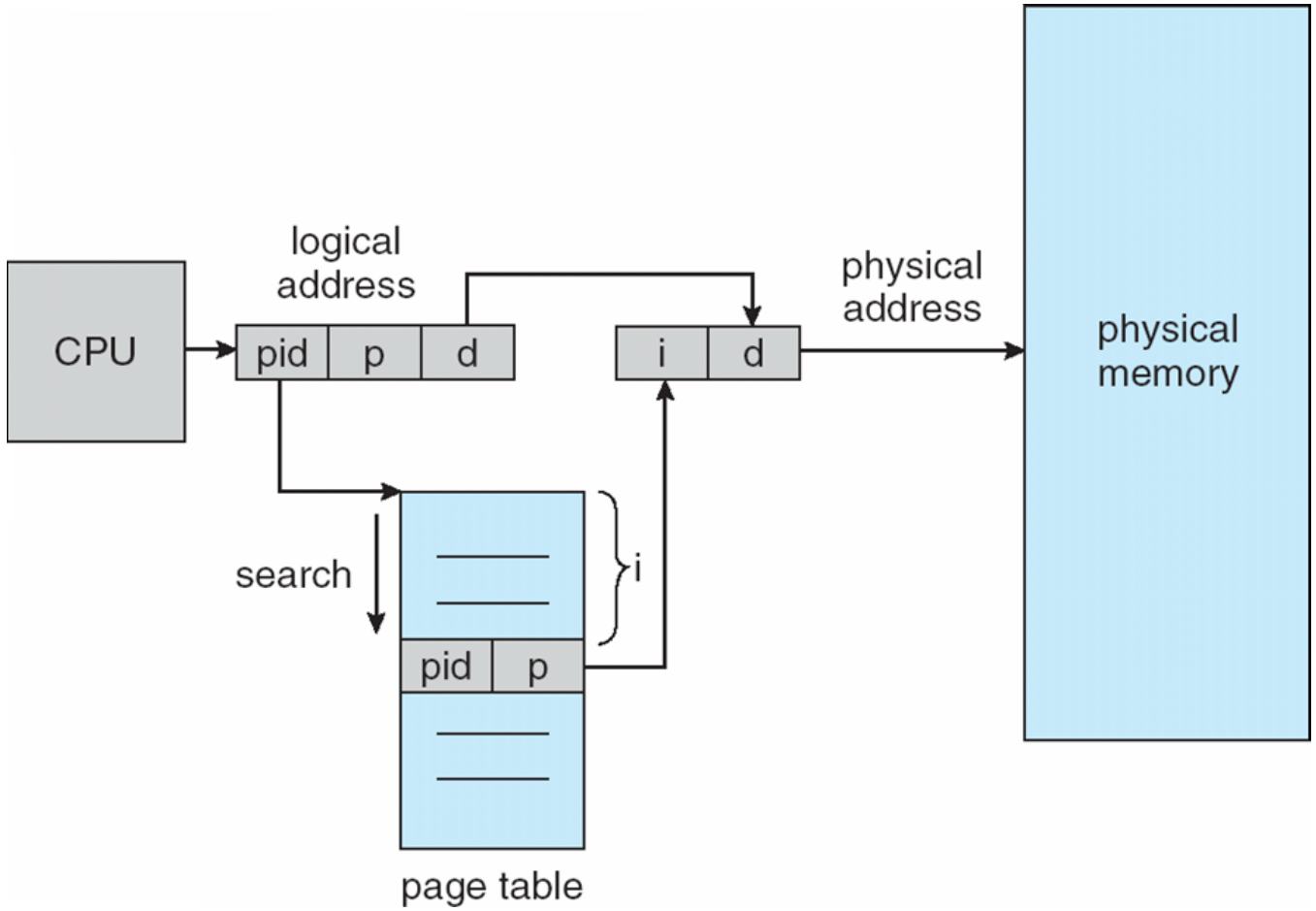
Inverted Page Table

- A simplified version of the inverted page table used in the *IBM RT*.
- IBM was the first major company to use inverted page tables
  - starting with the IBM System 38 to RS/6000 and the current IBM Power CPUs.
  - For the IBM RT, each virtual address in the system consists of a triple:
    - <process-id, page-number, offset>.

Inverted Page Table

- Each inverted page-table entry is a pair <process-id, page-number> where the process-id assumes the role of the address-space identifier.
- When a memory reference occurs, part of the virtual address, consisting of <process-id, page number>, *t*he inverted page table is then searched for a match.
- If a match is found—say, at entry *i*—then the physical address <*i*, offset> is generated.
- If no match is found, then an illegal address access has been attempted.

Inverted Page Table Architecture



Used in 64-bit UltraSPARC (from Sun Micro Systems) and PowerPC (created by Apple-IBM-Motorola alliance)

Inverted Page Table

- This scheme decreases the amount of memory needed to store each page table.
- It increases the amount of time needed to search the table when a page reference occurs.
- The inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched before a match is found.
- Hash table is used to over come this problem where the number of search's are limited one or at most few.
- each access to the hash table adds a memory reference to the procedure.
- One virtual memory reference requires at least two real memory reads—one for the hash-table entry and one for the page table

Inverted Page Table

- Systems that use inverted page tables have difficulty in implementing shared memory.
- Shared memory is usually implemented as multiple virtual addresses are mapped to one physical address.
- There is only one virtual page entry for every physical page
- One physical page cannot have two (or more) shared virtual addresses.
- A simple technique for addressing this issue is to allow the page table to contain only one mapping of a virtual address to the shared physical address.
- references to virtual addresses that are not mapped result in page faults

Page table example

Consider a computer system with a 32-bit logical address and 4-KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?

a. A conventional single-level page table

b. An inverted page table

THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Memory Management

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Intel 32 and 64-bit Architectures

Suresh Jamadagni
Department of Computer Science

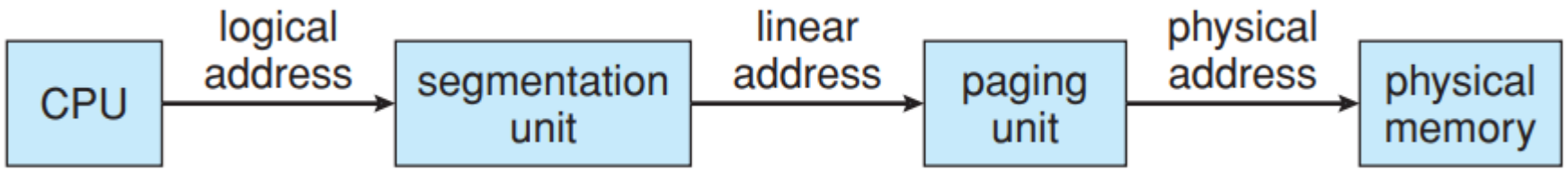
OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

Logical to physical address translation in IA-32



- Memory management in IA-32 systems is divided into two components— segmentation and paging:
- The CPU generates logical addresses, which are given to the segmentation unit.
- The segmentation unit produces a linear address for each logical address.
- The linear address is then given to the paging unit, which in turn generates the physical address in main memory.
- The segmentation and paging units form the equivalent of the memory-management unit (MMU).

OPERATING SYSTEMS

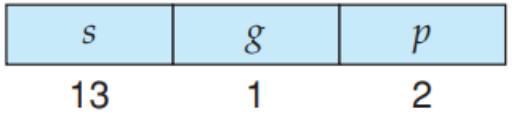
IA-32 Segmentation

- The IA-32 architecture allows a segment to be as large as 4 GB
- Maximum number of segments per process is 16 K.
- The logical address space of a process is divided into two partitions. The first partition consists of up to 8K segments that are private to that process. The second partition consists of up to 8 K segments that are shared among all the processes.
- Information about the first partition is kept in the **local descriptor table (LDT)** and information about the second partition is kept in the **global descriptor table (GDT).**
- Each entry in the LDT and GDT consists of an 8-byte segment descriptor with detailed information about a particular segment, including the base location and limit of that segment.

OPERATING SYSTEMS

IA-32 Segmentation (contd…)

- The logical address is a pair (selector, offset), where the selector is a 16-bit number
- s designates the segment number, g indicates whether the segment is in the GDT or LDT, and p deals with protection
- The offset is a 32-bit number specifying the location of the byte within the segment in question.

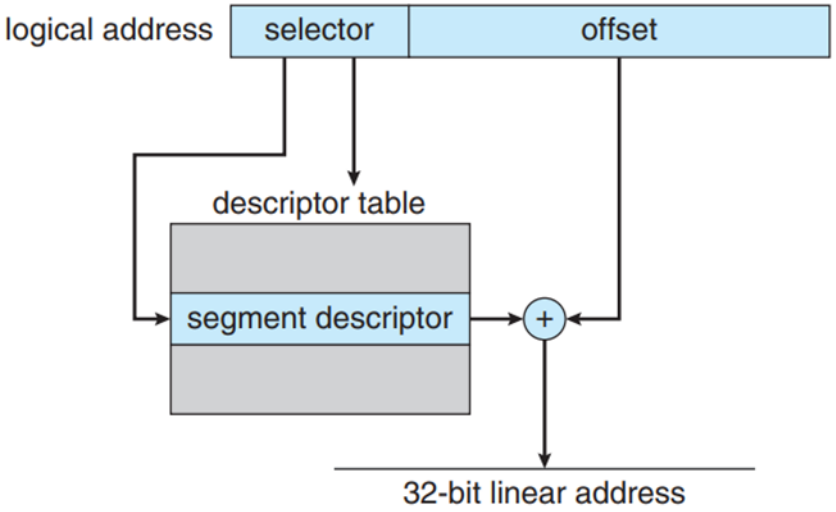


- The machine has **six segment registers**, allowing six segments to be addressed at any one time by a process.
- It also has **six 8-byte microprogram registers** to hold the corresponding descriptors from either the LDT or GDT. This cache lets the Pentium avoid having to read the descriptor from memory for every memory reference

OPERATING SYSTEMS

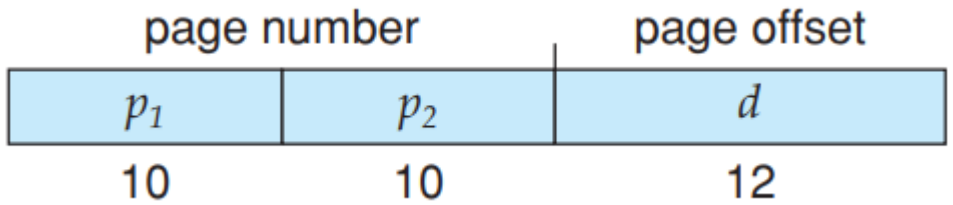
IA-32 Segmentation (contd…)

- The linear address on the IA-32 is 32 bits long and is formed as follows:
  - The segment register points to the appropriate entry in the LDT or GDT. The base and limit information about the segment in question is used to generate a linear address.
  - First, the limit is used to check for address validity. If the address is not valid, a memory fault is generated, resulting in a trap to the operating system.
  - If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address.



IA-32 Paging

- The IA-32 architecture allows a page size of either 4 KB or 4 MB.
- For 4-KB pages, IA-32 uses a two-level paging scheme in which the division of the 32-bit linear address is as follows:

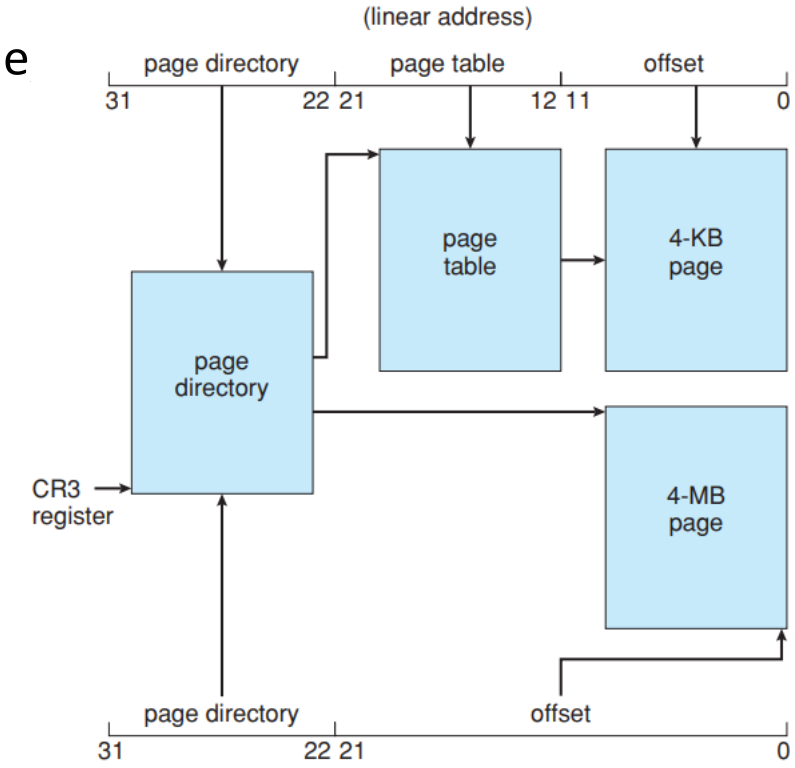


- The 10 high-order bits reference an entry in the outermost page table, which IA-32 terms the page directory. (The CR3 register points to the page directory for the current process.)
- The page directory entry points to an inner page table that is indexed by the contents of the innermost 10 bits in the linear address.
- The low-order bits 0–11 refer to the offset in the 4-KB page pointed to in the page table.

OPERATING SYSTEMS

IA-32 Paging

- One entry in the page directory is the Page Size flag, which if set, indicates that the size of the page frame is 4 MB and not the standard 4 KB.
- If this flag is set, the page directory points directly to the 4-MB page frame bypassing the inner page table; and the 22 low-order bits in the linear address refer to the offset in the 4-MB page frame



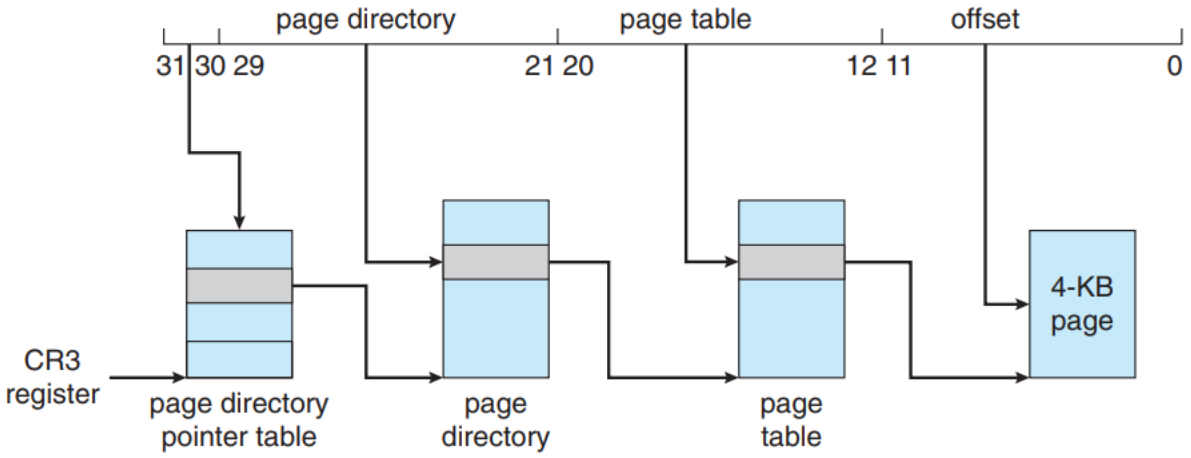
OPERATING SYSTEMS

IA-32 Paging

- To improve the efficiency of physical memory use, IA-32 page tables can be swapped to disk.
- An invalid bit is used in the page directory entry to indicate whether the table to which the entry is pointing is in memory or on disk.
- If the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table.
- The table can then be brought into memory on demand.

OPERATING SYSTEMS

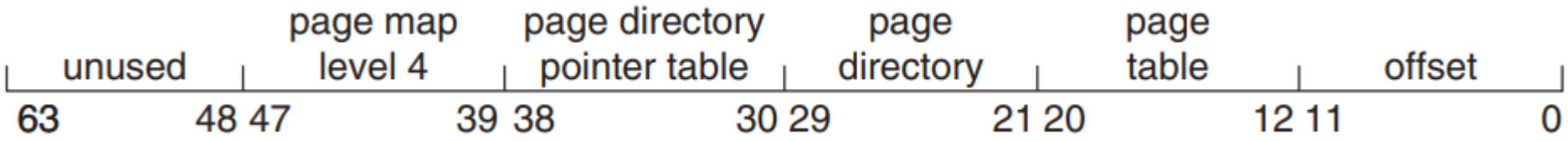
IA-32 Physical Address Extension



- Intel adopted a page address extension (PAE), which allows 32-bit processors to access a physical address space larger than 4 GB.
- The fundamental difference introduced by PAE support was that paging went from a two-level scheme to a **three-level scheme**, where the top two bits refer to a page directory pointer table.
- PAE also increased the page-directory and page-table entries from 32 to 64 bits in size, which allowed the base address of page tables and page frames to extend from 20 to 24 bits.
- Combined with the 12-bit offset, adding PAE support to IA-32 increased the address space to 36 bits, which supports up to 64 GB of physical memory.

OPERATING SYSTEMS

X86-64



- The x86-64 supported much larger logical and physical address spaces, as well as several other architectural advances.
- Support for a **64-bit address space yields an astonishing 264 bytes of addressable memory**—a number greater than 16 quintillion (or 16 exabytes).
- Even though 64-bit systems can potentially address this much memory, in practice far fewer than 64 bits are used for address representation in current designs.
- The x86-64 architecture currently provides a 48-bit virtual address with support for page sizes of 4 KB, 2 MB, or 1 GB using four levels of paging hierarchy
- Because this addressing scheme can use PAE, virtual addresses are 48 bits in size but support 52-bit physical addresses (4096 terabytes)

Additional reference

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Memory Management

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Virtual Memory

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

Background

- Virtual memory is a technique that allows the execution of processes that are not completely in memory.
- One major advantage of this scheme is that programs can be larger than physical memory.
- This technique frees programmers from the concerns of memory-storage limitations.
- Virtual memory also allows processes to share files easily and to implement shared memory

Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

Background (Cont.)

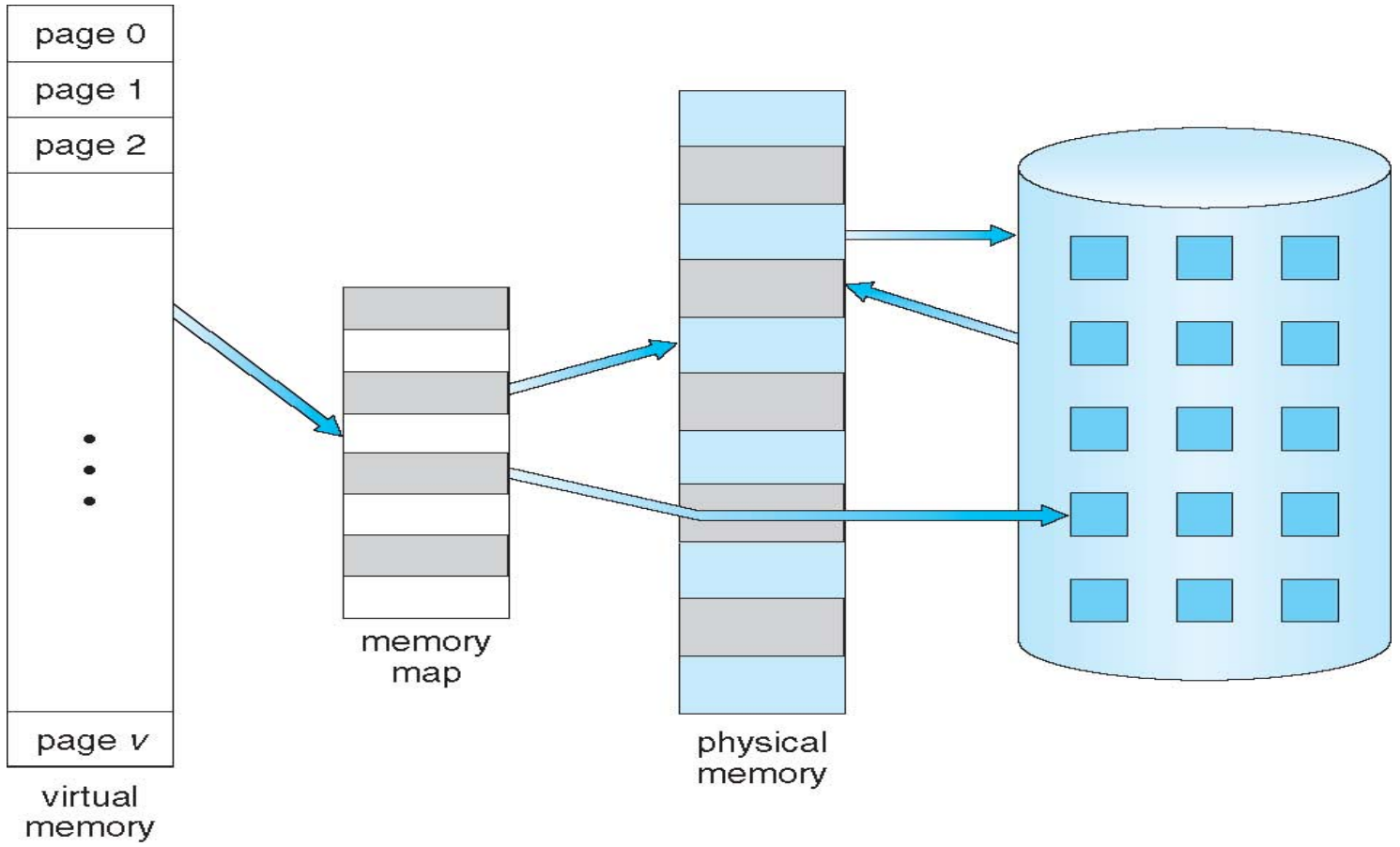
- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

Background (Cont.)

- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

OPERATING SYSTEMS

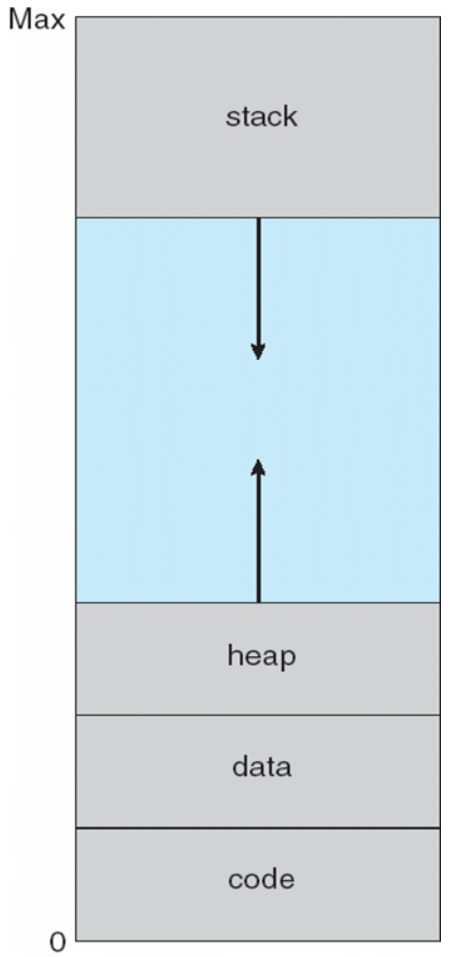
Virtual Memory That is Larger Than Physical Memory



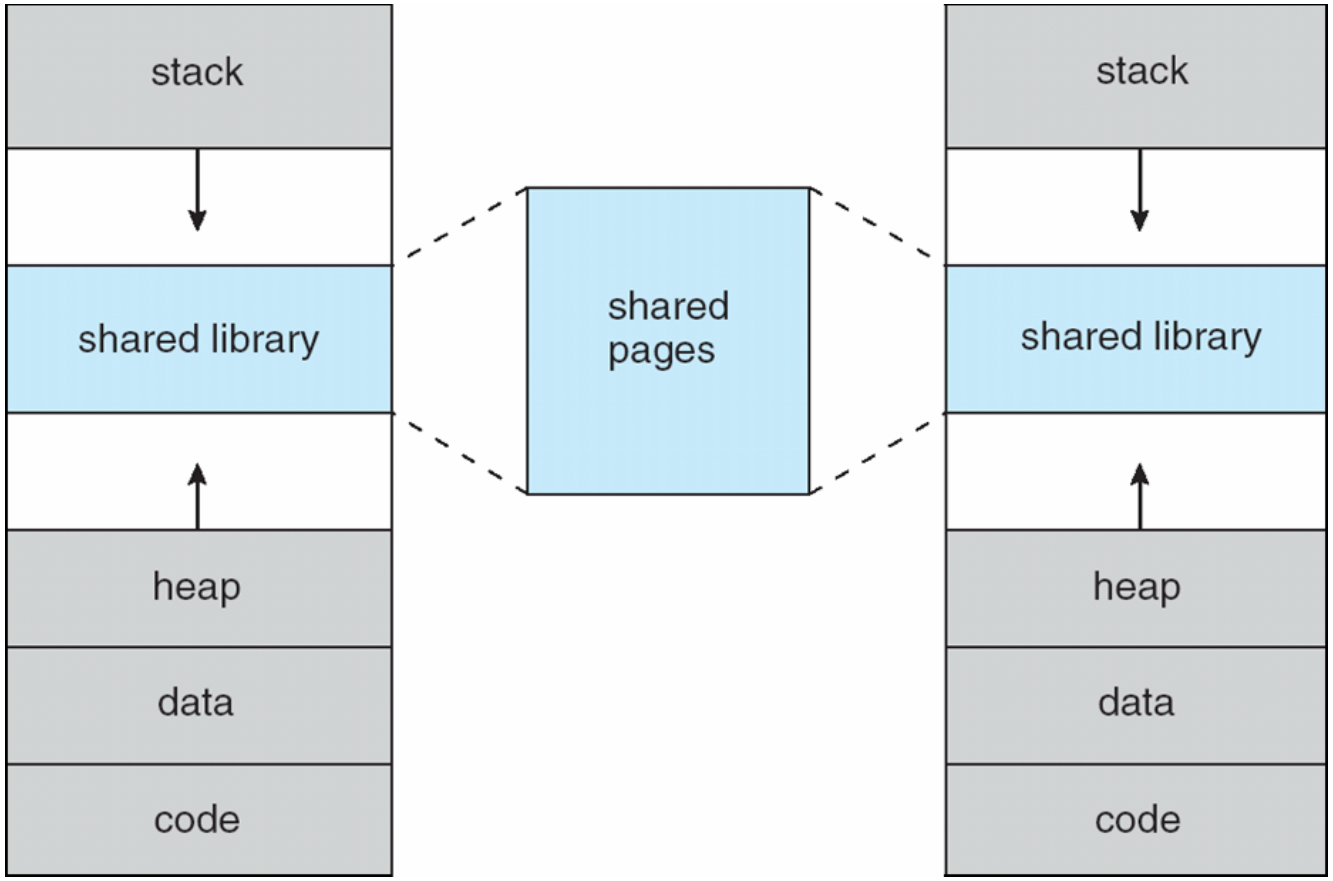
OPERATING SYSTEMS

Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during fork(), speeding process creation



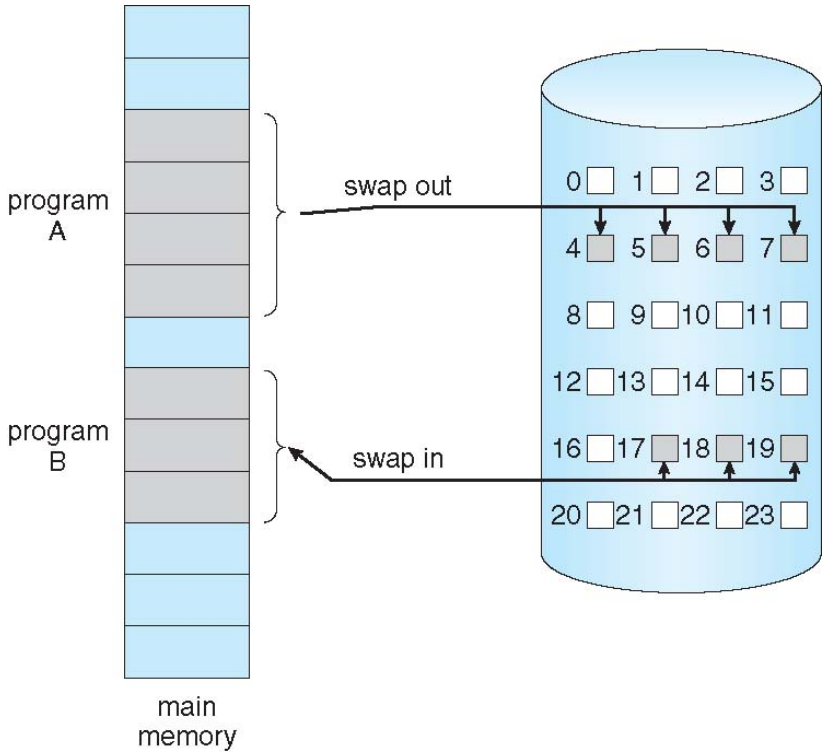
Shared Library Using Virtual Memory



OPERATING SYSTEMS

Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**



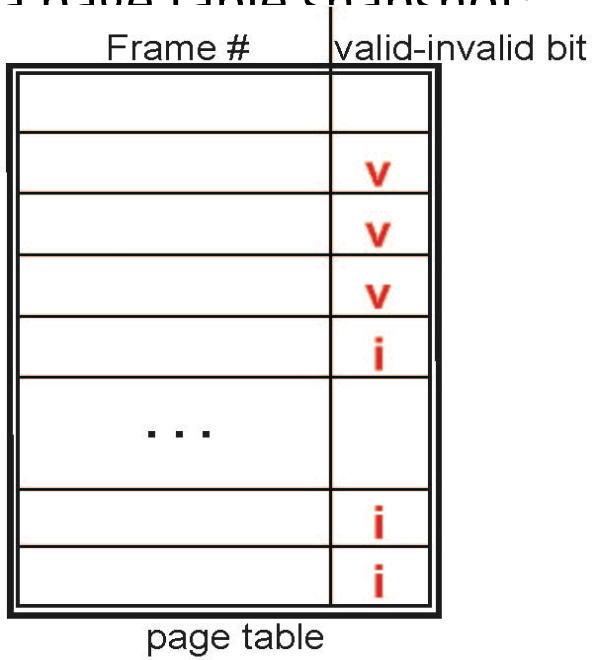
Demand Paging – Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code

OPERATING SYSTEMS

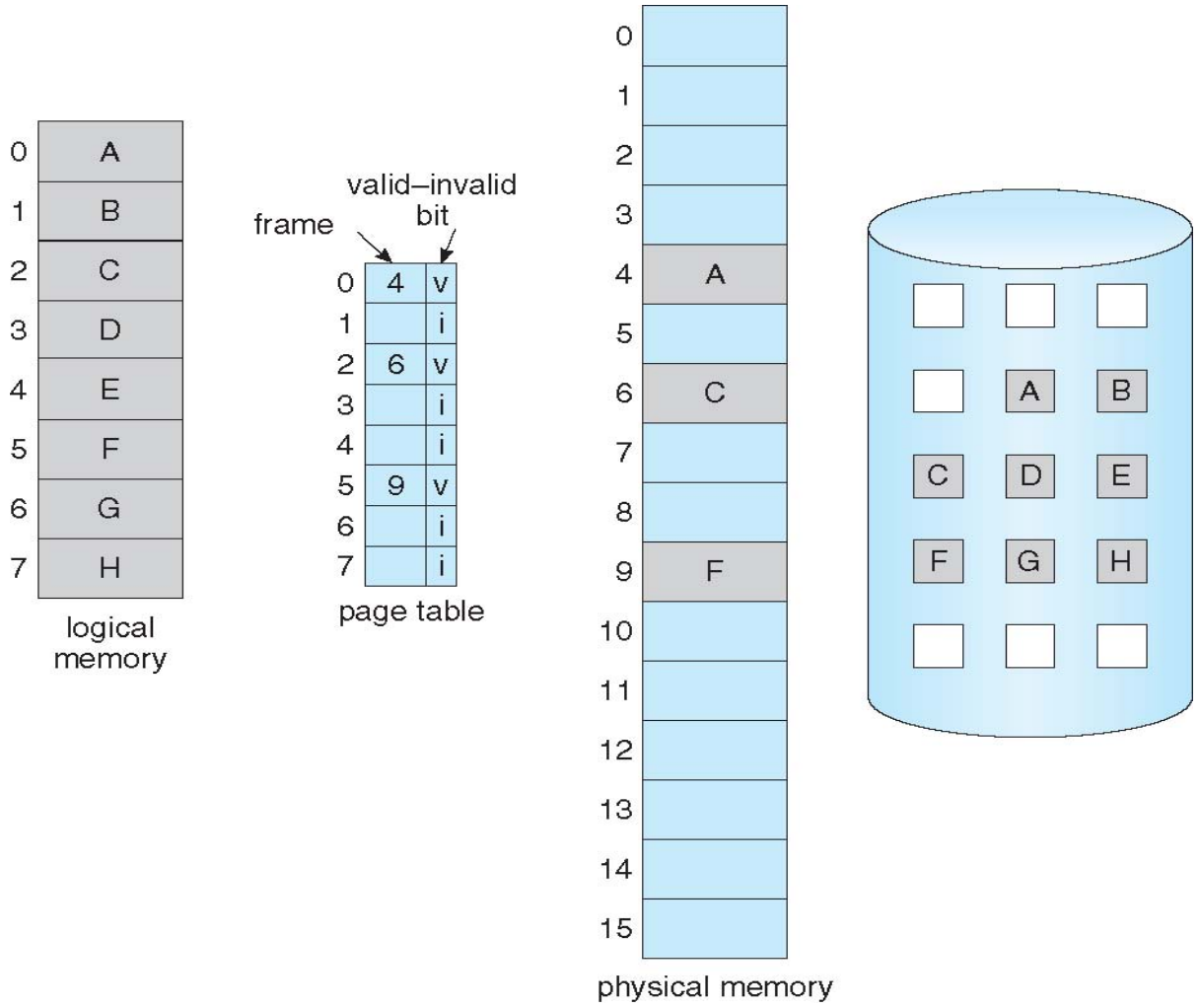
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

Page Table When Some Pages Are Not in Main Memory

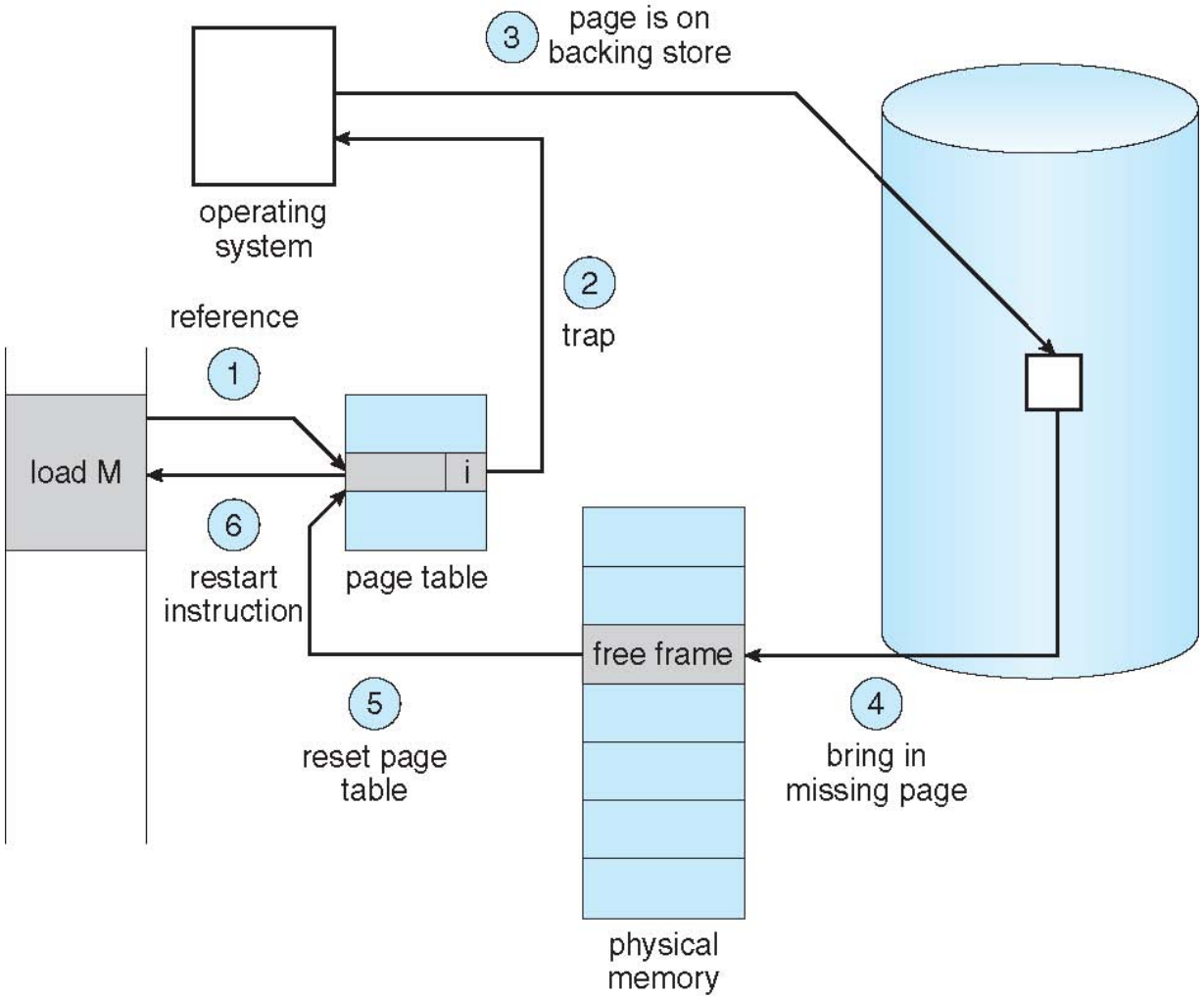


Steps in Handling Page Fault

1. If there is a reference to a page, first reference to that page will trap to operating system
   - **Page fault**
2. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
3. Find free frame
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory
   Set validation bit = **v**
6. Restart the instruction that caused the page fault

OPERATING SYSTEMS

Steps in Handling Page Fault (Cont.)



OPERATING SYSTEMS

Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference** - Process migrates from one locality (i.e., a set of pages that are actively used together) to another
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Instruction Restart

- Consider an instruction that could access several different locations (Ex: move some bytes from one location to another possibly overlapping location)
  - Source and destination blocks overlap i.e straddle a page boundary
    - Page fault might occur after the move is partially done
    - Source block may have been modified so we cannot simply restart the instruction
      - In one solution, the microcode computes and attempts to access both ends of both blocks.
        - Page fault can occur before anything is modified
      - The other solution uses temporary registers to hold the values of overwritten locations.
        - If Page fault occurs, all the old values are written back into memory before the trap occurs

Performance of Demand Paging

- Stages in Demand Paging (worse case)

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
    a) Wait in a queue for this device until the read request is serviced
    b) Wait for the device seek and/or latency time
    c) Begin the transfer of the page to a free frame

Performance of Demand Paging (Cont.)

6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging (Cont.)

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \le p \le 1$ (p is the probability of a page fault)
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

$$\textbf{EAT} = (1 - p) \times \text{memory access} + p \times \text{page-fault service time}$$

OPERATING SYSTEMS

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p) x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- EAT is directly proportional to page-fault rate
- If one access out of 1,000 causes a page fault i.e p = 0.001, then

  EAT = 8200 ns or 8.2 microseconds.

  This is a slowdown by a factor of 40!! (i.e. 8200/200)
- If we want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p

    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap space is allocated in larger blocks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix

Demand Paging Optimizations (Cont.)

- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code) from applications
    - if memory becomes constrained and demand page such data from file system later if needed

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Memory Management

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Virtual Memory

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

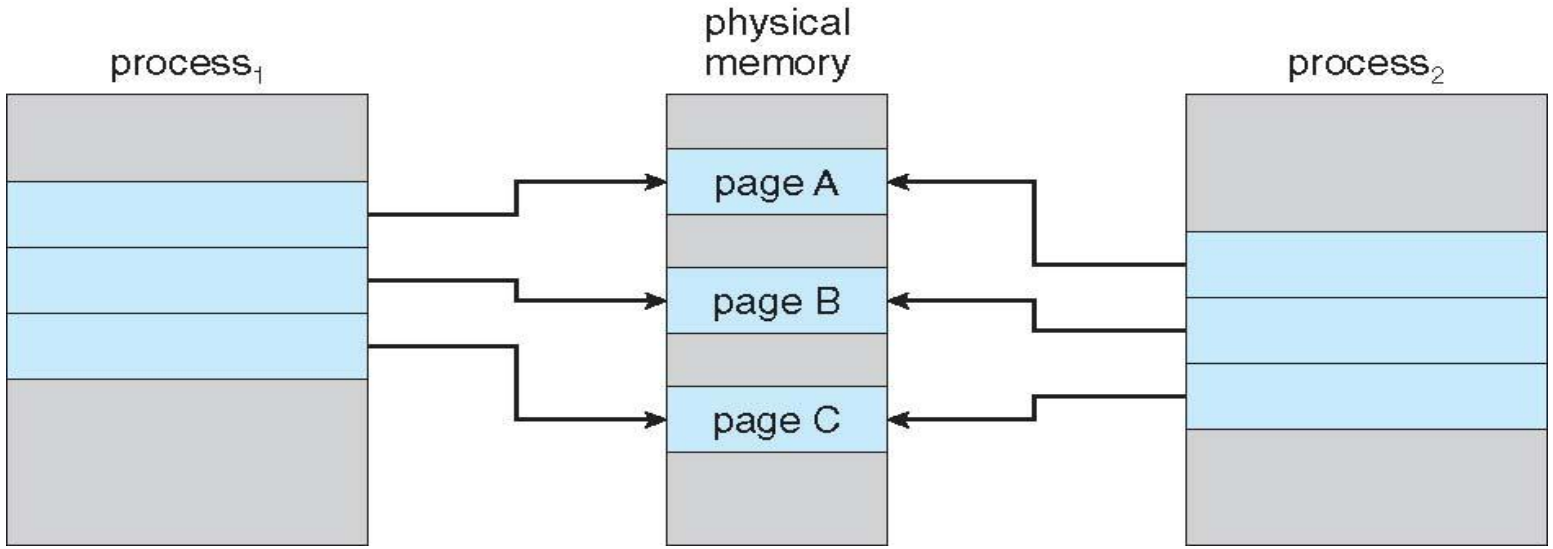
OPERATING SYSTEMS

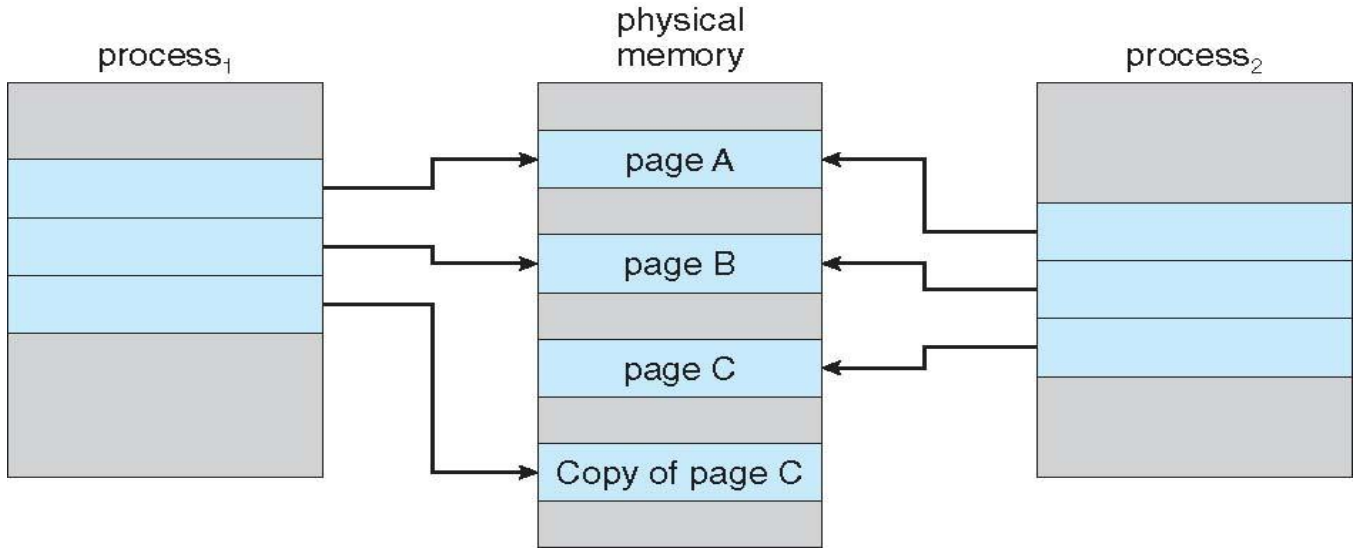
Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don’t want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it?
- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call exec()
  - Very efficient

OPERATING SYSTEMS

Before and After Process 1 Modifies Page C





What Happens if There is no Free Frame?

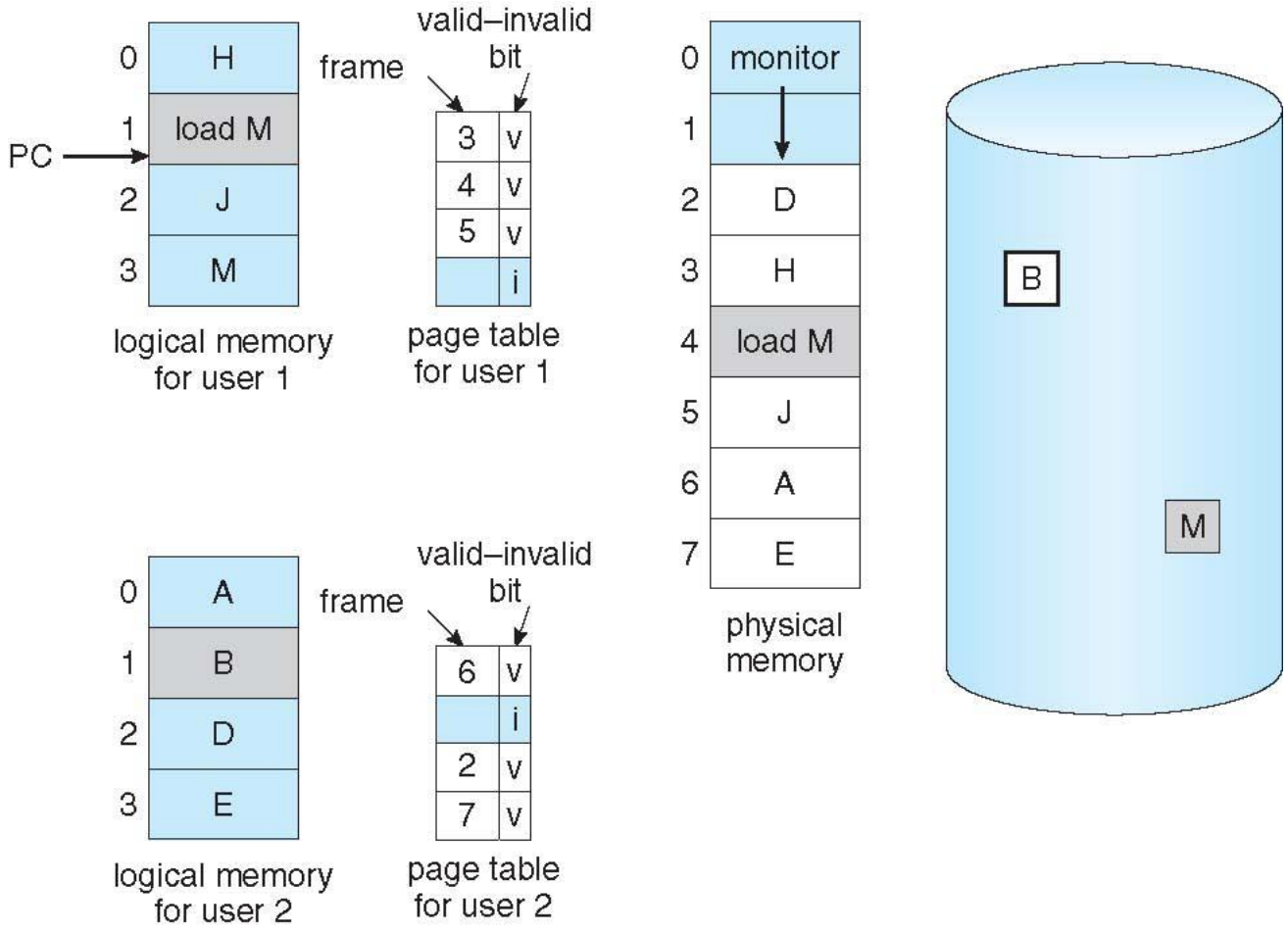
- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement (vs simply increasing degree of multiprogramming)
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

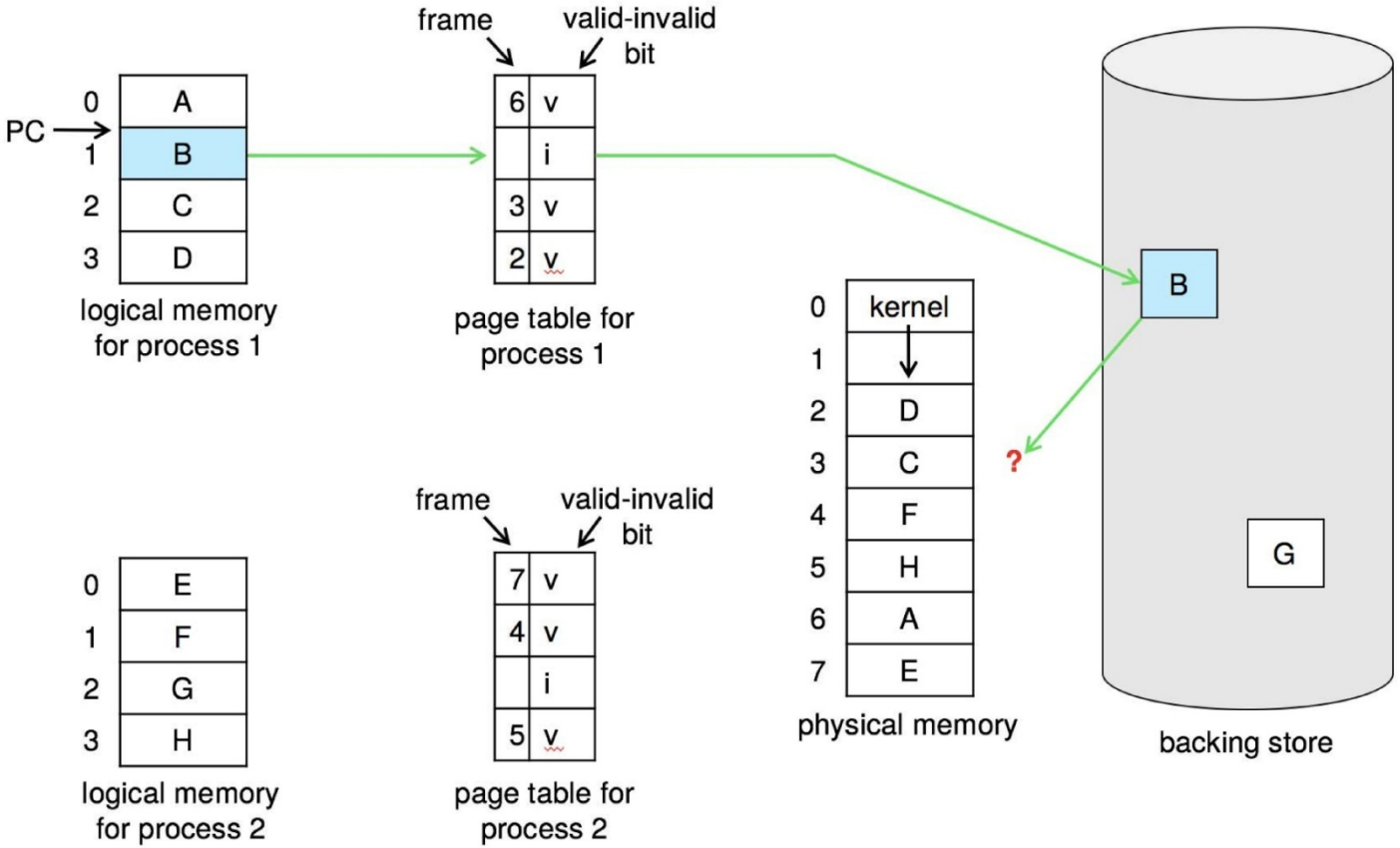
OPERATING SYSTEMS

Need For Page Replacement – Example 1



OPERATING SYSTEMS

Need For Page Replacement –Example 2

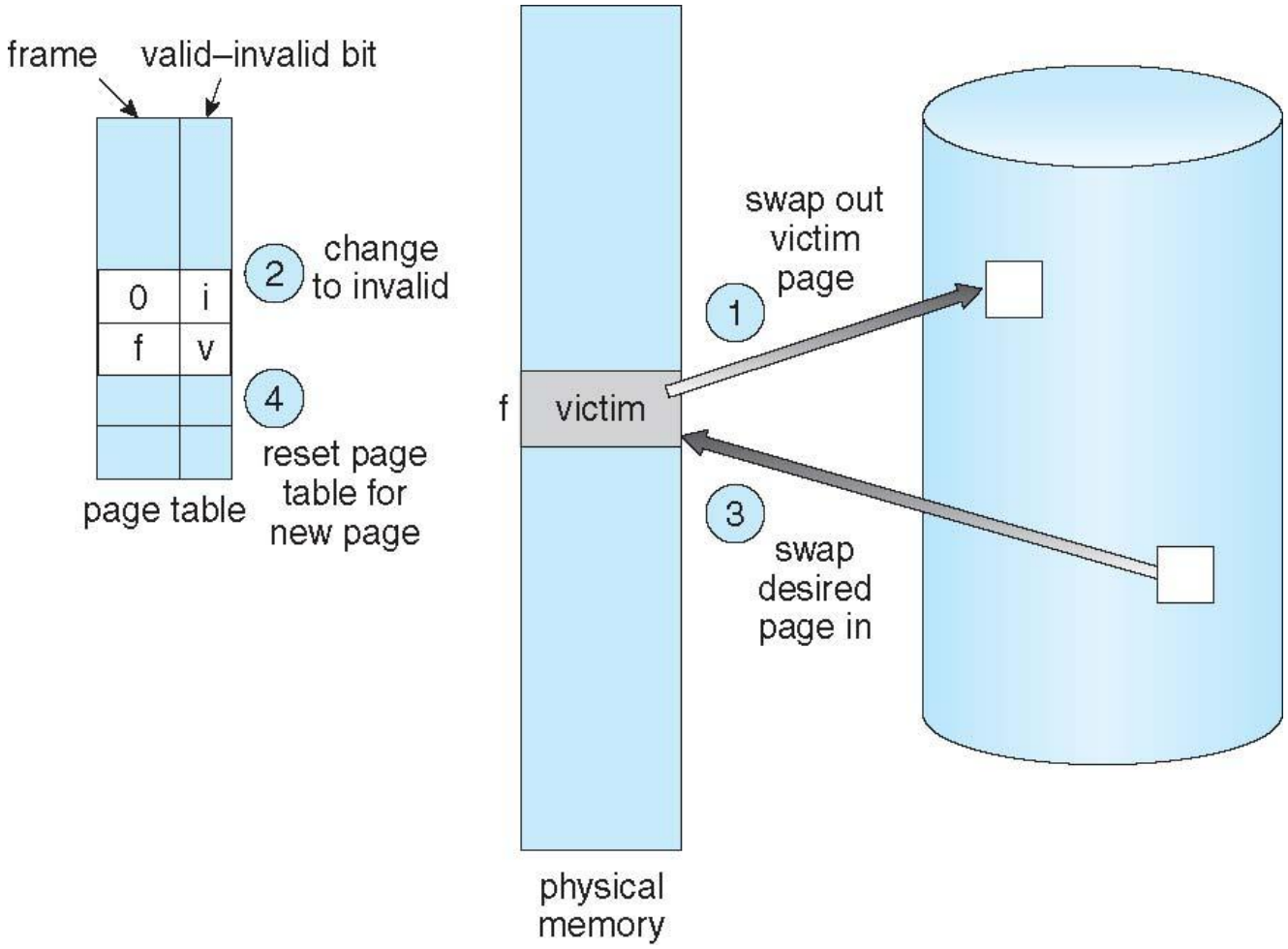


Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
     - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT

Basic Page Replacement (Cont.)



THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Memory Management

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Virtual Memory

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

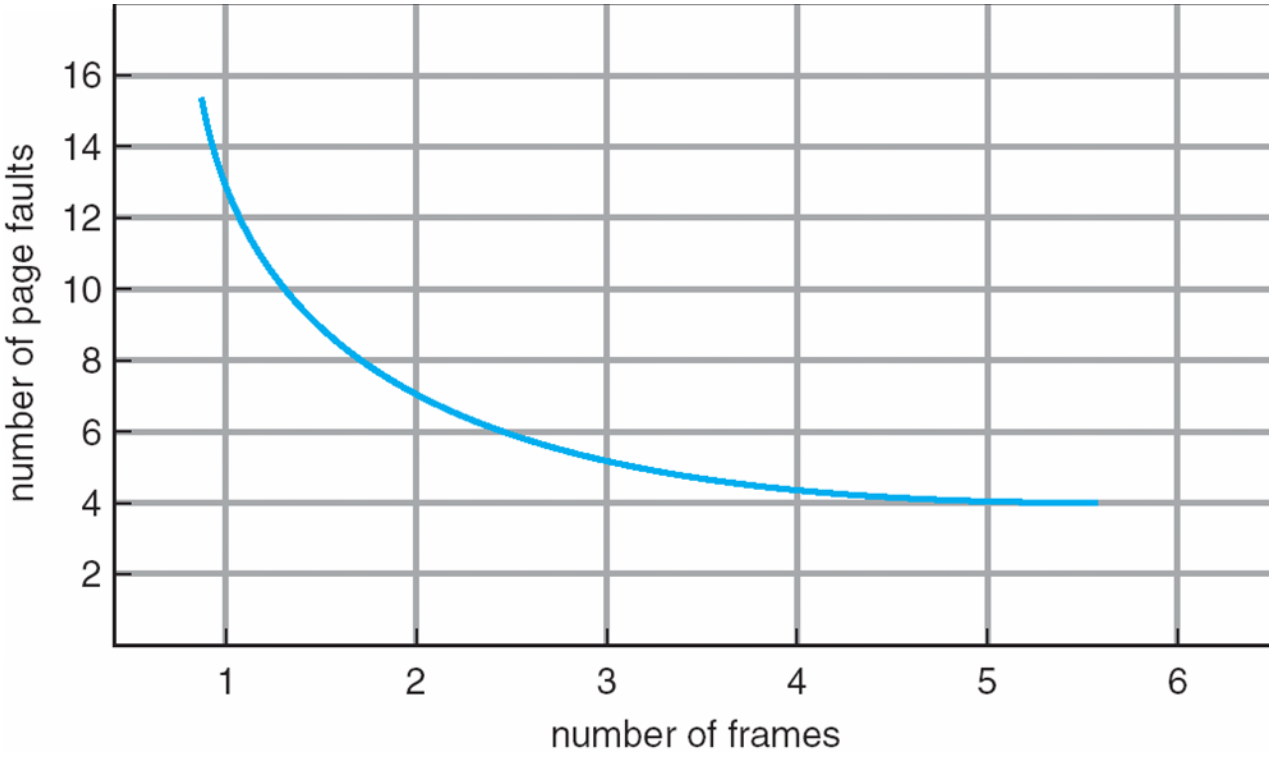
OPERATING SYSTEMS

Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

  7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

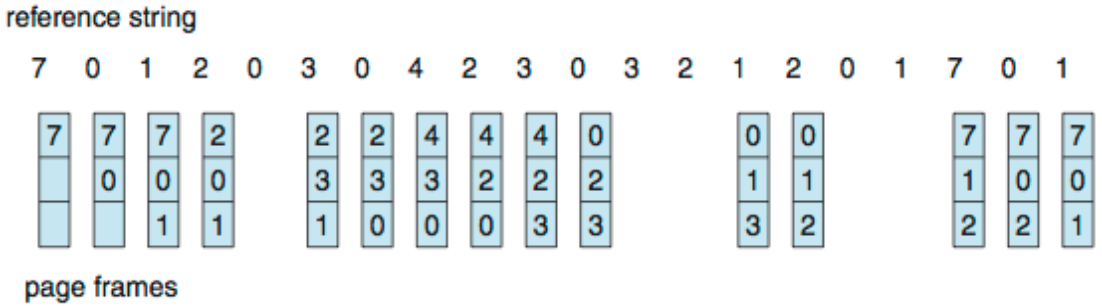
Graph of Page Faults Versus The Number of Frames



OPERATING SYSTEMS

First-In-First-Out (FIFO) Algorithm

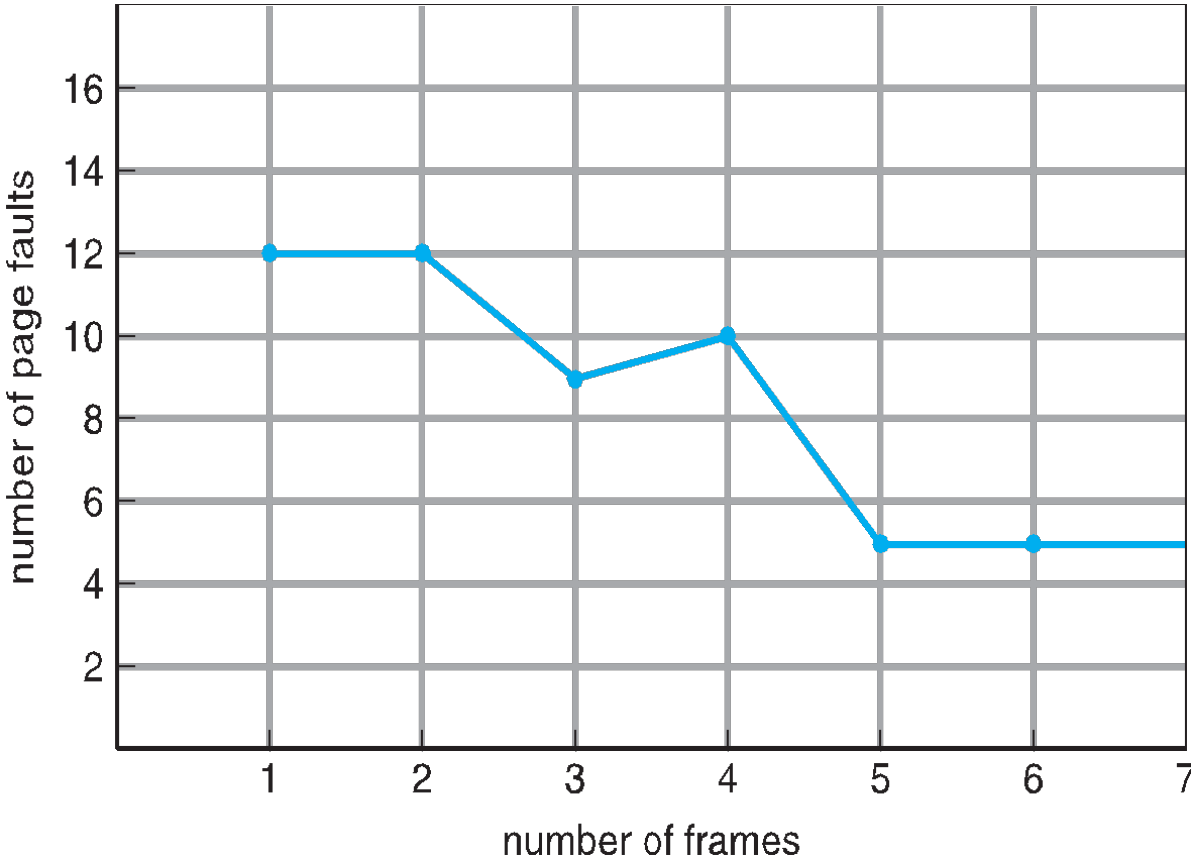
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - 4 **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

OPERATING SYSTEMS

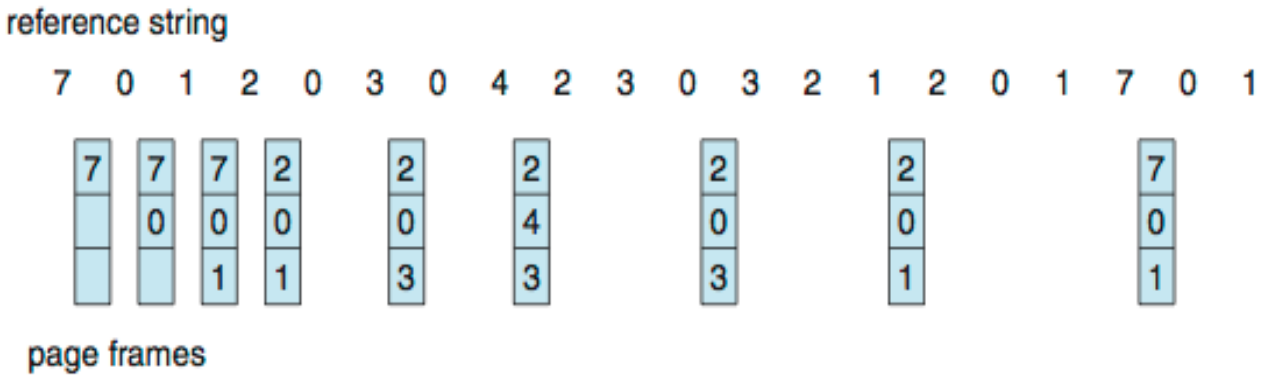
FIFO Illustrating Belady's Anomaly



OPERATING SYSTEMS

Optimal Algorithm

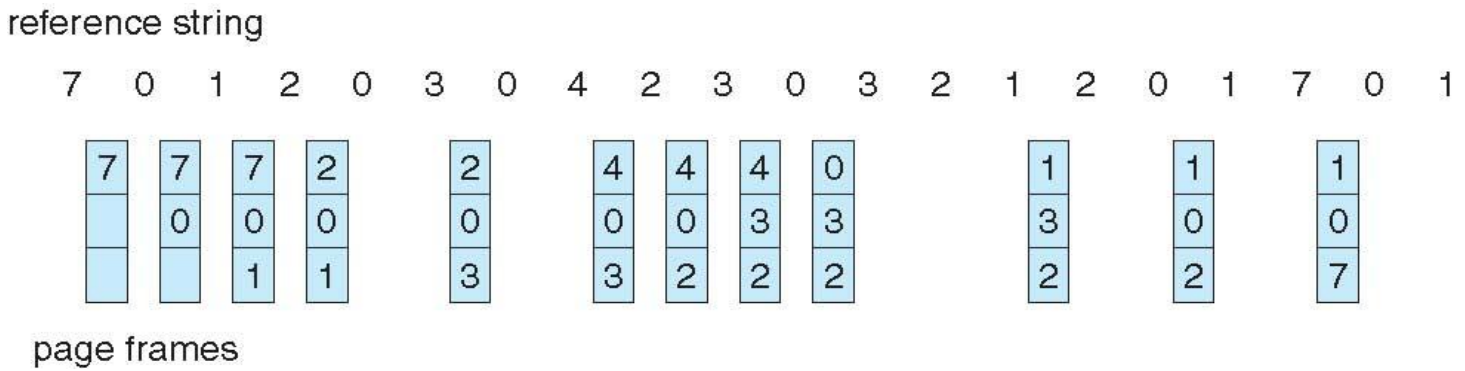
- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can’t read the future
- Used for measuring how well your algorithm performs



OPERATING SYSTEMS

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

OPERATING SYSTEMS

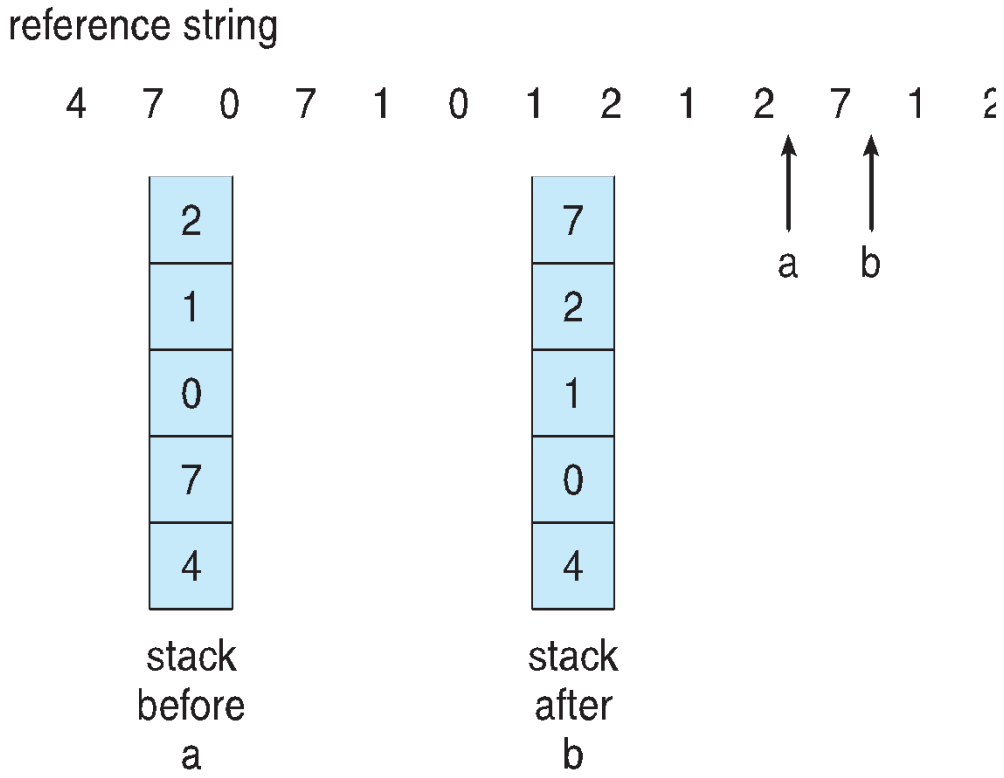
LRU Algorithm (Cont.)

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

OPERATING SYSTEMS

Use Of A Stack to Record Most Recent Page References

- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly
- Use Of A Stack to Record Most Recent Page References



THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Memory Management

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Allocation of Frames

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Allocation of Frames

- How do we allocate the fixed amount of free memory among the various processes?

Example:

- Consider a single-user system with 128 KB of memory composed of pages 1 KB in size. Therefore, this system has 128 frames.
- The operating system may take 35 KB, leaving 93 frames for the user process.
- When a user process started execution, under pure demand paging, it would generate a sequence of page faults.
- The first 93 page faults would all get free frames from the free-frame list.
- When the free-frame list is exhausted, a page-replacement algorithm would be used.
- When the process is terminated, the 93 frames would once again be placed on the free-frame list

Allocation of Frames

Minimum number of frames:

- One reason for allocating at least a minimum number of frames involves performance.
- As the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution
- When a page fault occurs before an executing instruction is complete, the instruction must be restarted. Consequently, we must have enough frames to hold all the different pages that any single instruction can reference
- The minimum number of frames is defined by the computer architecture.
- The maximum number is defined by the amount of available physical memory.
- In between, we are still left with significant choice in frame allocation

OPERATING SYSTEMS

Frame Allocation Algorithms

Equal allocation:

- The easiest way to split m frames among n processes is to give everyone an equal share, **m/n** frames (ignoring frames needed by the operating system)

  Example:

  - If there are 93 frames and 5 processes, each process will get 18 frames.
  - The three leftover frames can be used as a free-frame buffer pool.

Frame Allocation Algorithms

Proportional allocation:

- Allocate available memory to each process according to its size
- Let the size of the virtual memory for process p_i be s_i and define $S = \sum s_i$
- If the total number of available frames is m, allocate a_i frames to process p_i , where a_i is approximately $a_i = s_i / S \times m$
- Adjust each a_i to be an integer that is greater than the minimum number of frames required by the instruction set, with a sum not exceeding m
- **Example**: To allocate 62 frames between two processes, one of 10 pages and one of 127 pages
  - $10/137 \times 62 \approx 4$
  - $127/137 \times 62 \approx 57$

OPERATING SYSTEMS

Global versus Local Allocation

- Page-replacement algorithms can be classified into two broad categories: global replacement and local replacement
- **Global replacement** allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another.
- **Local replacement** requires that each process select from only its own set of allocated frames.
- **Example**: Consider a global replacement scheme wherein high-priority processes are allowed to select frames from low-priority processes for replacement.
- A process can select a replacement from among its own frames or the frames of any lower-priority process. This approach allows a high-priority process to increase its frame allocation at the expense of a low-priority process
- With a local replacement strategy, the number of frames allocated to a process does not change.
- Global replacement generally results in greater system throughput and is therefore the more commonly used method.

OPERATING SYSTEMS

Non-uniform Memory Access

- In systems with multiple CPUs, a given CPU can access some sections of main memory faster than it can access others.
- These performance differences are caused by how CPUs and memory are interconnected in the system.
- Systems in which memory access times vary significantly are known collectively as **non-uniform memory access (NUMA)** systems, and without exception, they are slower than systems in which memory and CPUs are located on the same motherboard
- Managing which page frames are stored at which locations can significantly affect performance in NUMA systems. If we treat memory as uniform in such a system, CPUs may wait significantly longer for memory access than if we modify memory allocation algorithms to take NUMA into account.

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Memory Management

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Virtual Memory

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

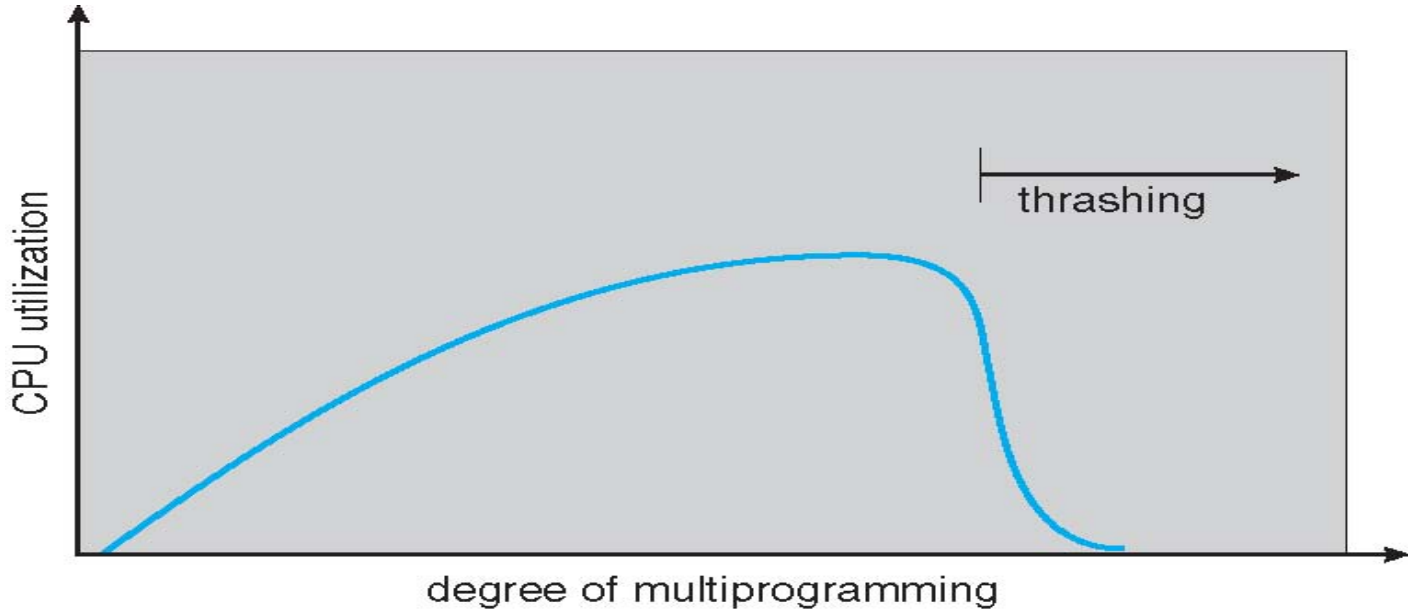
1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out

Cause of Thrashing

- Thrashing results in severe performance problems.
- Consider the scenario of the paging systems performance.
- As the degree of multiprogramming increases, CPU utilization also increases.
- Why there is a decrease in the CPU Utilization when the degree of multiprogramming is increased?.
- How this will be handled?

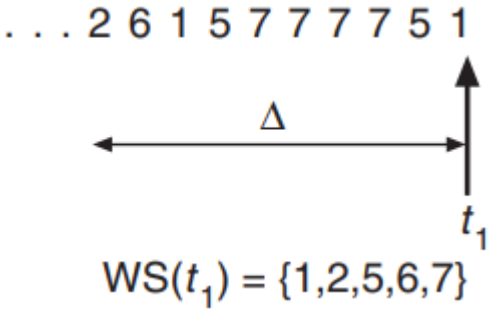


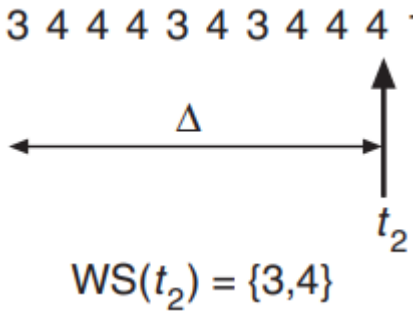
Cause of Thrashing contd…

- We can Limit the effects of thrashing by using a **local replacement algorithm** (or priority replacement algorithm)
  - If process starts thrashing, it cannot steal frames from another processes
- how does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?
  Σ size of locality > total memory size
  - Limit effects by using local or priority page replacement

Keeping Track of the Working Set

- The working-set model is based on the assumption of locality.
- The parameter, Δ , defines the working-set window.
- It is examine the most recent Δ page references.
- The set of pages in the most recent Δ page references is the working set.
- If a page is in active use, it will be in the working set.
- If it is no longer being used, it will drop from the working set time units after its last reference.
- Thus, the working set is an approximation of the program's locality
- Approximate the **W-S** model with interval timer + a reference bit





Keeping Track of the Working Set

- Example: Δ = 10,000 references
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
  - We cannot tell where, within an interval of 5000, a reference occurred.
- Improvement = 10 bits and interrupt every 1000 time units but overhead to service more frequent interrupts

OPERATING SYSTEMS

Working-Set Model

- Parameter $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references

  Example: 10,000 instructions

- WSS_i (working set of Process P_i) =

  total number of pages referenced in the most recent Δ (parameter that varies in time)

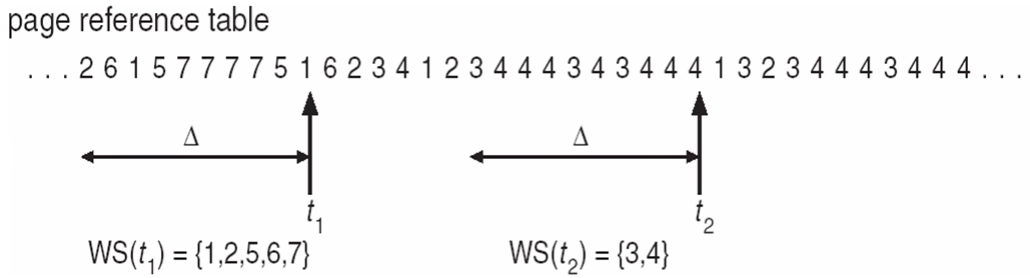
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program

- $D = \Sigma\ WSS_i \equiv$ total demand frames

  - Approximation of locality

- if $D > m$ *(available frames)* $\Rightarrow$ Thrashing will occur

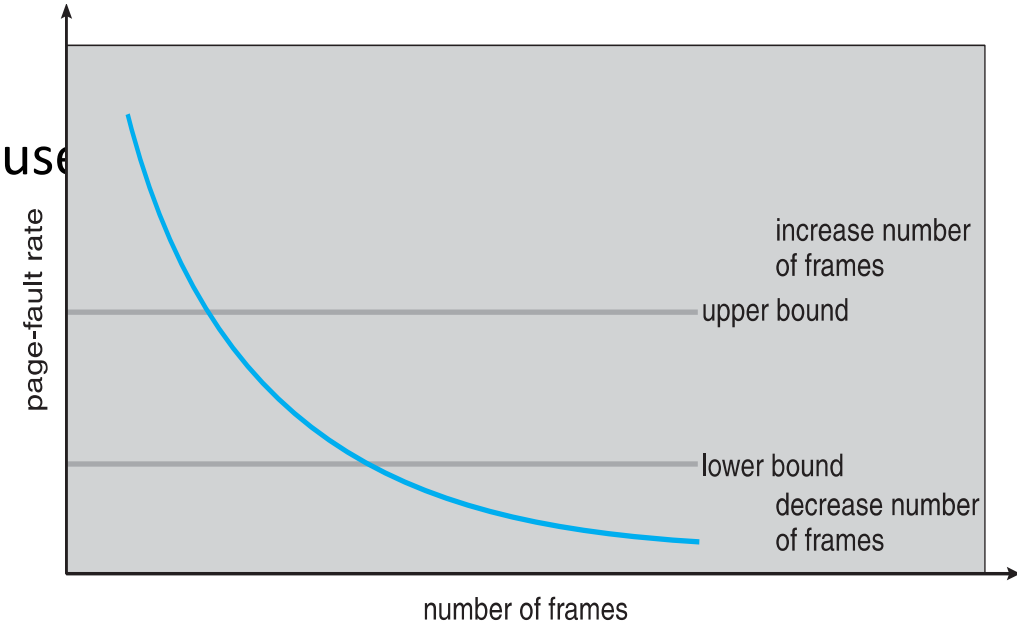
- Policy if $D >$ m, then suspend or swap out one of the processes



OPERATING SYSTEMS

Page-Fault Frequency

- More direct approach than W-S model
  - Control the page-fault rate
  - Looks clumsy to control thrashing
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame.
  - processes can be suspended if the free frames is zero
- With the working-set strategy, we may have to swap out a process.
  - If page fault Increases and no free frames available



THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Memory Management

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Virtual Memory – Case Study

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Virtual Memory Management in Windows

- Windows implements virtual memory using demand paging with clustering
- Clustering handles page faults by bringing in not only the faulting page but also several pages following the faulting page.
- When a process is first created, it is assigned a working-set minimum and maximum.
- The **working-set minimum** is the minimum number of pages the process is guaranteed to have in memory
- If sufficient memory is available, a process may be assigned as many pages as its **working-set maximum.**
- The virtual memory manager maintains a list of free page frames. Associated with this list is a threshold value that is used to indicate whether sufficient free memory is available.

OPERATING SYSTEMS

Virtual Memory Management in Windows

- If a page fault occurs for a process that is below its working-set maximum, the virtual memory manager allocates a page from this list of free pages.
- If a process that is at its working-set maximum incurs a page fault, it must select a page for replacement using a local LRU page-replacement policy
- When the amount of free memory falls below the threshold, the virtual memory manager uses a tactic known as **automatic working-set trimming** to restore the value above the threshold.
- Automatic working-set trimming works by evaluating the number of pages allocated to processes.
- If a process has been allocated more pages than its working-set minimum, the virtual memory manager removes pages until the process reaches its working-set minimum.
- A process that is at its working-set minimum may be allocated pages from the free-page-frame list once sufficient free memory is available. Windows performs working-set trimming on both user mode and system processes

OPERATING SYSTEMS

Virtual Memory Management in Linux Reference

- Linux uses demand paging to load executable images into a processes virtual memory.
- Whenever a command is executed, the file containing it is opened and its contents are mapped into the processes virtual memory. This is done by modifying the data structures describing this processes memory map and is known as *memory mapping*.
- However, only the first part of the image is actually brought into physical memory. The rest of the image is left on disk.
- As the image executes, it generates page faults and Linux uses the processes memory map in order to determine which parts of the image to bring into memory for execution.
- Linux on Alpha AXP systems uses 8 Kbyte pages and on Intel x86 systems it uses 4 Kbyte pages. Each of these pages is given a unique number; the **page frame number** (PFN).

OPERATING SYSTEMS

Virtual Memory Management in Linux

- In this paged model, a virtual address is composed of two parts; an offset and a virtual page frame number.
- If the page size is 4 Kbytes, bits 11:0 of the virtual address contain the offset and bits 12 and above are the virtual page frame number.
- Each time the processor encounters a virtual address it must extract the offset and the virtual page frame number.
- The processor must translate the virtual page frame number into a physical one and then access the location at the correct offset into that physical page. To do this the processor uses *page tables*.
- The page table is accessed using the virtual page frame number as an offset
- Linux uses a Least Recently Used (LRU) page aging technique to fairly choose pages which might be removed from the system.

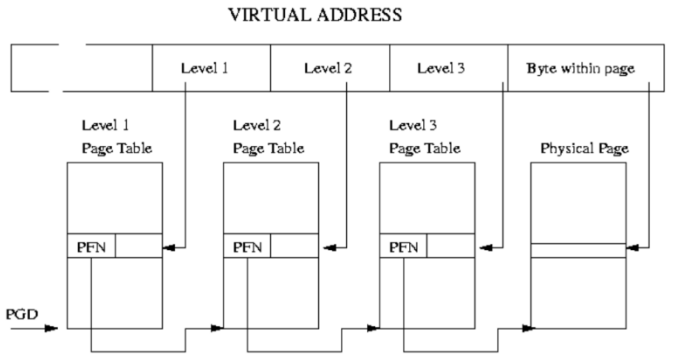


Figure 3.3: Three Level Page Tables

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu