

CS343 - Operating Systems

Module-4A

Introduction to Memory Management



Dr. John Jose

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam.

<http://www.iitg.ac.in/johnjose/>

Overview of Memory Management

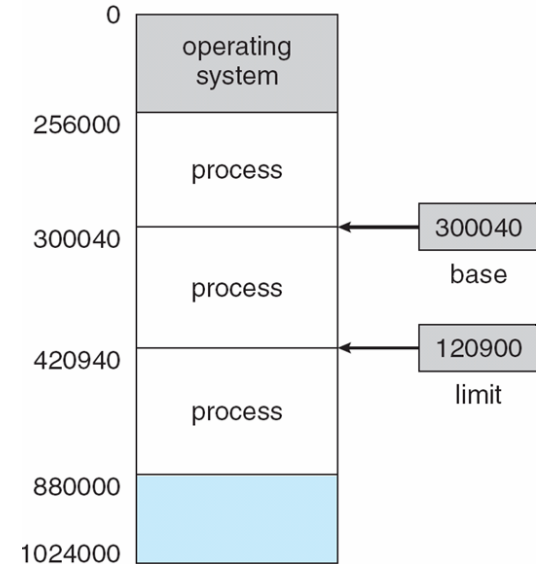
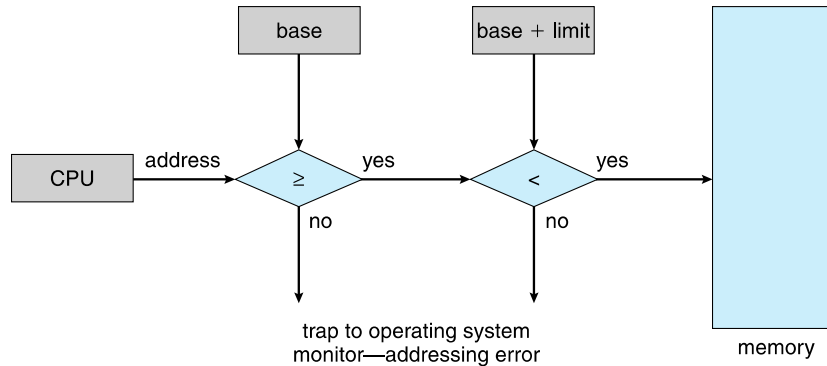
- ❖ Background
- ❖ Swapping
- ❖ Contiguous Memory Allocation
- ❖ Segmentation
- ❖ Paging
- ❖ Structure of the Page Table

Background

- ❖ Program must be brought (from disk) into memory and placed within a process for it to be run
- ❖ Main memory and registers are only storage CPU can access directly
- ❖ Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- ❖ Register access in one CPU clock
- ❖ Main memory can take many cycles, causing a **stall**
- ❖ **Cache** sits between main memory and CPU registers
- ❖ Protection of memory required to ensure correct operation

Hardware Protection using Base and Limit Registers

- ❖ 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



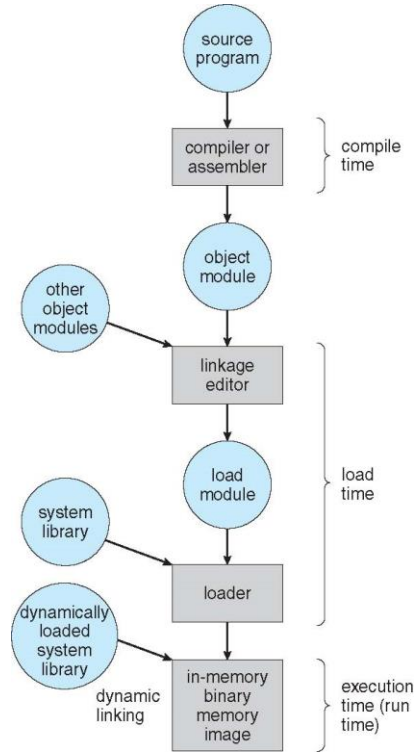
Address Binding

- ❖ Programs on disk, ready to be brought into memory to execute.
- ❖ Without support, must be loaded into address 0000
- ❖ Inconvenient to have user process physical address always at 0000
- ❖ Addresses represented in different ways in a program's life
 - ❖ Source code addresses usually symbolic
 - ❖ Compiled code addresses **bind** to relocatable addresses
 - ❖ i.e. "14 bytes from beginning of this module"
 - ❖ Linker or loader will bind relocatable addresses to absolute addresses
 - ❖ i.e. 74014

Binding of Instructions and Data to Memory

- ❖ Address binding of instructions and data to memory addresses can happen at three different stages
 - ❖ **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
 - ❖ **Load time:** Must generate **relocatable code** if memory location is not known at compile time
 - ❖ **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another
 - ❖ Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



Logical vs. Physical Address Space

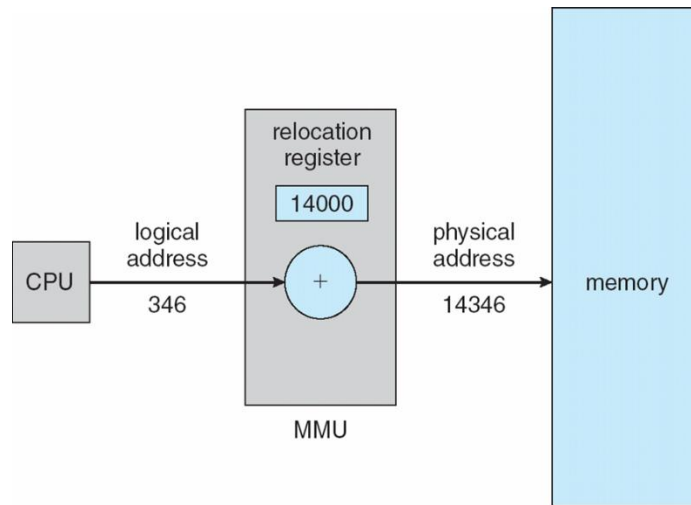
- ❖ **Logical address** – generated by the CPU; also referred to as **virtual address**
- ❖ **Physical address** – address seen by the memory unit
- ❖ Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in 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

Memory-Management Unit (MMU)

- ❖ Hardware device that at run time maps virtual to physical address
- ❖ To start, consider simple scheme where the value in the **relocation register** is added to every address generated by a user process at the time it is sent to memory
 - ❖ Base register now called **relocation register**
- ❖ 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

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



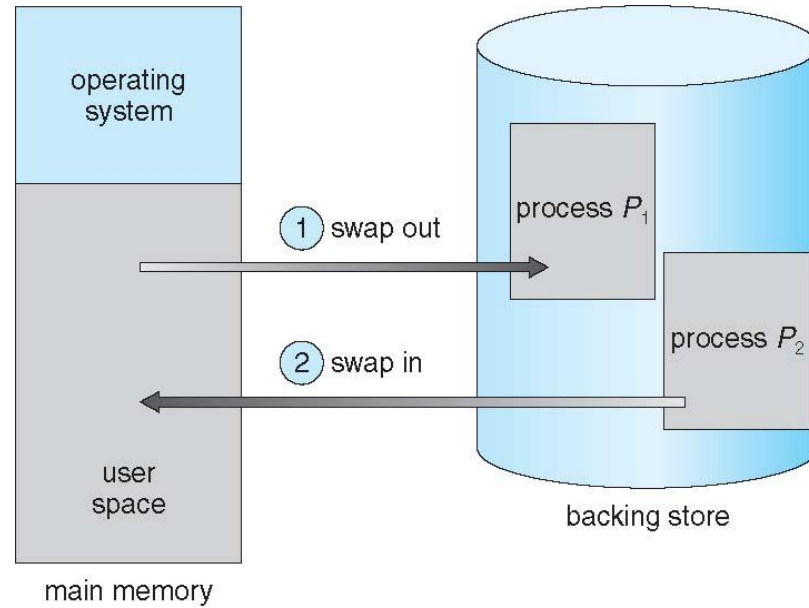
Dynamic Linking

- ❖ **Static linking** – system libraries and program code combined by the loader into the binary program image
- ❖ Dynamic linking –linking postponed until execution time
- ❖ 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
- ❖ System also known as **shared libraries**


Swapping

- ❖ A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
- ❖ Total virtual 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

Swapping



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
- ❖ 100MB process swapping to hard disk with transfer rate of 50MB/sec
 - ❖ Swap out time of 2000 ms
 - ❖ Plus swap in of same sized process
 - ❖ Total context switch swapping component time of 4000ms (4 seconds)
- ❖ Can reduce if reduce size of memory swapped – by knowing how much memory really being used
 - ❖ System calls to  OS: `request_memory()` and `release_memory()`

Swapping on Mobile Systems

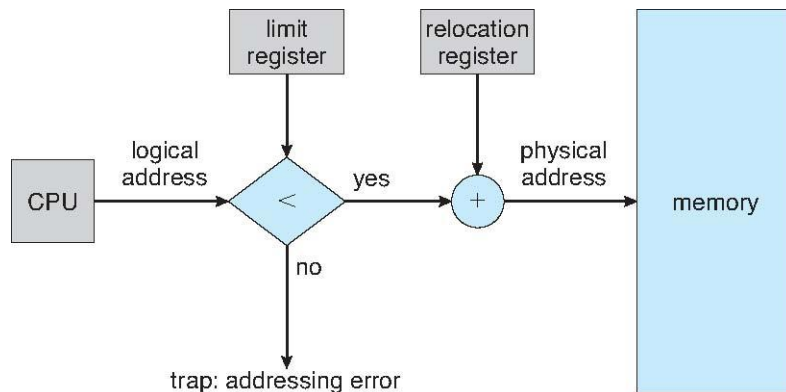
- ❖ Not typically supported in flash memory based systems
 - ❖ Small amount of space
 - ❖ Limited number of write cycles
 - ❖ Poor throughput between flash memory and CPU
- ❖ iOS **asks** apps to voluntarily relinquish allocated memory
 - ❖ Read-only data thrown out and reloaded from flash if needed
 - ❖ Failure to free can result in termination
- ❖ Android terminates apps if low free memory, but first writes **application state** to flash for fast restart

Contiguous Allocation

- ❖ Main memory must support both OS and user processes
- ❖ Limited resource, must allocate efficiently
- ❖ Contiguous allocation is one early method
- ❖ Main memory has 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 contiguous section of memory

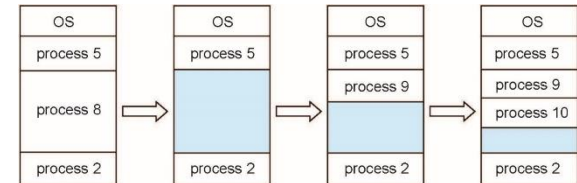
Contiguous Allocation

- ❖ 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*



Multiple-partition allocation

- ❖ Multiple-partition allocation
 - ❖ Degree of multiprogramming limited by number of partitions
 - ❖ **Variable-partition** sizes for efficiency as per size of process
 - ❖ **Hole** – block of available memory; holes of various size are scattered throughout memory
 - ❖ 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)



Dynamic Storage-Allocation Problem

- ❖ How to satisfy a request of size n from a list of free holes?
- ❖ **First-fit**: Allocate the **first** hole that is big enough
- ❖ **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

Fragmentation

- ❖ **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- ❖ **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- ❖ First fit analysis reveals that given N blocks allocated, $0.5 N$ blocks lost to fragmentation
 - ❖ $1/3$ may be unusable -> **50-percent rule**

Fragmentation

- ❖ 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
- ❖ Now consider that backing store has same fragmentation problems

Thank you

johnjose@iitg.ac.in

<http://www.iitg.ac.in/johnjose/>



CS343 - Operating Systems

Module-4B

Segmentation and Paging



Dr. John Jose

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam.

<http://www.iitg.ac.in/johnjose/>

Overview of Memory Management

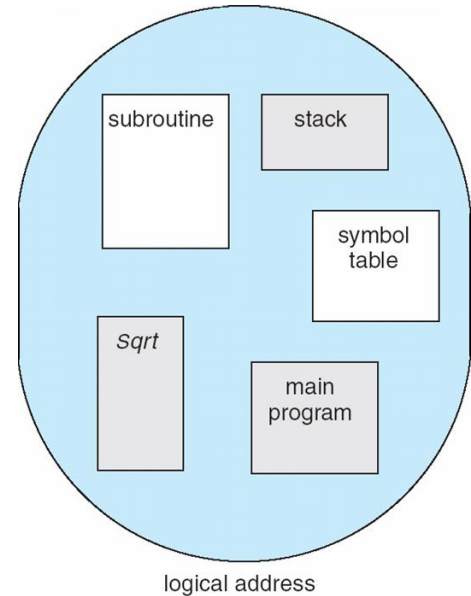
- ❖ Background
- ❖ Swapping
- ❖ Contiguous Memory Allocation
- ❖ Segmentation
- ❖ Paging
- ❖ Page Table

Principle of Locality

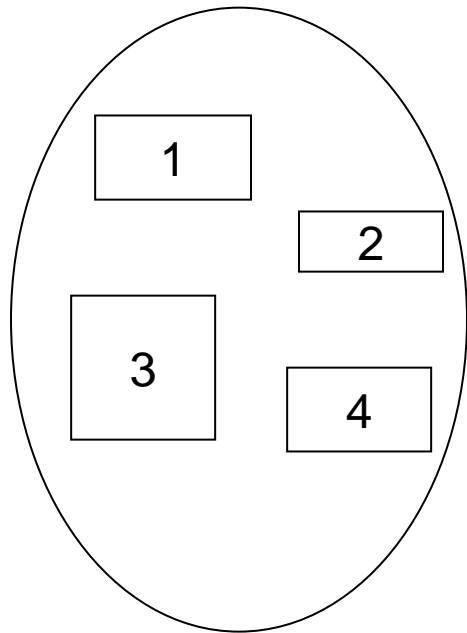
- ❖ Program and data references within a process tend to cluster around an address space. Only a few pieces of a process will be needed over a short period of time
- ❖ This help us to make intelligent guesses about which pieces will be needed in the future
- ❖ This suggests that virtual memory may work efficiently
- ❖ To implement virtual memory, hardware must support paging and segmentation
- ❖ OS must do the movement of pieces of process between main and secondary memory
- ❖ Physical address space of a process can be noncontiguous; - avoids external fragmentation

Segmentation

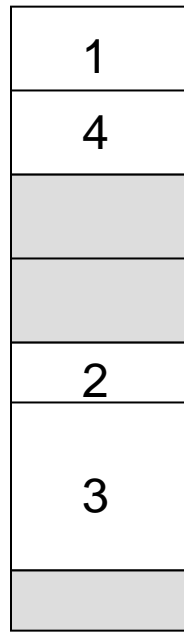
- ❖ Memory-management scheme that supports user view of memory
- ❖ A program is a collection of segments
 - ❖ A segment is a logical unit such as:
- ❖ main program
- ❖ procedure
- ❖ function
- ❖ method
- ❖ object
- ❖ local variables,
- ❖ global variables
- ❖ common block
- ❖ stack
- ❖ symbol table
-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



Logical View of Segmentation



user space

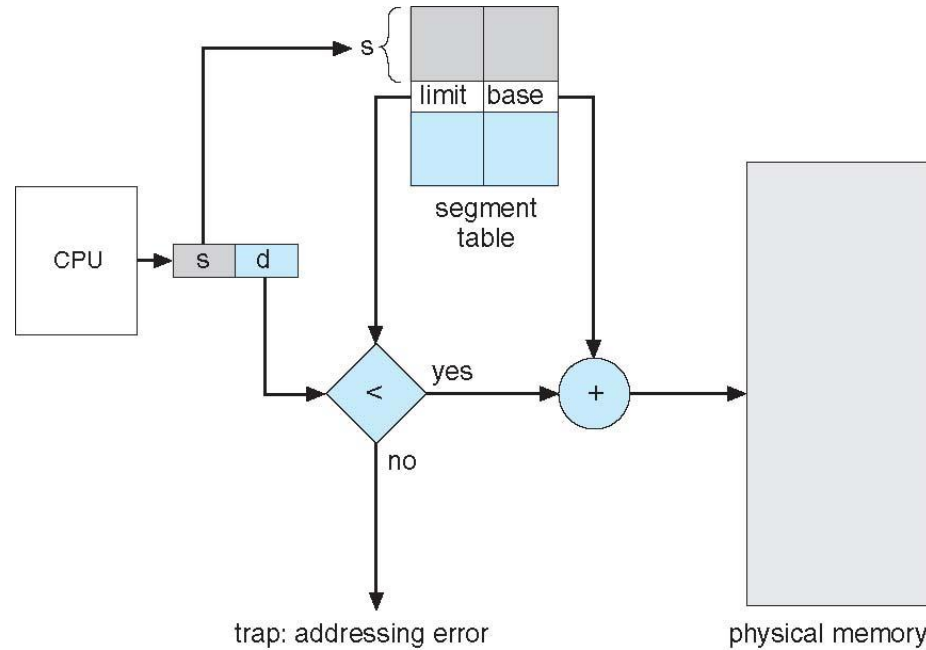


physical memory space

Segmentation Architecture

- ❖ Logical address consists of a two tuple: $\langle \text{segment-number}, \text{offset} \rangle$,
- ❖ **Segment table** – maps two-dimensional physical addresses; each table entry has:
 - ❖ **base** – contains the starting physical address where the segments reside in memory
 - ❖ **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 Hardware

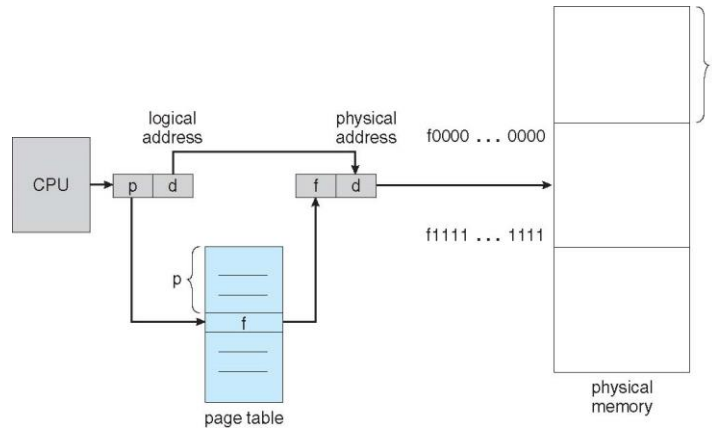


Paging

- ❖ Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
 - ❖ Avoids external fragmentation
 - ❖ Avoids problem of varying sized memory chunks
- ❖ 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**
- ❖ Keep track of all free frames
- ❖ To run a program of size **N** pages, need to find **N** free frames
- ❖ Set up a **page table** to translate logical to physical addresses

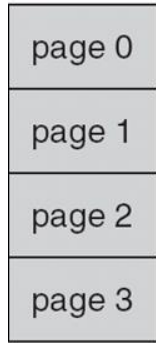
Address Translation Scheme

- ❖ Address generated by CPU is divided into:
 - ❖ **Page number** (p) – used as an index into a page table
 - ❖ **Page offset** (d) – displacement within a page



page number	page offset
p	d
m - n	n

Paging Model and Page Tables

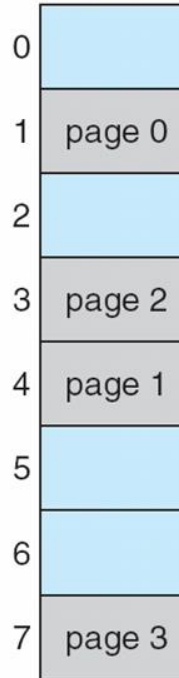


logical
memory

0	1
1	4
2	3
3	7

page table

frame
number



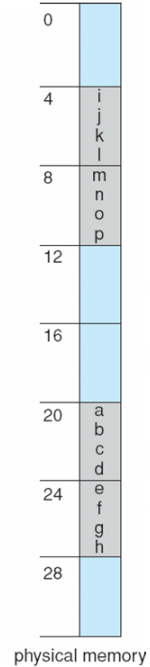
physical
memory

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

0	5
1	6
2	1
3	2

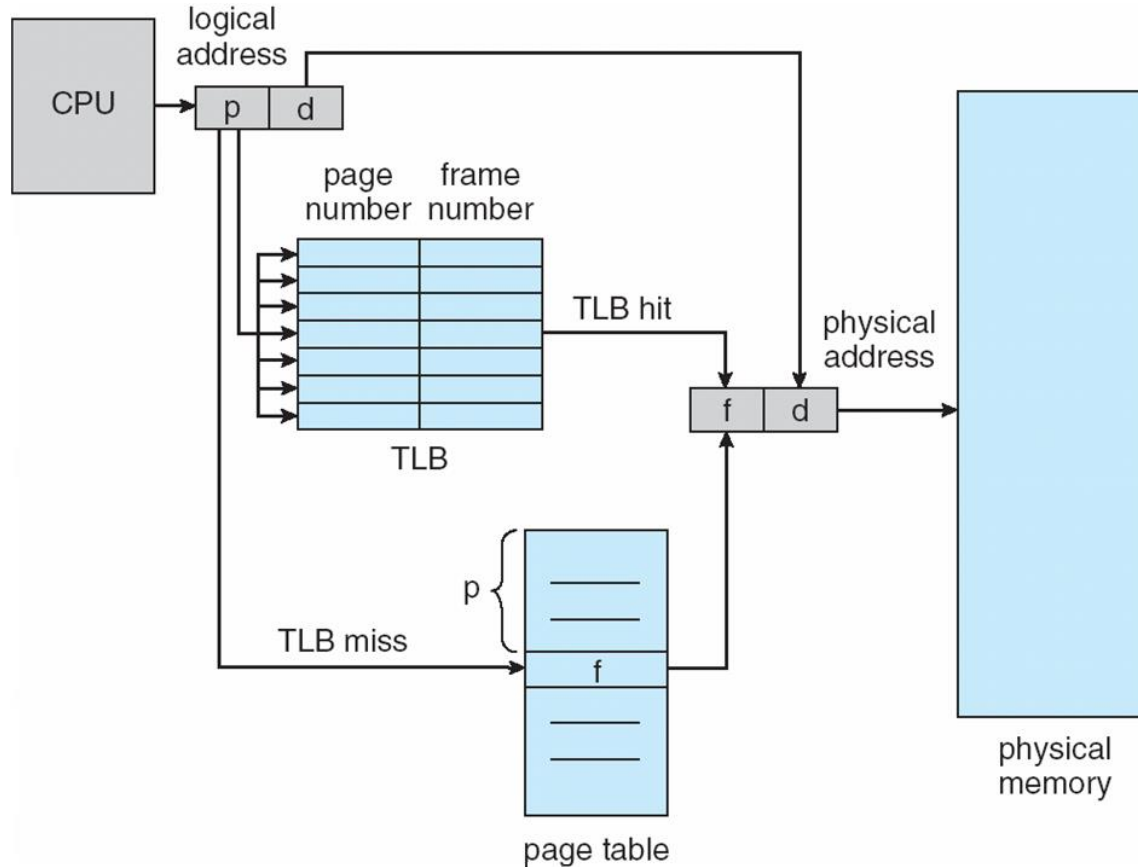
page table



Implementation of 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)**

Paging Hardware With TLB



Thank you

johnjose@iitg.ac.in

<http://www.iitg.ac.in/johnjose/>



CS343 - Operating Systems

Module-4C

Page Table Implementation



Dr. John Jose

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam.

<http://www.iitg.ac.in/johnjose/>

Overview of Memory Management

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

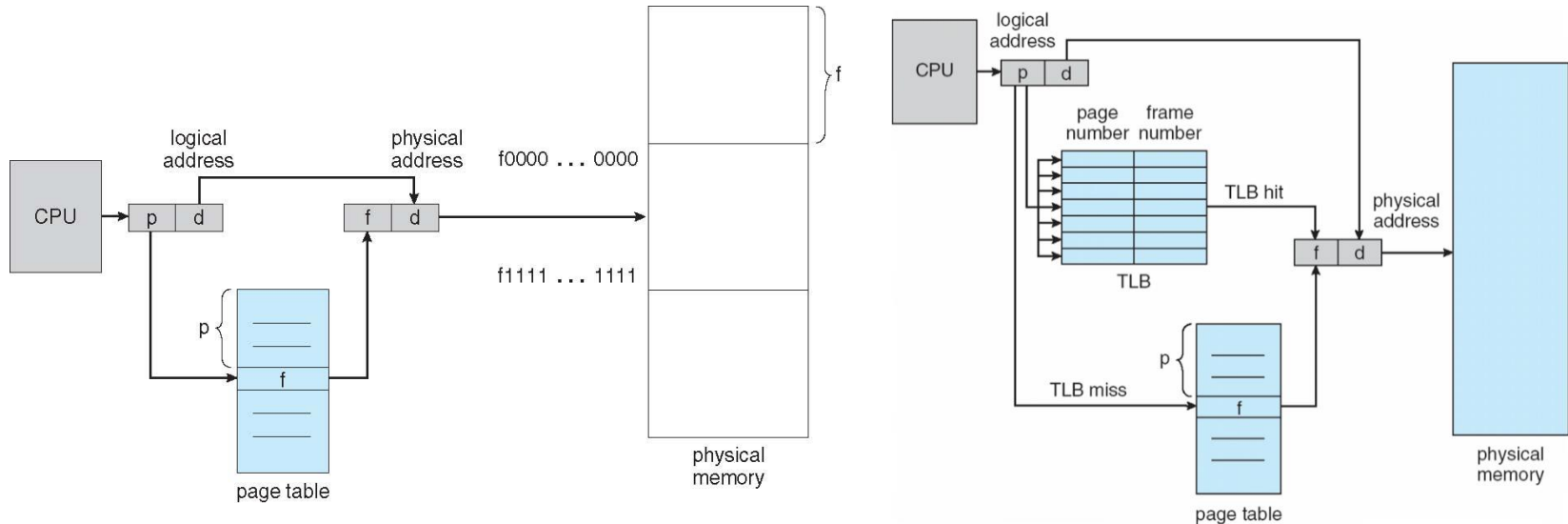
Paging & Address Translation Scheme

❖ Address generated by CPU is divided into:

❖ **Page number** (p) – used as an index into a page table

❖ **Page offset** (d) – displacement within a page

page number	page offset
p	d
m - n	n



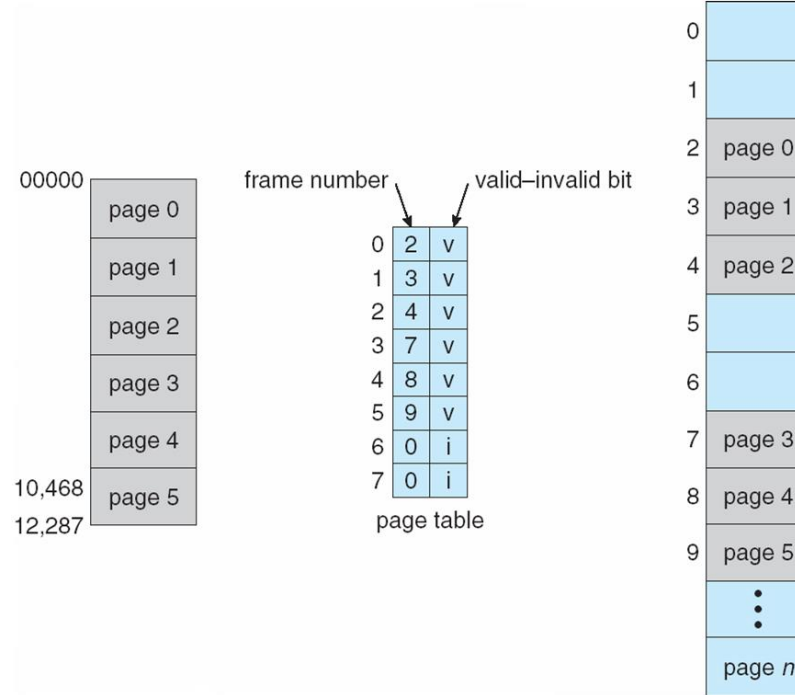
Implementation of Page Table

- ❖ 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
- ❖ Frequently used page table entries are kept in a fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**

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
- ❖ Any violations result in a trap to the kernel

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



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 address space / memory for page table alone
- ❖ Hierarchical Paging
- ❖ Hashed Page Tables
- ❖ Inverted Page Tables

Hierarchical Page Tables

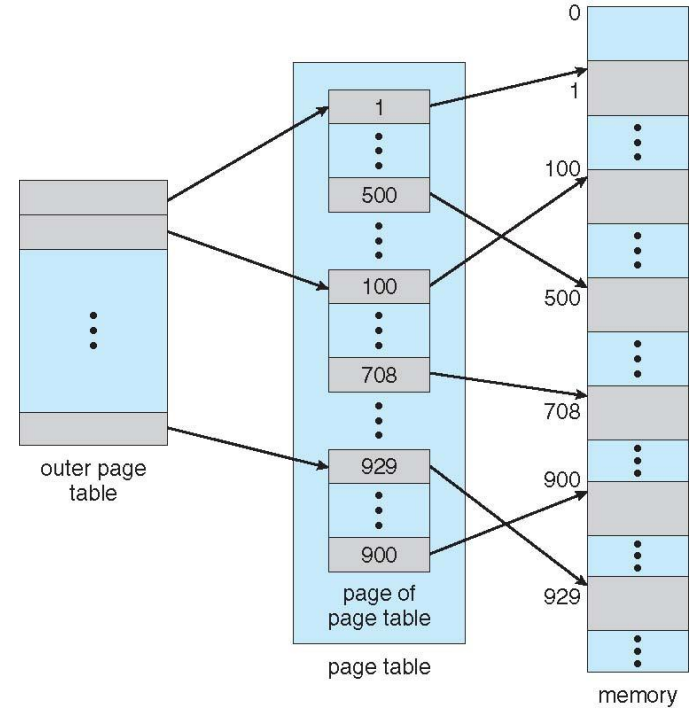
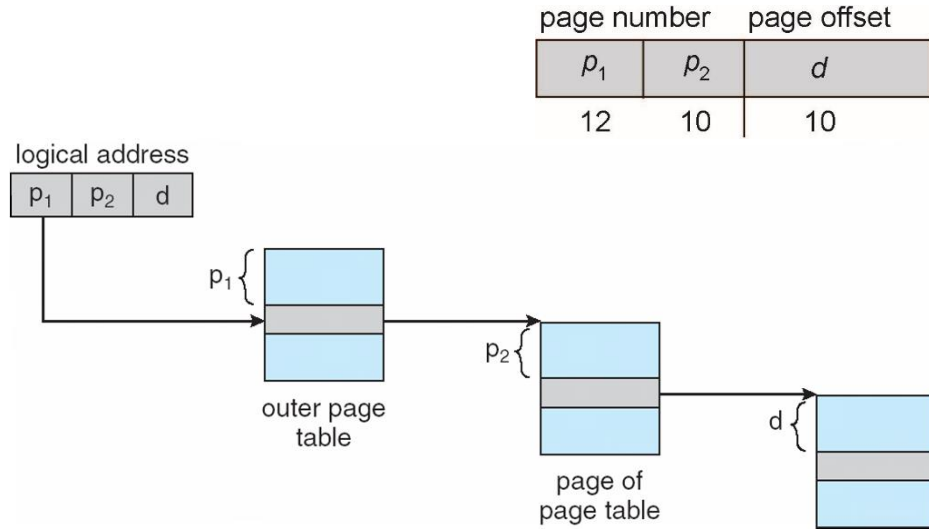
- ❖ Break up the logical address space into multiple page tables
- ❖ A simple technique is a two-level page table
- ❖ We then page the page table

Simple Two-Level Paging

- ❖ 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

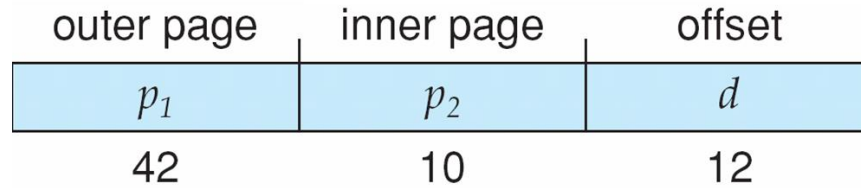
page number		page offset
p_1	p_2	d
12	10	10

Address-Translation Scheme

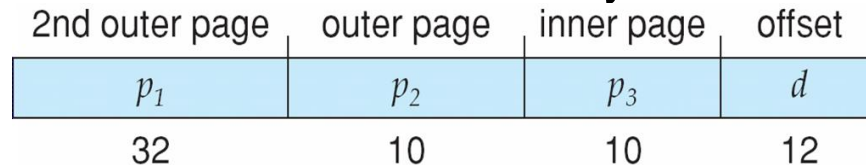


Larger Logical Address Space

- ❖ For 64 bits, even two-level paging scheme not sufficient
- ❖ If page size is 4 KB (2^{12})
 - ❖ Then page table has 2^{52} entries
 - ❖ Address would look like



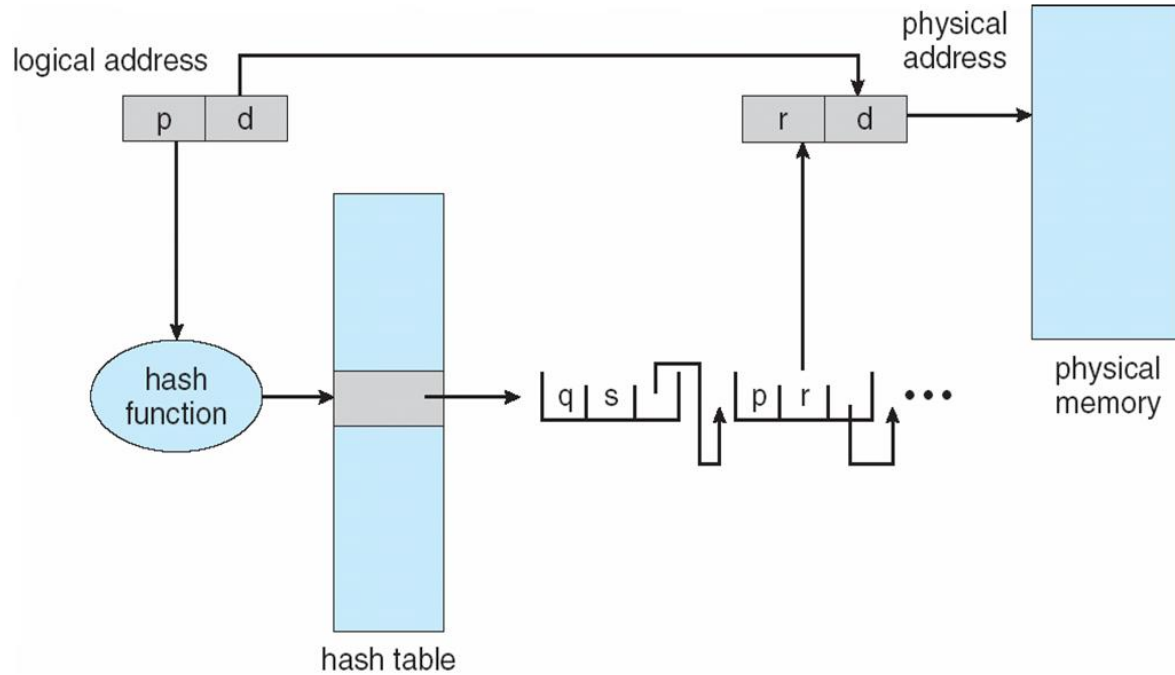
- ❖ Outer page table has 2^{42} entries or 2^{44} bytes



Hashed Page Tables

- ❖ The virtual page number is hashed into a page table
- ❖ This page table contains a chain of elements hashing to the same location
- ❖ Each element contains
 - (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 in this chain searching for a match
- ❖ If a match is found, the corresponding physical frame is extracted

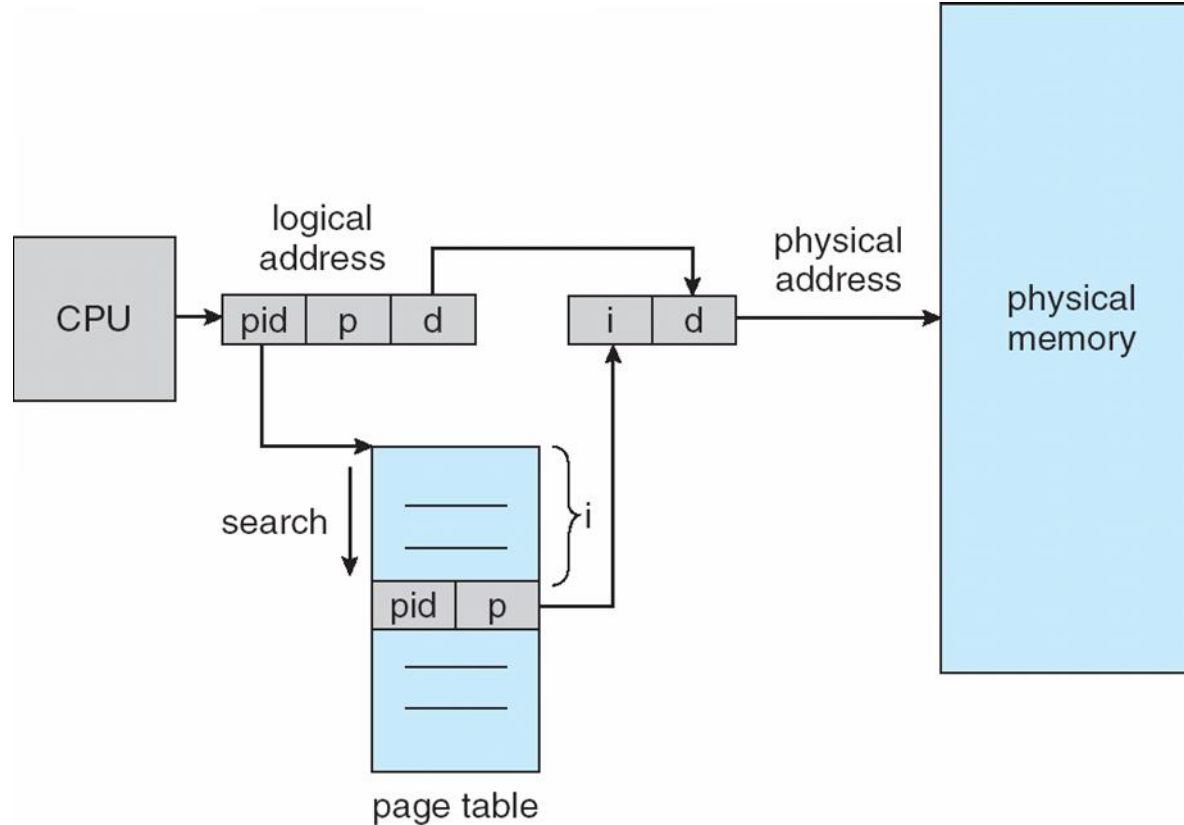
Hashed Page Tables



Inverted Page Table

- ❖ No page table per process
- ❖ 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,
- ❖ Increases time needed to search the table when a page reference occurs

Inverted Page Table Architecture



Shared Pages

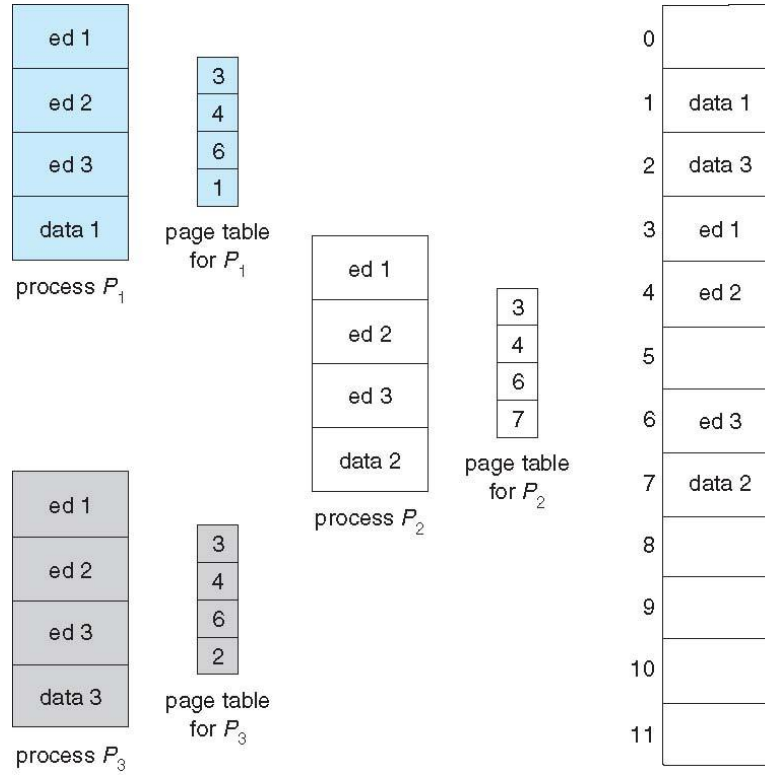
❖ Shared code

- ❖ One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
- ❖ 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

Shared Pages



Thank you

johnjose@iitg.ac.in

<http://www.iitg.ac.in/johnjose/>



CS343 - Operating Systems

Module-4D

Virtual Memory Techniques



Dr. John Jose

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam.

<http://www.iitg.ac.in/johnjose/>

Overview of Memory Management

- ❖ Demand Paging
- ❖ Copy-on-Write
- ❖ Page Replacement
- ❖ Allocation of Frames
- ❖ Thrashing
- ❖ Memory-Mapped Files

Objectives

- ❖ To describe the benefits of a virtual memory system
- ❖ To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- ❖ To discuss the principle of the working-set model
- ❖ To examine the relationship between shared memory and memory-mapped files

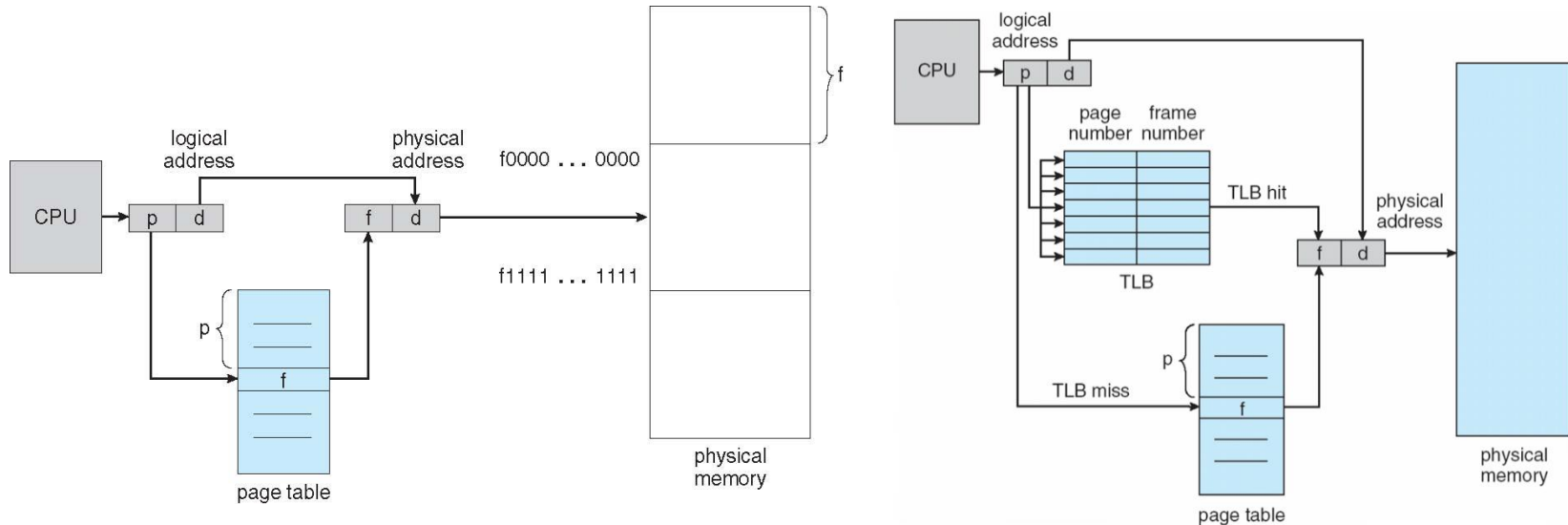
Paging & Address Translation Scheme

❖ Address generated by CPU is divided into:

❖ **Page number** (p) – used as an index into a page table

❖ **Page offset** (d) – displacement within a page

page number	page offset
p	d
m - n	n



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 address space / memory for page table alone
- ❖ Hierarchical Paging
- ❖ Hashed Page Tables
- ❖ Inverted Page Tables

Background

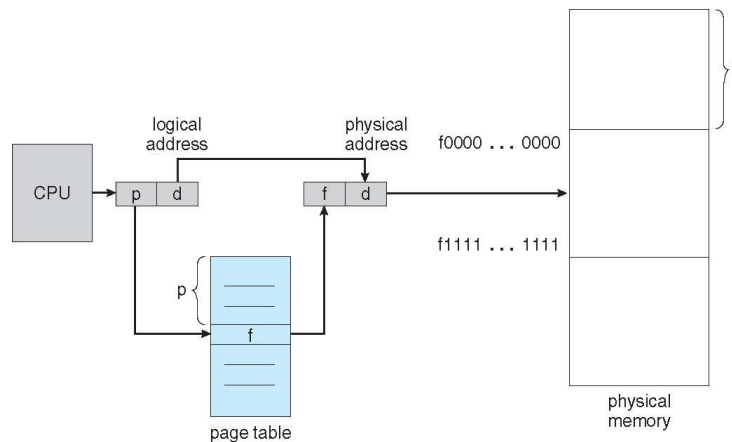
- ❖ Code needs to be in memory to execute, but entire program rarely used
 - ❖ Error code, unusual routines, large data structures
- ❖ Entire program code not needed at same time
- ❖ Consider ability to execute partially-loaded program
 - ❖ Program no longer constrained by limits of physical memory
 - ❖ 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

Background

- ❖ **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
- ❖ More programs running concurrently
- ❖ Less I/O needed to load or swap processes

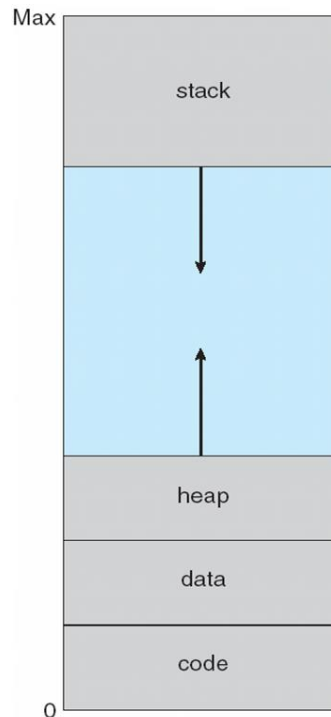
Background

- ❖ **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

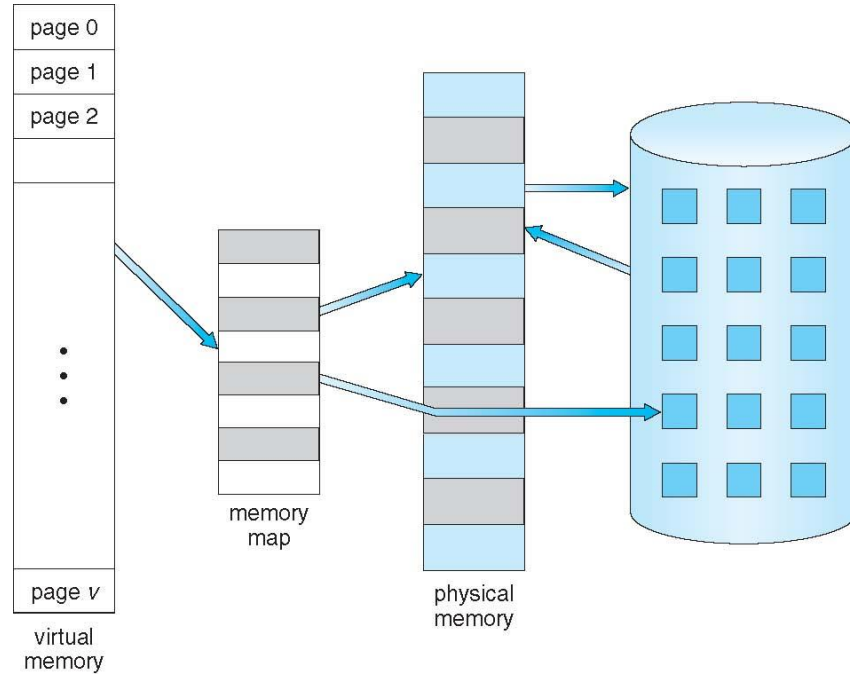


Virtual-address Space

- ❖ Logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
- ❖ Unused address space between stack and heap is hole
- ❖ Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc

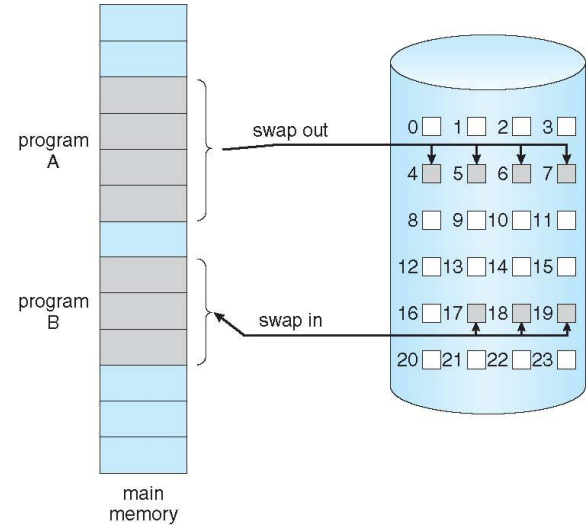


Virtual Memory To Physical Memory Mapping



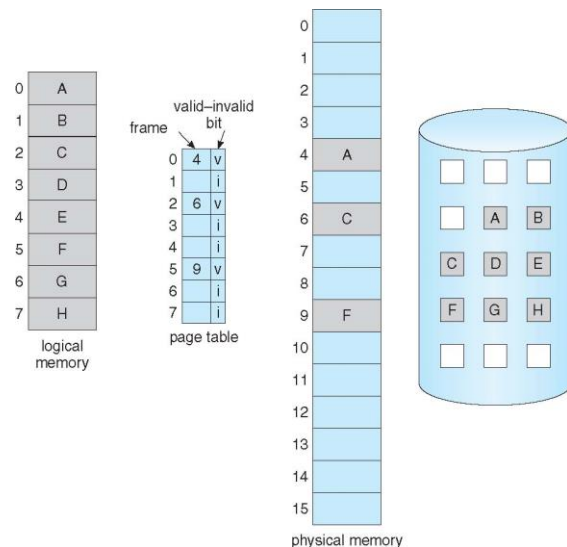
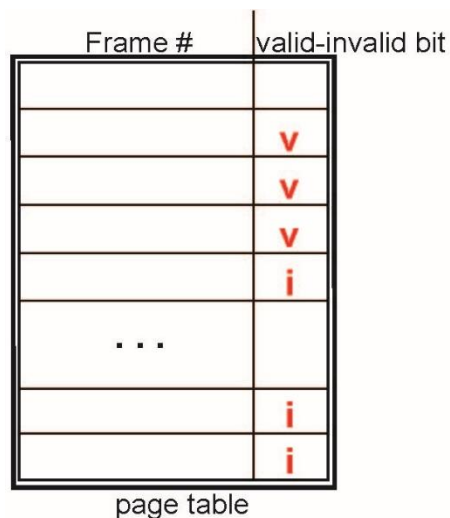
Demand Paging

- ❖ 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
- ❖ **Lazy swapper** – never swaps a page into memory unless page will be needed
 - ❖ Swapper that deals with pages is a **pager**



Valid-Invalid Bit

- ❖ With each page table entry a valid–invalid bit is associated (**v** \Rightarrow in-memory – **memory resident**, **i** \Rightarrow not-in-memory)
- ❖ Initially valid–invalid bit is set to **i** on all entries

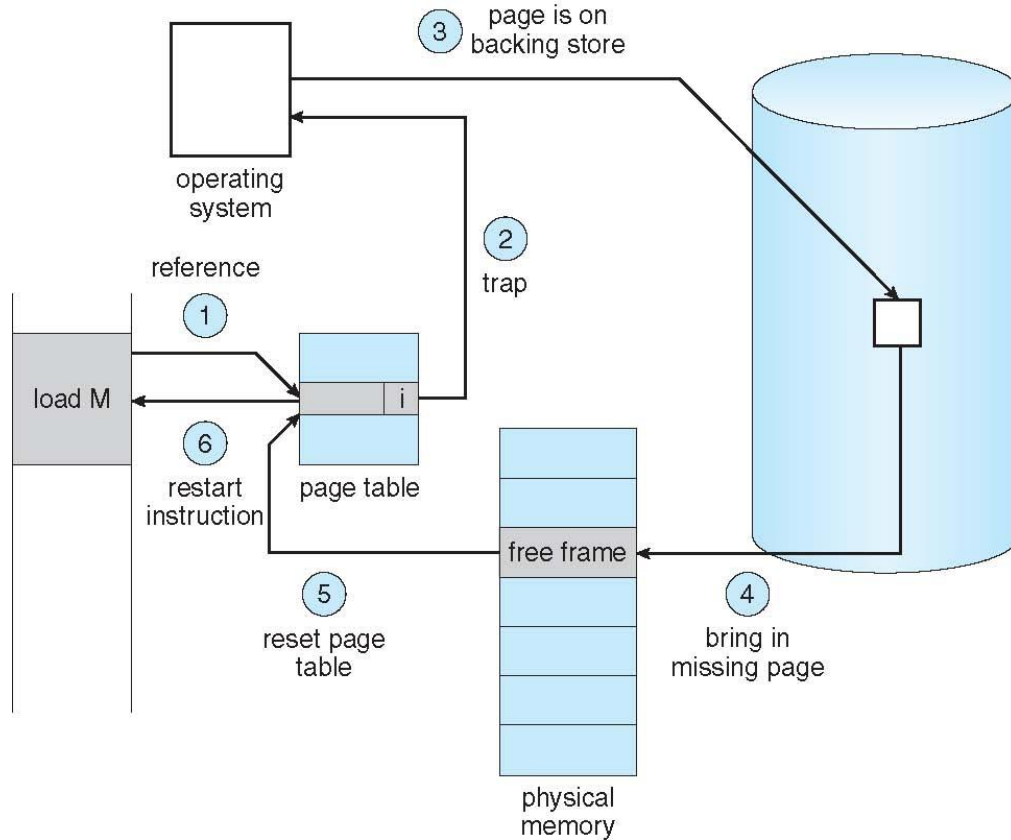


Page Table When Some Pages Are Not in Main Memory

Page Fault

- ❖ If there is a reference to a page, first reference to that page will trap to operating system: **page fault** (page not found in main memory)
- ❖ Find free frame
- ❖ Swap page into frame via scheduled disk operation
- ❖ Reset tables to indicate page now in memory: Set validation bit = **1**
- ❖ Restart the instruction that caused the page fault

Steps in Handling a Page Fault



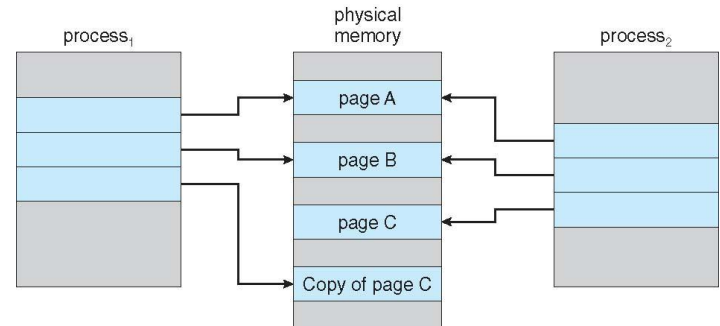
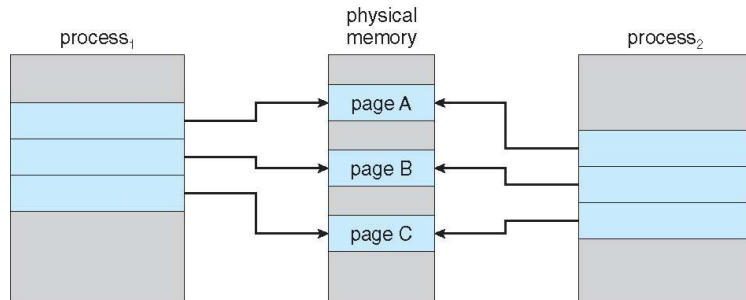
Demand Paging Overhead

- ❖ Page Fault Rate $0 \leq p \leq 1$
 - if $p = 0$ no page faults
 - if $p = 1$, every reference is a fault
- ❖ Effective Access Time (EAT)

$$\text{EAT} = (1 - p) \times \text{memory access} + p (\text{page fault overhead} \\ + \text{swap page out} + \text{swap page in})$$

Copy-on-Write

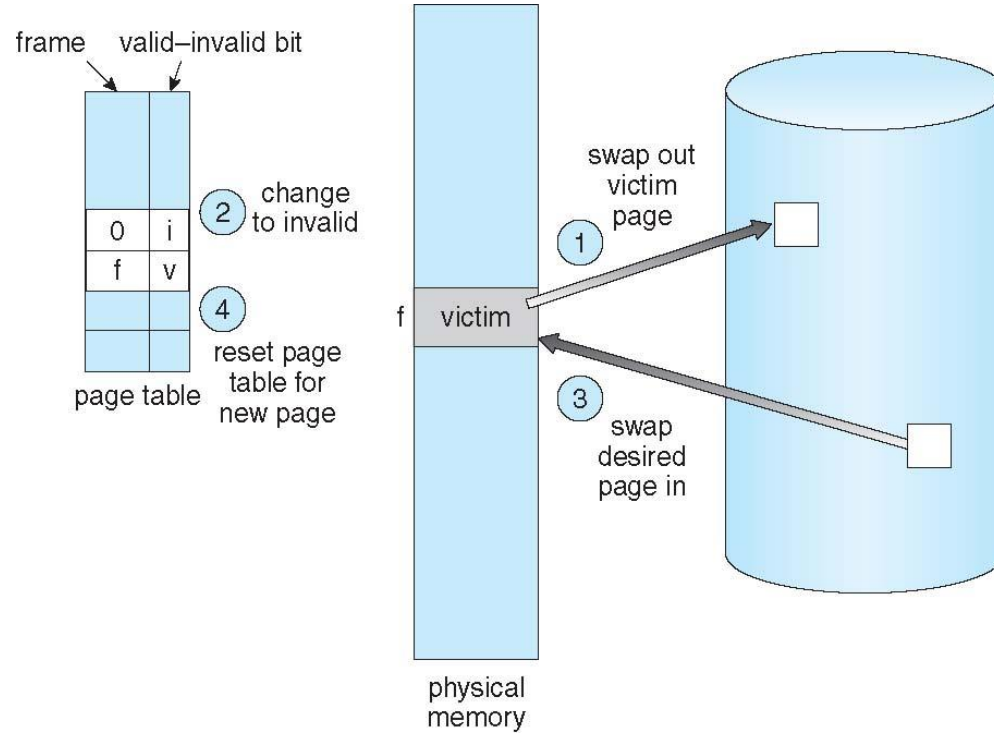
- ❖ **Copy-on-Write** (COW) allows both parent and child processes to initially **share** the same pages in memory
- ❖ If either process modifies a shared page, only then is the page copied
- ❖ COW allows more efficient process creation as only modified pages are copied
- ❖ In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages



Basic Page Replacement

- ❖ Find the location of the desired page on disk
- ❖ 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
- ❖ Bring the desired page into the (newly) free frame; update the page and frame tables
- ❖ Continue the process by restarting the instruction that caused the trap

Page Replacement

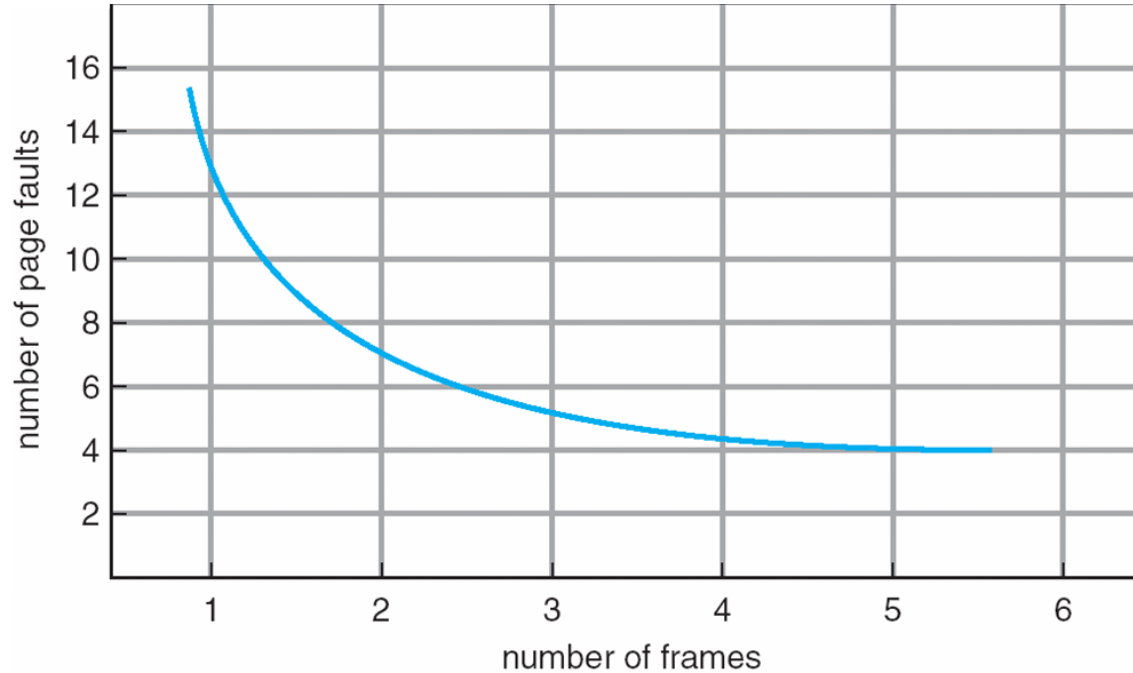


Page Replacement Algorithm

❖ 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
 - ❖ **FIFO, LIFO,**
 - ❖ **Optimal,**
 - ❖ **LRU, LRU approximations,**
 - ❖ **LFU, MFU**

Page Faults Vs Number of Frames



First-In-First-Out (FIFO) Algorithm

- ❖ Ref. 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)

reference string

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

7	7	7	2																		
	0	0	0																		
		1	1																		

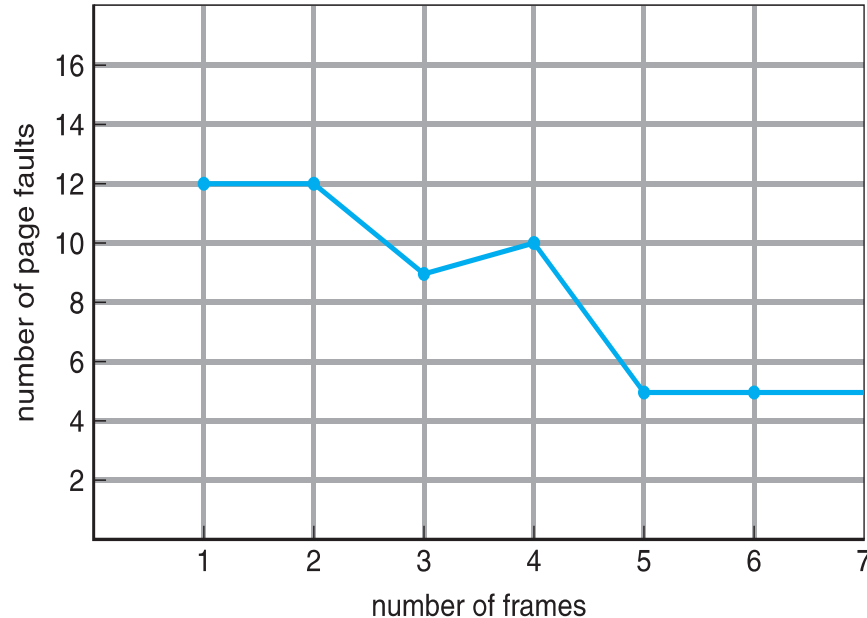
page frames

15 page faults

- ❖ How to track ages of pages? - Use a FIFO queue

Belady's Anomaly

- ❖ Consider 1,2,3,4,1,2,5,1,2,3,4,5
- ❖ Adding more frames can cause more page faults! - Belady's Anomaly



Optimal Algorithm

- ❖ Replace page that will not be used for longest period of time
 - ❖ 9 is optimal for the example
- ❖ Practical difficulty- Can't read the future
- ❖ Used for measuring how well your algorithm performs

reference string

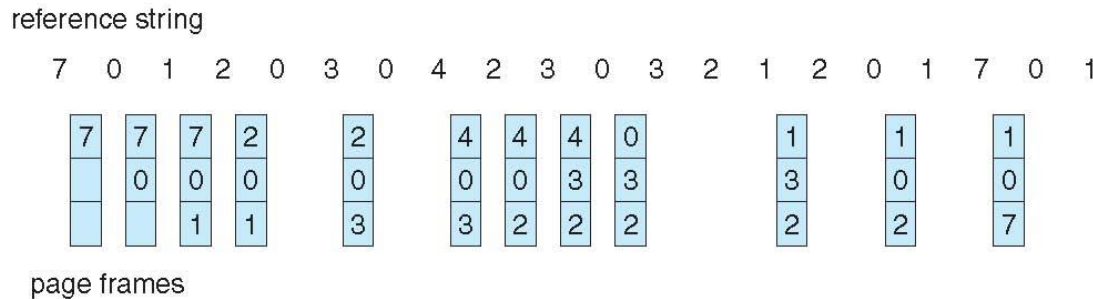
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2				2				2			2				7
	0	0	0				0				0			0				0
		1	1				3				3			1				1

page frames

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

LRU Algorithm Implementation

❖ 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.

LRU Approximation Algorithms

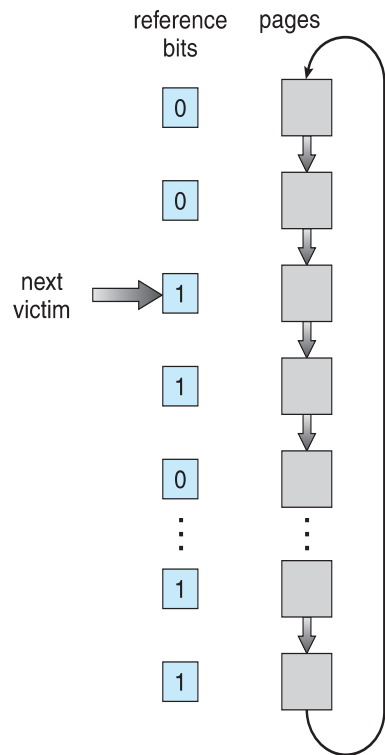
❖ Reference bit

- ❖ With each page associate a bit, initially = 0
- ❖ When page is referenced, bit set to 1
- ❖ Replace any with reference bit = 0 (if one exists)

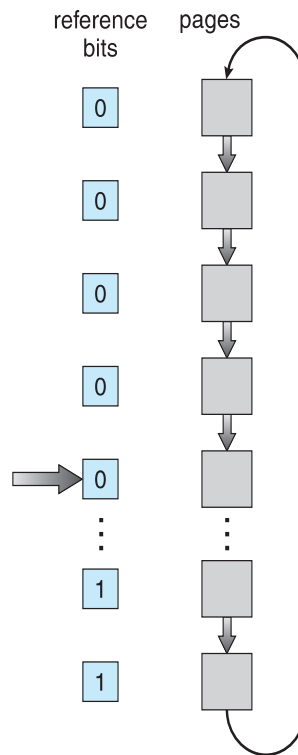
❖ Second-chance algorithm

- ❖ If page to be replaced has
 - ❖ Reference bit = 0 → replace it
 - ❖ reference bit = 1 then:
 - ❖ set reference bit 0, leave page in memory
 - ❖ replace next page, subject to same rules

Second-Chance Page-Replacement



(a)



(b)

Enhanced Second-Chance Algorithm

- ❖ Improve algorithm by using reference bit and modify bit
- ❖ Take ordered pair (reference, modify)
- ❖ (0, 0) neither recently used nor modified – best page to replace
- ❖ (0, 1) not recently used but modified – not quite as good, must write out before replacement
- ❖ (1, 0) recently used but clean – probably will be used again soon
- ❖ (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement



Counting Algorithms

- ❖ Keep a counter of the number of references that have been made to each
- ❖ **Least Frequently Used (LFU) Algorithm**: replaces page with smallest count
- ❖ **Most Frequently Used (MFU) Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page-Buffering Algorithms

- ❖ Always keep a pool of free frames
 - ❖ When needed, frame is always available, not found at fault time
 - ❖ Read page into free frame and select victim to evict and add to free pool
 - ❖ When convenient, evict victim
- ❖ Keep list of modified pages
 - ❖ When free, write to backing store and set to non-dirty
- ❖ Possibly, keep free frame contents intact and note what is in them
 - ❖ If referenced again before reused, no need to load contents again from disk
 - ❖ Generally useful to reduce penalty if wrong victim frame selected

Thank you

johnjose@iitg.ac.in

<http://www.iitg.ac.in/johnjose/>



CS343 - Operating Systems

Module-4E

Frame Allocation Techniques



Dr. John Jose

Assistant Professor

Department of Computer Science & Engineering

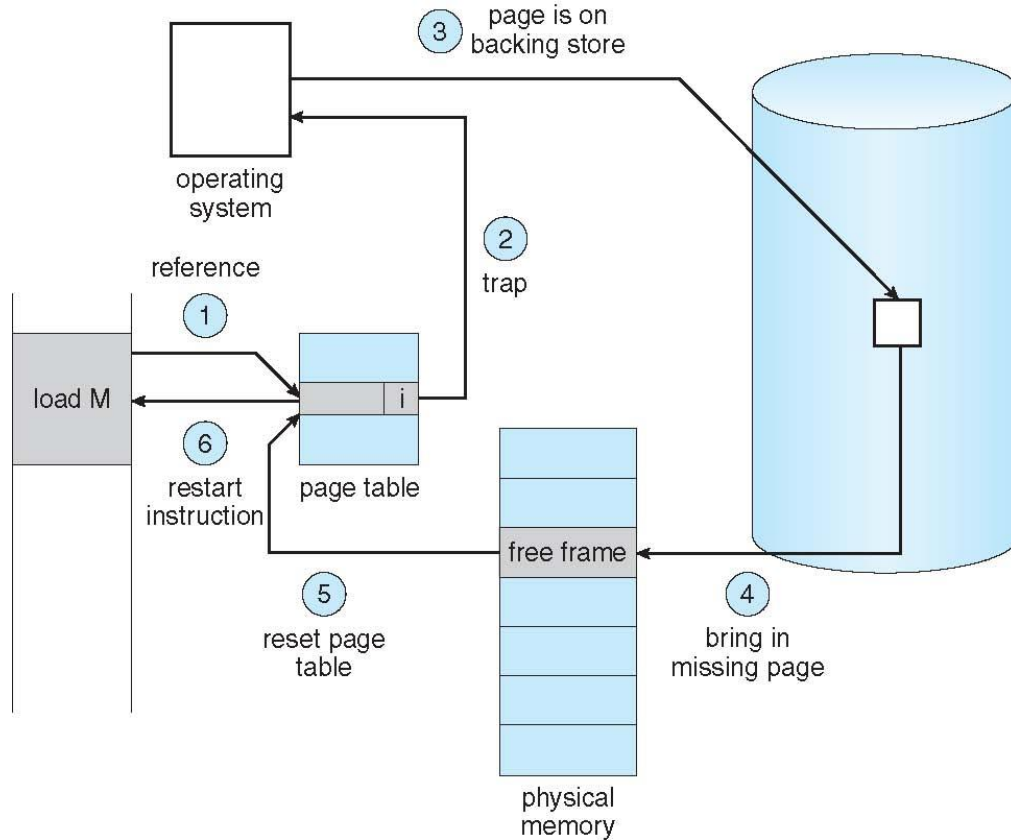
Indian Institute of Technology Guwahati, Assam.

<http://www.iitg.ac.in/johnjose/>

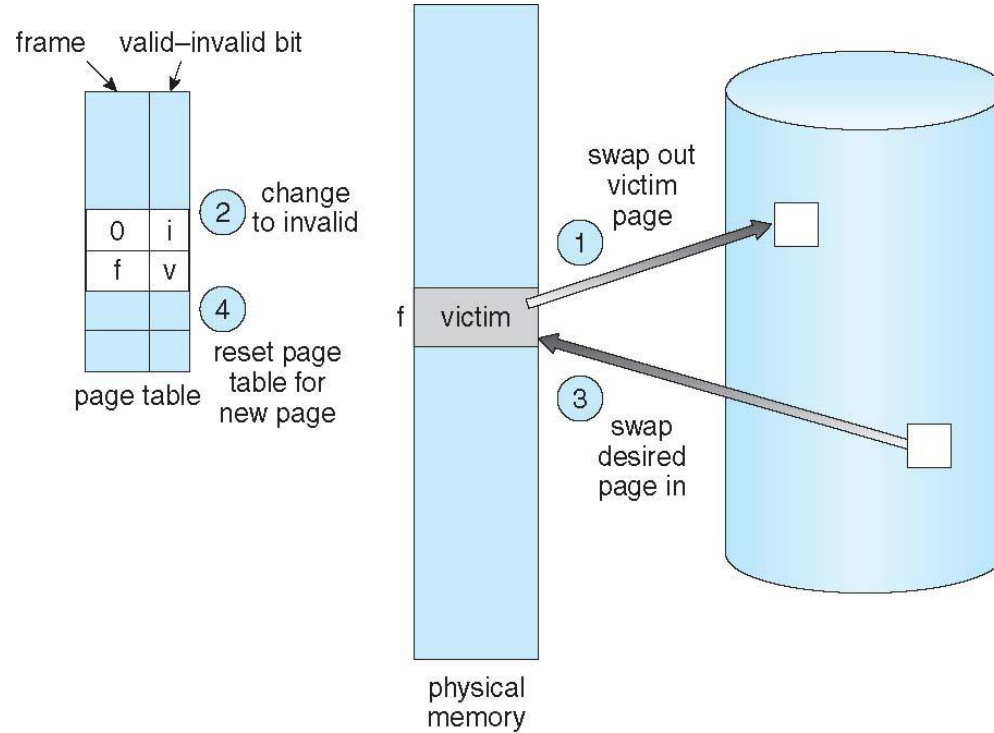
Overview of Memory Management

- ❖ Demand Paging
- ❖ Copy-on-Write
- ❖ Page Replacement
- ❖ Allocation of Frames
- ❖ Thrashing

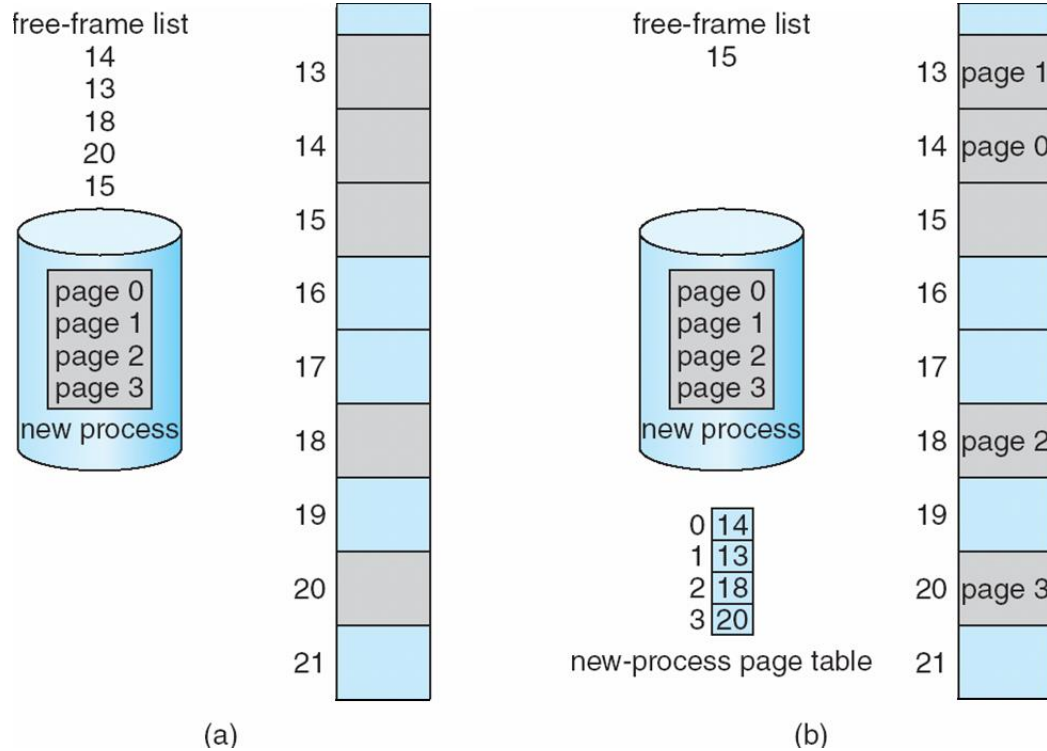
Steps in Handling a Page Fault



Page Replacement




Free Frames Allocation



Before allocation

After allocation

Allocation of Frames

- ❖ **Frame-allocation algorithm** determines
 - ❖ How many frames to give each process?
 - ❖ Which frames to replace?
- ❖ Two major allocation schemes
 - ❖ **fixed allocation** 
 - ❖ **priority allocation**

Fixed Allocation

- ❖ Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
 - ❖ Keep some as free frame buffer pool

Proportional Allocation

- ❖ Proportional allocation – Allocate according to the size of process

s_i = size of process p_i

$$S = \sum s_i$$

m = total number of frames

$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 4$$

$$a_2 = \frac{127}{137} \times 64 \approx 57$$

Priority Allocation

- ❖ Use a proportional allocation scheme using priorities rather than size
- ❖ If process P_i generates a page fault,
 - ❖ select for replacement one of its frames
 - ❖ select for replacement a frame from a process with lower priority number

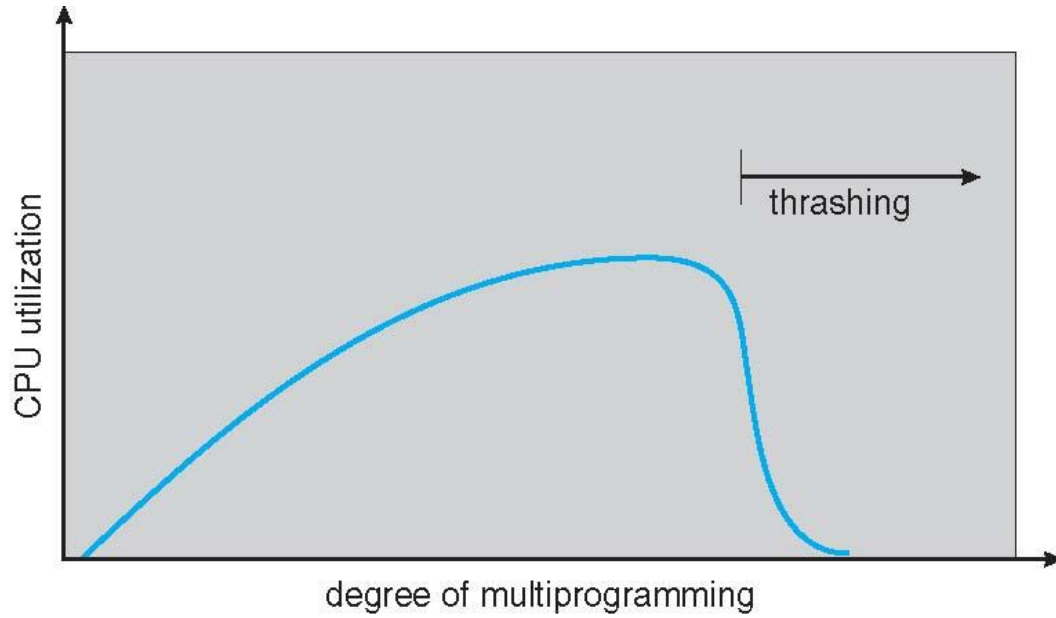
Global vs. Local Allocation

- ❖ **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
 - ❖ But then process execution time can vary greatly
 - ❖ But greater throughput so more common
- ❖ **Local replacement** – each process selects from only its own set of allocated frames
 - ❖ More consistent per-process performance
 - ❖ But possibly underutilized memory

Thrashing

- ❖ If a process does not have enough pages, the page-fault rate is 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** \equiv a process is busy swapping pages in and out

Thrashing

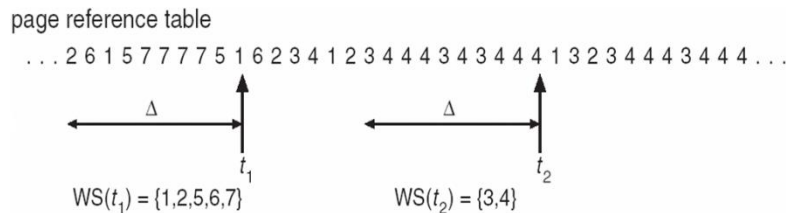


Demand Paging and Thrashing

- ❖ Why 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**

Working-Set Model

- ❖ $\Delta \equiv$ working-set window \equiv a fixed number of page references
Example: 10,000 instructions
- ❖ WS_i (working set of Process P_i) = Pages referenced in the most recent Δ
 - ❖ 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



- ❖ $D = \sum WS_i \equiv$ total demand frames (Approximation of locality)
- ❖ if $D > m \Rightarrow$ Thrashing; if $D > m$, then suspend/swap out processes

How to compute Working-Set?

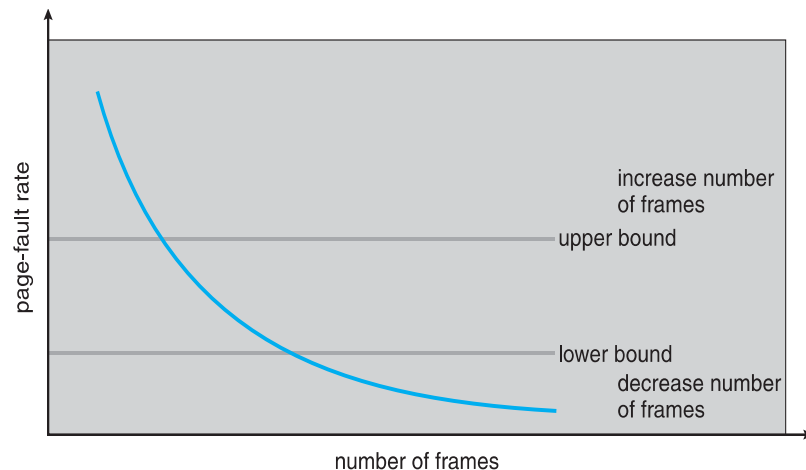
- ❖ Approximate with interval timer + a reference bit
- ❖ Example: $\Delta = 10,000$: Timer interrupts after every 5000 time units
 - ❖ 2 history bits for each page is kept in memory
 - ❖ Whenever a timer interrupts, copy the reference bit to history bit.
 - ❖ Sets the values of all reference bits to 0
 - ❖ During page fault, if one of the history bits = 1 \Rightarrow page in working set

How to compute Working-Set?

- ❖ Why counter, history and reference bits approach not completely accurate?
- ❖ Improvement = 10 bits and interrupt every 1000 time units

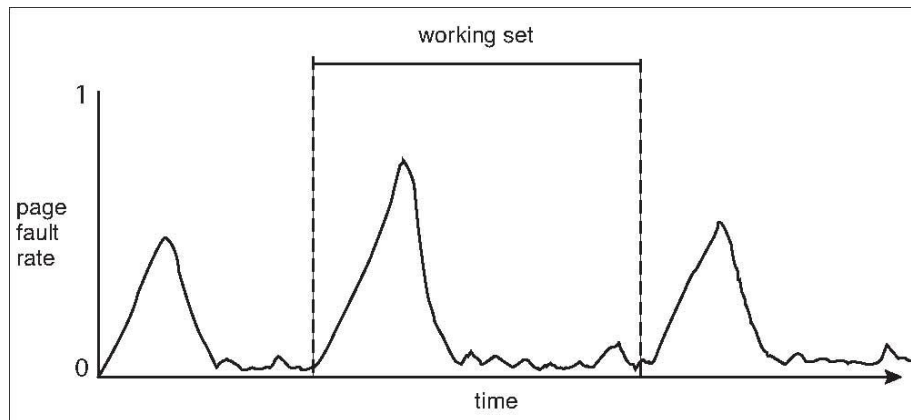
Page-Fault Frequency

- ❖ More direct approach than WSS
- ❖ 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



Working Sets and Page Fault Rates

- ❖ Direct relationship between working set of a process and its page-fault rate
- ❖ Working set changes over time
- ❖ Peaks and valleys over time

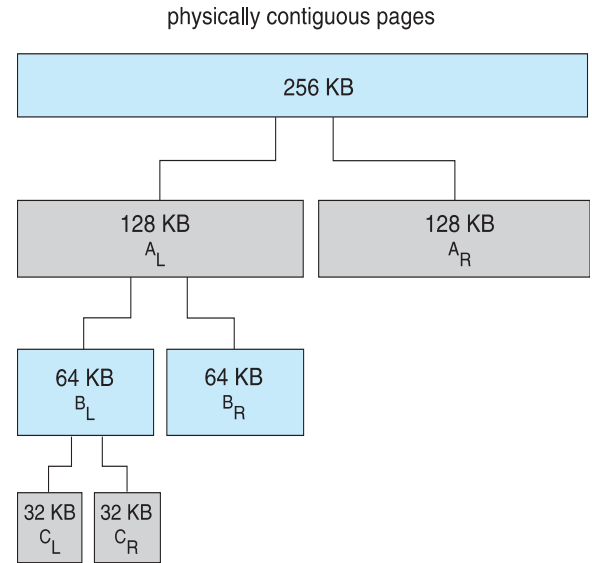


Buddy System

- ❖ Allocates memory from fixed-size segment consisting of physically-contiguous pages
- ❖ Memory allocated using **power-of-2 allocator**
 - ❖ Satisfies requests in units sized as power of 2
 - ❖ Request rounded up to next highest power of 2
 - ❖ When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
 - ❖ Continue until appropriate sized chunk available
- ❖ Advantage – quickly **coalesce** unused chunks into larger chunk
- ❖ Disadvantage - fragmentation

Buddy System Allocator

- ❖ Assume 256KB chunk available
- ❖ Kernel requests 21KB
 - ❖ Split into A_L and A_R of 128KB each
 - ❖ One further divided into B_L and B_R of 64KB
 - ❖ One further into C_L and C_R of 32KB each
 - ❖ One used to satisfy request



Prepaging

- ❖ Reduce the large number of page faults that occurs at process startup
- ❖ Prepage all or some of the pages a process will need, before they are referenced
- ❖ But if prepaged pages are unused, I/O and memory was wasted
- ❖ Assume s pages are prepaged and α of the pages is used
- ❖ Cost of $s * \alpha$ saved pages faults vs cost of prepaging $s * (1 - \alpha)$ unnecessary pages
- ❖ α near zero \Rightarrow prepaging loses

Page Size

- ❖ Sometimes OS designers have a choice on custom-built CPU
- ❖ Page size selection criteria:
 - ❖ Fragmentation and Resolution
 - ❖ Page table size
 - ❖ I/O overhead
 - ❖ Number of page faults
 - ❖ Locality
 - ❖ TLB size and effectiveness
- ❖ Always power of 2, usually in the range 2^{12} to 2^{22}

TLB Reach

- ❖ TLB Reach - The amount of memory accessible from the TLB
- ❖ $\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$
- ❖ Ideally, the working set of each process is stored in the TLB
 - ❖ Otherwise there is more time spend in resolving memory references in page table (delay).
- ❖ Increase the Page Size
 - ❖ This may lead to an increase in fragmentation as not all applications require a large page size
- ❖ Provide Multiple Page Sizes
 - ❖ This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Program Structure

- ❖ `int[128,128] data;` Each row is stored in one page
- ❖ A page can store 128 words

Program 1 [128 x 128 = 16,384 page faults]

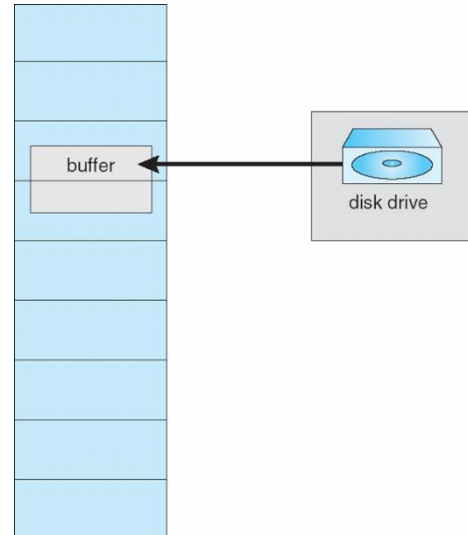
```
for (j = 0; j < 128; j++)  
    for (i = 0; i < 128; i++)  
        data[i,j] = 0;
```

Program 2 [128 page faults]

```
for (i = 0; i < 128; i++)  
    for (j = 0; j < 128; j++)  
        data[i,j] = 0;
```

I/O interlock

- ❖ **I/O Interlock** – Pages must sometimes be locked into memory
- ❖ Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- ❖ When I/O is complete pages are unlocked



Thank you

johnjose@iitg.ac.in

<http://www.iitg.ac.in/johnjose/>

