

Operating Systems – 1BCS304

Module -IV

Course Coordinator:
Mrs. Pallavi A R
Assistant professor
Dept. of CSE-DS
ATMECE, Mysuru

Memory Management

Objectives

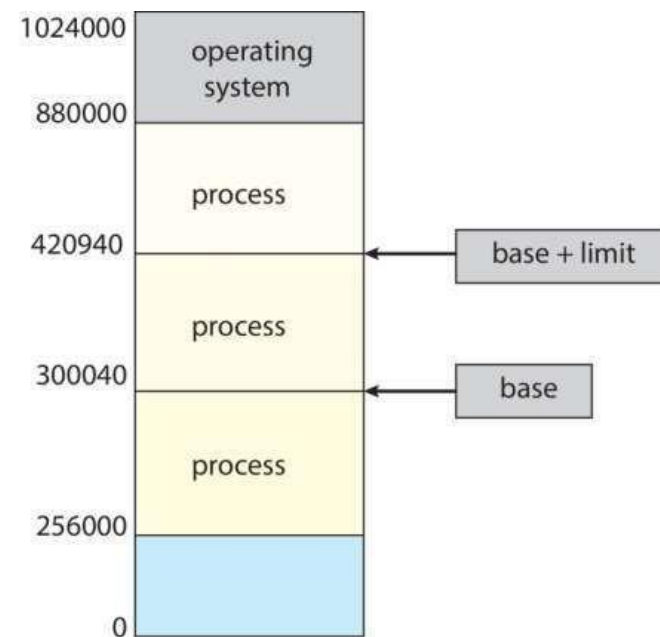
- To provide a detailed description of various ways of
- organizing memory hardware
- To discuss various memory-management techniques,

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 is done in one CPU clock (or less)
- 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

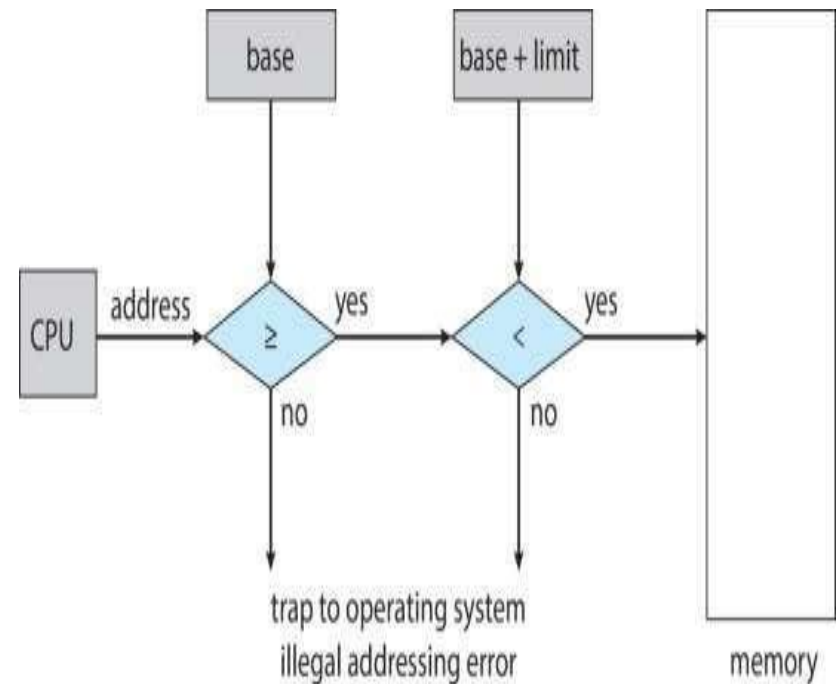
Protection

- Need to ensure that a process can only access those addresses in its address space.
- We can provide this protection by
- using a pair of base and limit registers define the logical address space of a process



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



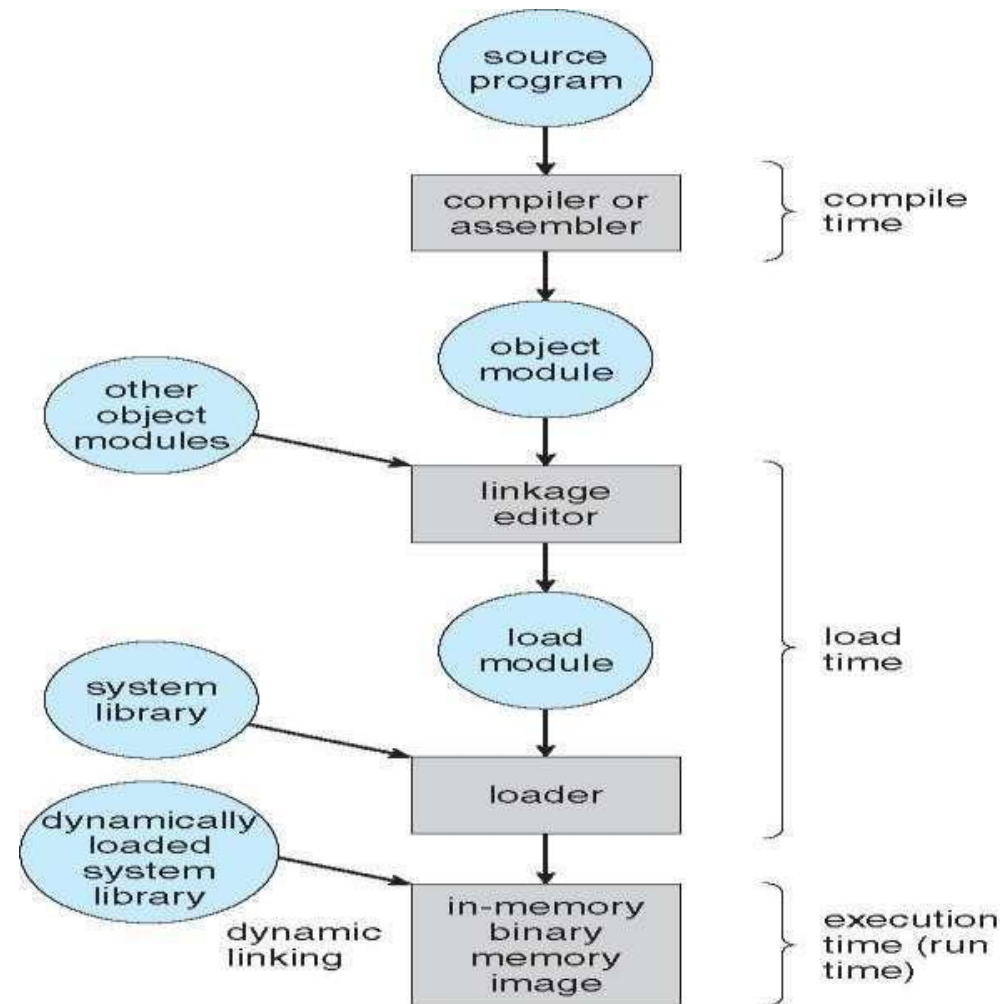
Address Binding

- Programs on disk, ready to be brought into memory to execute form an input queue
 - – Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
 - How can it not be?
- Addresses represented in different ways at different stages of 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
 - – Each binding maps one address space to another

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
 - – **Compiletime**: 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
 - – **Executiontime**: 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 Graph Example

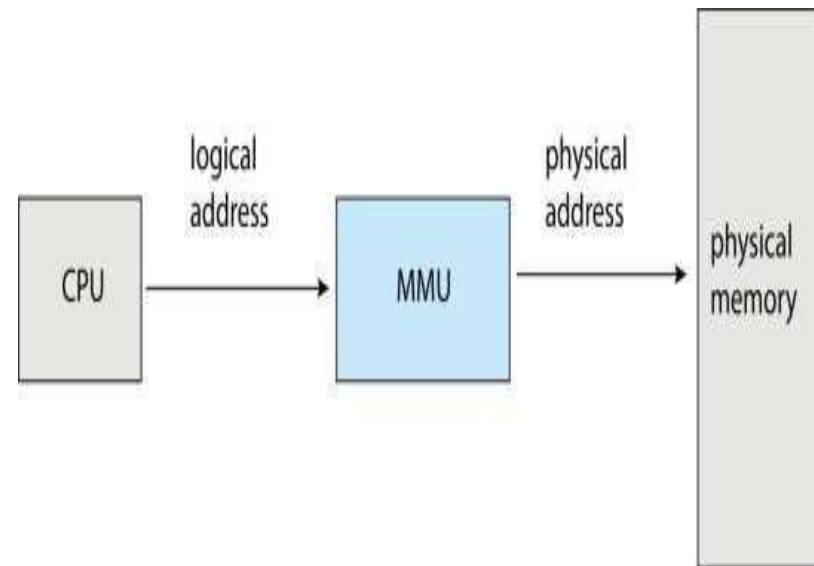


Logical vs. Physical Address Space

- 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_x0002_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
- Many methods possible, covered in the rest of this chapter

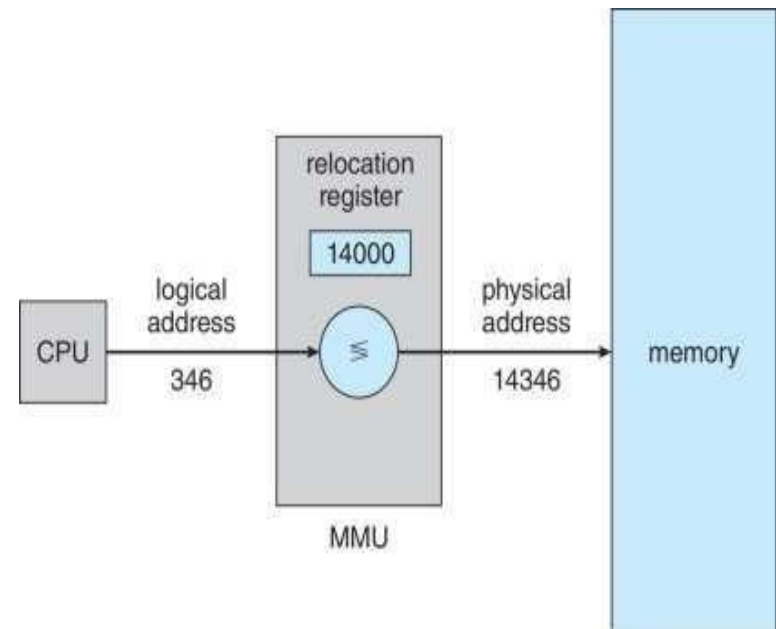


Memory-Management Unit (Cont.)

- Consider simple scheme. which is a generalization of the base-register scheme.
- The base register now called relocation register
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with logical addresses; it never sees the real physical addresses
- – Execution-time binding occurs when reference is made to location in memory
- – Logical address bound to physical addresses

Memory-Management Unit (Cont.)

- Consider simple scheme. which is a generalization of the base register scheme. □
- The base register now called relocation register
- The value in the added relocation Register is to every address generated by a user process at the time it is sent to memory



Dynamic Loading

- The entire program does need to be in memory to execute
- 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

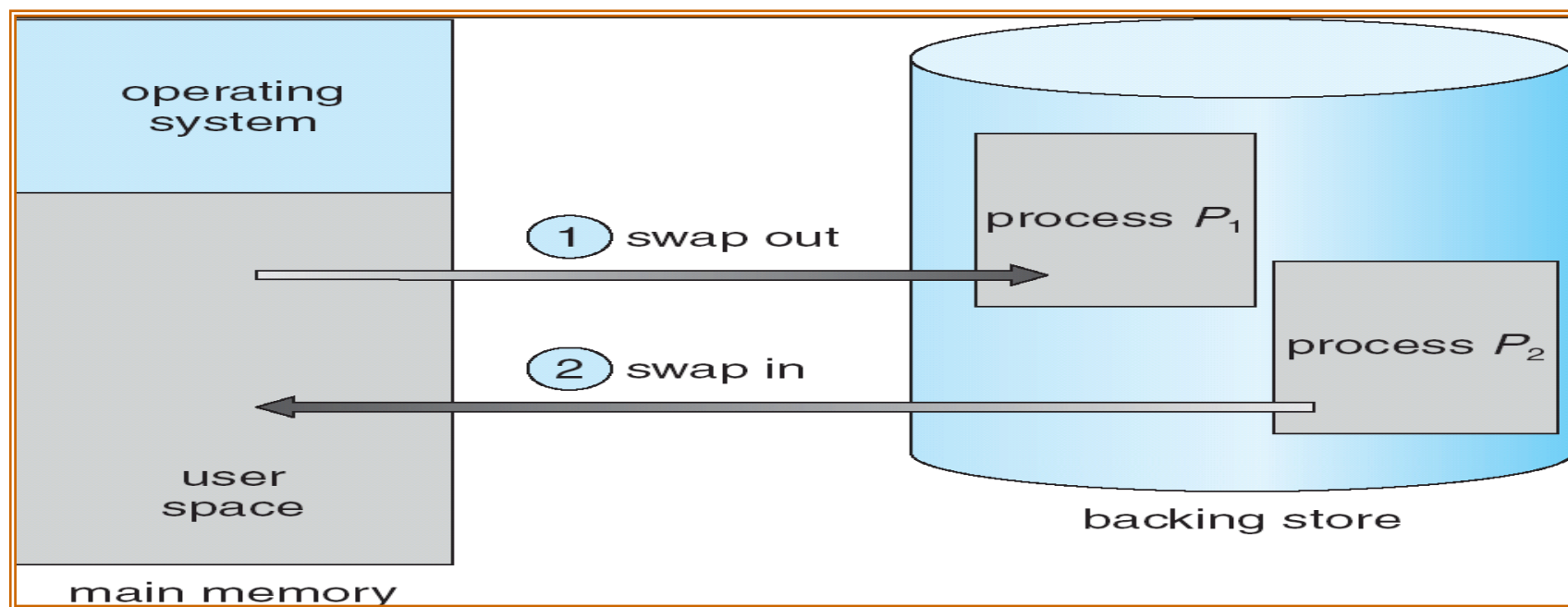
Dynamic Linking

- 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
- System also known as shared libraries
- Consider applicability to patching system libraries
 - – Versioning may be needed

Swapping

- A process can be swapped temporarily out of memory to a **backing store**, and then brought back into memory for continued execution
- 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
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
- System maintains a ready queue of ready-to-run processes which have memory images on disk

Schematic View of Swapping



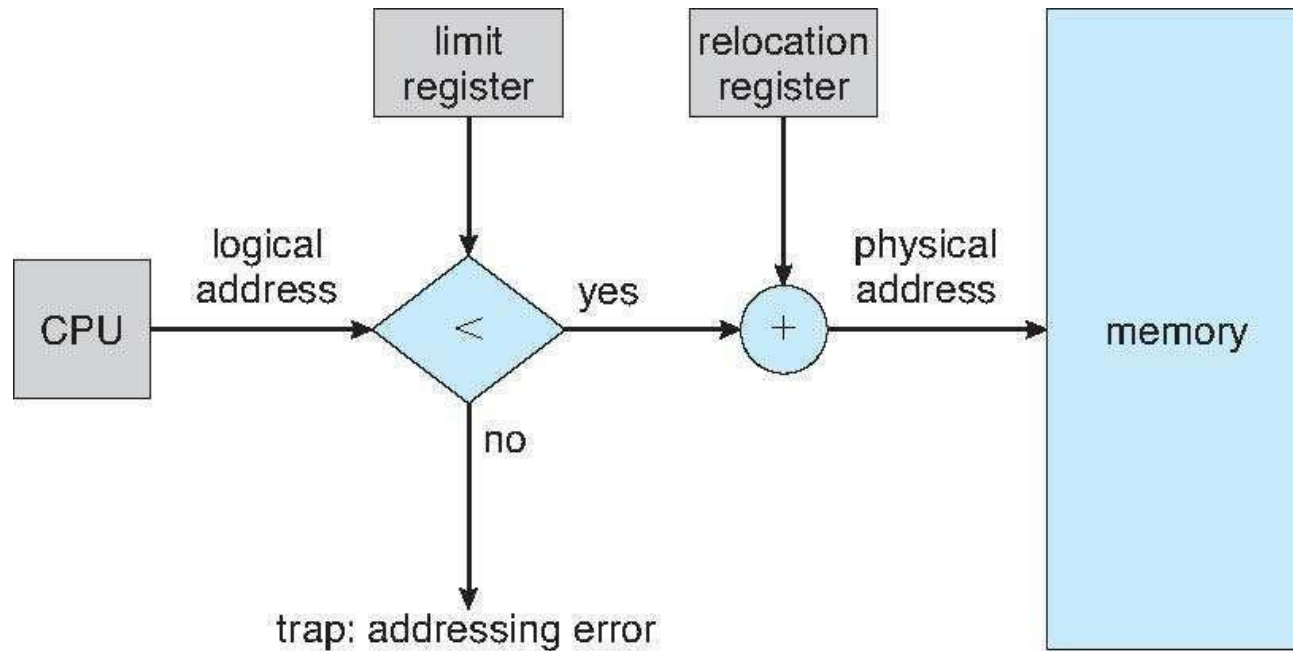
Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate 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 contiguous section of memory

Contiguous Allocation (Cont.)

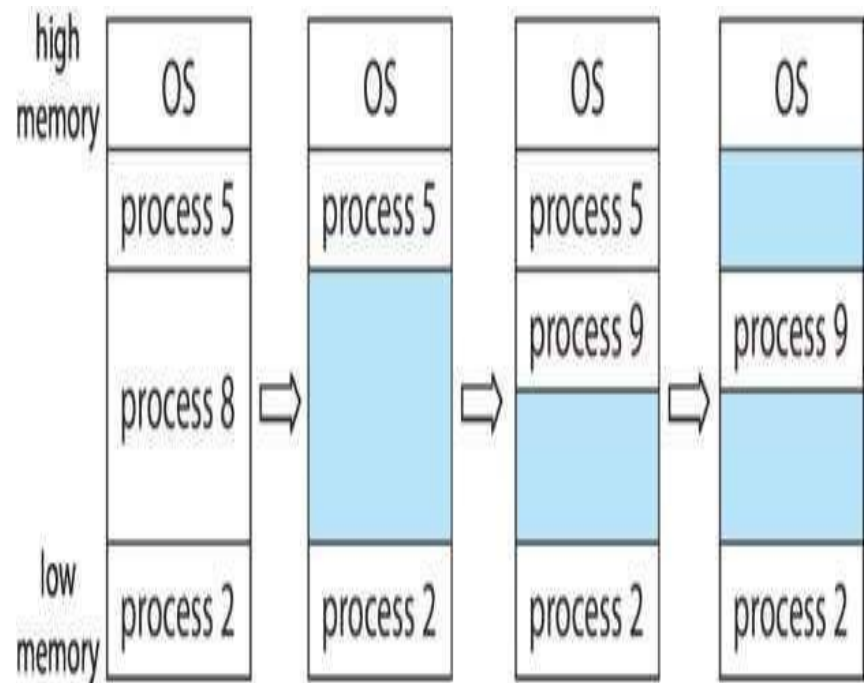
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
 - – Base register contains value of smallest physical address
 - – Limit register contains range of logical addresses
 - – each logical address must be less than the limit register
 - – MMU maps logical address dynamically
 - – Can then allow actions such as kernel code being transient and kernel changing size

Hardware Support for Relocation and Limit Registers



Variable Partition

- Multiple-partition allocation large enough to
- – Degree of multiprogramming limited by number of partitions
- – Variable-partition sizes for efficiency (sized to a given process' needs)
- – Hole –block of available memory; holes of various size are scattered throughout memory
- – When a process arrives, it is allocated memory from a hole 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

Fragmentation

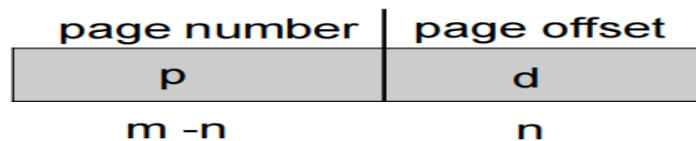
- External Fragmentation –total memory space exists to satisfy a request, but it is not contiguous
- Internal Fragmentation –allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given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

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 and load program
- Set up a page table to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation

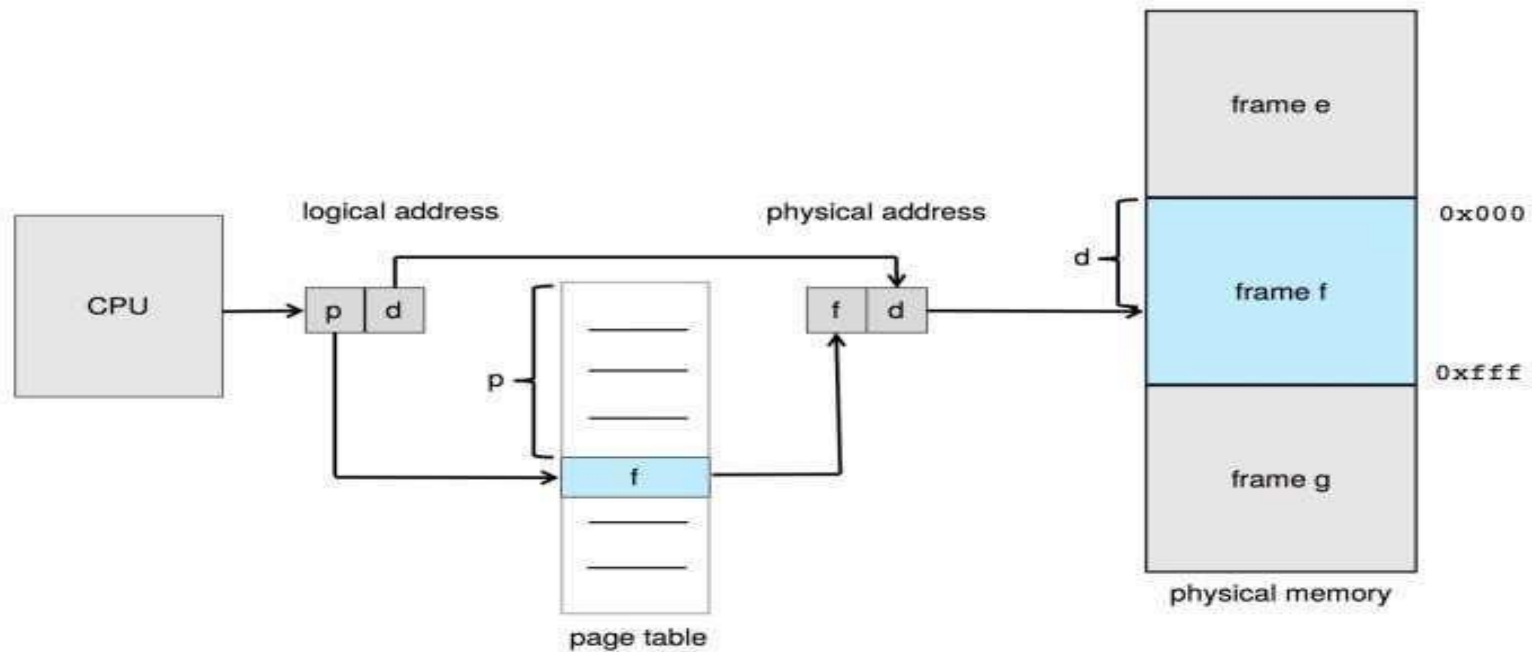
Address Translation Scheme

- 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

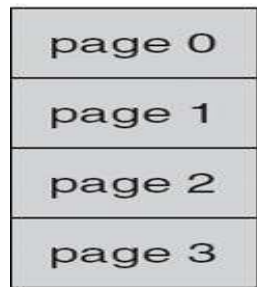


- For given logical address space 2^m and page size 2^n

Paging Hardware



Paging Model of Logical and Physical Memory

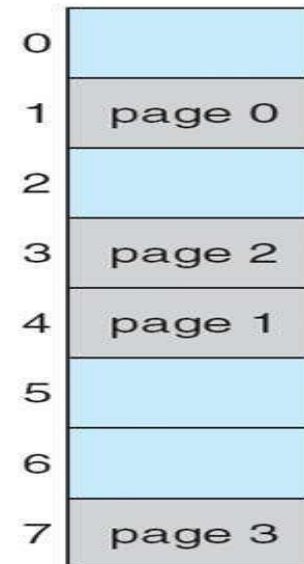


logical
memory

0	1
1	4
2	3
3	7

page table

frame
number



physical
memory

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)

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

0	
4	i
	j
	k
8	m
	n
	o
12	
16	
20	a
	b
24	c
	d
	e
	f
	g
28	

physical memory

Paging - 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
- – Solaris supports two page sizes – 8KB and 4 MB

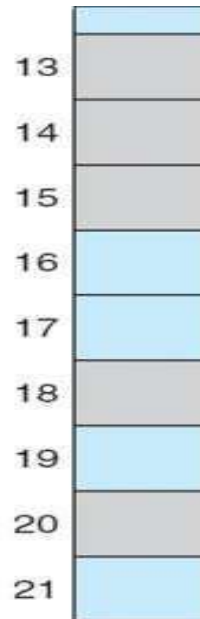
Free Frames

free-frame list

14
13
18
20
15

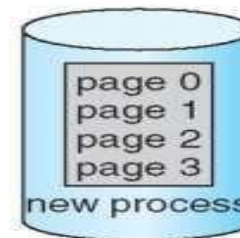


(a)



free-frame list

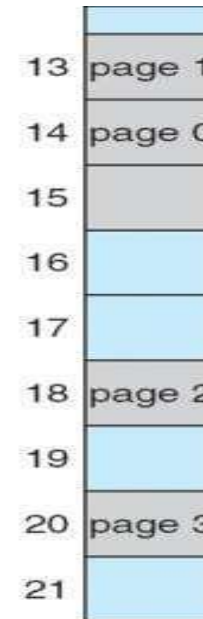
15



new-process page table

0	14
1	13
2	18
3	20

(b)



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 translation look aside buffers (TLBs) (also called associative memory).

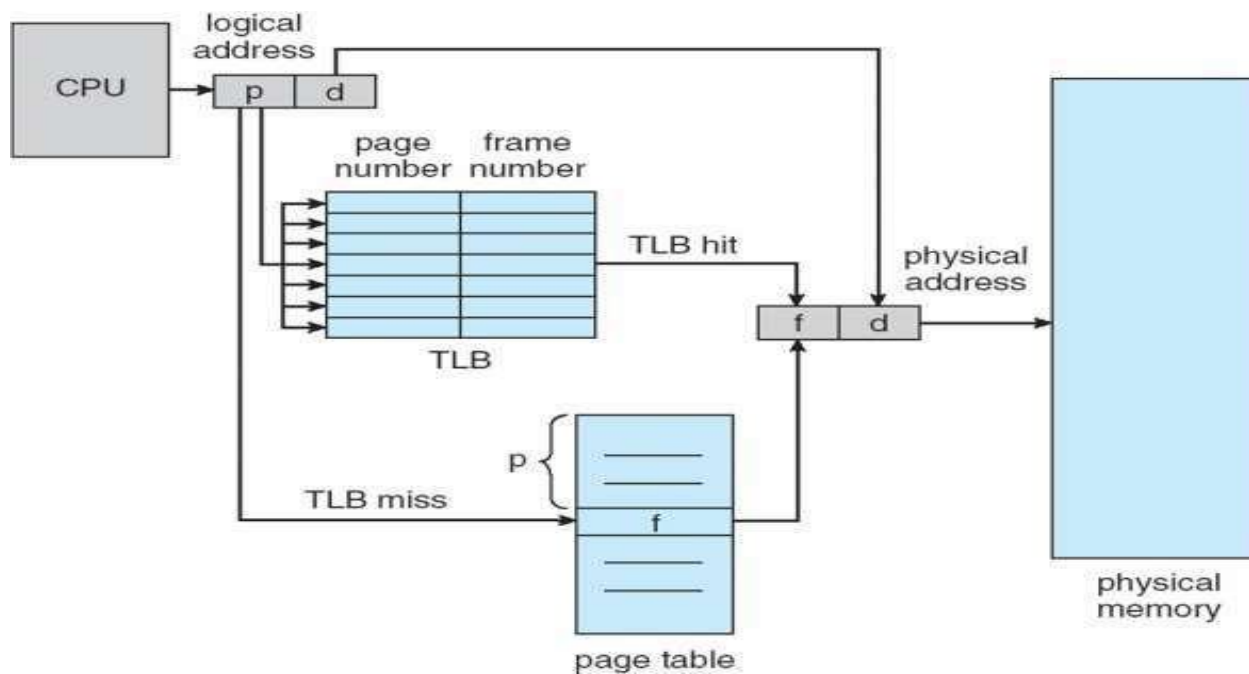
Translation Look-Aside Buffer

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

Hardware

- Associative memory –parallel search
- Address translation (p, d)
 - – If p is in associative register, get frame # out
 - – Otherwise get frame # from page table in memory

Paging Hardware With TLB



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
- Effective Access Time (EAT)
- $EAT = 0.80 \times 10 + 0.20 \times 20 = 12$ nanoseconds
- implying 20% slowdown in access time
- Consider amore realistic hit ratio of 99%,
- $EAT = 0.99 \times 10 + 0.01 \times 20 = 10.1$ ns
- implying only 1% slowdown in access time

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

00000

page 0
page 1
page 2
page 3
page 4
page 5

10,468
12,287

frame number		valid-invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

page table

0	
1	
2	page 0
3	page 1
4	page 2
5	
6	
7	page 3
8	page 4
9	page 5
	⋮
	page n

Shared Pages

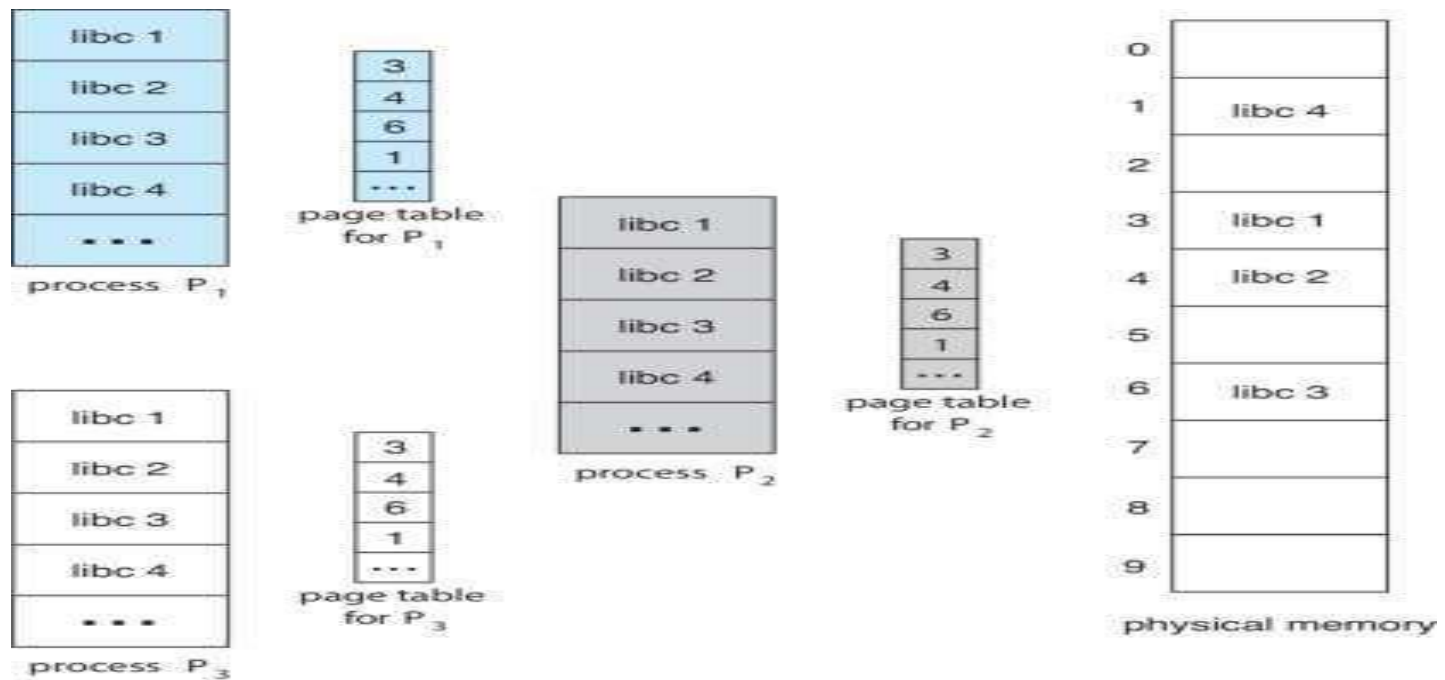
Shared code

- – 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 Example



Structure of the Page Table

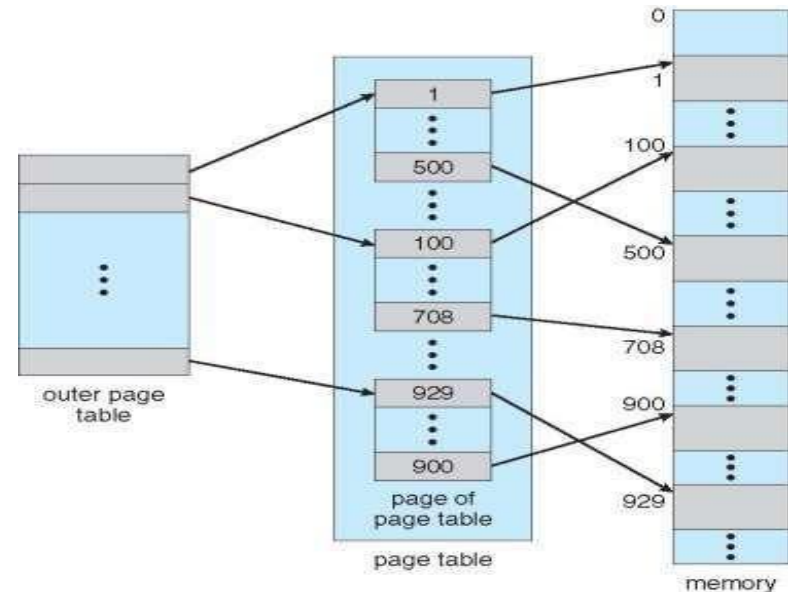
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Structure of the Page Table

- Memory structures for paging can get huge using straightforward 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 $\square$ each process 4 MB of physical address space for the page table alone
- Don't want to allocate that contiguously in main memory
- – One simple solution is to divide the page table into smaller units
- Hierarchical Paging

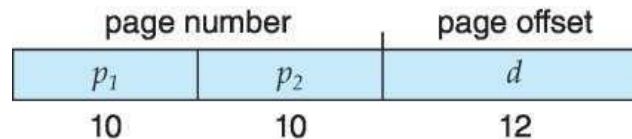
Hierarchical Page Tables

- 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



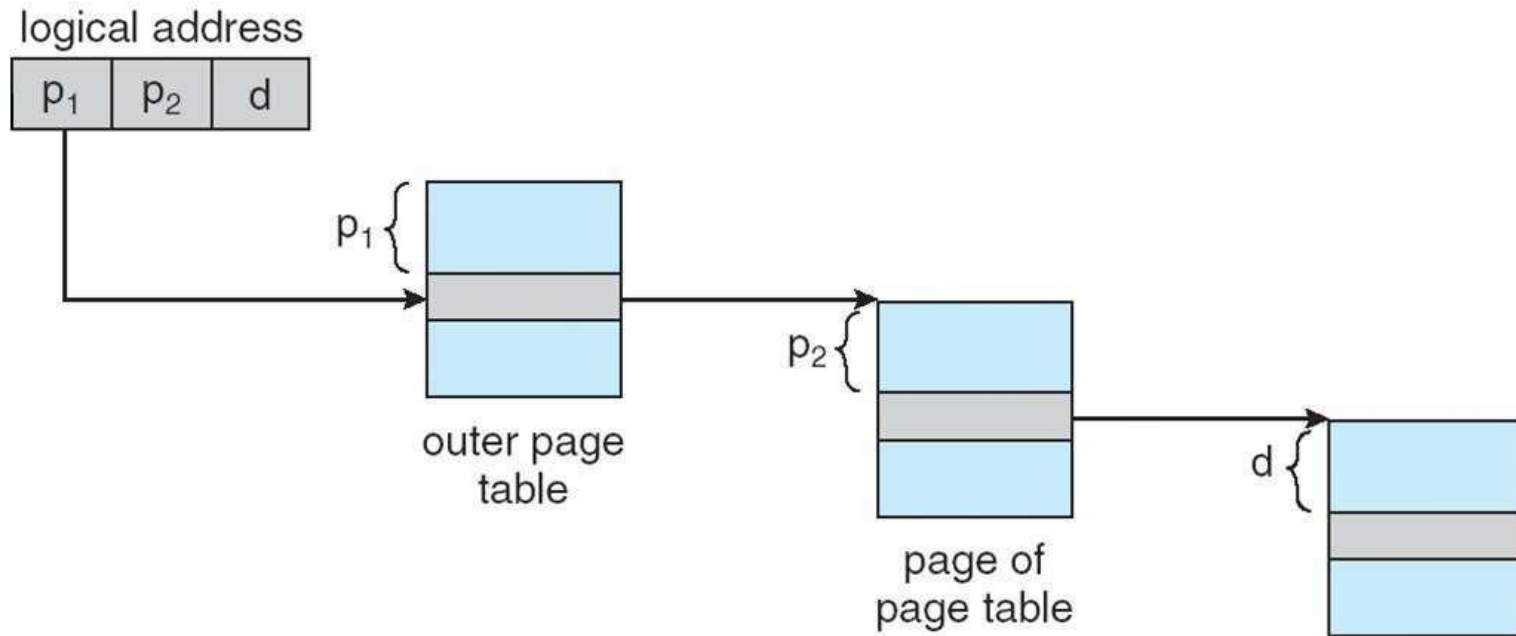
Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
 - – a page number consisting of 20bits – a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
 - – a 10-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 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
- Known as forward-mapped page table

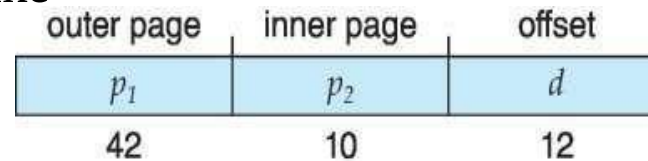
Address-Translation Scheme



64-bit Logical Address Space

1. Even two-level paging scheme not sufficient

- If page size is 4 KB (2¹²) – Then page table has 252 entries
 - If two level scheme, inner page tables could be 210 4-byte entries
 - Address would look like



Outer page table has 242 entries or 244 bytes

- One solution is to add a 2nd outer page table
- But in the following example the 2nd outer page table is still 234 bytes in size
- And possibly 4 memory access to get to one physical memory location

Three-level Paging Scheme

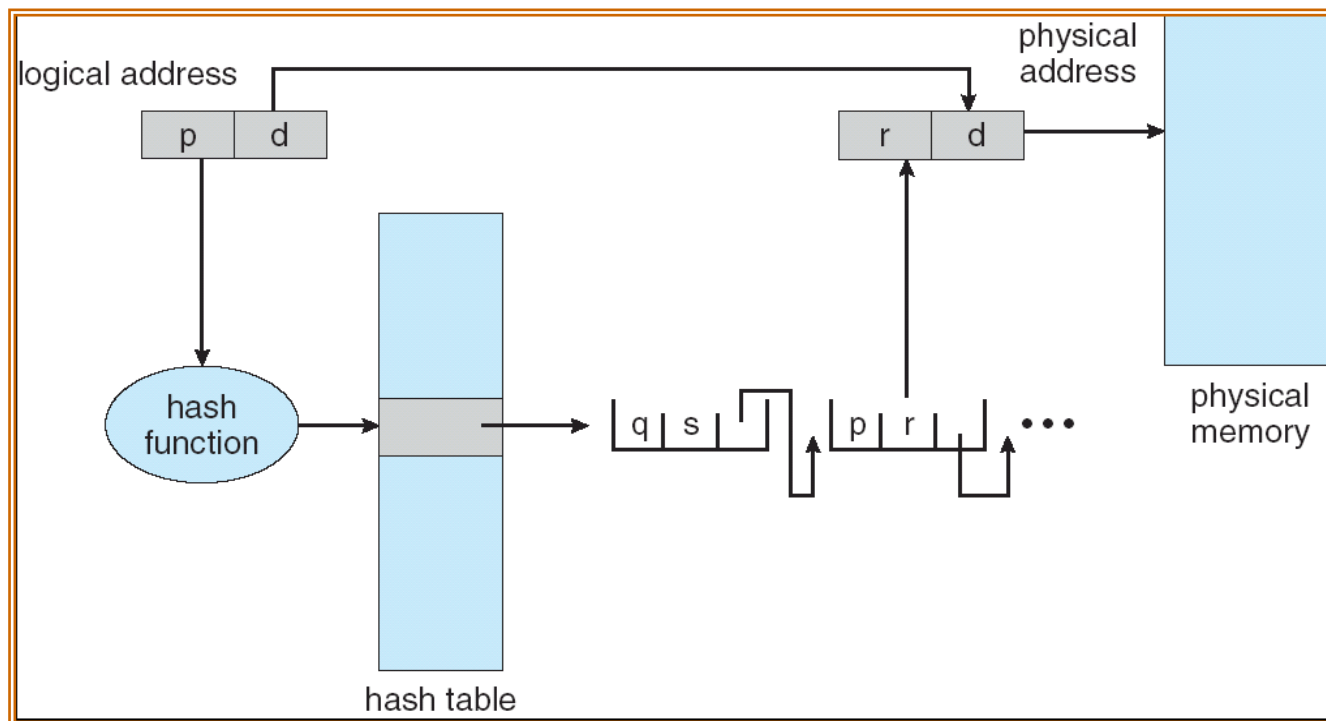
outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

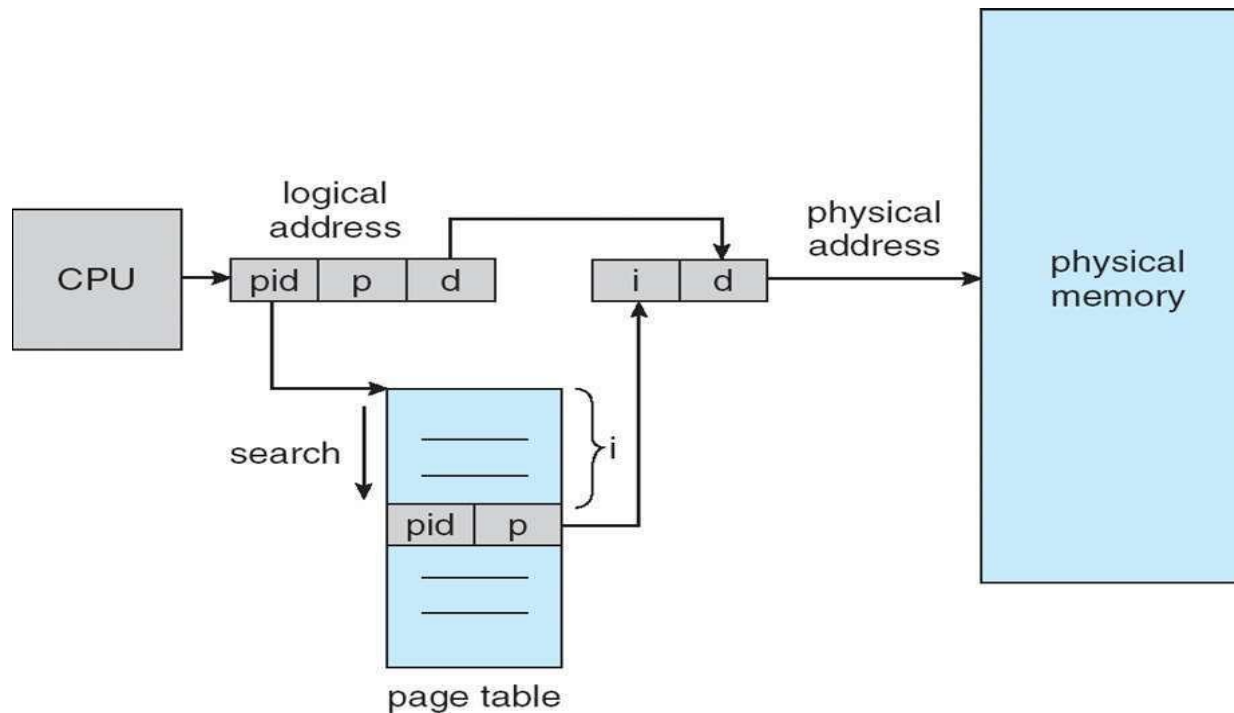
Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table.
- This page table contains a chain of elements hashing to the same location.
- 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 Architecture

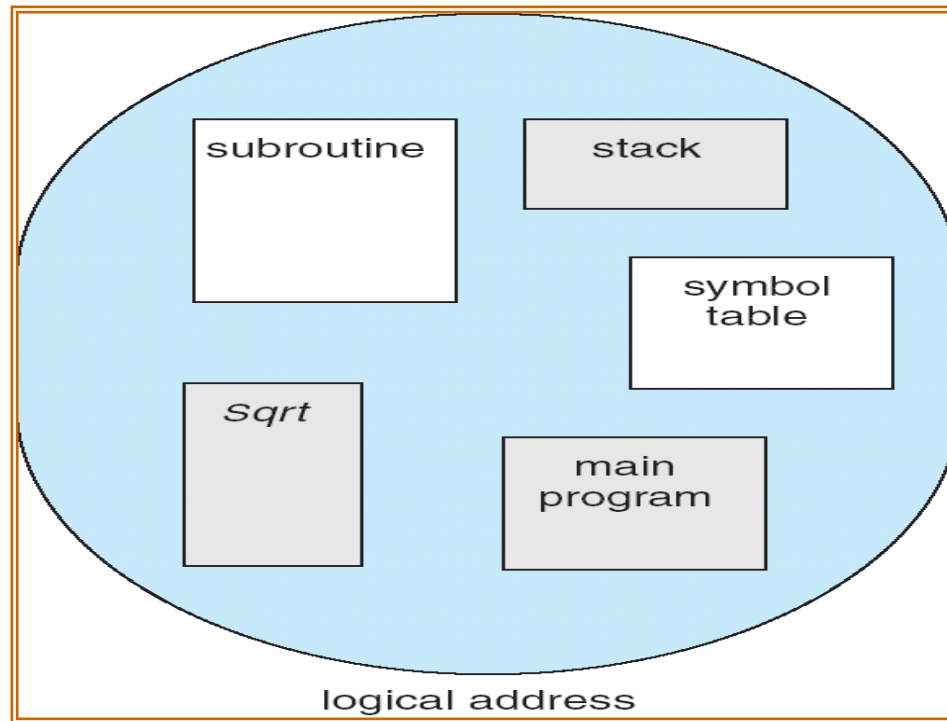


Segmentation

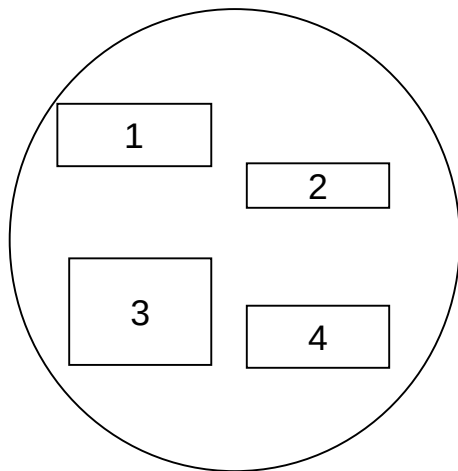
- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:

main program,
procedure,
function,
method,
object,
local variables, global variables,
common block,
stack,
symbol table, arrays

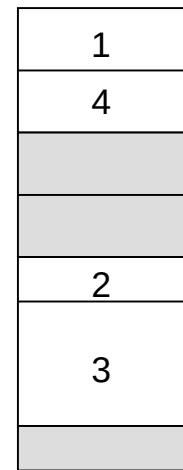
User's View of a Program



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

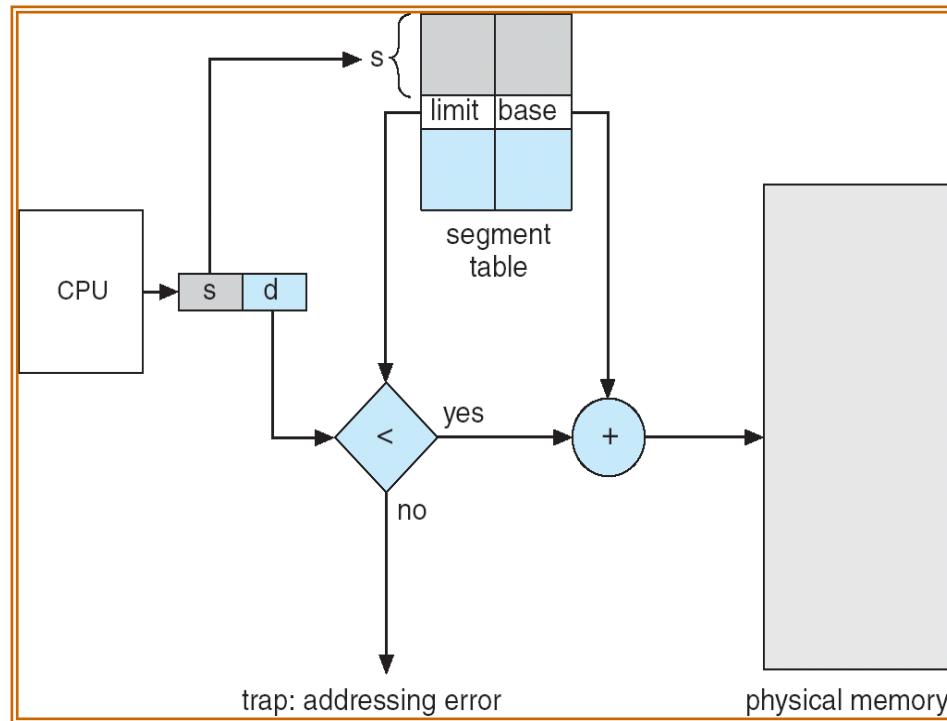
Segment-table length register (STLR) indicates number of segments used by a program;

segment number s is legal if $s < \text{STLR}$

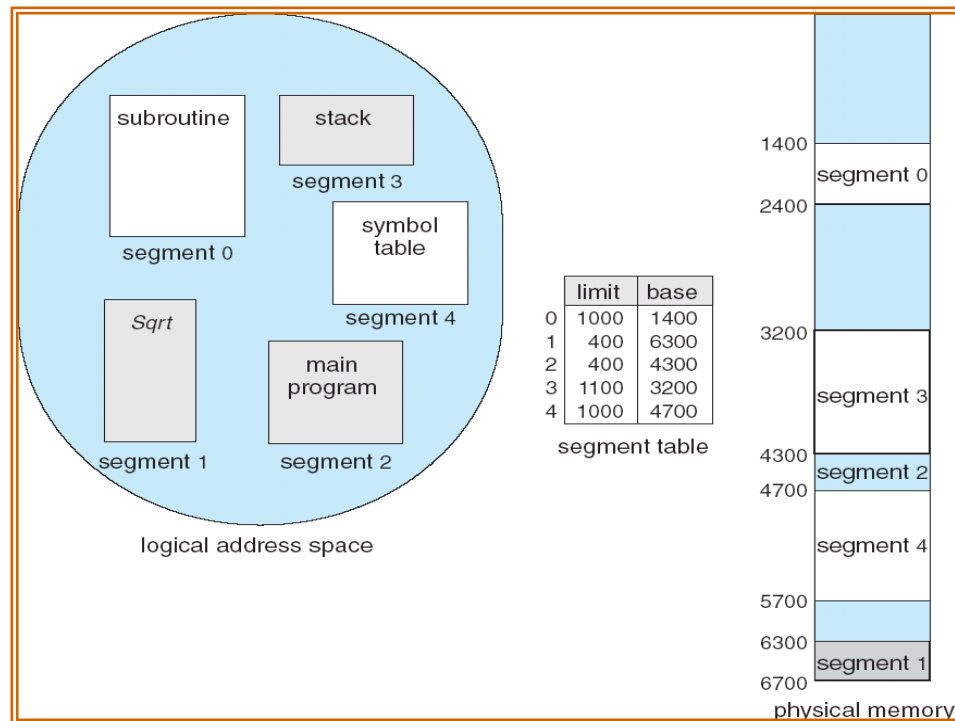
Segmentation Architecture (Cont.)

- Protection
 - With each entry in segment table associate:
 - validation bit = 0 ☐ illegal segment
 - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Segmentation Hardware



Example of Segmentation



Virtual Memory Management

Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page_x0002_replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model

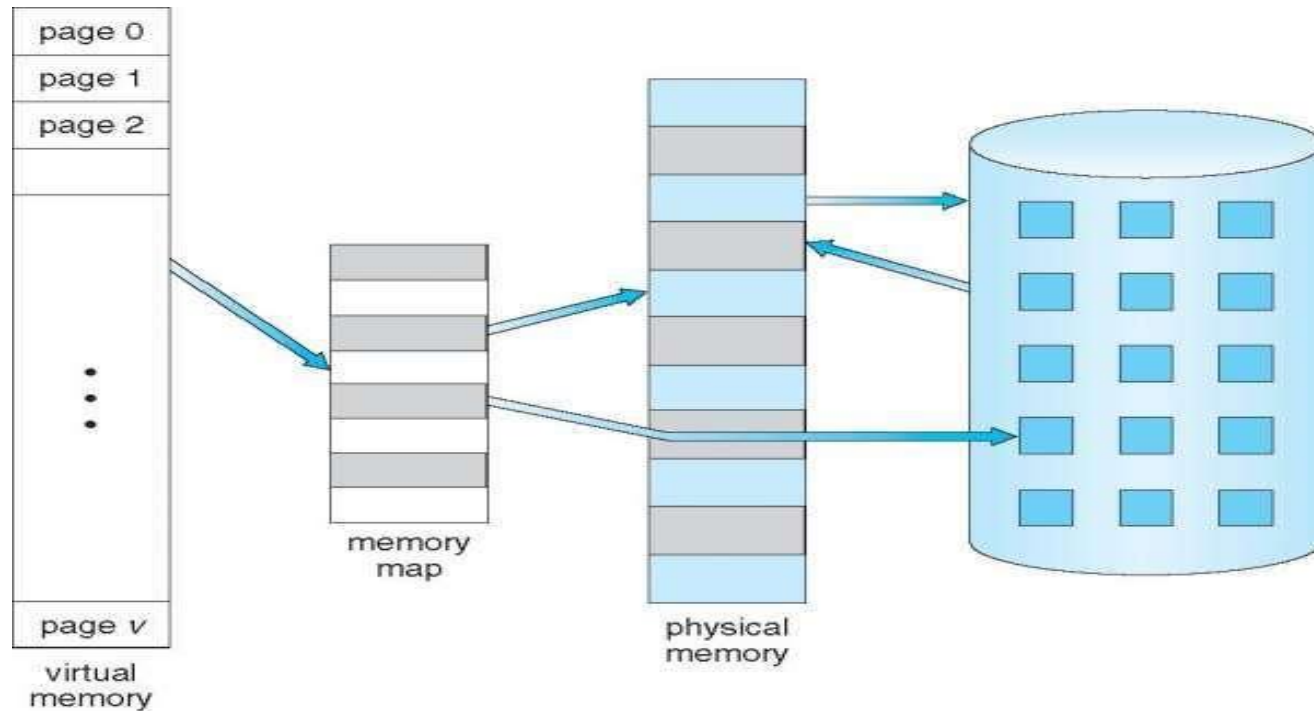
Background

- 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
 - – Program and programs could be larger than physical memory

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
 - – Allows for more efficient process creation
 - – More programs running concurrently
 - – Less I/O needed to load or swap processes
- Virtual memory can be implemented via:
 - – Demand paging
 - – Demand segmentation

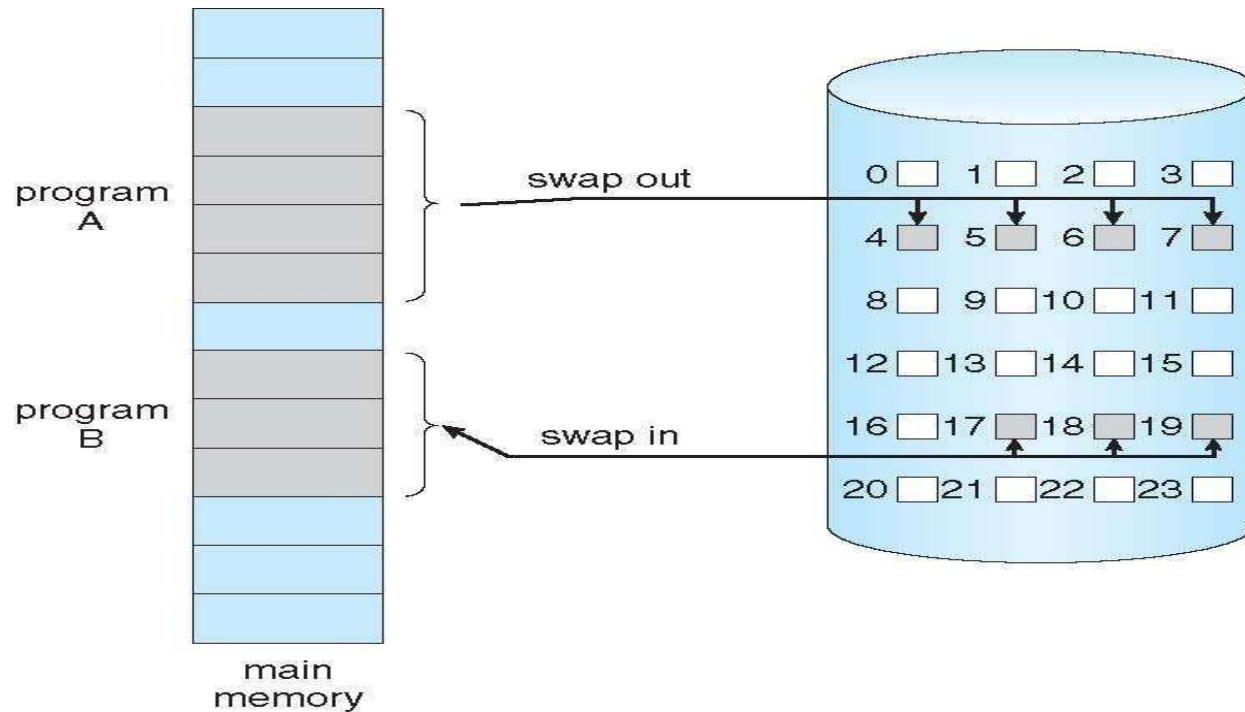
Virtual Memory That is Larger Than Physical Memory



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
- 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

Transfer of a Paged Memory to Contiguous Disk Space



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:

Frame #	valid-invalid bit
	v
	v
	v
	i
....	
	i
	i

page table

- During address translation, if valid–invalid bit in page table entry
- is I ☐ page fault

Page Table When Some Pages Are Not in Main Memory

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

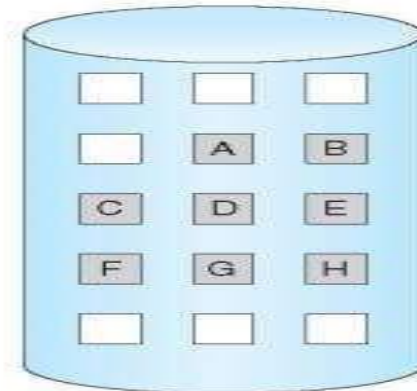
logical memory

frame		valid-invalid bit
0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

page table

0	
1	
2	
3	
4	A
5	
6	C
7	
8	
9	F
10	
11	
12	
13	
14	
15	

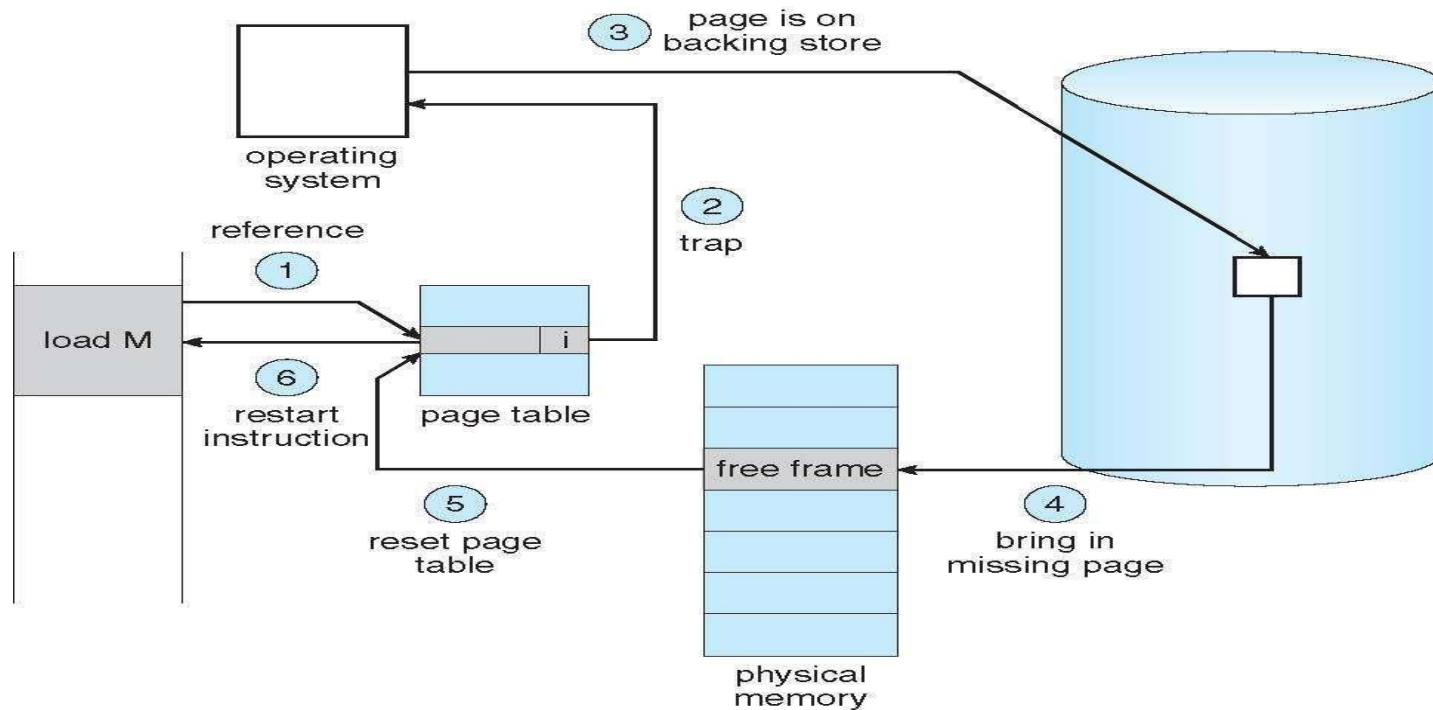
physical memory



Page Fault

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

Steps in Handling a Page Fault



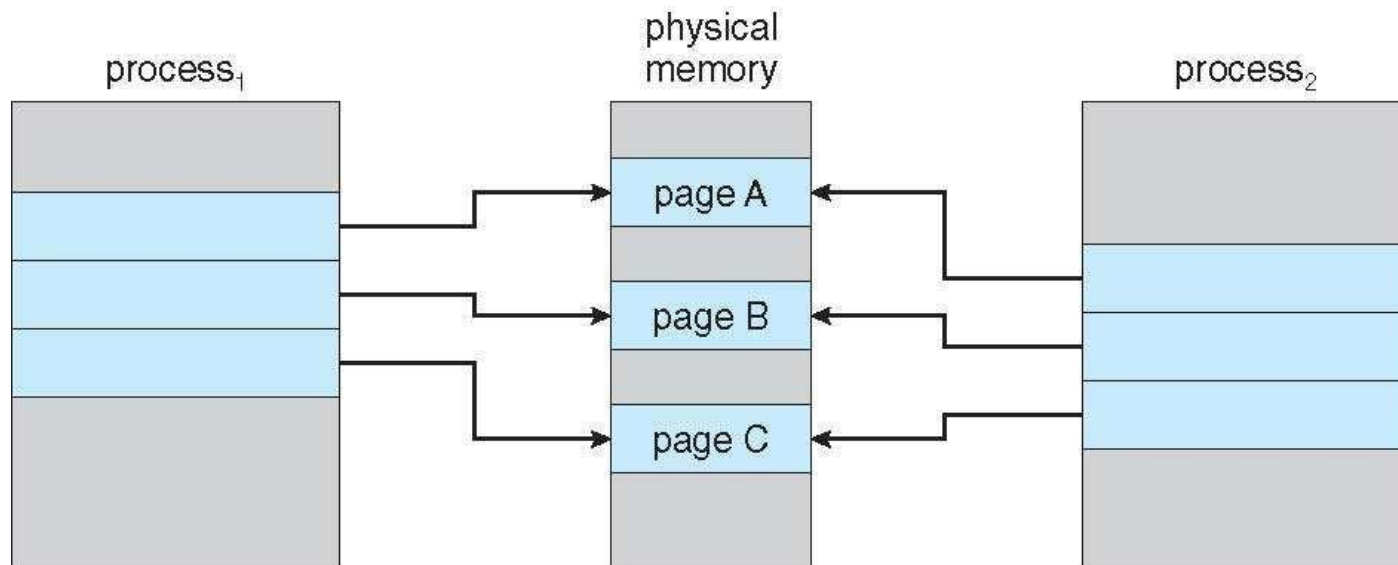
Performance of Demand Paging

- Stages in Demand Paging
- 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:
 - 1. Wait in a queue for this device until the read request is serviced
 - 2. Wait for the device seek and/or latency time
 - 3. Begin the transfer of the page to a free frame
- 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

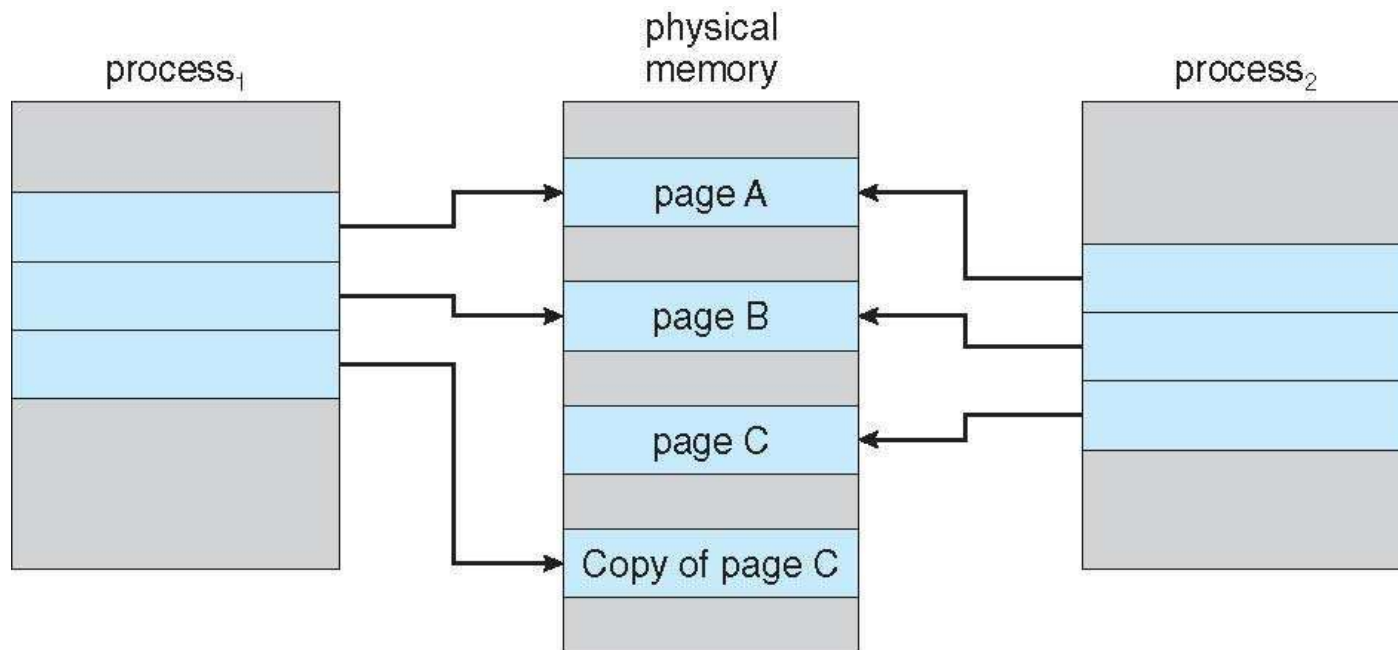
Copy-on-Write

- 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
 - – 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

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



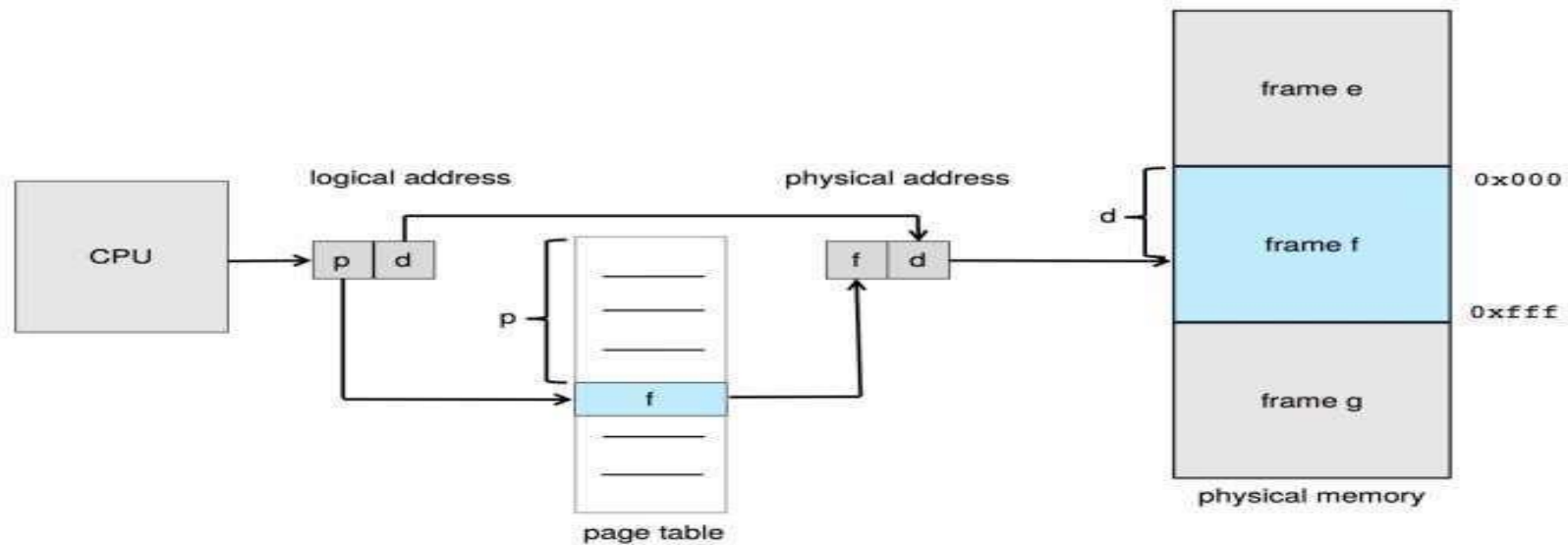
What Happens if There is no Free Frame?

- 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
- 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

Page Replacement

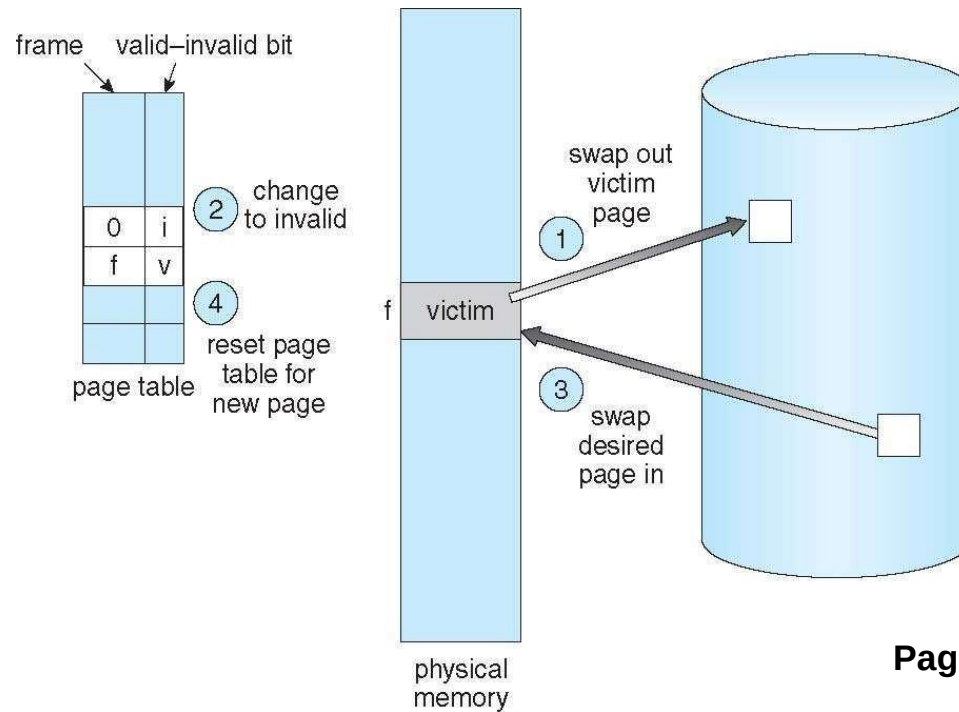


Need For Page Replacement

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

Basic Page Replacement

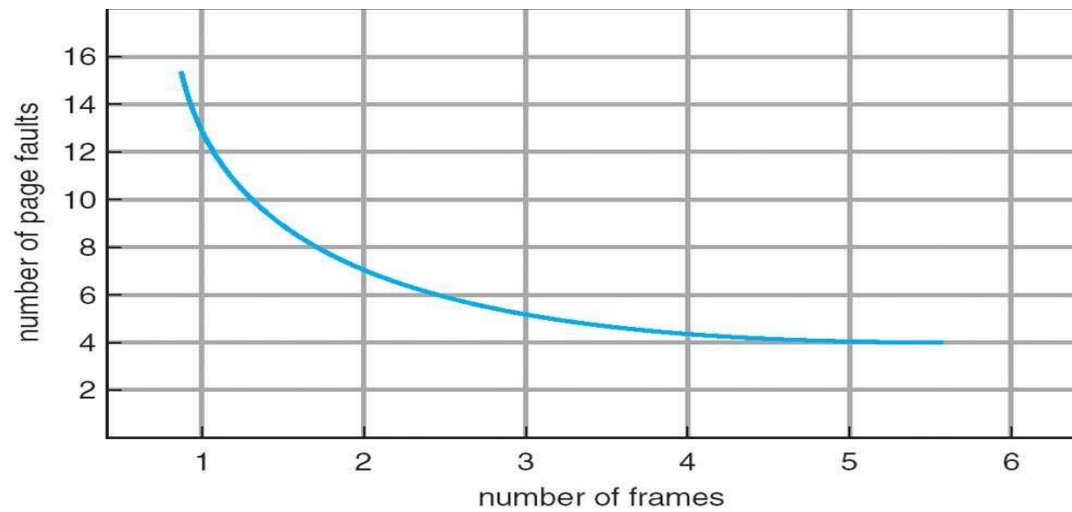


Page Replacement

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
- In all our examples, the reference string 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



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

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

15 page faults

- 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!
- Belady's Anomaly
- How to track ages of pages?
 - Just use a FIFO queue

FIFO Page Replacement

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
	0	0	0
		1	1

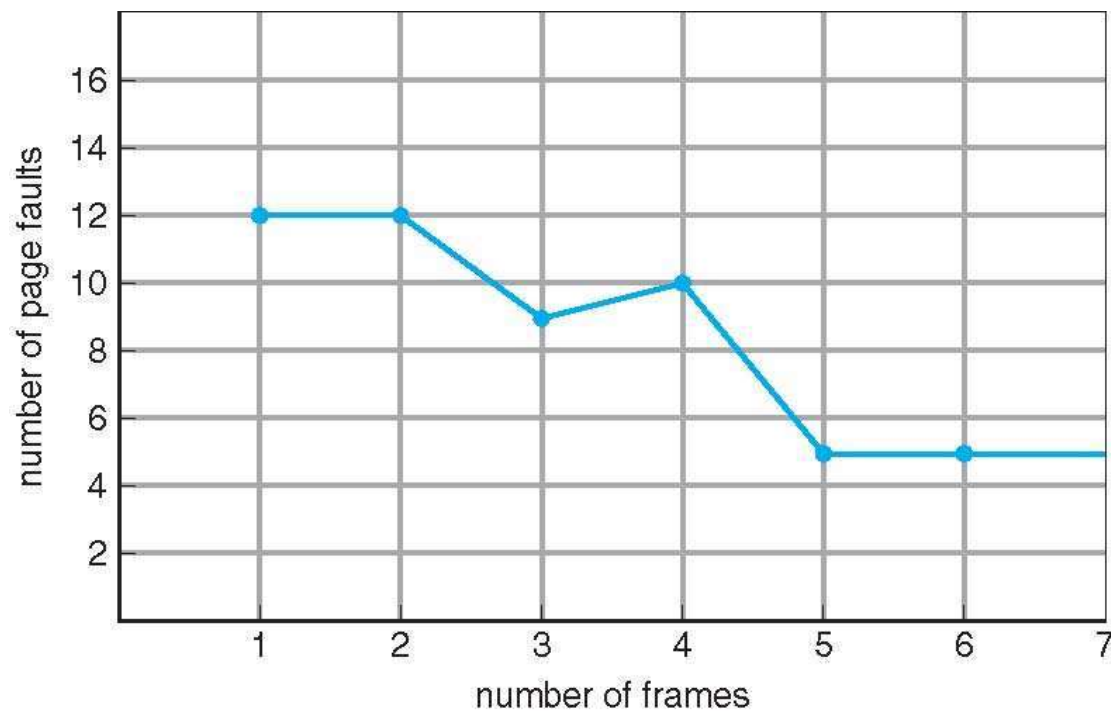
2	2	4	4	4	0
3	3	3	2	2	2
1	0	0	0	3	3

0	0
1	1
3	2

7	7	7
1	0	0
2	2	1

page frames

FIFO Illustrating Belady's Anomaly



Optimal Algorithm

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

Optimal Page Replacement

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

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

page frames

12 faults – better than FIFO but worse than OPT

Generally good algorithm and frequently used But how to implement?

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

reference string

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

2
1
0
7
4

stack
before
a

7
2
1
0
4

stack
after
b

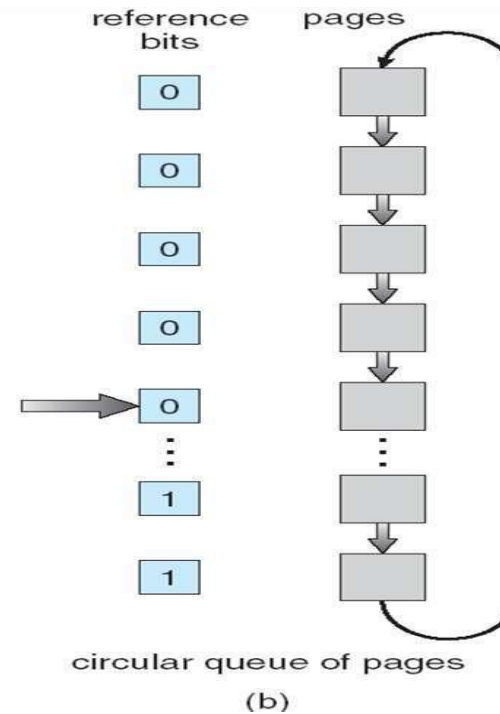
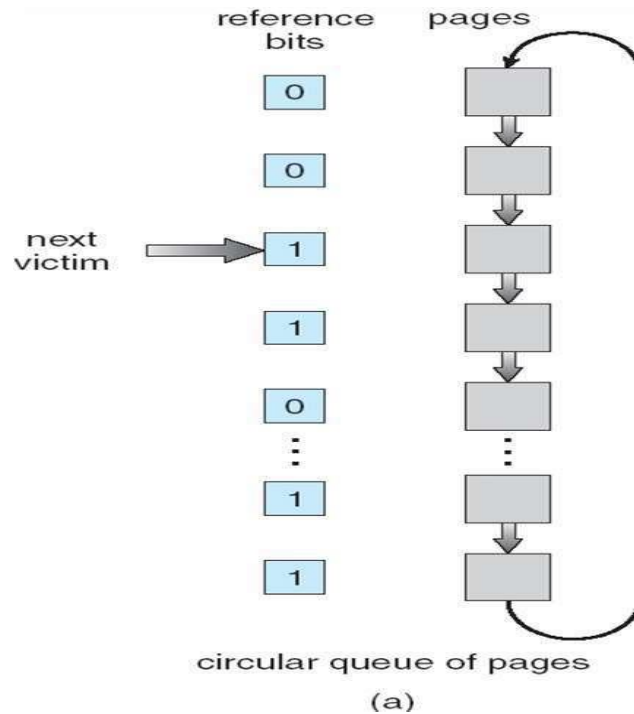
a ↑
b ↑

Use Of A Stack to Record The Most Recent Page References

LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **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)
- • We do not know the order, however
- **Second-chance algorithm**
 - – Generally FIFO, plus hardware-provided reference bit
 - – Clock replacement
 - – If page to be replaced has
 - Reference bit = 0 $\rightarrow$ replace it
 - reference bit = 1 then:
 - set reference bit 0, leave page in memory
 - replace next page, subject to same rule

Second-Chance (clock) Page-Replacement Algorithm



Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- – Not common
- L F U Algorithm: replaces page with smallest count
- M F U 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

- Keep a pool of free frames, always
 - – Then frame available when needed, not found at fault time
 - – Read page into free frame and select victim to evict and add to free pool
 - – When convenient, evict victim Possibly, keep list of modified pages
 - – When backing store otherwise idle, write pages there 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

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
 - – OS keeps copy of page in memory as I/O buffer
 - – Application keeps page in memory for its own work
- Operating system can given direct access to the disk, getting out of the way of the applications
 - – Raw disk mode
- Bypasses buffering, locking, etc

Allocation of Frames

- Each process needs minimum number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
 - – instruction is 6 bytes, might span 2 pages
 - – 2 pages to handle from
 - – 2 pages to handle to
- Maximum of course is total frames in the system
- Two major allocation schemes
 - – fixed allocation
 - – priority allocation
- Many variations

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 – Allocate according to the size of process
- – Dynamic as degree of multiprogramming,
- process sizes change S

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

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

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

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

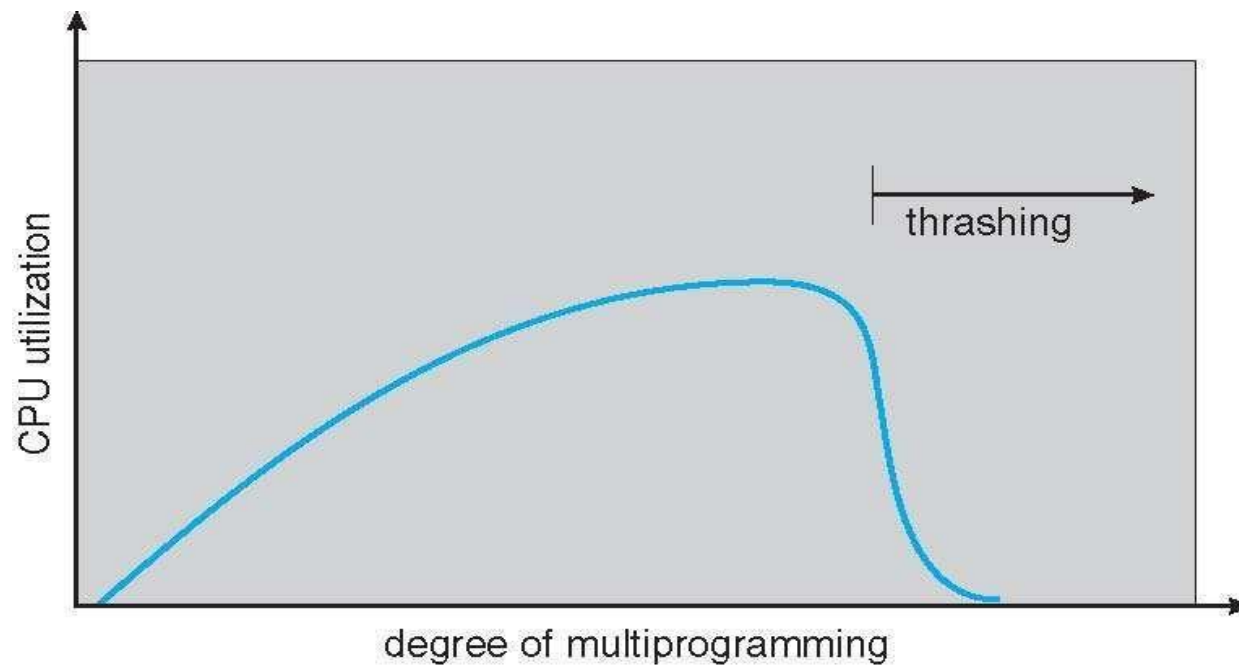
Non-Uniform Memory Access

- So far all memory accessed equally
- Many systems are NUMA – speed of access to memory varies
 - – Consider system boards containing CPUs and memory, interconnected over a system bus
- Optimal performance comes from allocating memory “close to” the CPU on which the thread is scheduled
 - – And modifying the scheduler to schedule the thread on the same system board when possible
- – Solved by Solaris by creating lgroups
- Structure to track CPU / Memory low latency groups
- Used my schedule and pager
- When possible schedule all threads of a process and allocate all memory for that process

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

Thrashing (Cont.)



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?
- ☐ size of locality > total memory size
 - – Limit effects by using local or priority page replacement

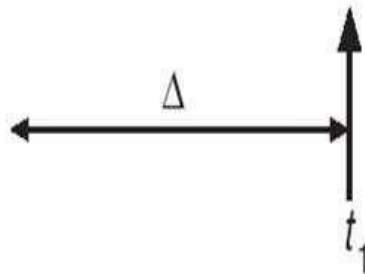
Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
Example: 10,000 instructions
 - WSS_i (working set of Process P_i) =
total number of pages referenced in the most recent Δ (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
 - $D = \sum WSS_i \equiv$ total demand frames
 - Approximation of locality
 - if $D > m \Rightarrow$ Thrashing
 - Policy if $D > m$, then suspend or swap out one of the processes
-

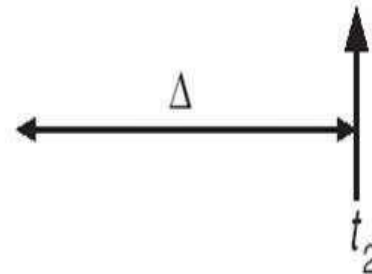
Working-set model

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



$$WS(t_1) = \{1, 2, 5, 6, 7\}$$



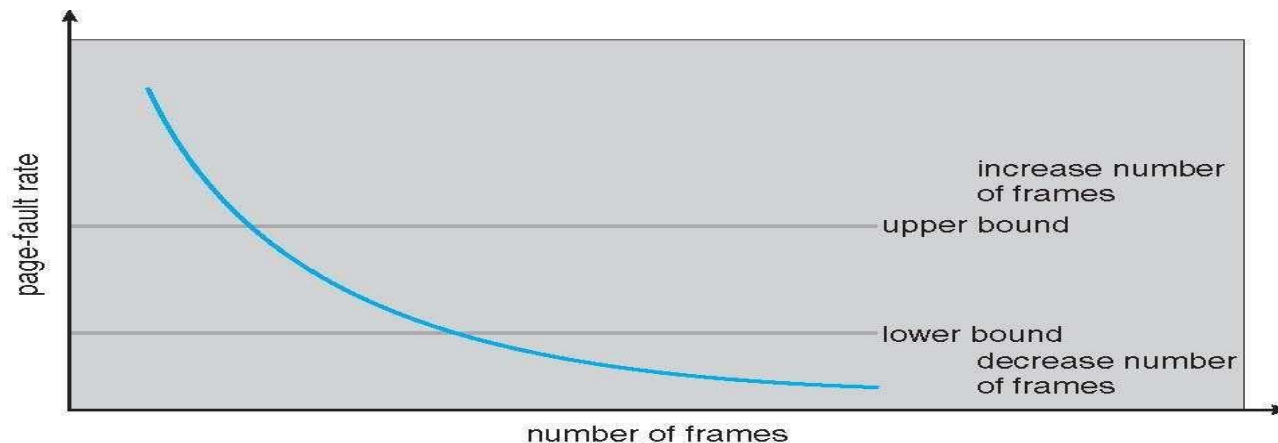
$$WS(t_2) = \{3, 4\}$$

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
 - 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?
- Improvement = 10 bits and interrupt every 1000 time units

Page-Fault Frequency

- More direct approach than WSS
- Establish “acceptable” page-fault frequency rate
- and use local replacement policy



Working Sets and Page Fault Rates

