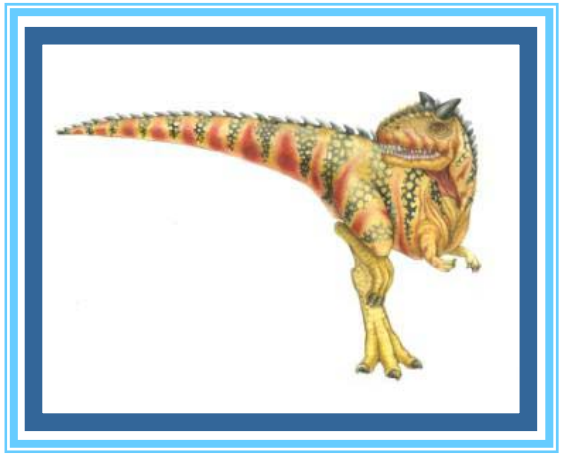
Topic 3: Memory Management



- Chapter 8: Main Memory
- Chapter 9: Virtual Memory

Contents

- **Background and Memory hierarchy**
- **Memory binding**
  - Logical and Physical Addresses
  - Memory binding stages
- **Memory Allocation Methods**
- **Paging and segmentation**
  - One and two level Paging
  - Inverted page tables and TLB
  - Segmented and hybrid memory models
- **Virtual Memory**
  - Required HW and SW
  - Allocation and page replacement algorithms
  - Thrashing and PFF strategy

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

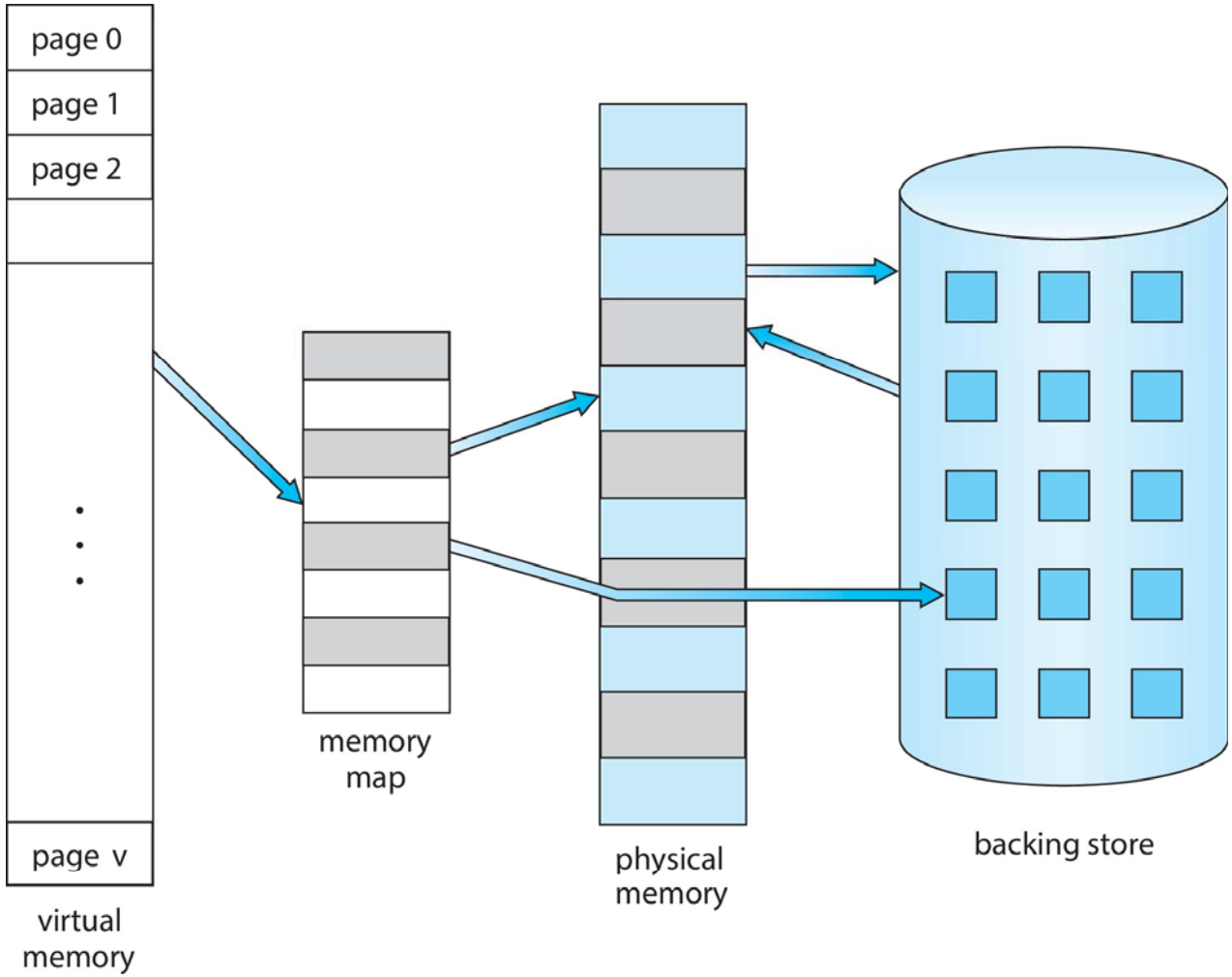
Virtual memory

- Process does not require complete loading
  - keep only the part in use in memory
  - rest of the part of the process stays in swap
    - Partition (unix) or file (windows) on the hard disk
- Works due to the locality concept
  - in space and time
  - the 90/10 rule of locality
- Separation of user logical memory from physical memory
  - Logical address space can therefore be much larger than physical address space
- Advantage
  - increases possible muliprogramming level
  - processes are not bounded by physical memory
  - reduction of swapping time
    - just loading/unloading parts of the program (not all)

Virtual memory (Cont.)

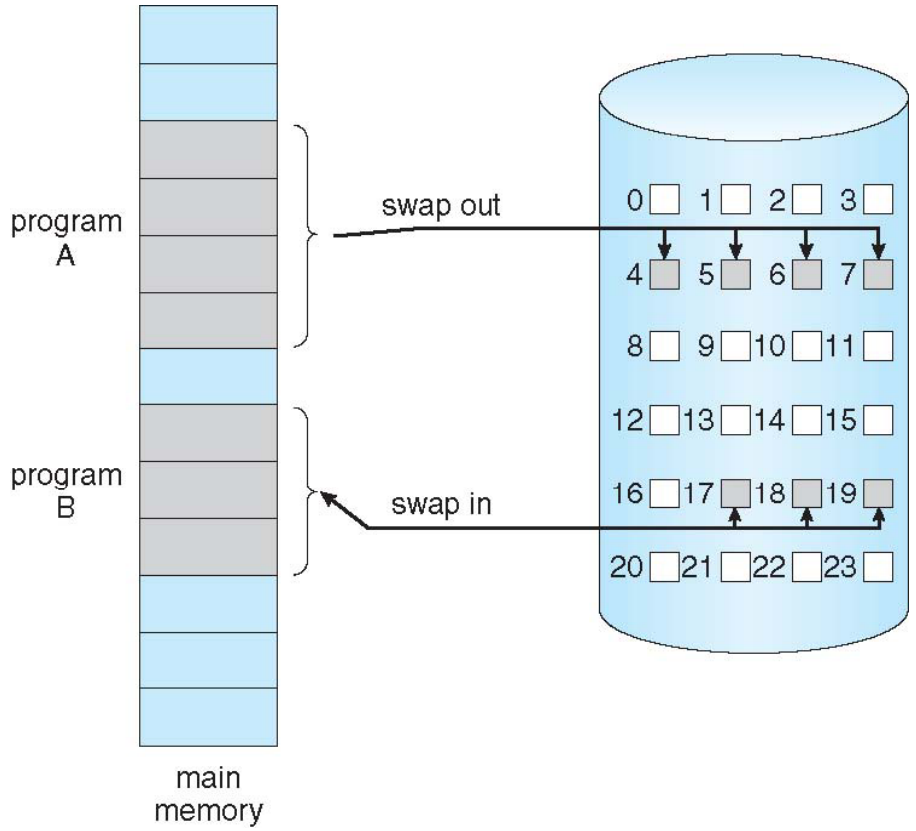
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging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

Virtual Memory That is Larger Than Physical Memory



Implementation (paged virtual memory)

- When process looks for logical address @L
  - If present in physical space (in a frame) →HIT
  - If not present → MISS (except page fault)
    - OS takes over executing RTI (real-time interrupt) solving the fault
    - RTI fetches page (swap-in) of the swap and loads into phys space
    - possible replacement if the physical space is full (swap-out)



Implementation (paged virtual memory)

- Implementation variants
  - on demand: only load a page if requested
    - 1st program instruction already generates a page fault
  - prepaging: on process creation load some pages
  - prefetch: on each page fault, load additional page

Virtual memory HW requirements

- More bits in page table
  - Bit P (present) indicates if page is loaded or not
  - Bit D (dirty) 1 if page has changed in memory
    - replacing a dirty page requires swap-out
    - if not dirty, we do not have to write to disk; it is already there
- Swap zone on HD and in DMA
  - swap zone may be
    - file (flexible in size) like in Windows
    - a partition (specific high performance file system) Linux

Virtual memory HW requirements

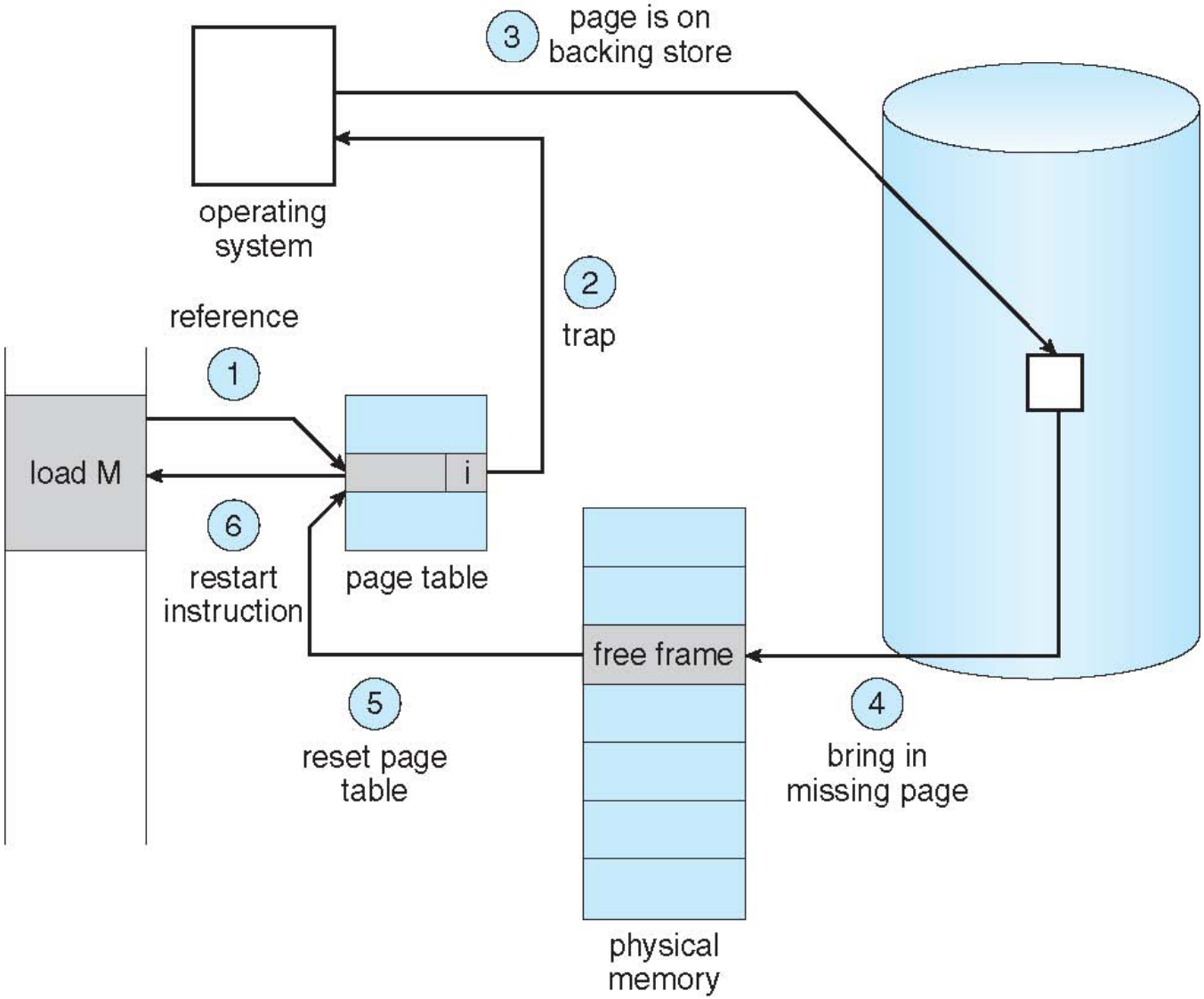
- Interruption mechanism
  - the MMU should load an exception on P=0
  - Resume instructions. Example:
    - instruction lw r5,0(r4) generates a miss in searching operand
    - instruction not completed → run RTI (swap-in)
    - resume instruction after RTI

Steps in Handling a Page Fault (Cont.)

- RTI dealing with page fault:

```
IF V=0 or insufficient permission THEN
        error message or finish process
IF V=1 and P=0 THEN
  search free frame in memory
  IF Not EXISTS free frame THEN
        run replacement algorithm (select victim page)
        IF victim page has D=1 (dirty) THEN
            SWAP-OUT the victim (several milliseconds)
            (during swap-out → process is blocked)
  SWAP-IN of page giving fault (blocked process)
  update page table
  Restore blocked process and resume instruction
```

Steps in Handling a Page Fault (Cont.)



What Happens if There is no Free Frame?

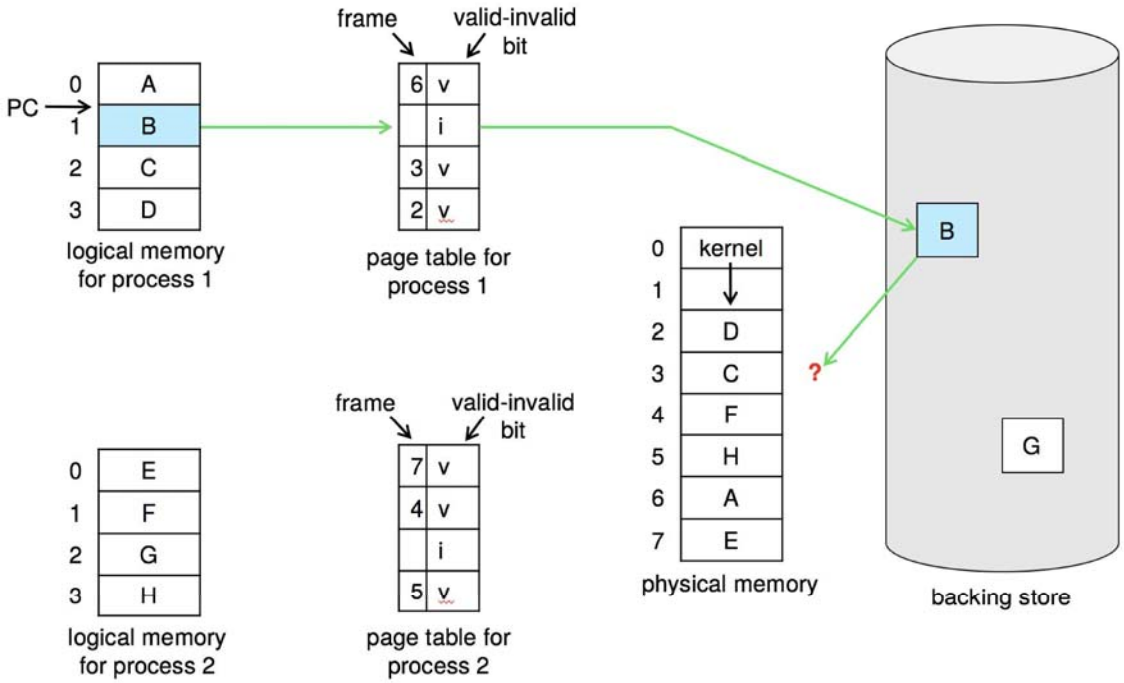
- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result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

Allocation and Replacement Algorithms

- Allocation algorithms (how many to each process)
  - Local: frames can be distributed statically
    - each process is assigned n frames according to criteria
    - on page fault → replace your own memory frames
    - drawback:
      - on process creation → redistribute frames
      - does not adapt to locality of each process
  - Global: assign frames to processes on request
    - all frames can be victim candidate
    - better adaptation to locality of each process
    - drawback: process with more page faults gets more frames

Allocation and Replacement Algorithms

- Replacement algorithms
  - On full memory → select victim
  - Allocation: select several candidate pages
  - Replace: select victim out of candidates
  - Working set: Assignment+replacement



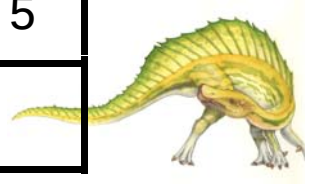
Replacement Algorithms

- Could bring Select victim out of candidates
- Minimize nº of future Page faults
- Algorithms (tested on reference strings)
  - Optimal: knowing the future. Not implementable
  - Random: select arbitrary victim
  - FIFO: first that entered, first to leave. Anomaly of Belady
  - LRU: Least Recently Used
  - Approximation LRU
  - NFU: Not Frequently Used
  - NRU: Not Recently Used
  - LFU: Least Frequently Used
  - Clock or second chance
  - MFU: Most Frequently Used

Replacement Algorithm: Optimal

- Optimal:
  - Select as victim the page that
    - Not needed anymore in the future
    - or is needed farthest in the future
  - requires knowing the future!!
  - example 7 page faults, the minimum
    - 3 frames in physical memory
    - sequence of accesses to these pages (reference string):

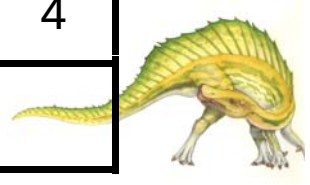
p:	1	2	3	4	1	2	5	1	2	3	4	5
frames	1	1	1	1	1	1	1	1	1	3	4	4
		2	2	2	2	2	2	2	2	2	2	2
			3	4	4	4	5	5	5	5	5	5
Miss	x	x	x	x			x			x	x	



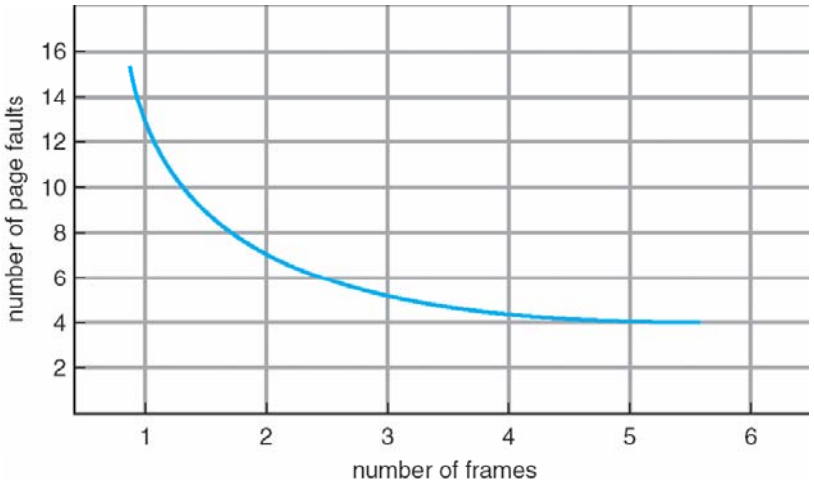
Replacement Algorithm: FIFO

- FIFO:
  - Select as victim the page that
    - has been most time in memory (the 1st enters, 1st leaves)
  - Two possible implementations:
    - link to each frame the arrival time
    - implement a software queue
      - the pages enter at one side and leave at the other
      - Example: 3 frames in physical memory -> 9 page faults

p:	1	2	3	4	1	2	5	1	2	3	4	5
Frames	1	1	1	2	3	4	1	1	1	2	5	5
		2	2	3	4	1	2	2	2	5	3	3
			3	4	1	2	5	5	5	3	4	4
Miss	x	x	x	x	x	x	x			x	x	



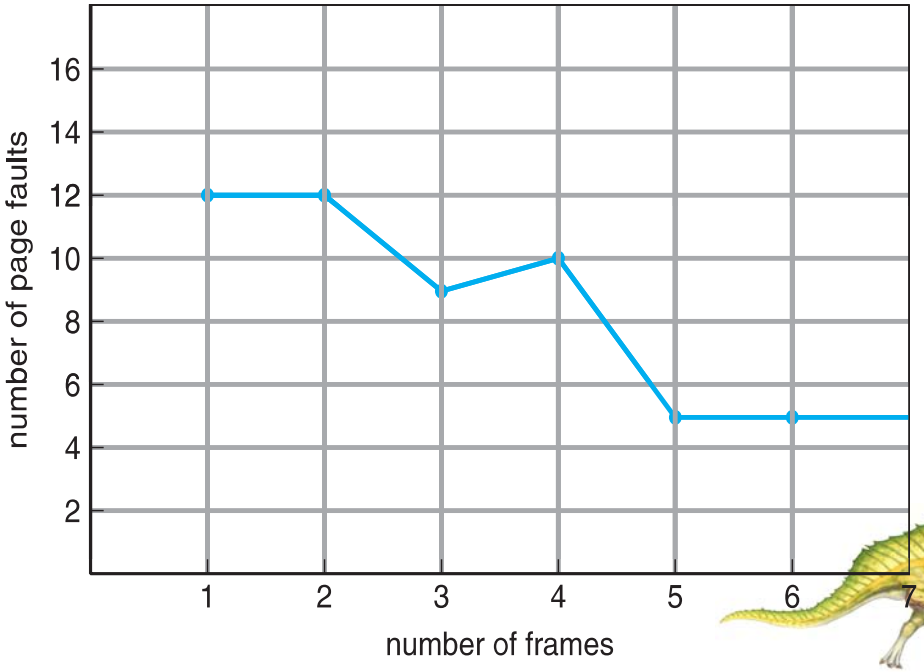
Belady's Anomaly



In principle: more nº of frames → less nº of faults

Page Faults Vs Number of Frames

- FIFO:
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**



Belady's Anomaly Example

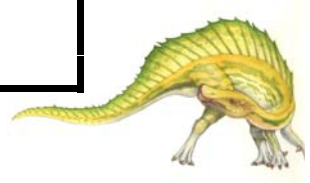
- Belady's Anomaly example:
  - 4 frames in physical memory (10 miss). Before 3 frames (9 miss)

p:	1	2	3	4	1	2	5	1	2	3	4	5
frames	1	1	1	1	1	1	2	3	4	5	1	2
		2	2	2	2	2	3	4	5	1	2	3
			3	3	3	3	4	5	1	2	3	4
				4	4	4	5	1	2	3	4	5
Miss	x	x	x	x			x	x	x	x	x	x

Replacement Algorithm: LRU

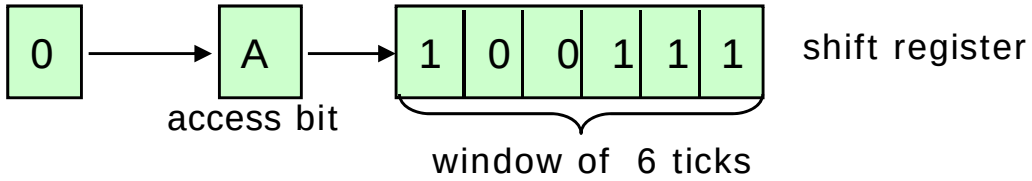
- LRU (Least Recently Used):
  - learning, approximating optimal algorithm
  - use the past to predict the future
    - remove that page that has been referenced least recently
  - Implementations:
    - link the time of use to each page, counters
    - ordered queue, at each reference put at the top.
  - Example:

p:	7	0	1	2	0	1	0	4	0	3	0	3
frames	7	7	7	0	1	2	2	1	1	4	4	4
		0	0	1	2	0	1	0	4	0	3	0
			1	2	0	1	0	4	0	3	0	3
Miss	x	x	x	x				x		x		



Replacement Algorithms: LRU App., NFU

- LRU Approximation page replacement
  - easier to implement:
    - add a bit A (access) to each entry of the page table
    - the bit A is initially set to 0
    - if the page is referenced change the bit to 1
    - or use shift register: each “n” clock ticks: copy A left bit and shift



    - each page has its shift register (e.g. a byte)
    - page with 010011 is used less recently than one with 100000
    - remove that page that has been referenced least recently

- NFU (Not Frequently Used):
  - sum bit A in a counting register
    - **each n ticks**, IF A=1 increase the counting register
    - victim: the page with smallest value in the counting register

Replacement Algorithms: NRU, LFU

- NRU (Not Recently Used)
  - as function of bits A and D (dirty) define classes, for selecting the victim:
    - inspect IF exist in class (0,0)
    - IF not, select from class (0,1)
    - IF not, select from class (1,0)
    - and IF no select from class (1,1)

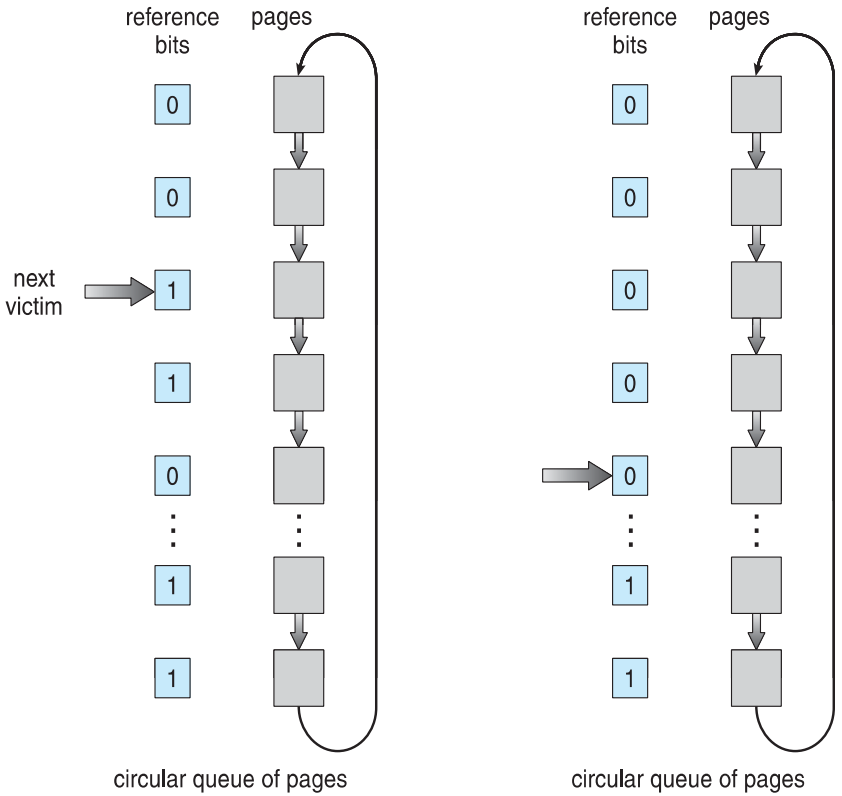
A	D	Class
0	0	0
0	1	1
1	0	2
1	1	3

- LFU (Least Frequently Used)
  - Counter for each frame. Increment on each access
  - Replace frame with lowest value
  - Problem:
    - IF a page is accessed a lot, gains many points
    - if not used anymore, still has all points
  - Solution: now and then divide counter by 2

Second-chance Algorithm

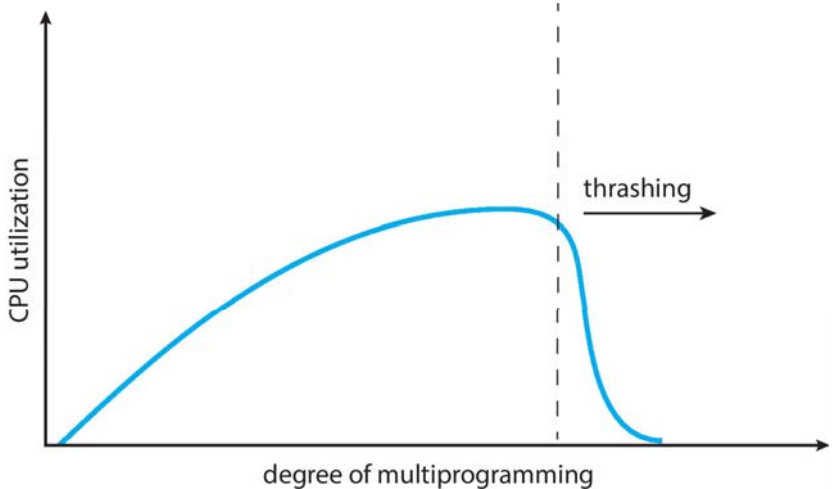
- Second-chance algorithm (Clock)
  - only apply **FIFO** to pages with A=0
  - a circular pointer scans the bits in a queue
    - IF found A=1 put it to 0 and follow (second chance)
    - IF found A=0 → this is the victim page
    - worst case: all pages have A=1 → do a complete round
    - example:

use a queue of page frames in order of filling them. If all frames are full, the last is linked to the first



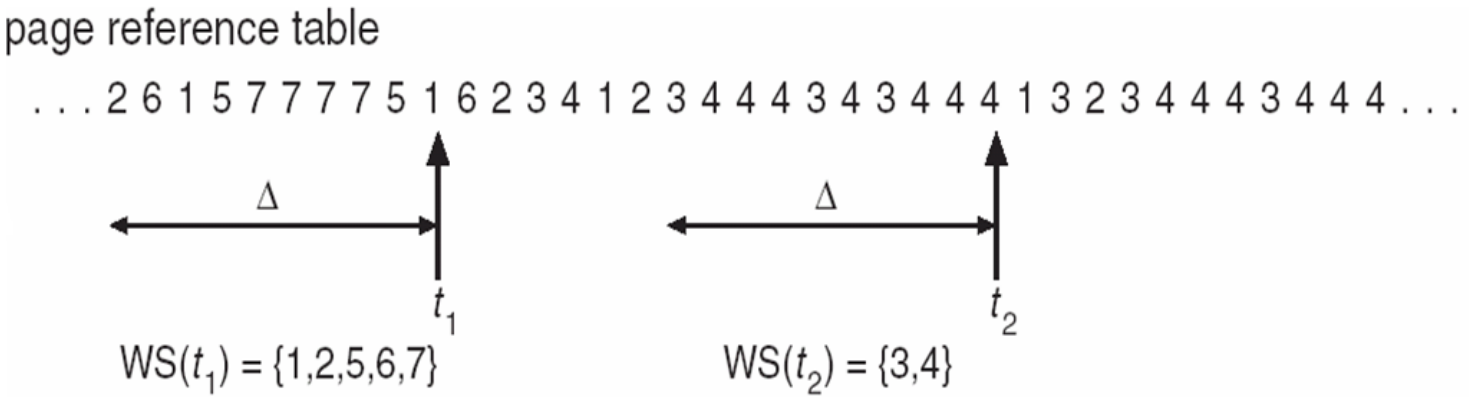
Thrashing

- 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



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

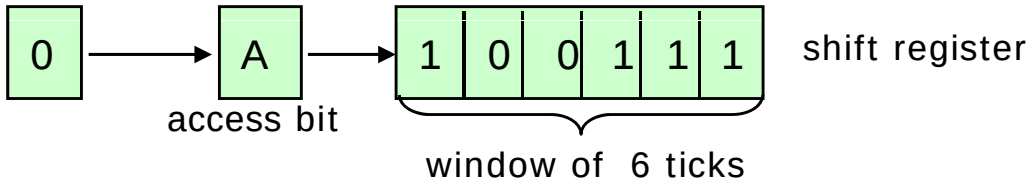


Working-Set Model (Cont.)

- $D = \Sigma\ WSS_i \equiv$ 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: Assignment+replacement

Working-Set Implementation

- Implementation of Working Set:
  - Use shift register for each entry of PT
  - nº of bits of the register → size of the window



  - each time interval (timer int.):
    - move the register of each page of the actual proc. to the right
    - insert access bit at the left of the page register
    - put access bit of each page to 0
  - register contains history of the use of the pages in each time interval:
    - pages not used have the complete register at 0 (not in WS)
    - used pages have a value different from 0
    - pages used most recently have the largest values

Operating System Examples

- Intel 32 and 64-bit Architectures
- Windows
- Solaris

Example: The Intel 32 and 64-bit Architectures

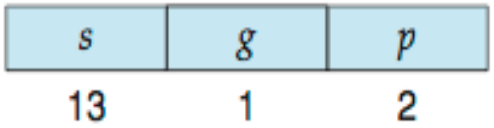
- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

Example: The Intel IA-32 Architecture

- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process
  - Divided into two partitions
    - First partition of up to 8 K segments are private to process (kept in **local descriptor table** (**LDT**))
    - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table** (**GDT**))

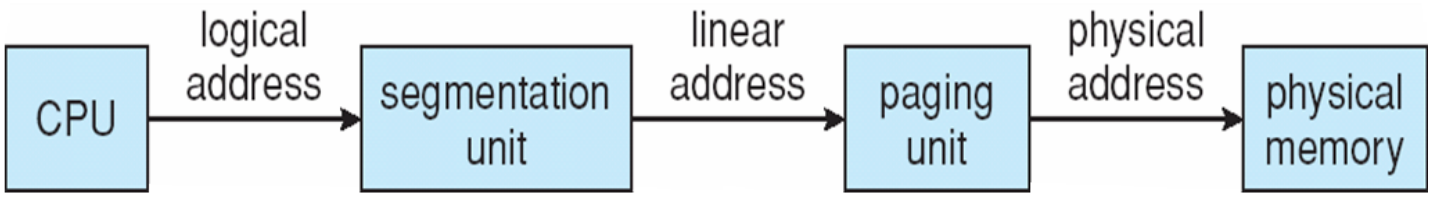
Example: The Intel IA-32 Architecture (Cont.)

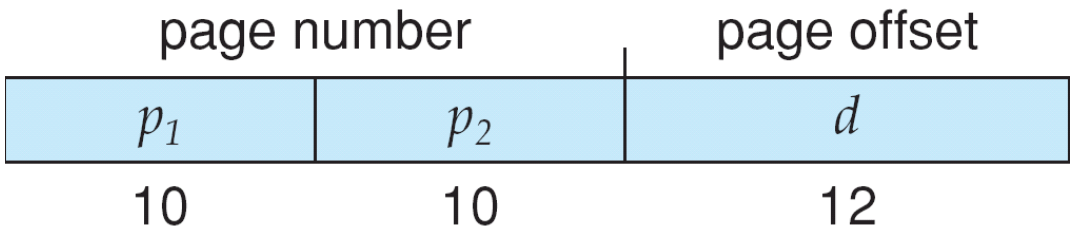
- CPU generates logical address
  - Selector given to segmentation unit
    - Which produces linear addresses



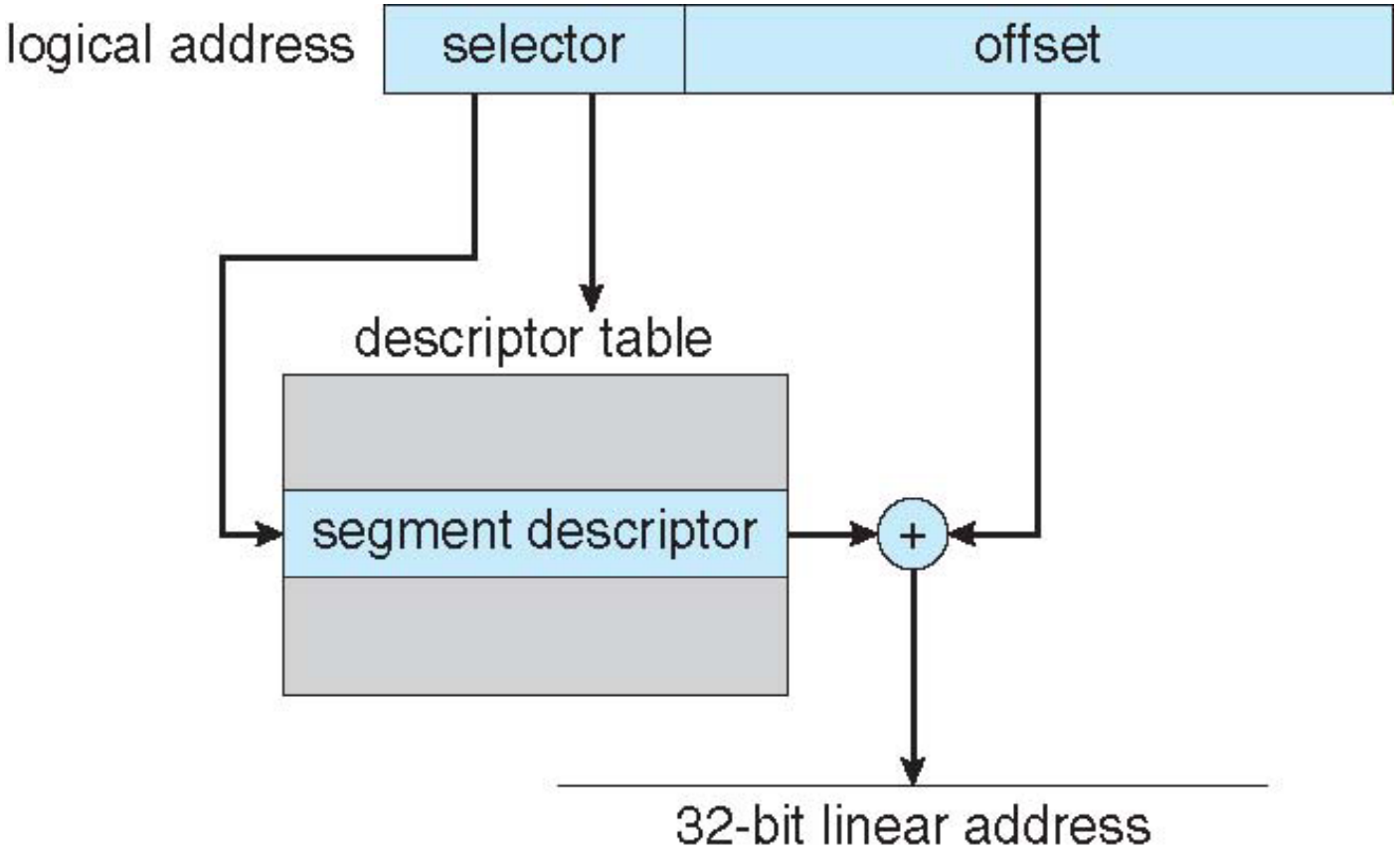
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB

Logical to Physical Address Translation in IA-32

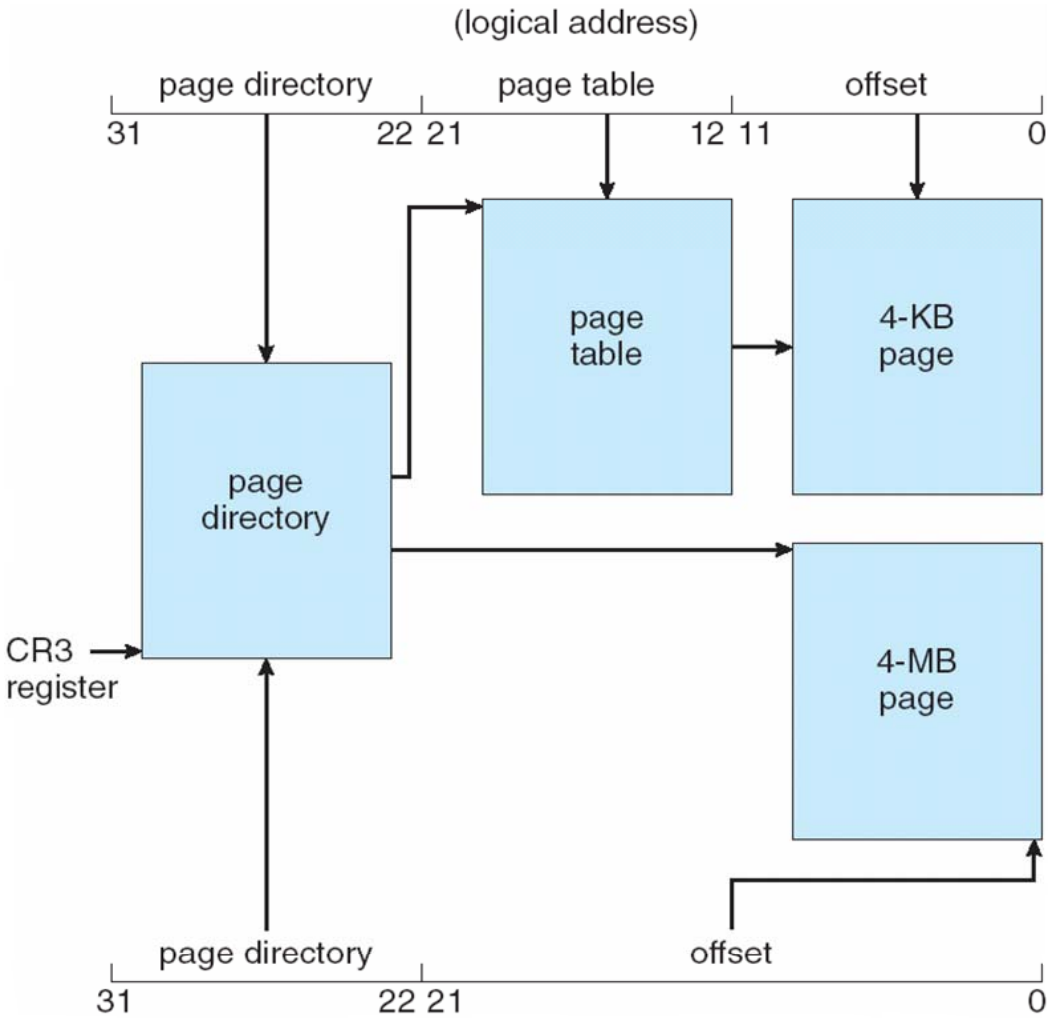




Intel IA-32 Segmentation

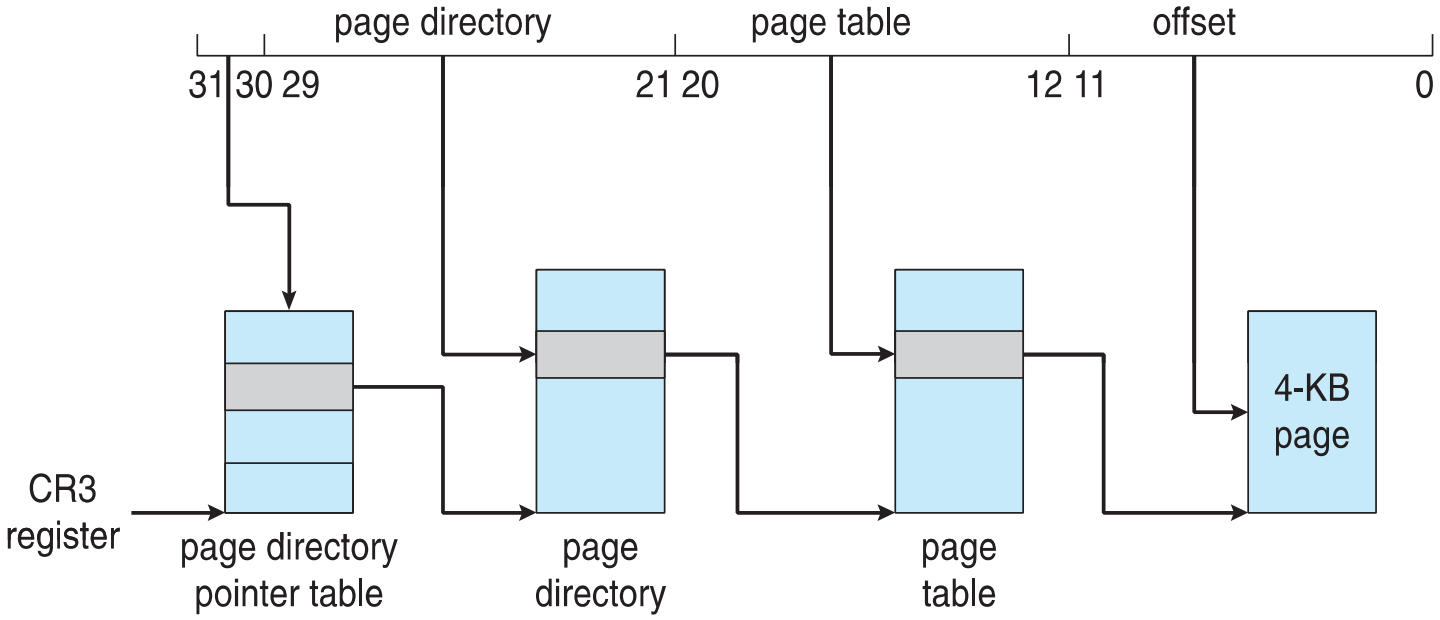


Intel IA-32 Paging Architecture



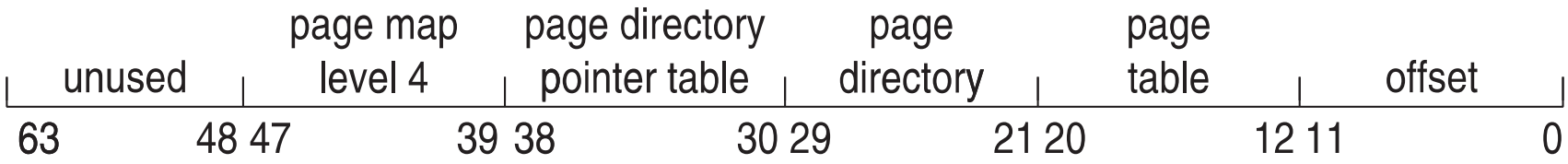
Intel IA-32 Page Address Extensions

- 32-bit address limits led Intel to create **page address extension** (**PAE**), allowing 32-bit apps access to more than 4GB of memory space
  - Paging went to a 3-level scheme
  - Top two bits refer to a **page directory pointer table**
  - Page-directory and page-table entries moved to 64-bits in size
  - Net effect is increasing address space to 36 bits – 64GB of physical memory



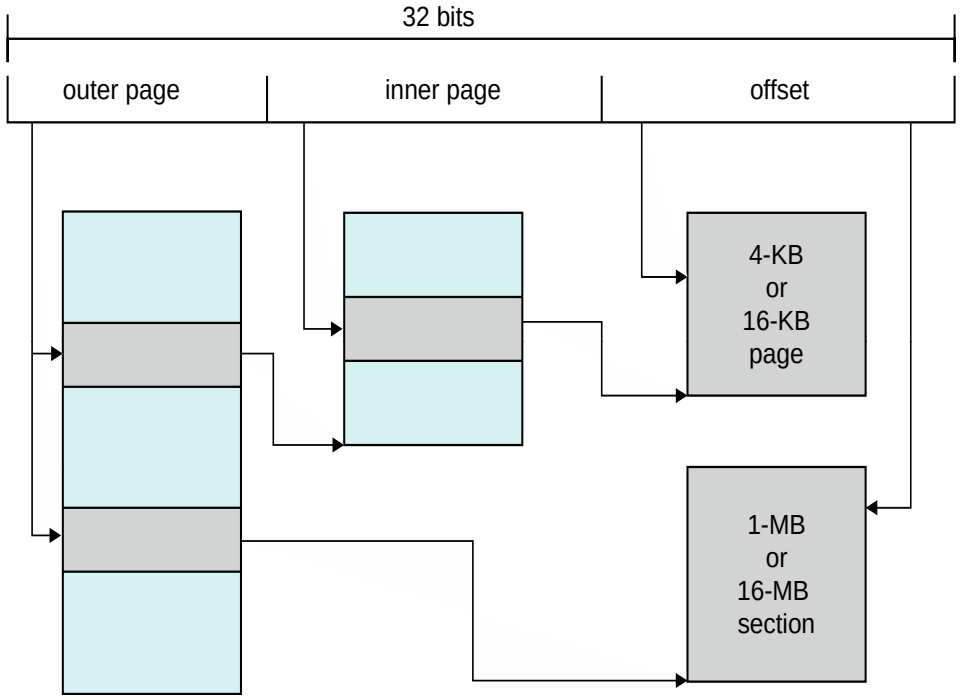
Intel x86-64

- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits



Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU

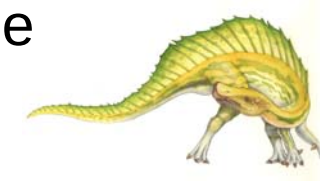


Windows

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic**

Oracle SPARC Solaris

- Consider modern, 64-bit operating system example with tightly integrated HW
  - Goals are efficiency, low overhead
- Based on hashing, but more complex
- Two hash tables
  - One kernel and one for all user processes
  - Each maps memory addresses from virtual to physical memory
  - Each entry represents a contiguous area of mapped virtual memory,
    - More efficient than having a separate hash-table entry for each page
  - Each entry has base address and span (indicating the number of pages the entry represents)



Oracle SPARC Solaris (Cont.)

- TLB holds translation table entries (TTEs) for fast hardware lookups
  - A cache of TTEs reside in a translation storage buffer (TSB)
    - Includes an entry per recently accessed page
- Virtual address reference causes TLB search
  - If miss, hardware walks the in-memory TSB looking for the TTE corresponding to the address
    - If match found, the CPU copies the TSB entry into the TLB and translation completes
    - If no match found, kernel interrupted to search the hash table
      - The kernel then creates a TTE from the appropriate hash table and stores it in the TSB, Interrupt handler returns control to the MMU, which completes the address translation.

Solaris

- Maintains a list of free pages to assign faulting processes
- **`Lotsfree`** – threshold parameter (amount of free memory) to begin paging
- **`Desfree`** – threshold parameter to increasing paging
- **`Minfree`** – threshold parameter to being swapping
- Paging is performed by **`pageout`** process
- **`Pageout`** scans pages using modified clock algorithm