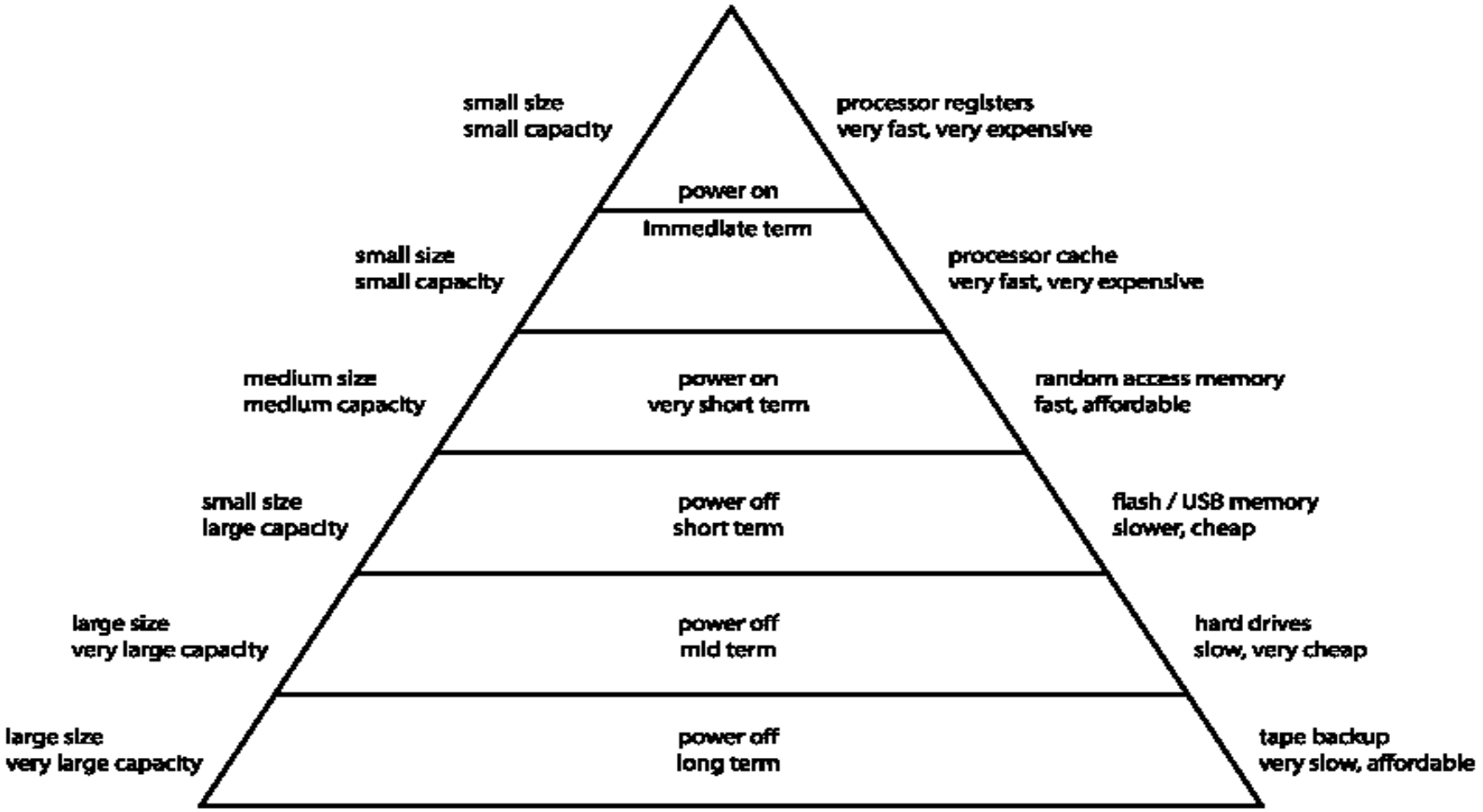
Memory Management

The entire program and data of a process must be in main memory for the process to execute.

How to keep the track of processes currently being executed?
Which processes to load when memory space is available?
How to load the processes that are larger than main memory?
How do processes share the main memory?

OS component that is responsible for handling these issues is a memory manager.

Computer Memory Hierarchy



Memory Manager

- The part of the operating system that manages the memory hierarchy is called the **memory manager**.
- Its job are:
  - to keep track of which parts of memory are in use and which parts are not in use,
  - to allocate memory to processes when they need it and de-allocate it when they are done,
  - to manage **swapping** between main memory and disk when main memory is too small to hold all the processes.

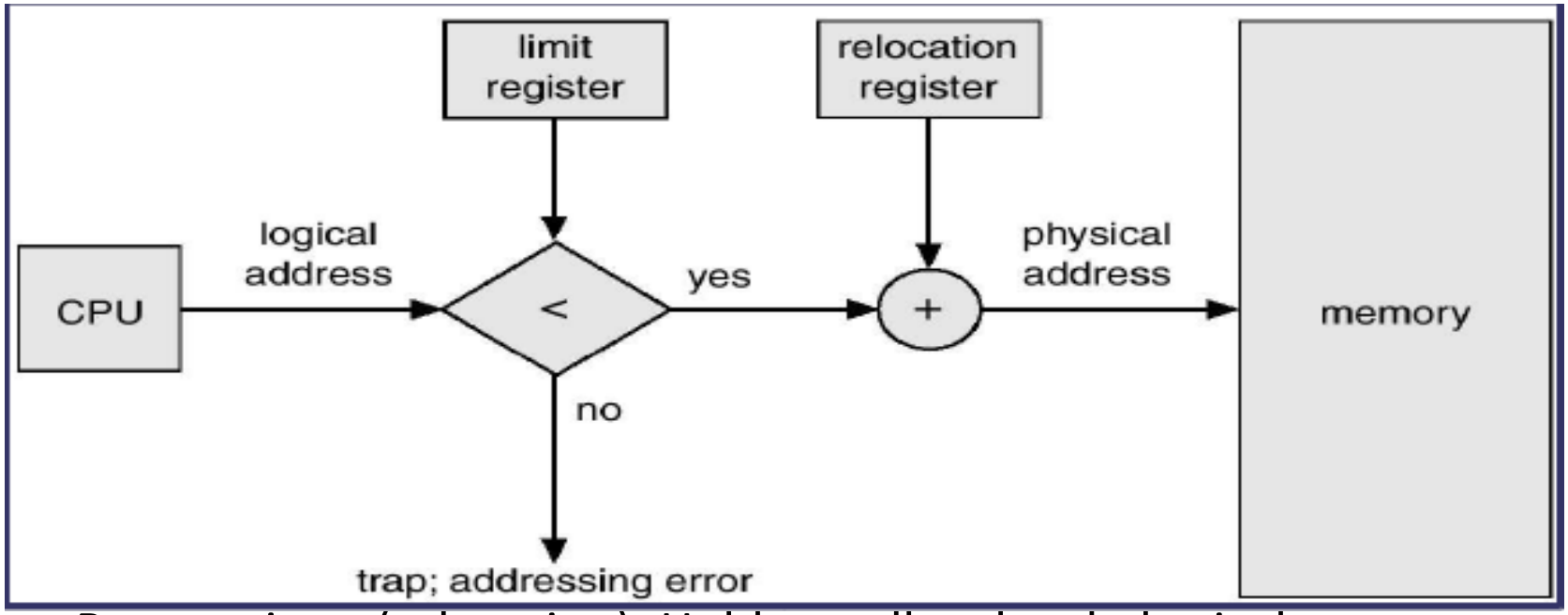
Memory Management System

- Memory management systems can be divided into two classes:
  - Those that move processes back and forth between main memory and disk during execution (swapping and paging),
  - and those with out swapping and paging.
- The latter are simpler, so we will study them first.

Memory Addresses

- Logical Address: The address generated by CPU. Also known as virtual address.
- Physical Address: Actual address loaded into the memory address register.
- Mapping from logical address to physical address is relocation.
- Two registers:
  - Base and Limit are used in mapping.
- This mechanism also used in memory protection .
  - memory outside the range are protected.

Address Translation



- Base register (relocation): Holds smallest legal physical memory address.
- Limit register: contain the size of range.
- Example:
  - logical address = 300
  - if Base = 1500
    - physical address = 1500 + 300 = 1800.

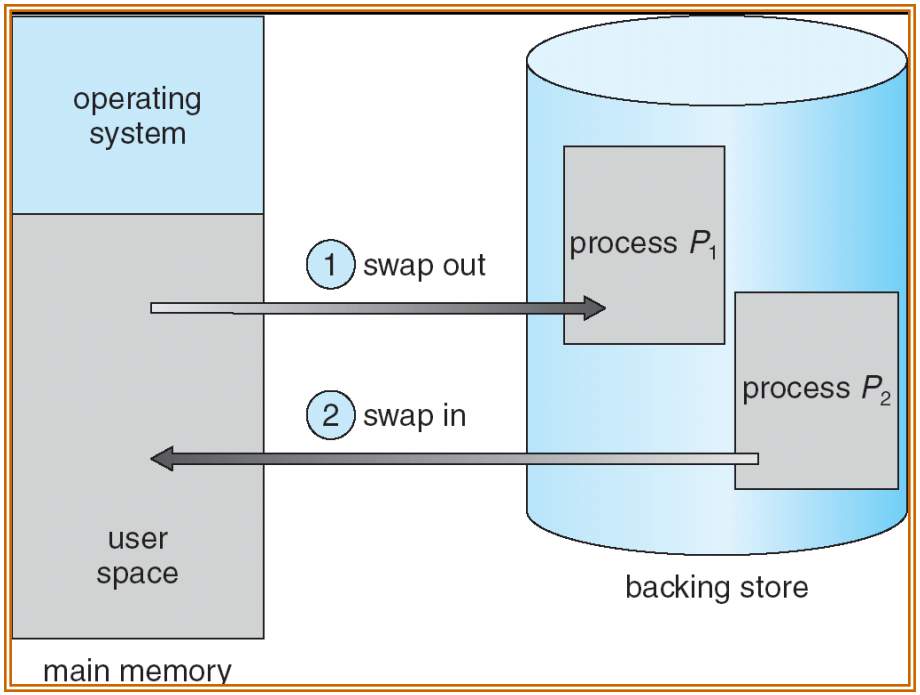
Virtual Memory

- The basic idea behind the virtual memory is that the combined size of the of the program, data, and stack may exceed the amount of physical memory available for it.
- The OS keeps those part of the program currently in use in main memory, and the rest on the disk.

Review: 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 and Microsoft Windows.

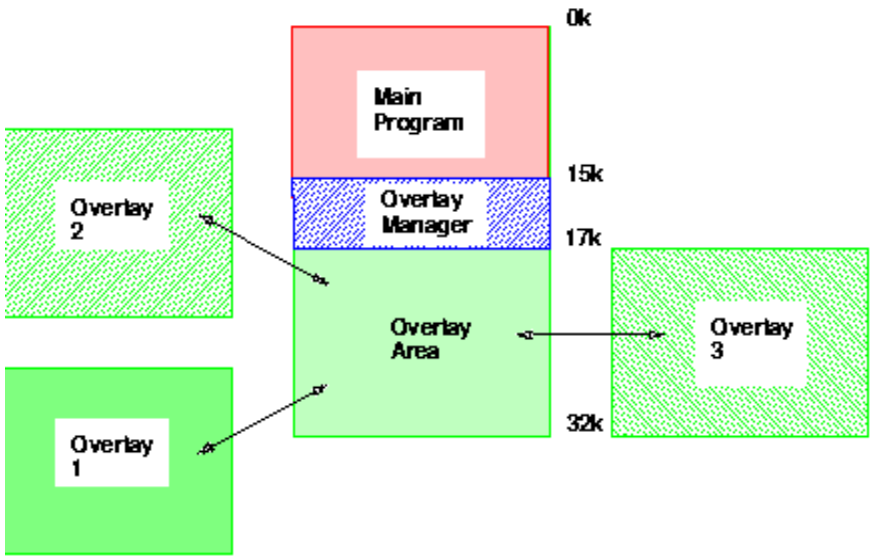
Schematic view of swapping



Review: Overlays

- Keep in memory only those instructions and data that are needed at any given time.
- Needed when process is larger than amount of memory allocated to it.
- Implemented by user, no special support from operating system; programming design of overlay structure is complex.

Overlaying

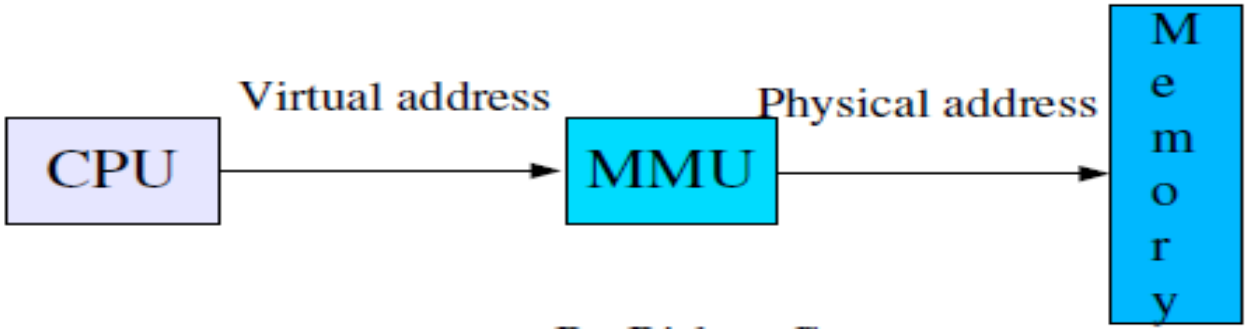

0k
Main Program
15k
Overlay Manager
17k
Overlay 2
Overlay Area
Overlay 3
Overlay 1
32k

Virtual Memory

- Virtual memory can be implemented by two most commonly used methods :
  - Paging
  - Segmentation
  - or mix of both.

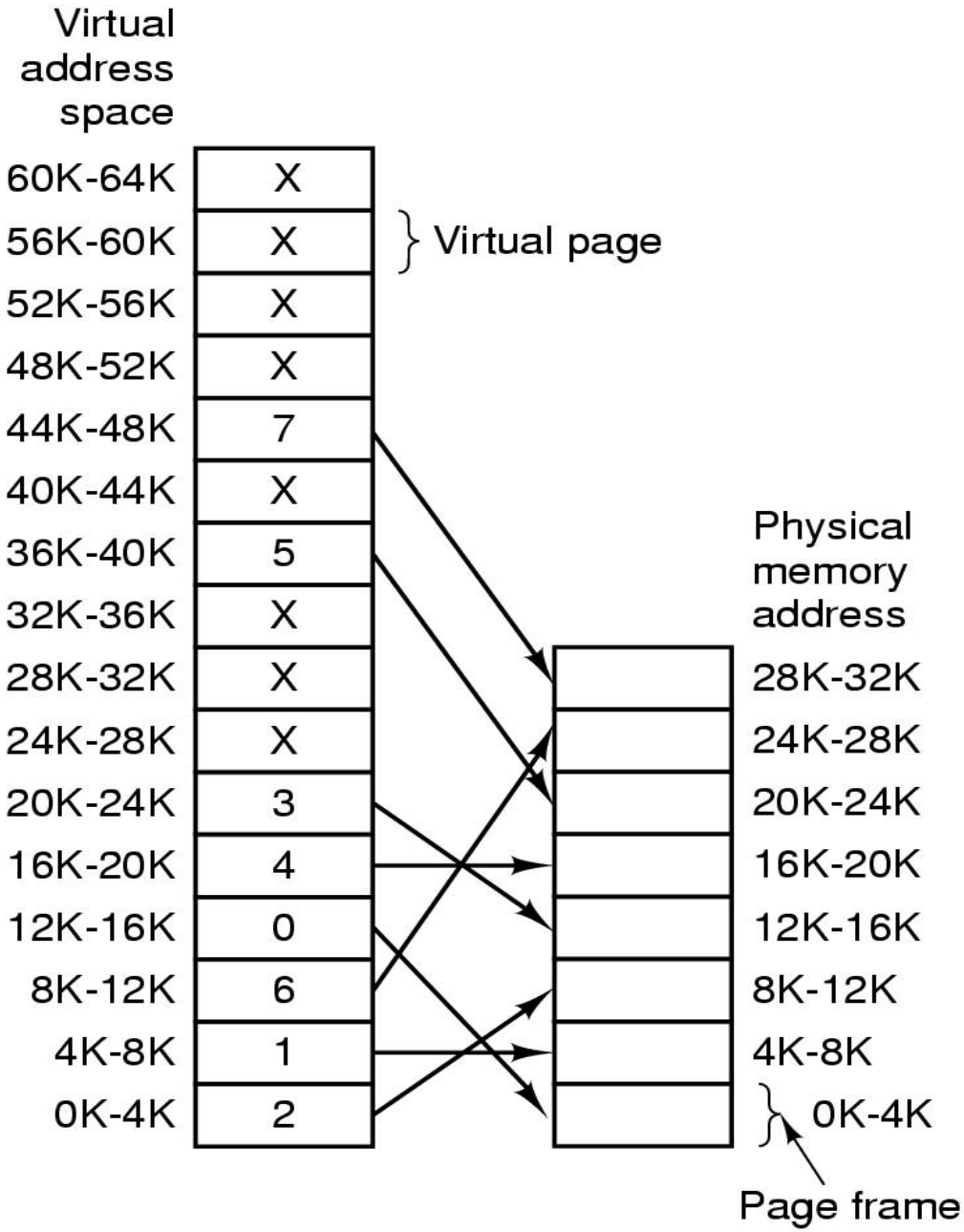
Background
MMU

- The run time mapping from virtual address to physical address is done by hardware devices called memory-management-unit (MMU).



Paging

- Logical address space of a process can be non-contiguous;
  - process is allocated physical memory wherever the latter is available.
  - Divide physical memory into fixed size blocks called ***frames***
    - size is power of 2, 512 bytes - 8K
  - Divide logical memory into same size blocks called ***pages.***
    - Keep track of all free frames.
    - To run a program of size n pages, find n free frames and load program.
  - Set up a page table to translate logical to physical addresses.
  - Note:: Internal Fragmentation possible!!



This example shows 64KB program can run in 32KB physical Memory. The complete copy is stored in the disk and the pieces can be brought into memory as needed.

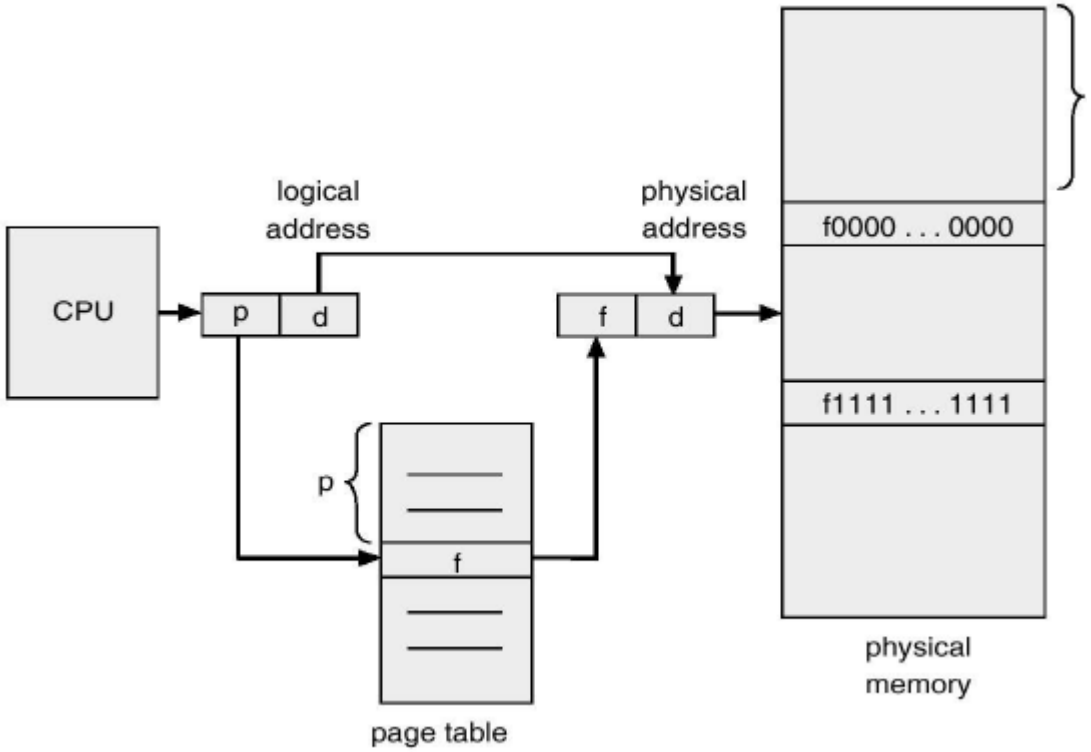
Paging

- With 64KB of virtual address space and 32KB of physical memory and 4KB page size, we get 16 virtual pages and 8 frames.
- What happen in following instruction?
- MOV REG, 0
  - This virtual address, 0, is sent to the MMU. The MMU sees that this virtual address falls in page 0 (0-4095), which is mapping to frame 2 (8192 -12287). Thus the address 0 is transformed to 8192 and output address is 8192.

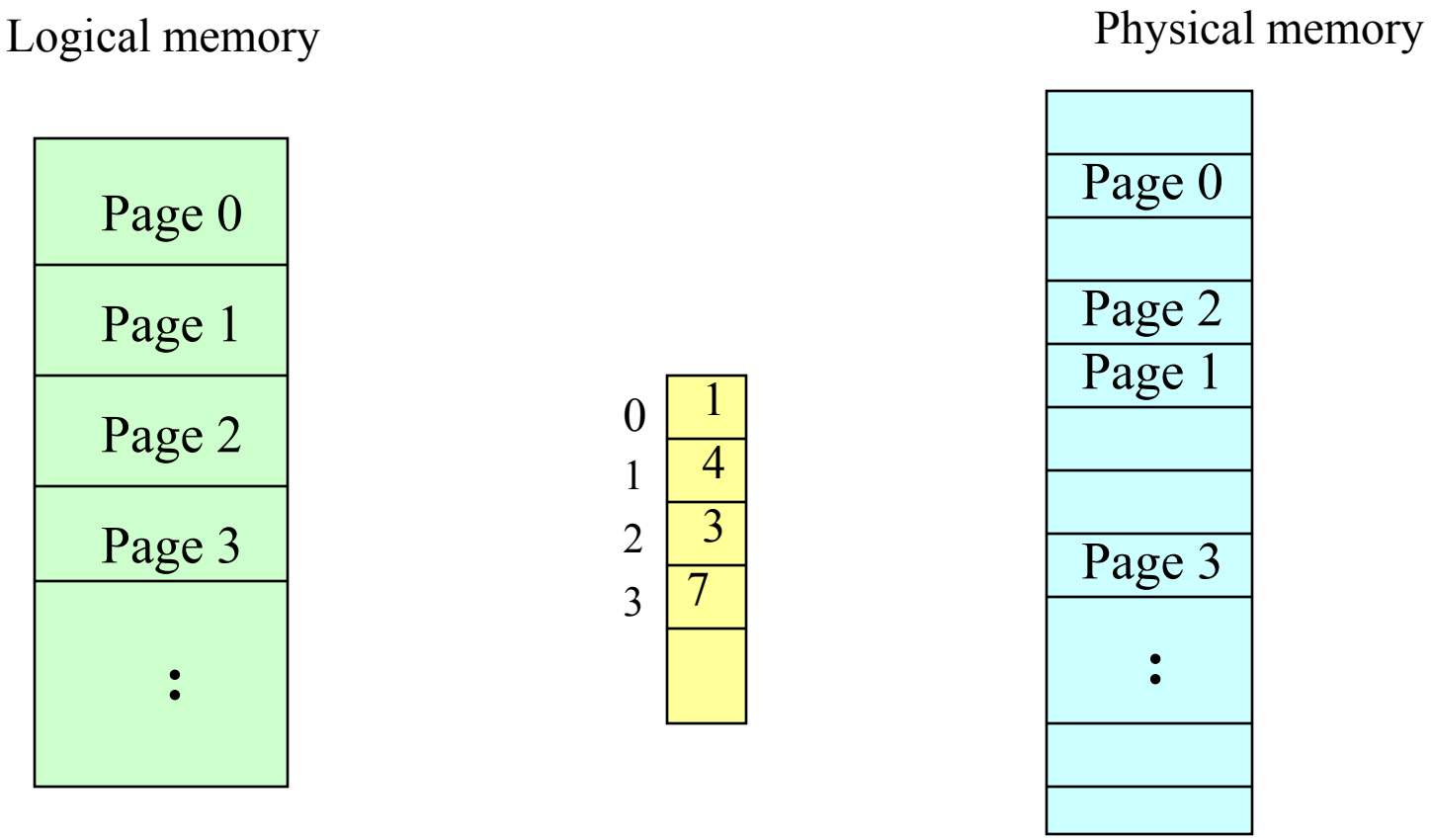
Address Translation

- 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

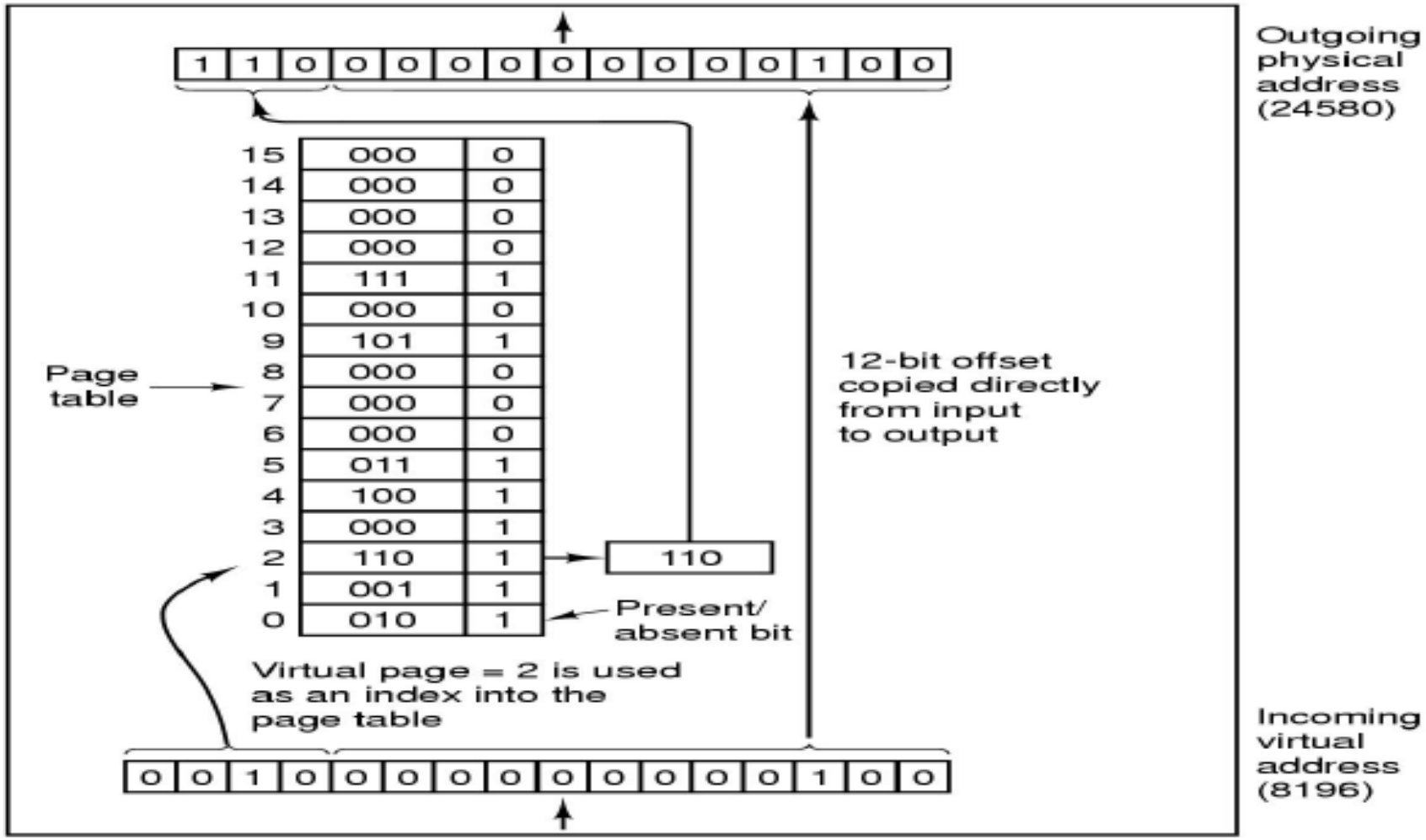
Address Translation



Example of Paging



Example


Outgoing physical address (24580)
1 1 0 0 0 0 0 0 0 0 0 0 1 0 0
15 000 0
14 000 0
13 000 0
12 000 0
11 111 1
10 000 0
9 101 1
8 000 0
7 000 0
6 000 0
5 011 1
4 100 1
3 000 1
2 110 1
1 001 1
0 010 1
Page table
110
Present/ absent bit
12-bit offset copied directly from input to output
Virtual page = 2 is used as an index into the page table
0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
Incoming virtual address (8196)

Page Tables

For each process, page table stores the number of frame, allocated for each page.

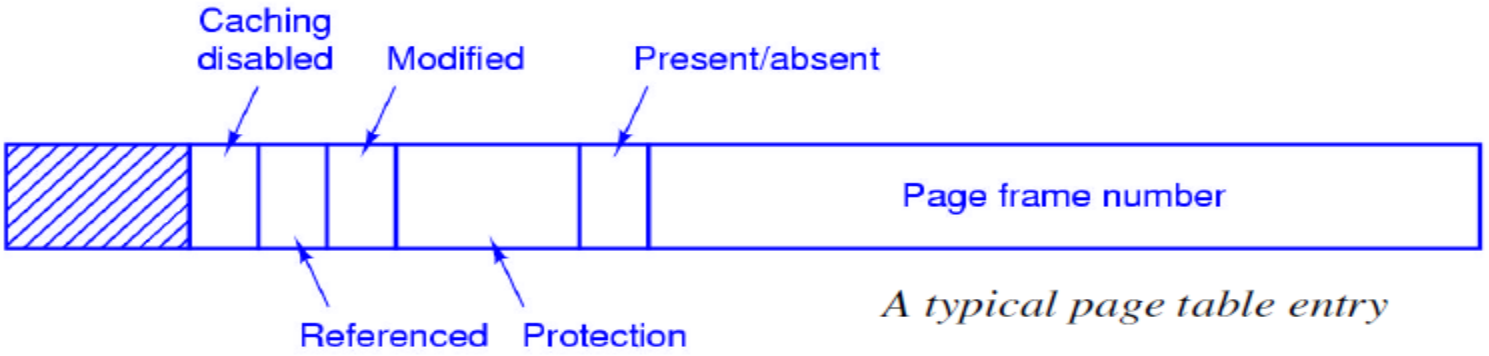
- The purpose of the page table is to map virtual pages into pages frames. This function of page table can be represented in mathematical notation as:

  page_frame = page_table(page_number)

- The virtual page number is used as an index into the page table to find the corresponding page frame.

Page Table Structure

- The exact layout of page table entry is highly machine dependent, but more common structure for 32-bit system is as:



A typical page table entry

- Frame number: The goal is to locate this value

- **Present/absent bit:** If present/absent bit is present, the virtual addresses is mapped to the corresponding physical address. If present/absent is absent the trap is occur called page fault.
- **Protection bit:** Tells what kinds of access are permitted read, write or read only.
- **Modified bit (dirty bit):** Identifies the changed status of the page since last access; if it is modified then it must be rewritten back to the disk.
- **Referenced bit:** set whenever a page is referenced; used in page replacement.
- **Caching disabled**: used for that system where the mapping into device register rather than memory.

Page Tables Issues

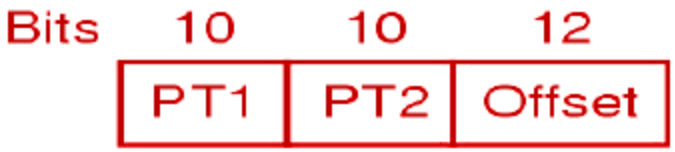
- Size of page table.
  - Most modern computers support a large virtual-address space (2^32 to 2^64). If the page size is 4KB, a 32-bit address space has 1 million pages. With 1 million pages, the page table must have 1 million entries.
  - think about 64-bit address space???
- Efficiency of mapping.
  - If a particular instruction is being mapped, the table lookup time should be very small than its total mapping time to avoid becoming CPU idle.
- What would be the performance, if such a large table have to load at every mapping.

How to handle these issues?

Multilevel Page Tables

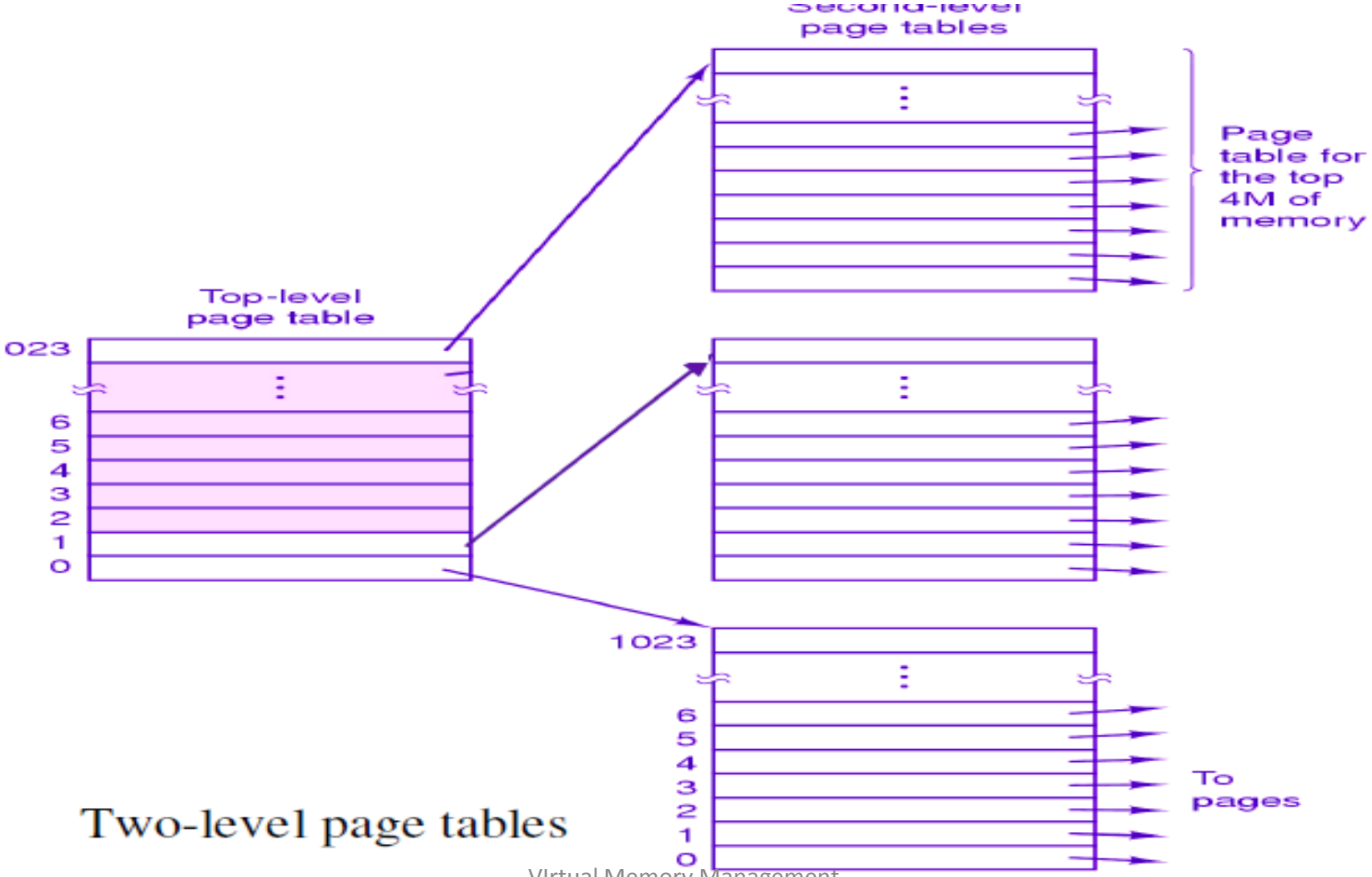
- To get around the problem of having to store huge page tables in memory all the time, many computers use the multilevel page table in which the page table itself is also paged.
- Pentium II -2 level, 32-bit Motorola -4 level, 32 -bit SPARC-3 level etc.

- Example: Two-Level Page Tables
- A 32-bit virtual address space with a page size of 4 KB, the virtual address space is partitioned into a 10-bit PT1 field, a 10-bit PT2 field, and a 12-bit offset field.



- The top level have 1024 entries, corresponding to PT1. At mapping, it first extracts the PT1 and uses this value as an index into the top level page table. Each of these entries have again 1024 entries, the resulting address of top-level yields the address or page frame number of second-level page table.

Multilevel Page tables



Two-level page tables

Translation Look aside Buffers (TLBs)

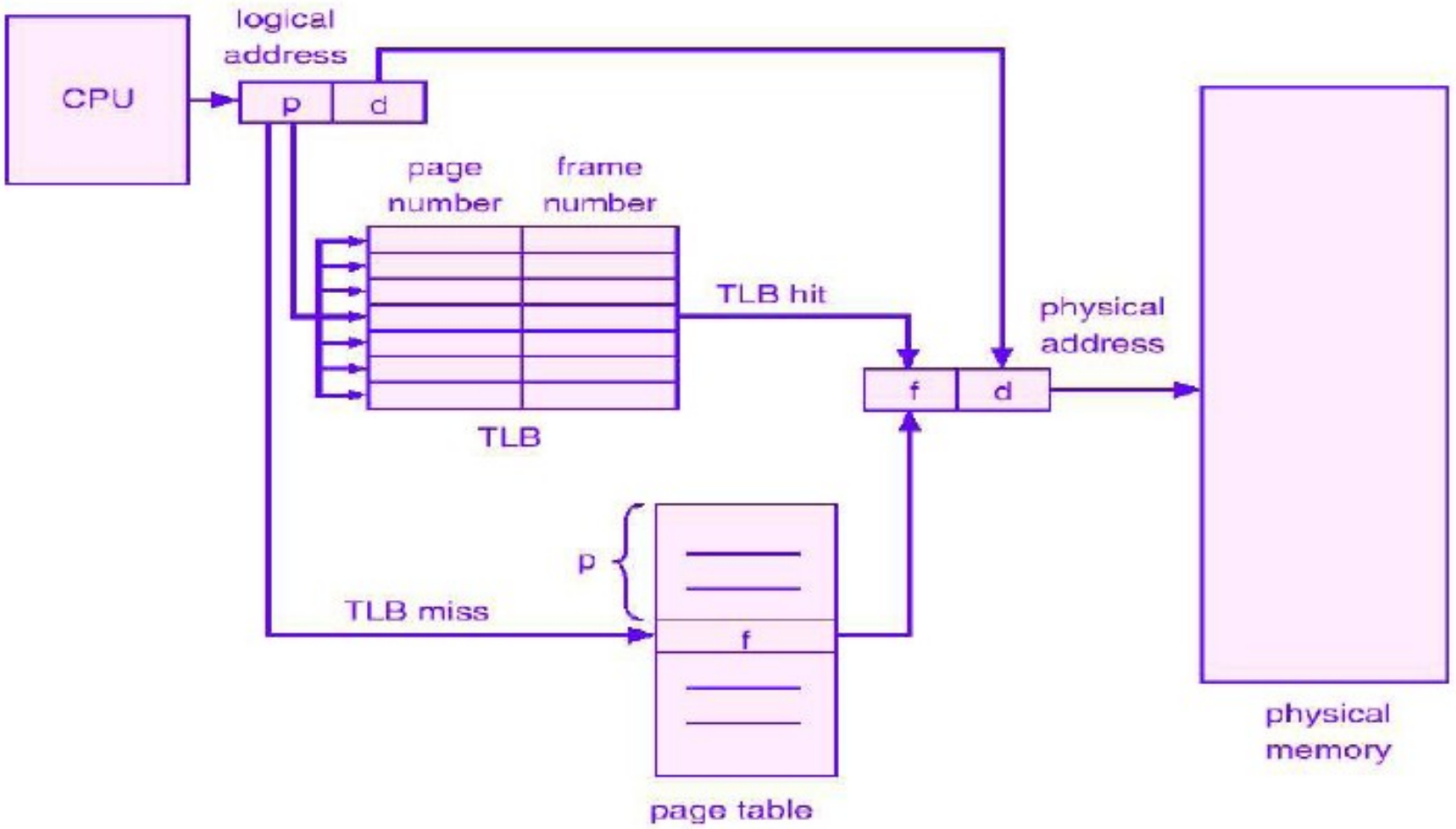
How to speed up address translation?

Locality: Most processes use large number of reference to small number of pages.

- The small, fast, lookup hardware cache, called TLB is used to overcome this problem, by mapping logical address to physical address without page tables.
- The TLB is associative, high-speed, memory; each entry consists information about virtual page-number and physical page frame number.
- Typical sizes: only 64 to 1024 entries.

- Key of improvement: Parallel search for all entries.

TLB


CPU
logical address
p
d
page number
frame number
TLB hit
physical address
f
d
TLB
p
TLB miss
f
page table
physical memory

TLB

- When logical address is generated by CPU, its page number is presented to the TLB; if page number is found (TLB hit), its frame number is immediately available, the whole task would be very fast because it compare all TLB entries simultaneously.

- If the page number is not in TLB (TLB miss), a memory reference to the page table must be made. It then replace one entry of TLB with the page table entry just referenced. Thus in next time, for that page, TLB hit will found.

Advantages/Disadvantages of Paging

- Advantages:
  - Fast to allocate and free:
    - Alloc: keep free list of free pages, grab first page in the list.
    - Free: Add pages to free list.
  - Easy to swap-out memory to disk.
    - Frame size matches disk page size.
    - Swap-out only necessary pages.
    - Easy to swap-in back from disk.
- Disadvantages:
  - Additional memory reference.
  - Page table are kept in memory.
  - Internal fragmentation: process size does not match allocation size.

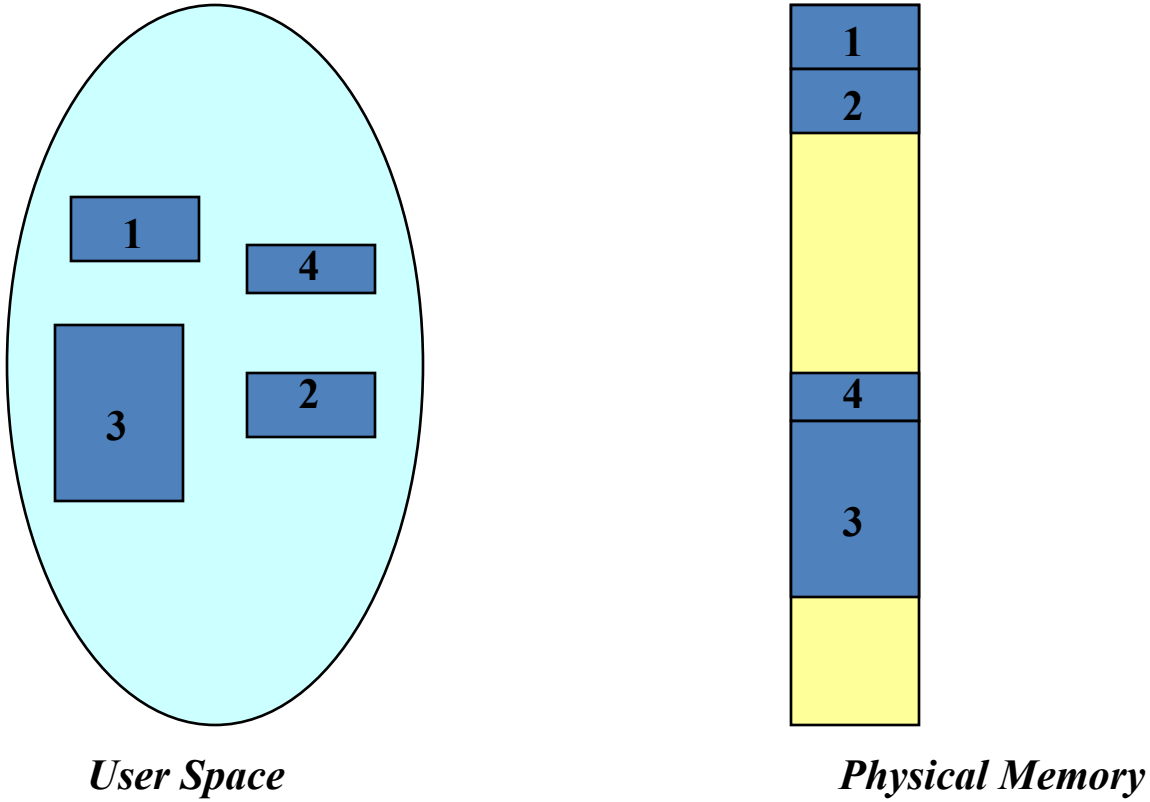
Exercises

- Why are page sizes always a power of 2?
- On a simple paging system with 2^24 bytes of physical memory, 256 pages of logical address space, and a page size of 2^10 bytes, how many bits are in a logical address?
- Describe, how TLB increase performance in paging.

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
  - local variables, global variables,common block
  - stack, symbol table, arrays
  - Protect each entity independently
  - Allow each segment to grow independently
  - Share each segment independently

Logical view of segmentation



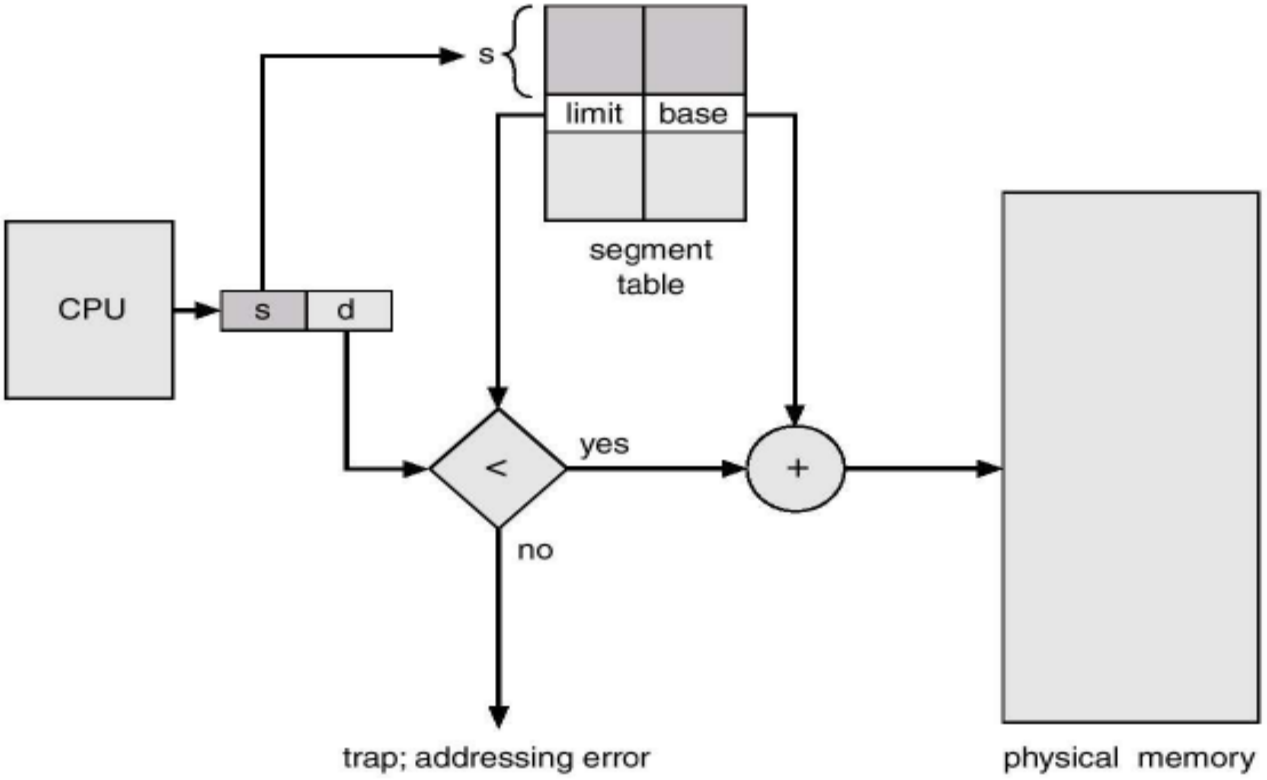
Segmentation Architecture

- Logical address consists of a two tuple

  <segment-number, offset>

- Segment Table
  - Maps two-dimensional user-defined addresses into one-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. We call it **Base register**
  - *Segment-table length register* (STLR) indicates the number of segments used by a program; segment number is legal if s < STLR.
  - **Limit register**

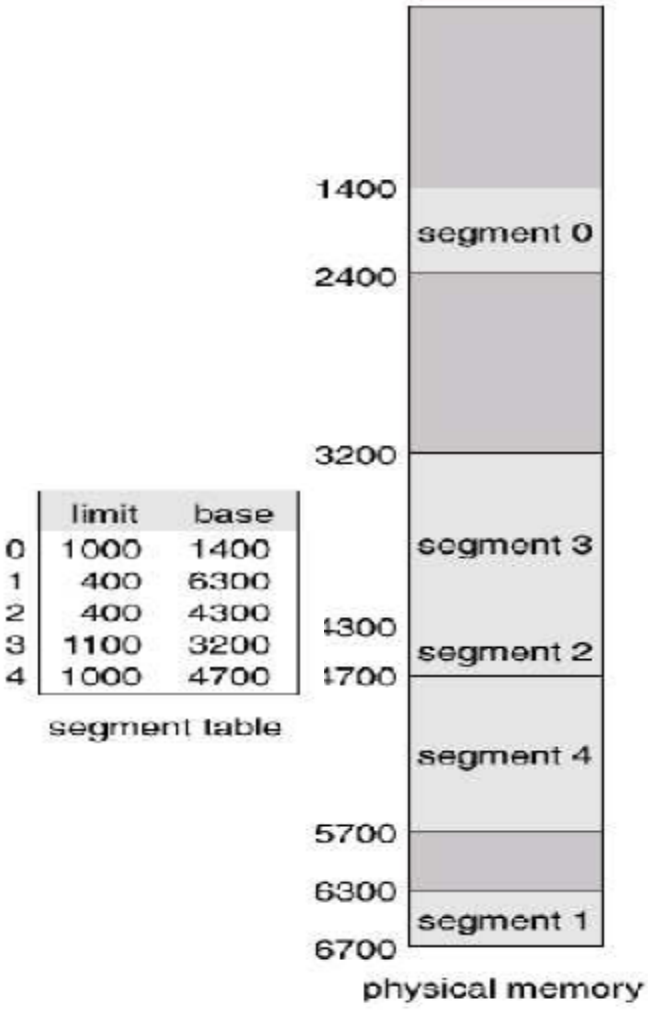
Address Translation in segmentation



Segmentation Architecture (cont.)

- Relocation is dynamic - by segment table
- Sharing
  - Code sharing occurs at the segment level.
  - Shared segments must have same segment number.
- Allocation - dynamic storage allocation problem
  - use best fit/first fit, may cause external fragmentation.
- Protection
  - protection bits associated with segments
    - read/write/execute privileges
    - array in a separate segment - hardware can check for illegal array indexes.

- The segment number used as index into the segment table. The offset d of the logical address must be between 0 and the segment limit. If not , trap occur, if it is legal it is added to the segment base to produce the address in the physical memory.
- The segmentation scheme causes External fragmentation, this can be handle by memory compaction technique.



Example
Address translation in segmentation

Compute the Physical address for

a) 099 b) 278 c)1265

Ans:

a)099, here segment no is 0 and offset is 99. since 99<124 and segment 0 begins at location 330, the physical address is 330+99=429

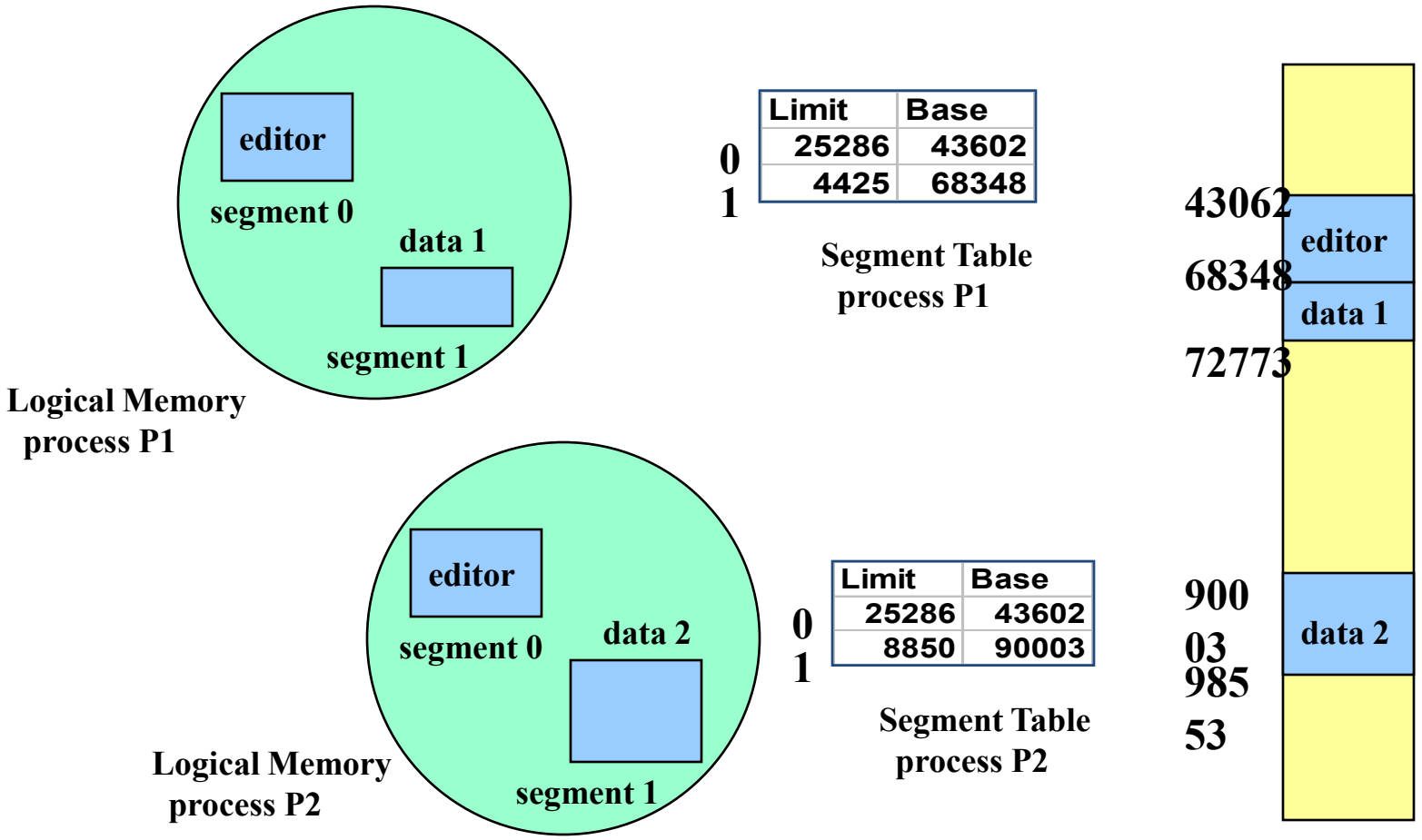
b)278, here segment no is 2 and offset is 78<99. Segment 2 begins at 111, the physical address is 111+78=189

c) 1265, here segment no is 1 and offset is 265>211. this address result in segment fault.

Consider the segment table:

Segment	Base	Size
0	330	124
1	876	211
2	111	99
3	498	302

Shared segments



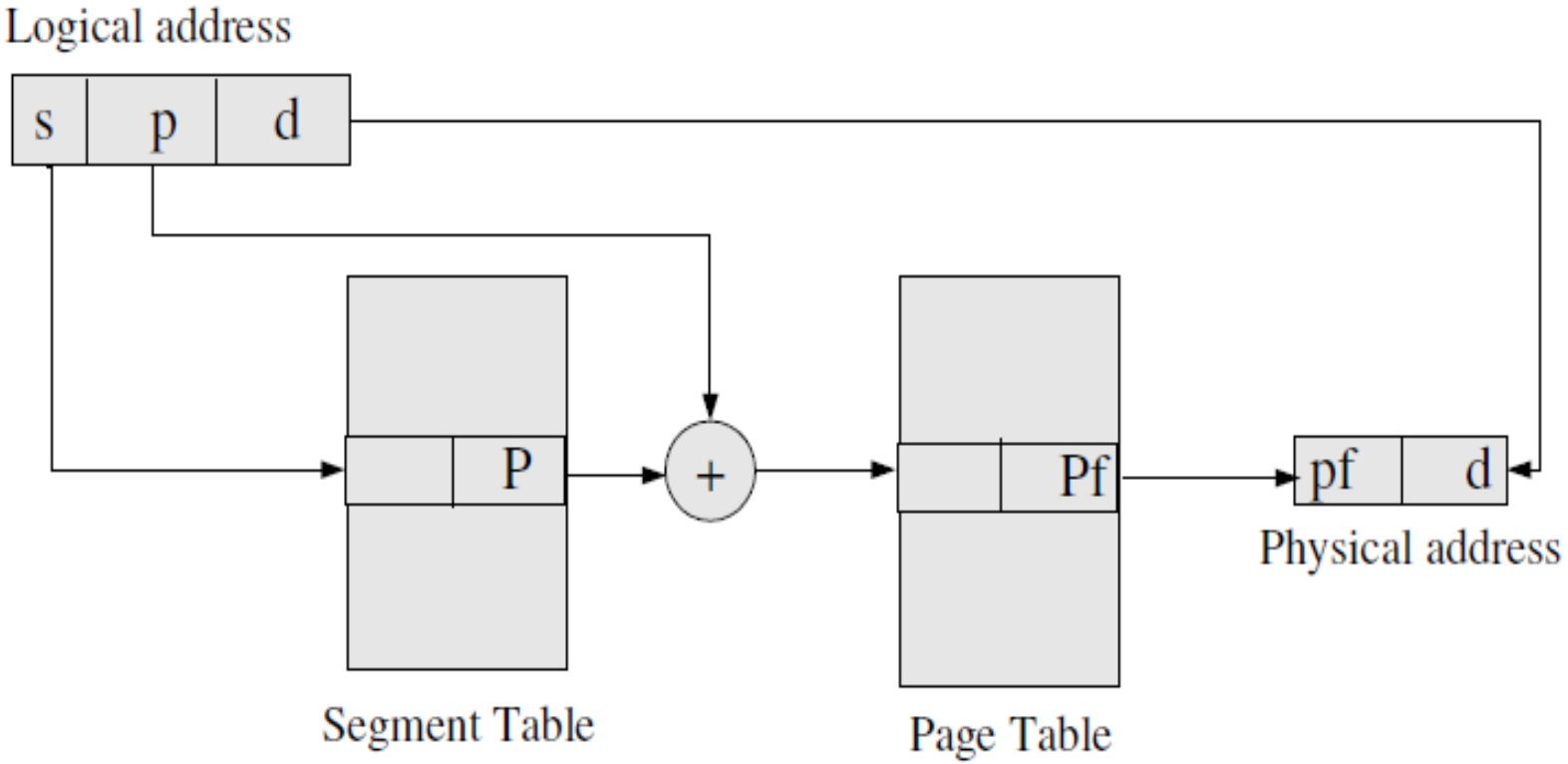
Paging Vs Segmentation

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Segmentation with Paging

- What happen when segment are larger than main memory?
- Segmentation can be combined with paging to provide the efficiency of paging with the protection and sharing capabilities of segmentation.
- As with simple segmentation, the logical address specifies the segment number and the offset within the segment.
- When paging is added, the segment offset is further divided into a page number and page offset.
- The segment table entry contains the address of the segment's page table.

Segmentation with Paging



Questions?

- A smaller page size leads to smaller page tables –
  - » False -> need more entries because we have more pages
- A smaller page size leads to more TLB misses
  - True –> less likely that page will encompass address we are after.
- A smaller page size leads to fewer page faults
  - True

Continue?

- **What is swapping? Can swapping permit an application requiring 16M memory to run on a machine with 8M of RAM?**
- Swapping is the act of running each whole process in main memory for a time then placing back onto disk and vice versa. It is used when the system does not have enough main memory to hold all the currently active processes.

- Assuming that an application is one program and hence a single process, swapping will not permit an application(process) requiring 16M of memory to run on a machine with 8M RAM (main memory) as the whole process is too large to fit into main memory.

Continue?

- **Enumerate some pros and cons for increasing the page size.**
- pros:
  - reduces page table size
  - increases TLB coverage
  - increases swapping I/O throughput, as small disk

- cons:
  - increases page fault latency, as swap response time is slower due to more data.
  - increases internal fragmentation of pages, as there is more 'wasted page' in the working set

Exercise

- Why are segmentation and paging sometimes combined into one scheme?
- Consider the following segment table:

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

- What are the physical address for the following logical address?
  - a) 0430 b)110 c)2500 d) 3400 e) 4112

- Distinguish the paging and segmentation.

Paging/Segmentation Policies

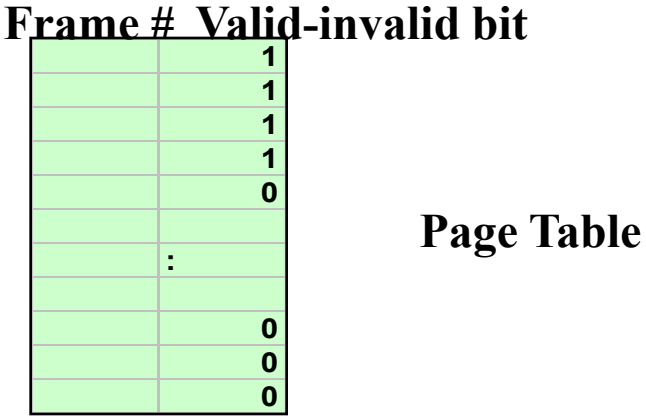
- Fetch Strategies
  - When should a page or segment be brought into primary memory from secondary (disk) storage?
    - Demand Fetch
    - Anticipatory Fetch
- Placement Strategies
  - When a page or segment is brought into memory, where is it to be put?
    - Paging - trivial
    - Segmentation - significant problem
- Replacement Strategies
  - Which page/segment should be replaced if there is not enough room for a required page/segment?

Demand Paging

- Bring a page into memory only when it is needed.
  - Less I/O needed
  - Less Memory needed
  - Faster response
  - More users
- The first reference to a page will trap to OS with a page fault.
- OS looks at another table to decide
  - Invalid reference - abort
  - Just not in memory.

Valid-Invalid Bit

- With each page table entry a valid-invalid bit is associated (1 ⇒ in-memory, 0 ⇒ not in memory).
- Initially, valid-invalid bit is set to 0 on all entries.
  - During address translation, if valid-invalid bit in page table entry is 0 --- ***page fault*** occurs.
  - Example of a page-table snapshot



Handling a Page Fault: Process

- Page is needed - reference to page
  - Step 1: Page fault occurs - trap to OS (process suspends).
  - Step 2: Check if the virtual memory address is valid. Kill job if invalid reference. If valid reference, and page not in memory, continue.
  - Step 3: Bring into memory - Find a free page frame, map address to disk block and fetch disk block into page frame. When disk read has completed, add virtual memory mapping to indicate that page is in memory.
  - Step 4: Restart instruction interrupted by illegal address trap. The process will continue as if page had always been in memory.

What happens if there is no free frame?

- Page replacement - find some page in memory that is not really in use and swap it.
  - Need page replacement algorithm
  - Performance Issue - need an algorithm which will result in minimum number of page faults.
  - Same page may be brought into memory many 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
    - large virtual memory can be provided on a smaller physical memory.

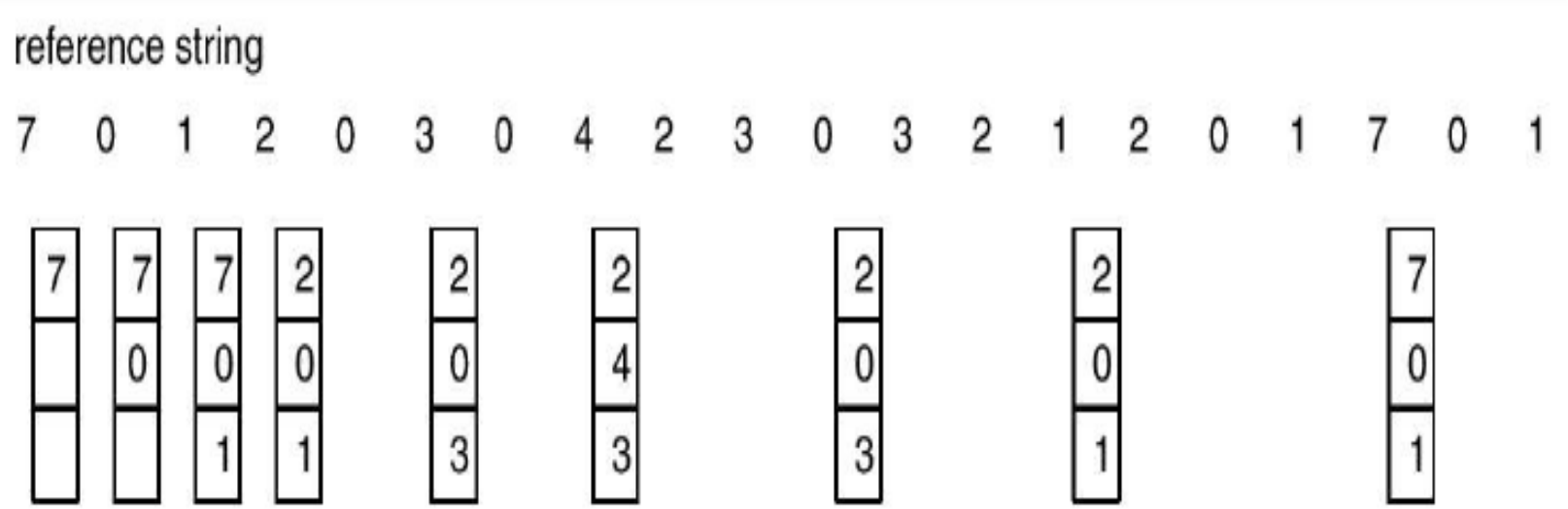
Page Replacement Strategies (algorithms)

- The Principle of Optimality
  - Replace the page that will not be used again the farthest time into the future.
- Random Page Replacement
  - Choose a page randomly
- FIFO - First in First Out
  - Replace the page that has been in memory the longest.
- LRU - Least Recently Used
  - Replace the page that has not been used for the longest time.
- LFU - Least Frequently Used
  - Replace the page that is used least often.
- NUR - Not Used Recently
  - An approximation to LRU
- Working Set
  - Keep in memory those pages that the process is actively using

Optimal Page Replacement (OPR)

Replace the page that will not be used for the longest period of time.

- Each page can be labeled with number of instructions that will be executed before that page is first referenced.
- Ex: For three page frames and 8 pages system the optimal page replacement is as:



No of page fault?

For three page frames and page references in the order

- 0 1 2 3 2 1 0 3 2 3

- Note: Not Implementable “Future reference can not be determined exactly”. It depend upon the process behavior.
- Used to compare the efficiency of other page replacement algorithms

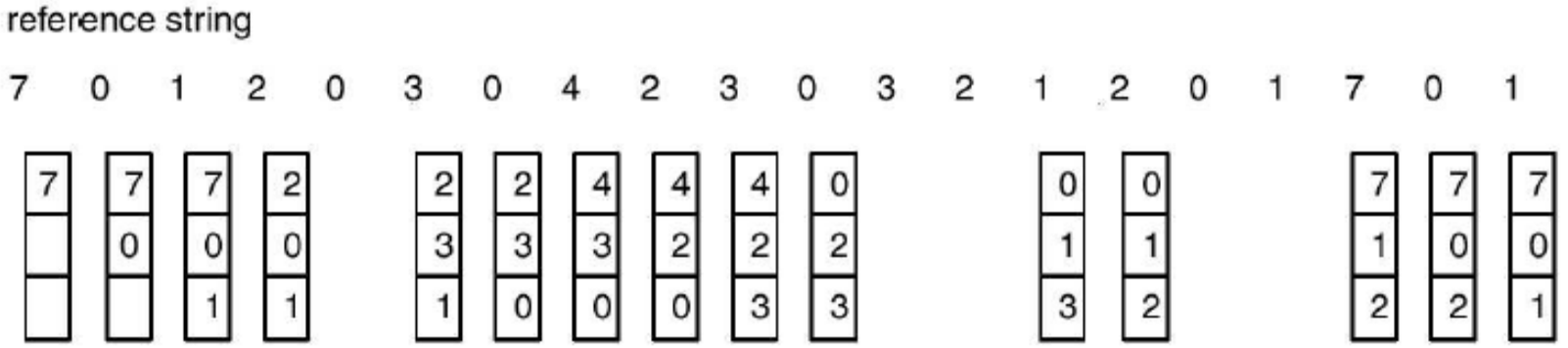
OPR: Discussion

- Advantages:
  - An optimal page-replacement algorithm; it guarantees the lowest possible page fault rate.
- Problems:
  - Unrealizable, at the time of the page fault, the OS has no way of knowing when each of the pages will be referenced next.
- This is not used in practical system.

First-In-First-Out (FIFO)

- This associates with each page the time when that page was brought into the memory. The page with highest time is chosen to replace.

- This can also be implemented by using queue of all pages in memory.



How many Page Fault?

for four page frames and page references in the order

- 0 1 2 3 2 1 0 3 2 3

FIFO

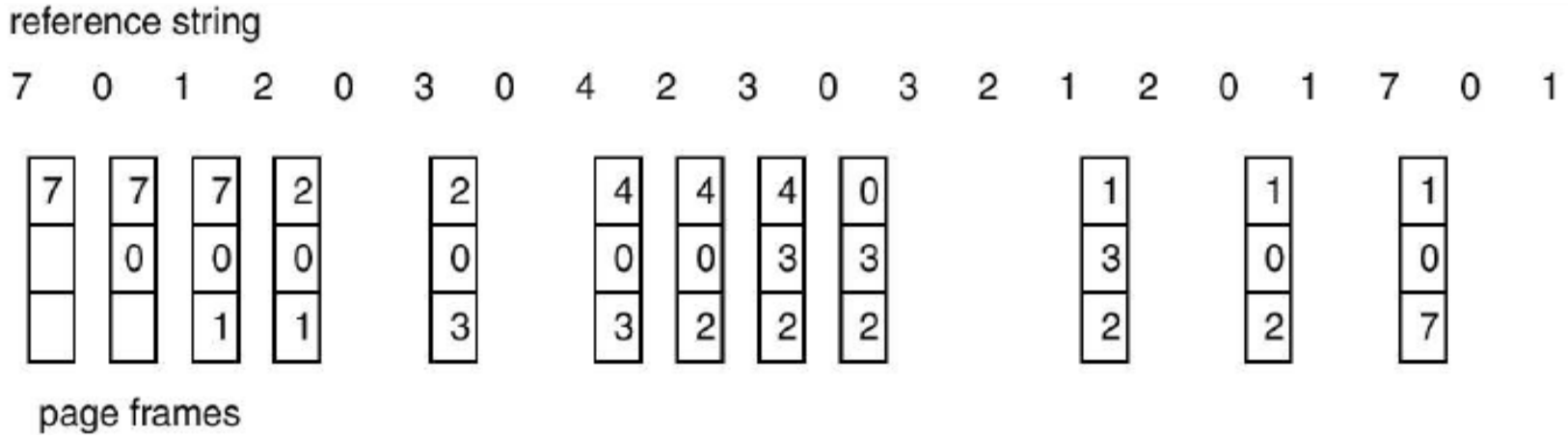
- Advantages:
  - Easy to understand and program.
  - Distributes fair chance to all.
- Problems:
  - FIFO is likely to replace heavily (or constantly) used
  - pages and they are still needed for further processing.

Least Recently Used (LRU)

Recent past is a good indicator of the near future.

- When a page fault occurs, throw out the page that has been unused for longest time.

- It maintain a linked list of all pages in memory with the most recently used page at the front and least recently used page at the rare. The list must be updated on every memory reference.



Calculate no of Page fault

- For 3 frame and the page reference in order of
  - 0 1 2 3 2 1 0 3 2 3

Implementation of LRU algorithm

- 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 changes, look at the counters to determine which page to change (page with smallest time value).
- Stack Implementation
  - Keeps a stack of page numbers in a doubly linked form
  - Page referenced
    - move it to the top
    - requires 6 pointers to be changed
  - No search required for replacement

Least Frequently Used (LFU)

One approximation to LRU, software implementation.

- LFU requires that the page with the smallest count be replaced.
- The reason for this selection is that an actively used page should have a large reference count.
- It requires a software counter associated with each page. When page fault occur the page with lowest counter is chosen for replacement.

- **Problem**: likely to replace highly active pages. This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.

Not Recently Used (NRU)

- Pages not recently used are not likely to be used in near future and they must be replaced with incoming pages.

- To keep useful statistics about which pages are being used and which pages are not, most computers have two status bits associated with each page . referenced and modified.

- These bits must be updated on every memory reference.
- When a page fault occurs, the OS inspects all the pages and divides them into four categories based on the current value of their referenced and modified bits.

NRU(Cont)

Class 0: not referenced, not modified(00).
Class 1: not referenced, modified(01).
Class 2: referenced, not modified(10).
Class 3: referenced, modified(11).

- Pages in the lowest numbered class should be replaced first, and those in the highest numbered groups should be replaced last.
- Pages within the same class are randomly selected.

Working Set (WS) Page Replacement

What to do when a process just swapped out and another process has to load?

The set of pages that a process is currently using is called its working set.

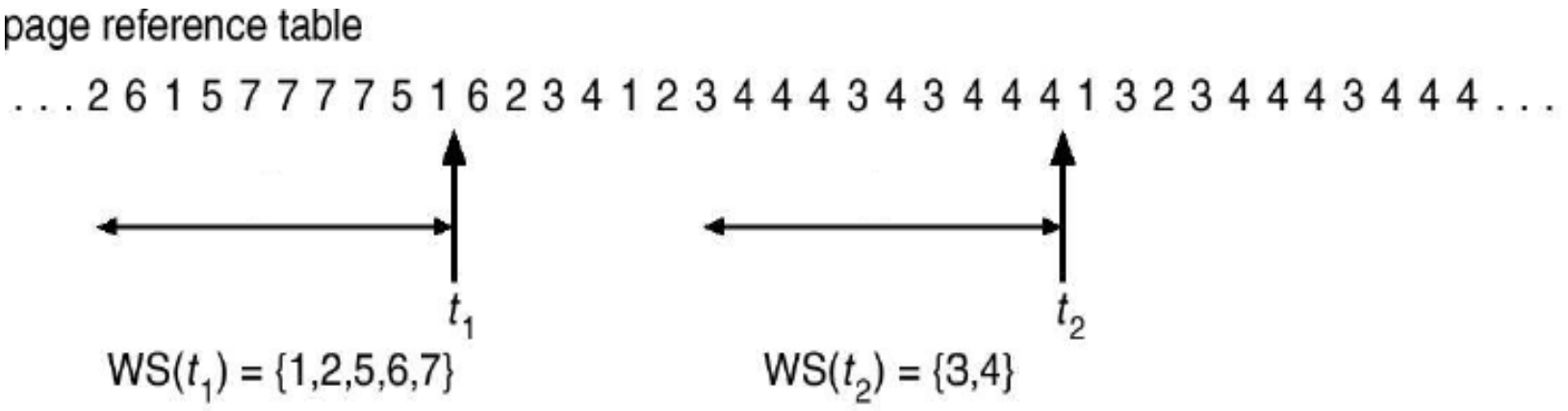
If the entire working set is in memory, the process will run without causing Many faults until it moves into another execution.

Otherwise, excessive page fault might occur called thrashing.

Many paging systems try to keep track of each process' working set and make sure that it in memory before the process Run--working set model Or pre-paging.

Working Set (WS) Page Replacement

- The working set of pages of process, ws(t, k) at time t, is the set of pages referenced by the processes in time interval t-k to t.
- The variable k is called working-set-window, the size of k is central issue in this model.
- For example: working set with k=10

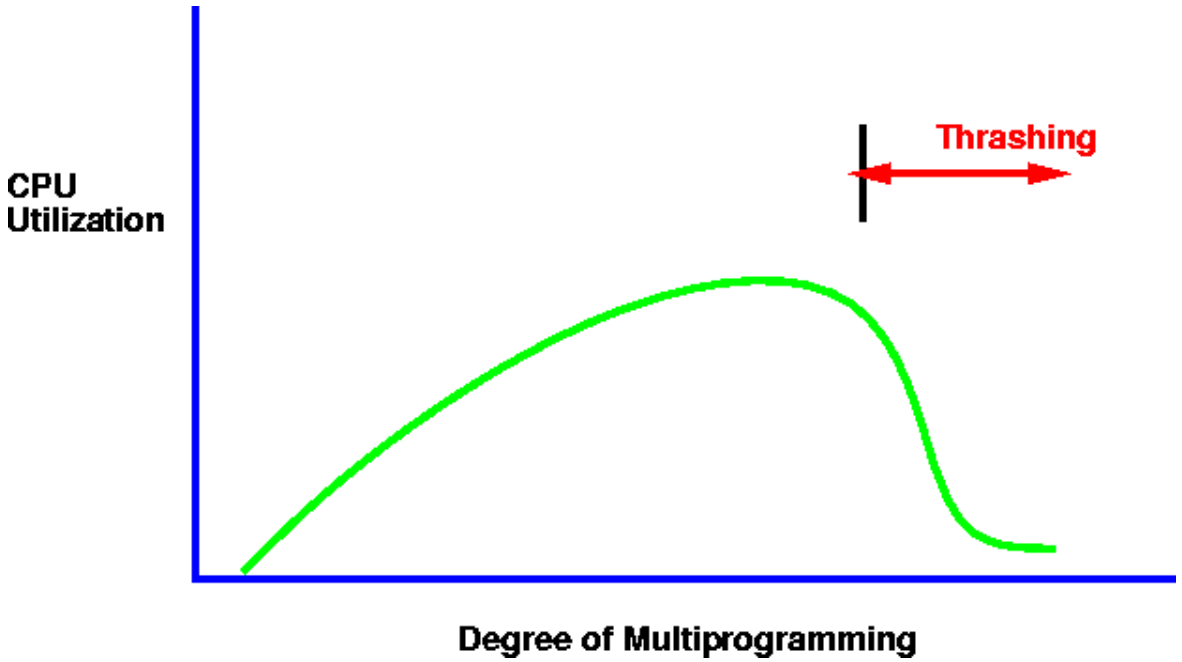


Thrashing

- If a process does not have enough pages, the page-fault rate is very high. This leads to:
  - low CPU utilization.
  - OS thinks that it needs to increase the degree of multiprogramming
  - Another process is added to the system.
  - System throughput plunges or falls down...
- *Thrashing*
  - A process is busy swapping pages in and out.
  - In other words, a process is spending more time in paging than in executing.

Thrashing

– Why does thrashing occur?



Belady's Anomaly

- Intuitively, it might seem that the more page frames the memory has, the fewer page faults a program will get.
- Surprisingly enough, this is not always the case. Belady et al. (1969) discovered a counterexample, in which FIFO caused more page faults with four page frames than with three.
- This strange situation has become known as **Belady's anomaly**.

Example from Belady's paper

- The pages are referenced in the order
- 0 1 2 3 0 1 4 0 1 2 3 4
- Now Calculate the Page faults
  - Case with 3 frame
  - Case with 4 frame
- Now you will find the more page fault in case of 4 frame: Belady's Anmoly

Thank You