USAL
جامعة العلوم والاداب اللبنانية
UNIVERSITY OF SCIENCES & ARTS IN LEBANON

CSCI320 – Operating Systems

Ahmad Fadlallah

References

- These slides are based on the official slides of the following textbooks
  - Operating Systems: Internals and Design Principles (9th Edition), William Stallings, Pearson
  - Operating System Concepts with Java (10th Edition), Abraham Silberschatz, Peter B. Galvin and Greg Gagne, Wiley

Memory Management

Main Memory

Outline

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

Objectives

- To provide a detailed description of various ways of **organizing memory hardware**.
- To explore various techniques of **allocating memory to processes**.
- To discuss in detail **how paging works** in contemporary computer systems.

Background

Swapping

Contiguous Memory Allocation

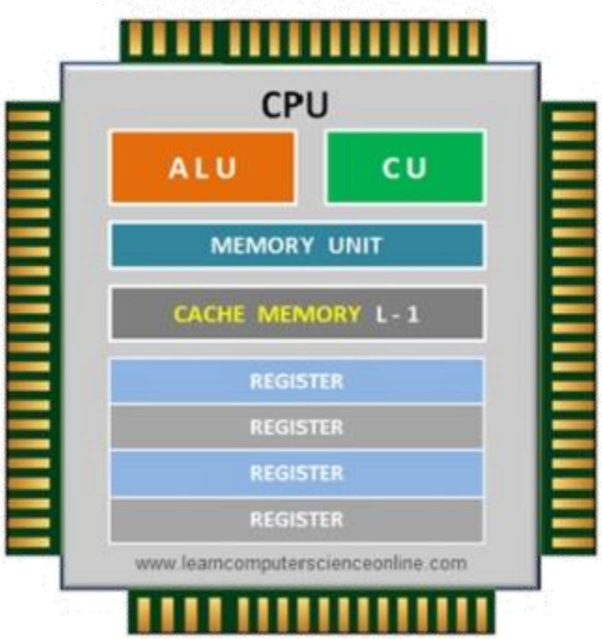
Segmentation

Paging

Structure of the Page Table

Introduction

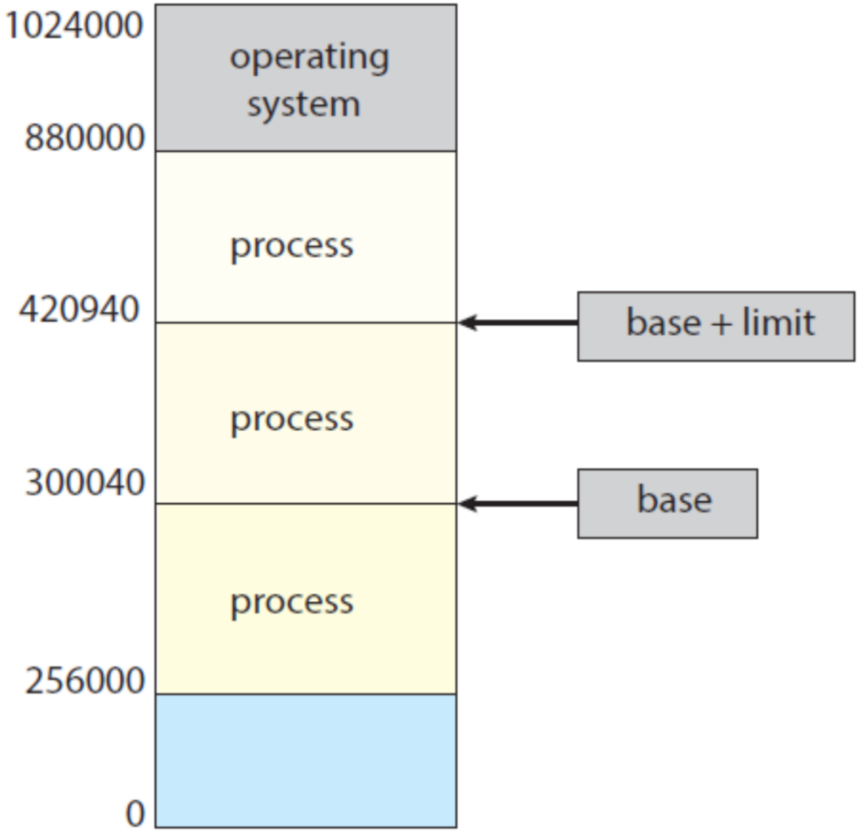
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory (including Cache) and registers are the only storage directly accessible by CPU
- Memory unit only sees a stream of addresses + read requests, or address + data + write requests
- Register access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



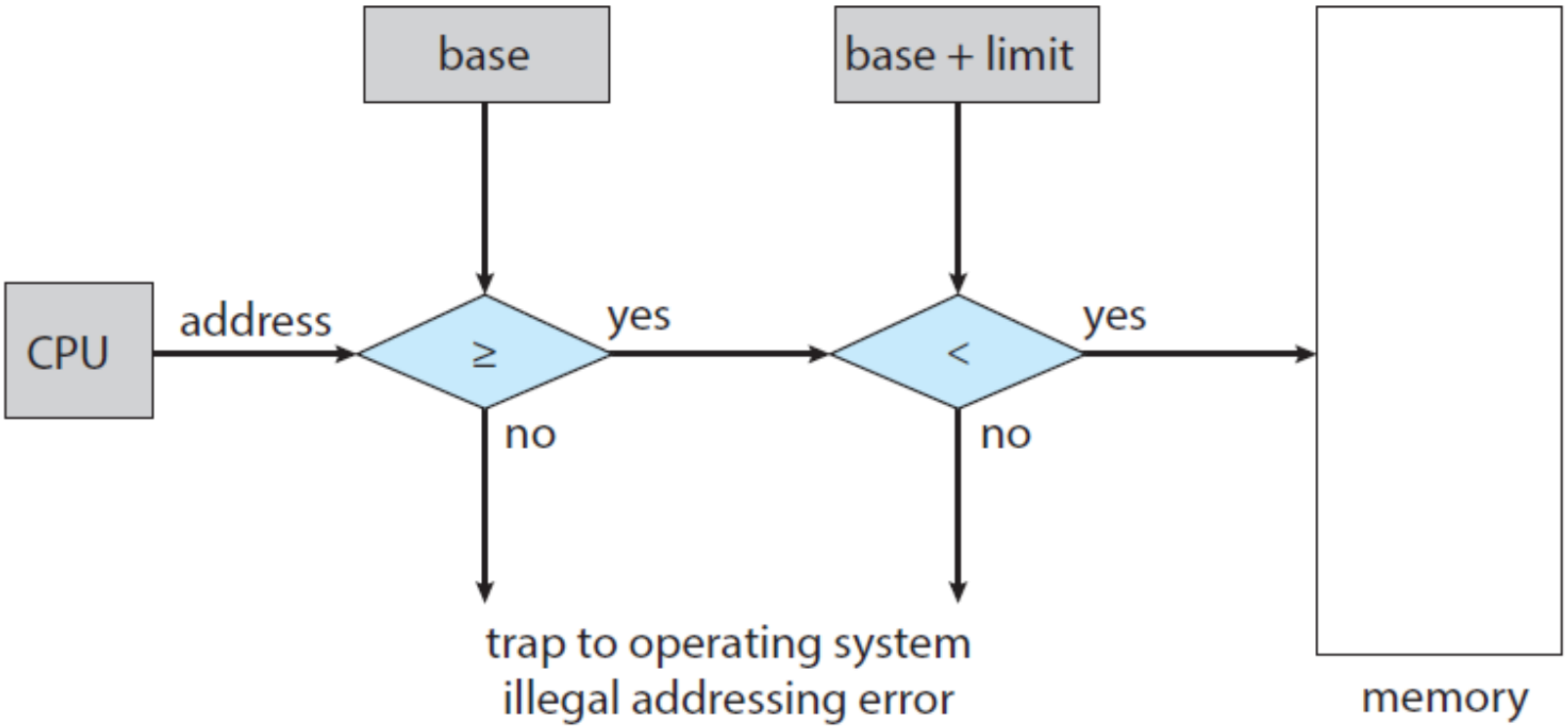
Hardware Address Protection

- A pair of **base** and **limit registers** define the **logical address space**

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

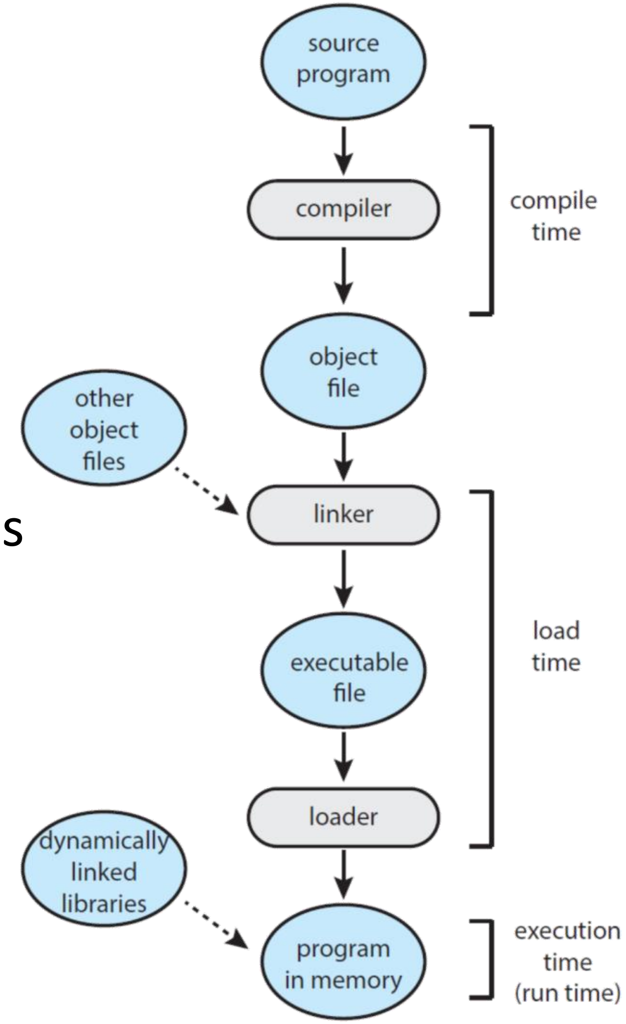


Hardware Address Protection (cont’d)



Address Binding

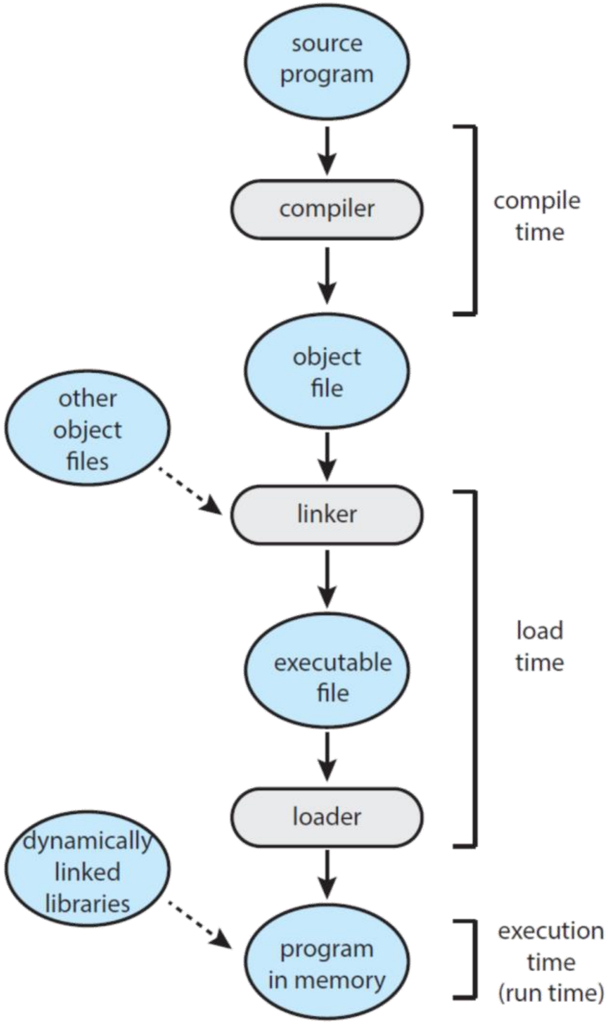
- *Programs on disk*, ready to be brought into memory to execute form an **input queue**
- **Addresses** represented *in different ways* at different stages of a program's life
  - ○ *Source code addresses* usually **symbolic** (e.g., variable names)
  - ○ *Compiled code addresses* bind to **relocatable/relative addresses** (e.g., "14 bytes from beginning of this module")
  - ○ Linker or loader will bind **relocatable/relative** addresses to **absolute addresses**
    - i.e. 0x74014
  - ○ Each binding maps **one address space to another**



Binding of Instructions and Data to Memory

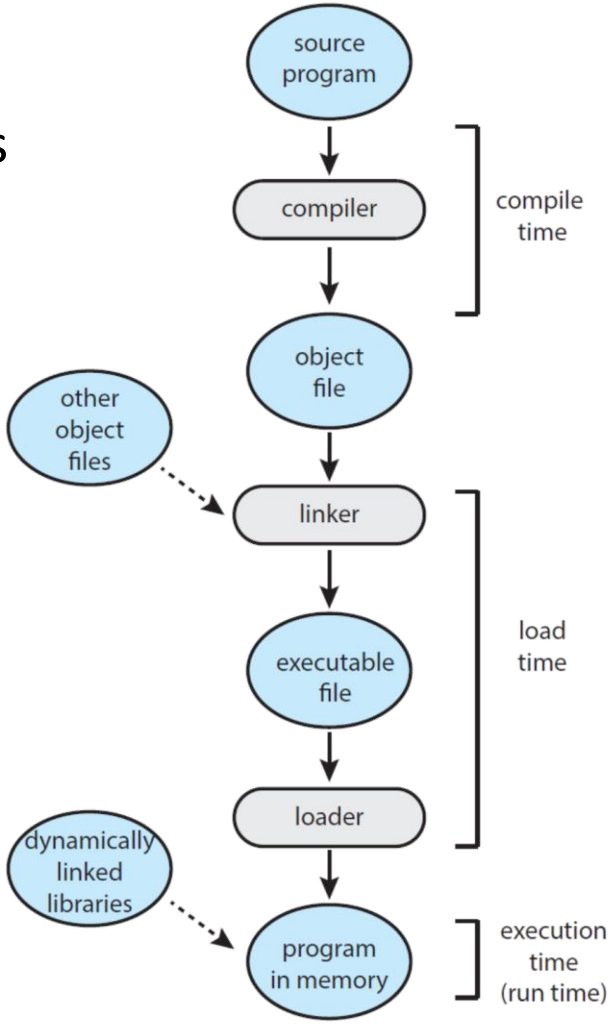
- **Address binding** of instructions and data to memory addresses **can happen** at three different stages

1. **Compile time**: If memory location known a priori, **absolute code** can be generated
   - ✓ Must recompile code if starting location changes
2. **Load time**: Must generate **relocatable code** if memory location is not known at compile time
3. **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another

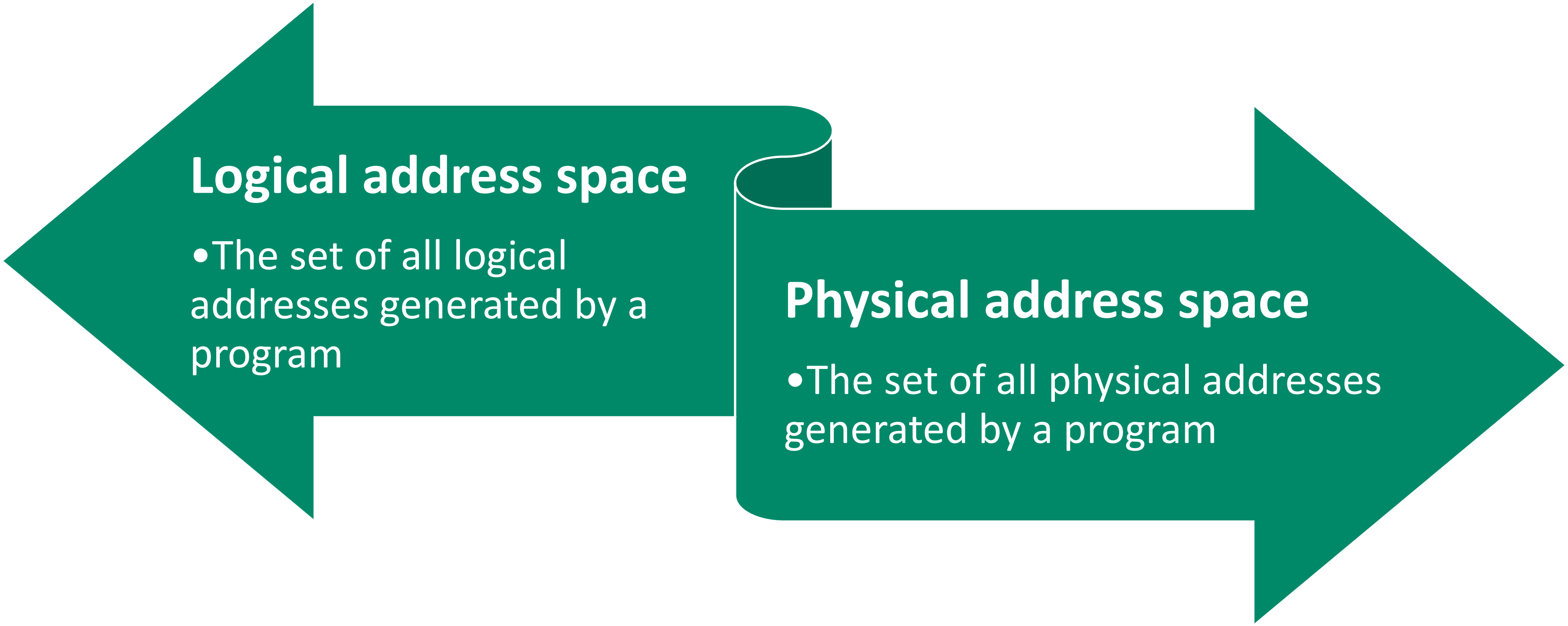


Logical vs. Physical Address Space

- The concept of a **logical address space** that is **bound** to a separate **physical address space** is central to proper memory management
- **Logical/Virtual Address:** Generated by CPU
- **Physical address:** Address seen by the memory unit
- Logical and physical addresses **are the same in compile-time and load-time address-binding schemes**
- Logical and physical addresses **differ in execution-time address-binding** scheme



Logical vs. Physical Address Space



Logical address space

- The set of all logical addresses generated by a program

Physical address space

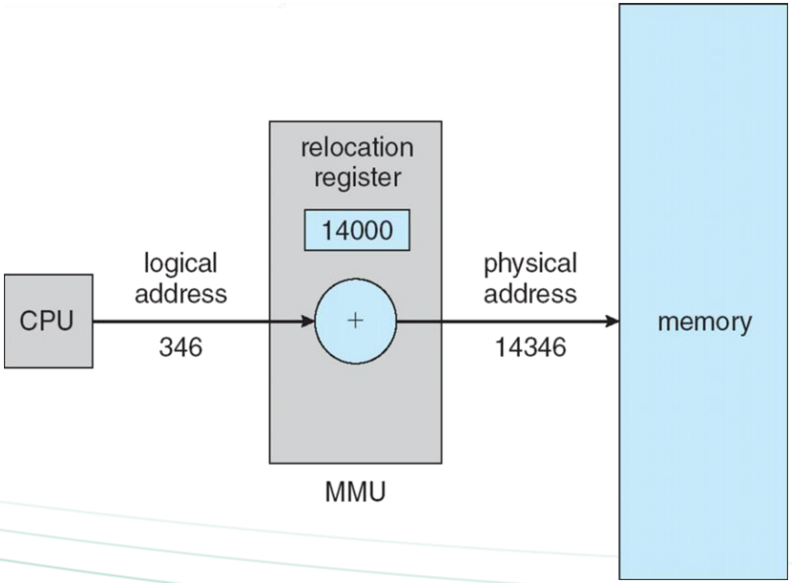
- The set of all physical addresses generated by a program

Memory-Management Unit (MMU)

- **Hardware device** that at run time maps virtual to physical address
- **Different mapping methods**
- Basic scheme:

  physical address = virtual address + base register

- **Base register** also called **Relocation register**

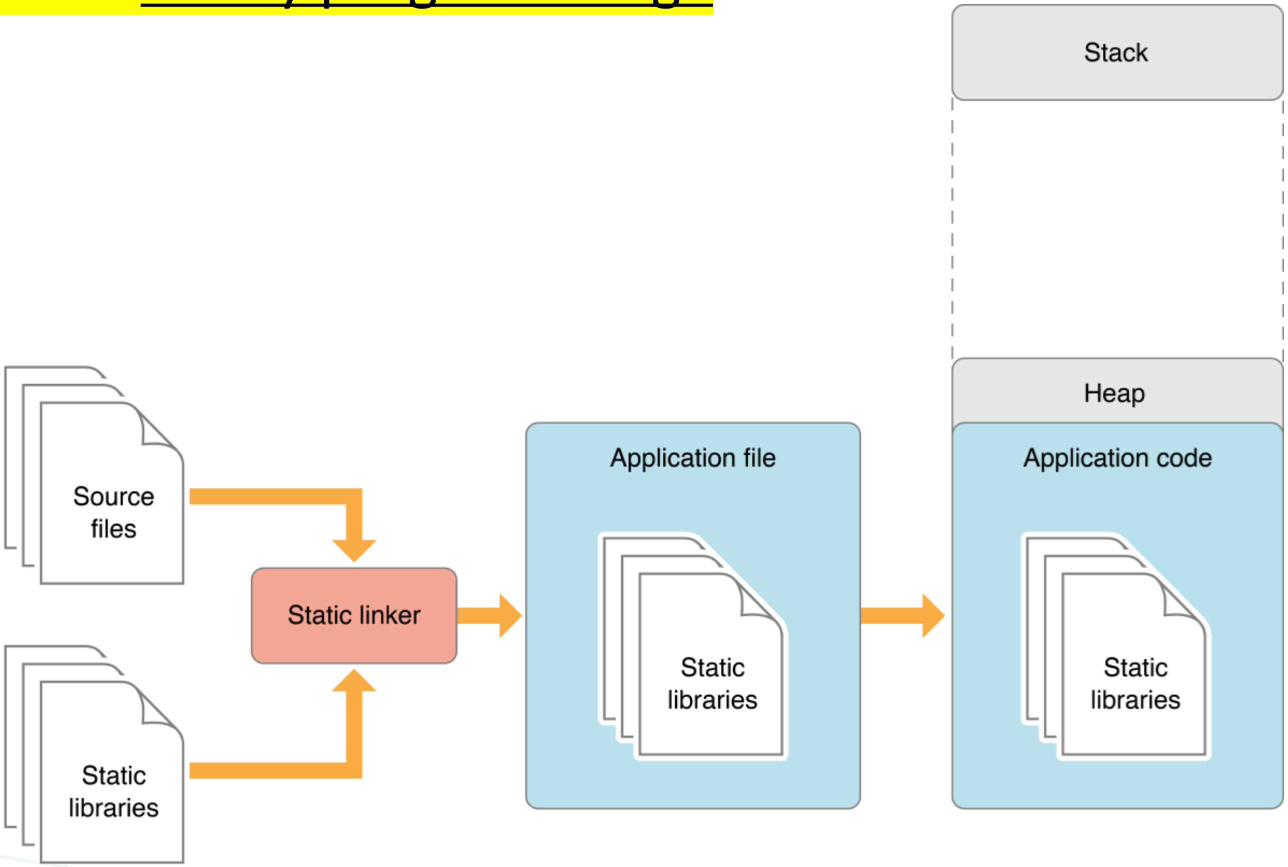


Dynamic relocation using a relocation register

- The **user program deals with logical addresses**
  - It never sees the **real physical addresses**
  - **Execution-time binding** occurs when **reference is made to location in memory**
- 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

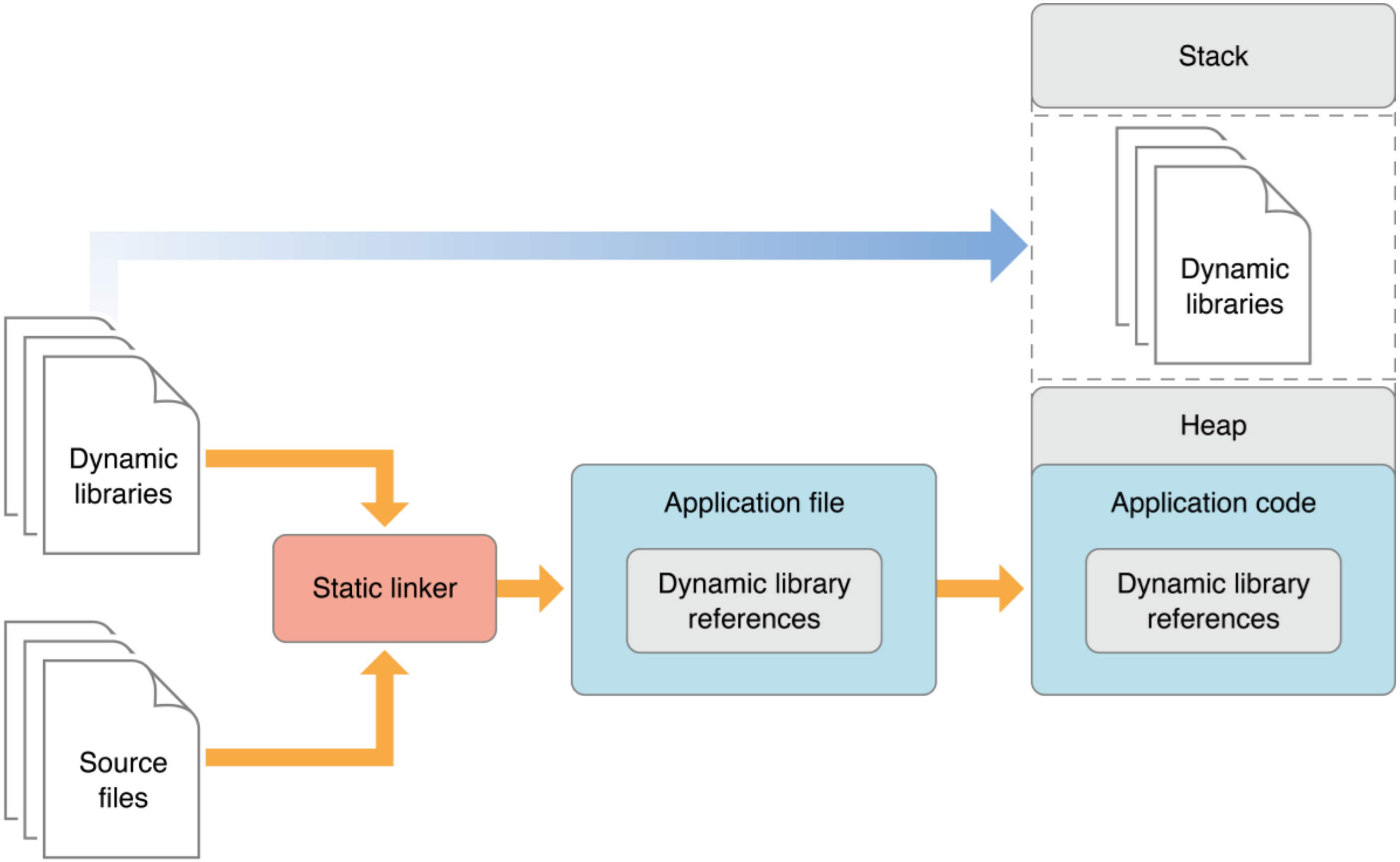
Static Linking

- System libraries and program code combined **by the loader** into the binary program image



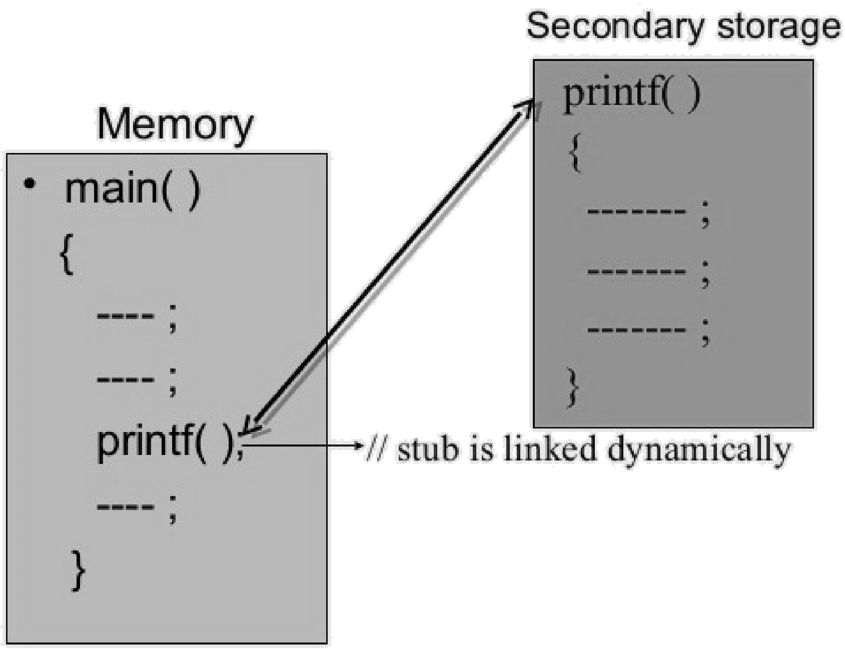
Dynamic Linking

- Linking postponed until **execution time**



Dynamic Linking (cont'd)

- **Stub**: Small piece of code 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 (cont’d)

- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed

Background

Contiguous Memory Allocation

Segmentation

Paging

Structure of the Page Table

Swapping

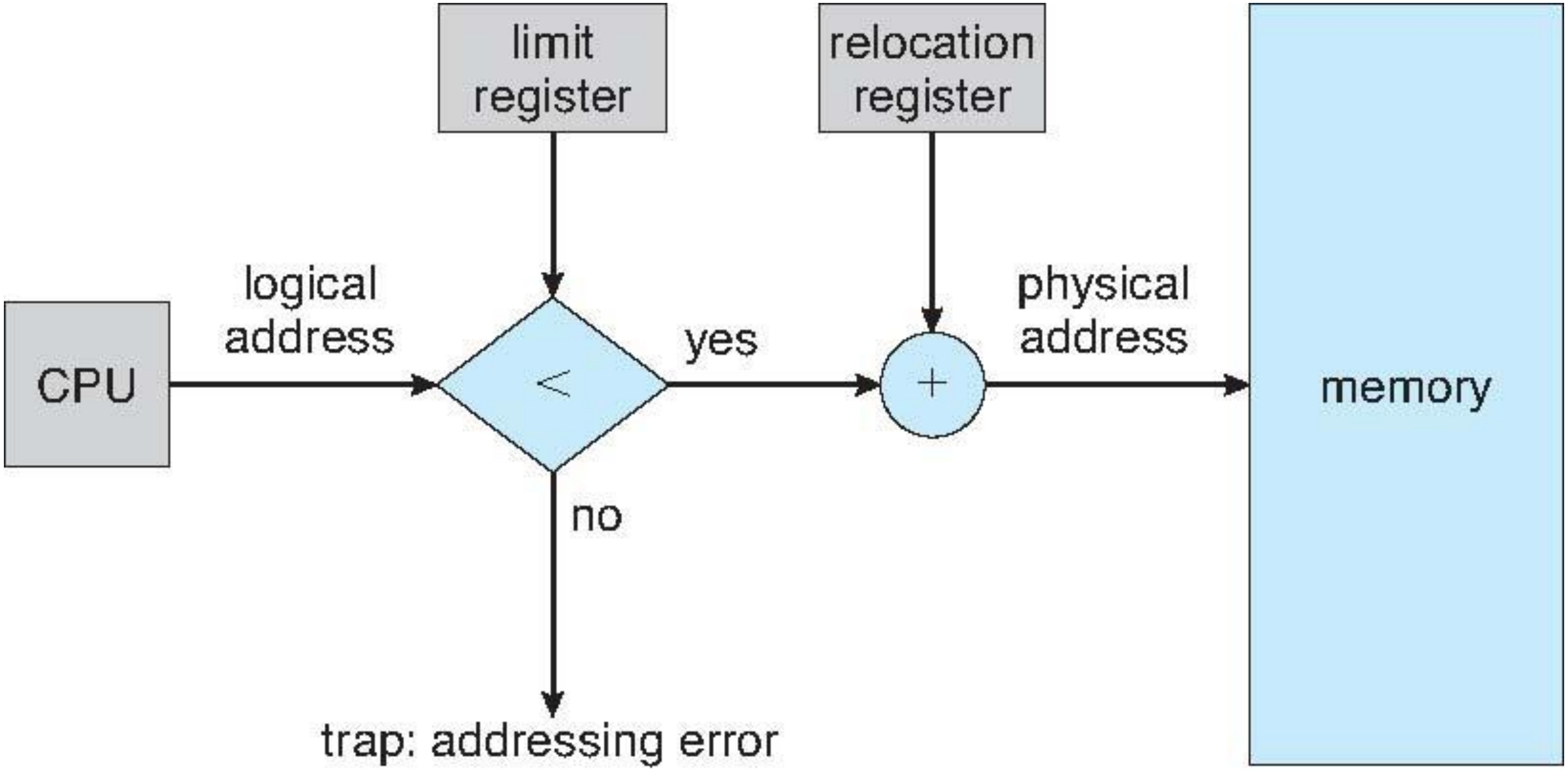
Contiguous Allocation

- Main memory must support both **OS and user processes**
- **Limited resource** => must be allocated efficiently
- **Contiguous allocation** is one early method
- Main memory usually divided into two **partitions**:
  - Resident operating system
    - Held in **in either low memory addresses or high memory addresses** (along with interrupt vector)
  - User processes held accordingly in **high memory addresses** or **low memory addresses**
  - Each process contained in single contiguous section of memory

Contiguous Allocation - Memory Protection

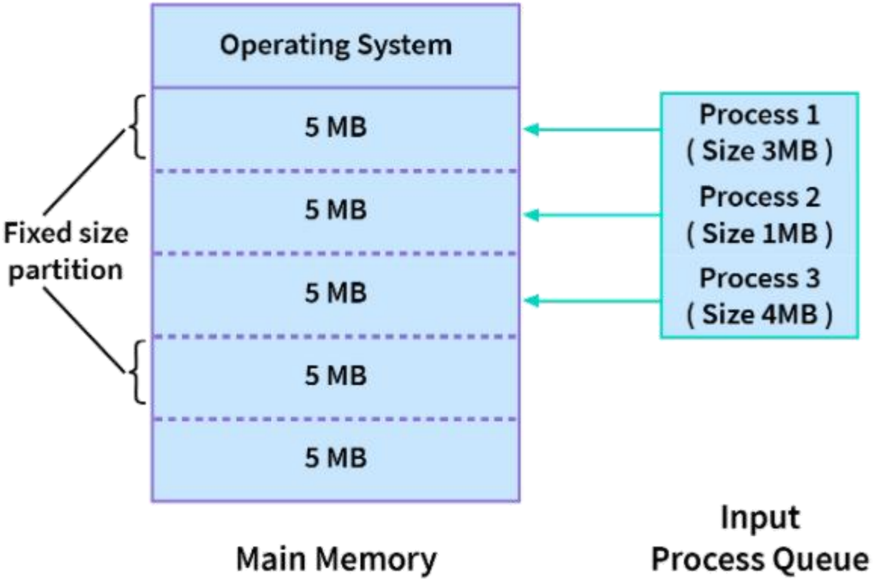
- **Relocation registers** used to protect user processes
  - From each other
  - From changing operating-system code and data
- **Base register** contains value of smallest physical address
- **Limit register** contains range of logical addresses
  - Each logical address must be ***less than*** the limit register
- **MMU maps logical address *dynamically***

Contiguous Allocation - Memory Protection



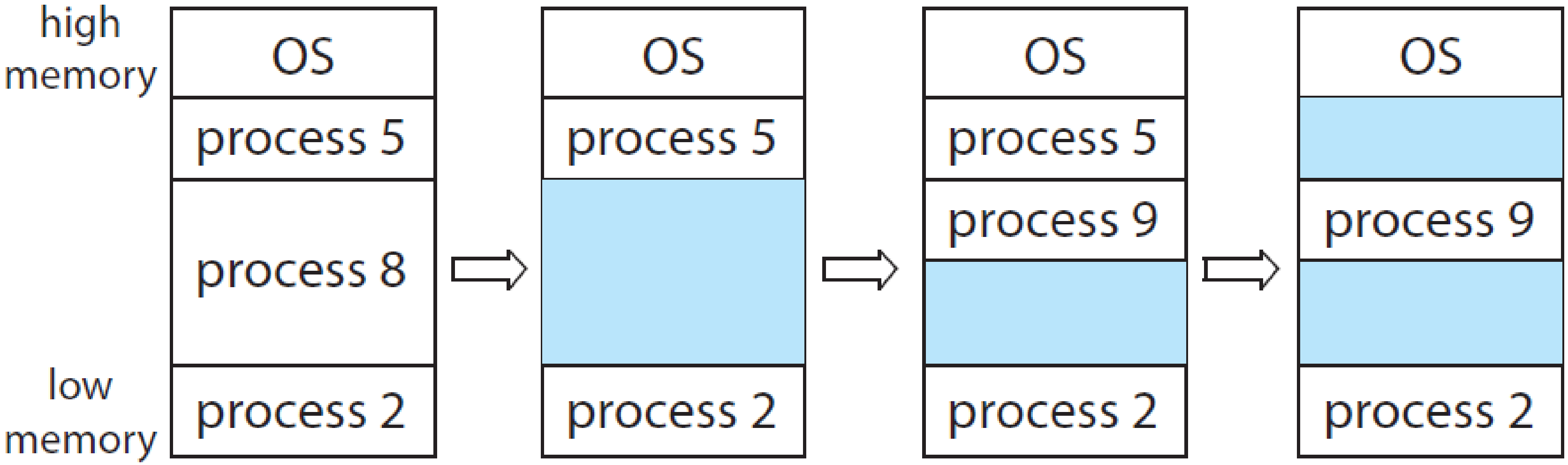
Memory Allocation

- **Fixed partition sizes**
  - + Simple method
  - ○ Divide Memory into several fixed-sized **partitions**
  - - Degree of multiprogramming limited by number of partitions
  - - Cannot allocate space for processes with greater size of the partition
  - - Waste of space in case of many “small” processes



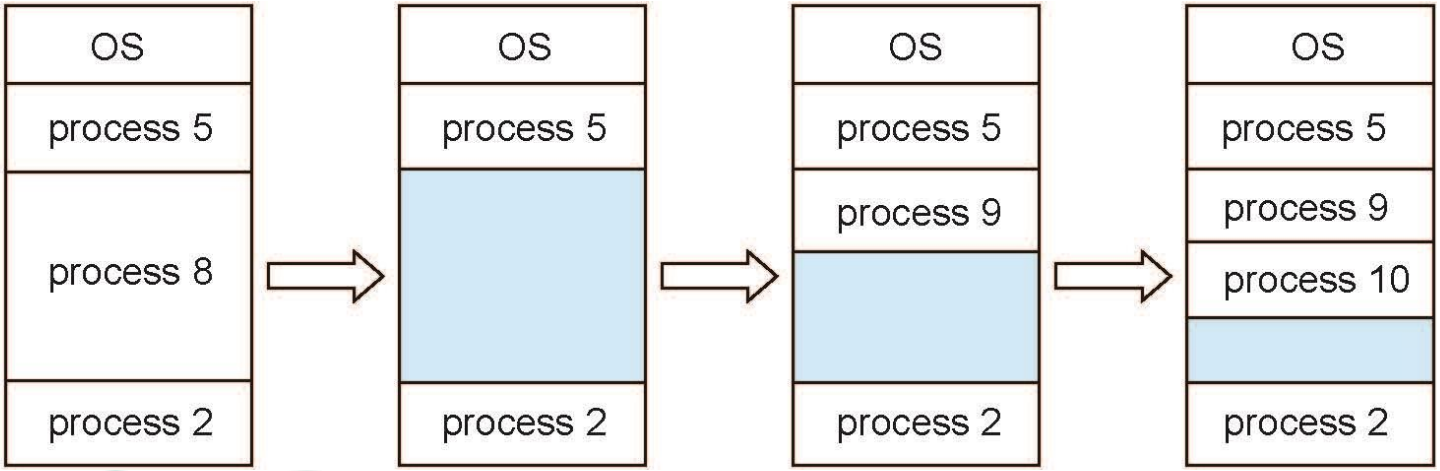
Memory Allocation (cont’d)

- **Variable-partition** sizes for efficiency (sized to a given process’ needs)
- **Hole:** block of available memory
- Holes of various size are scattered throughout memory



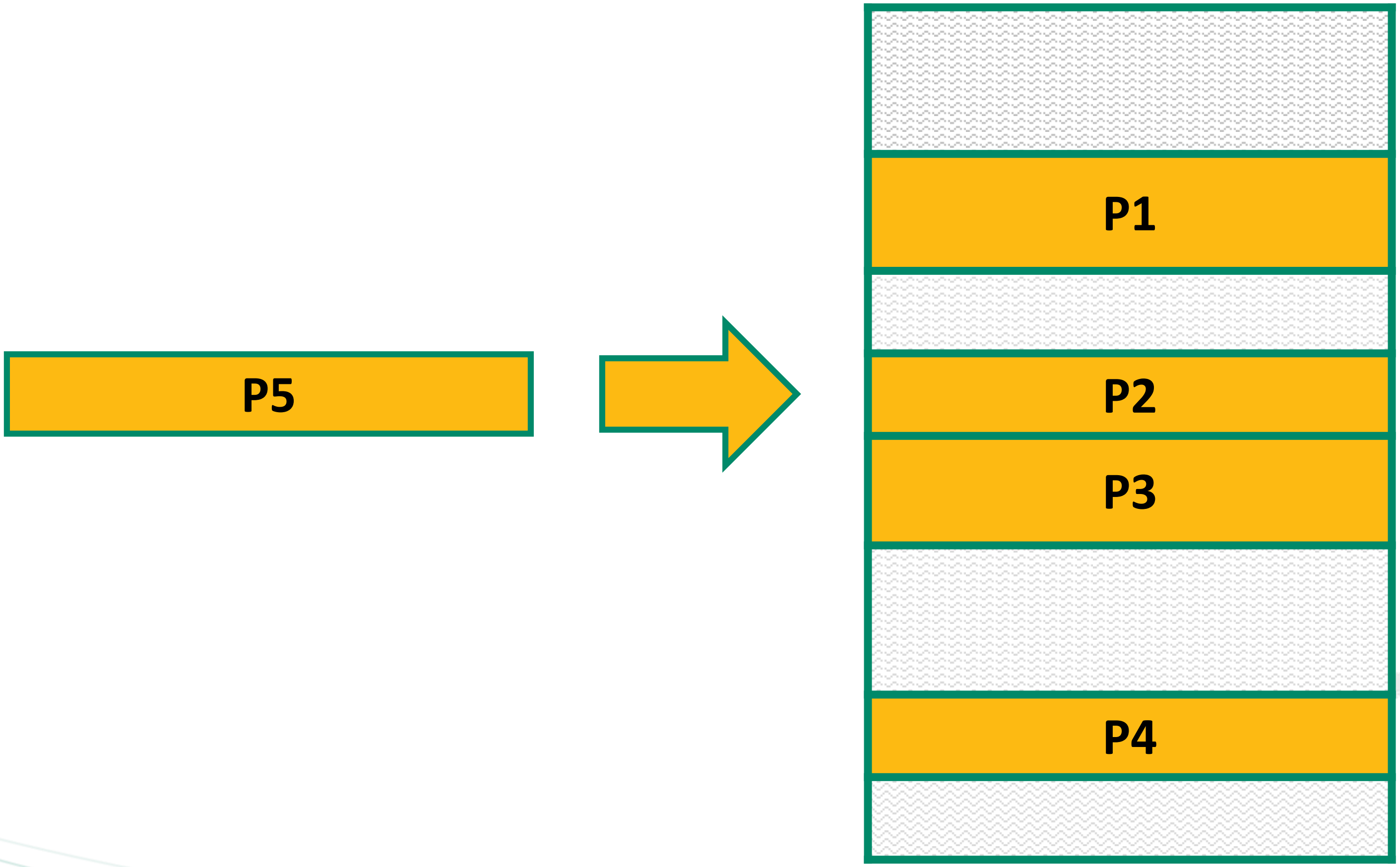
Memory Allocation (cont'd)

- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting **frees** its partition
- Adjacent free partitions **combined**
- Operating system maintains information about:
  a) **allocated partitions** b) **free partitions (hole)**

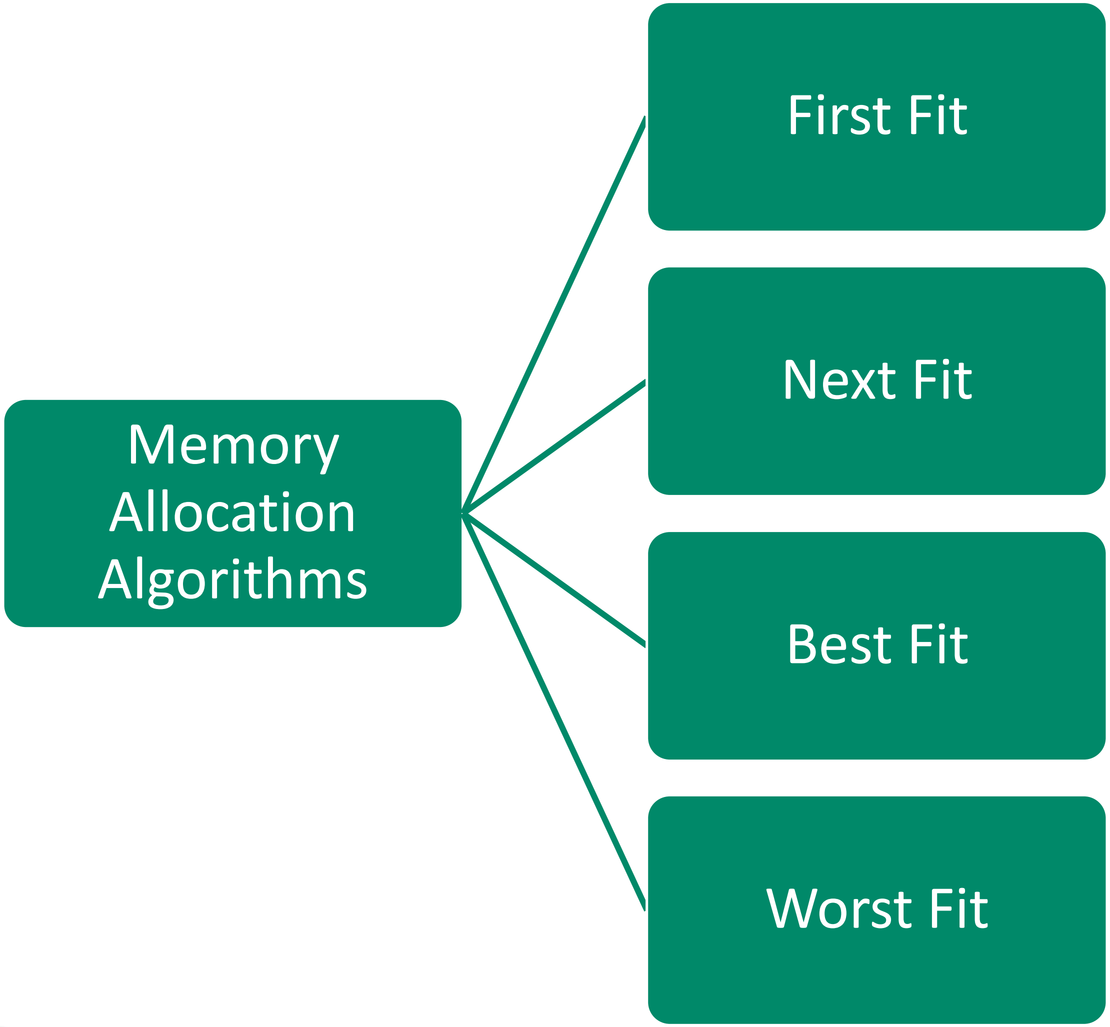


Memory Allocation Problem

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



Memory Allocation Problem



Memory Allocation Algorithms

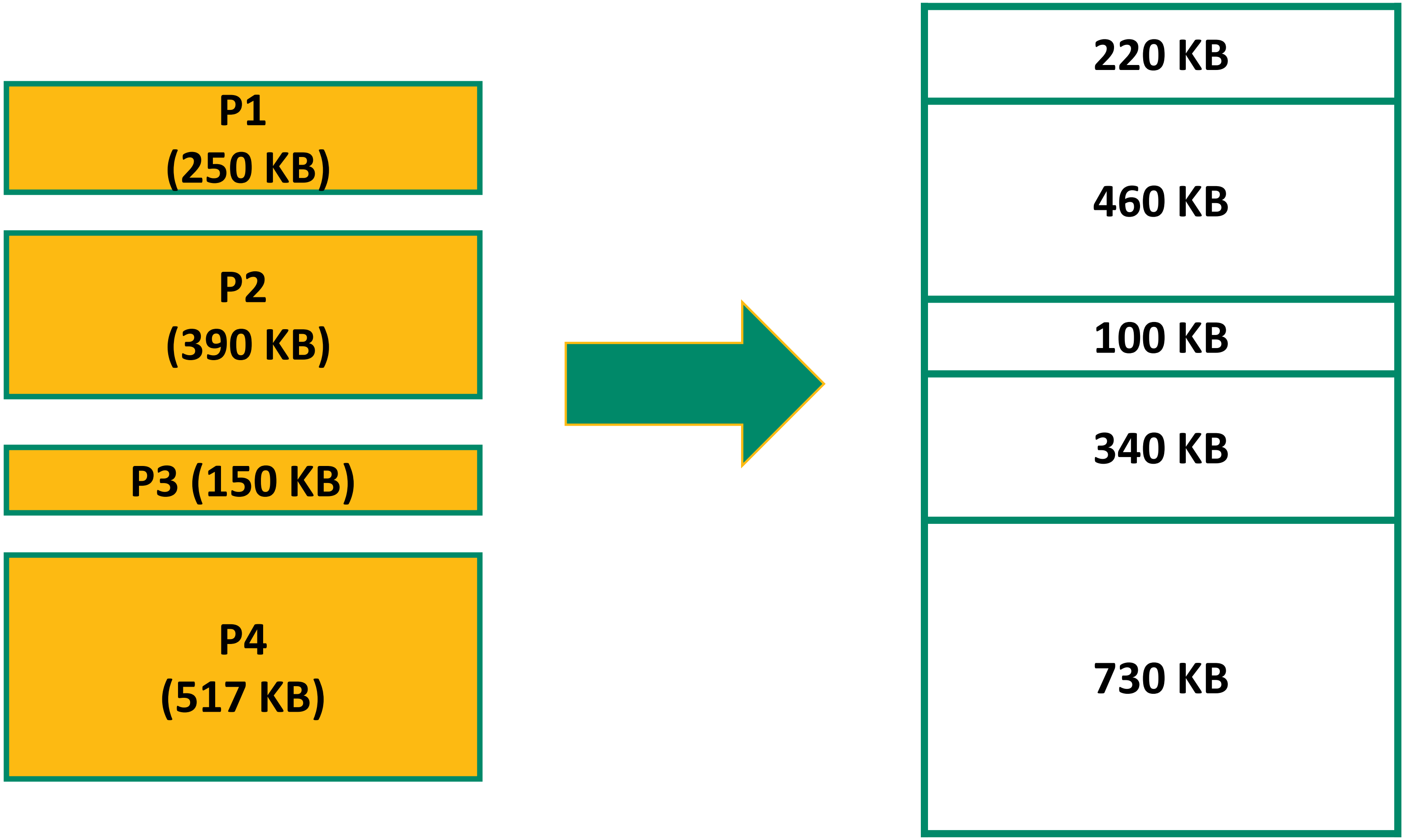
First Fit

Next Fit

Best Fit

Worst Fit

Memory Allocation Problem



P1
(250 KB)

P2
(390 KB)

P3 (150 KB)

P4
(517 KB)

220 KB

460 KB

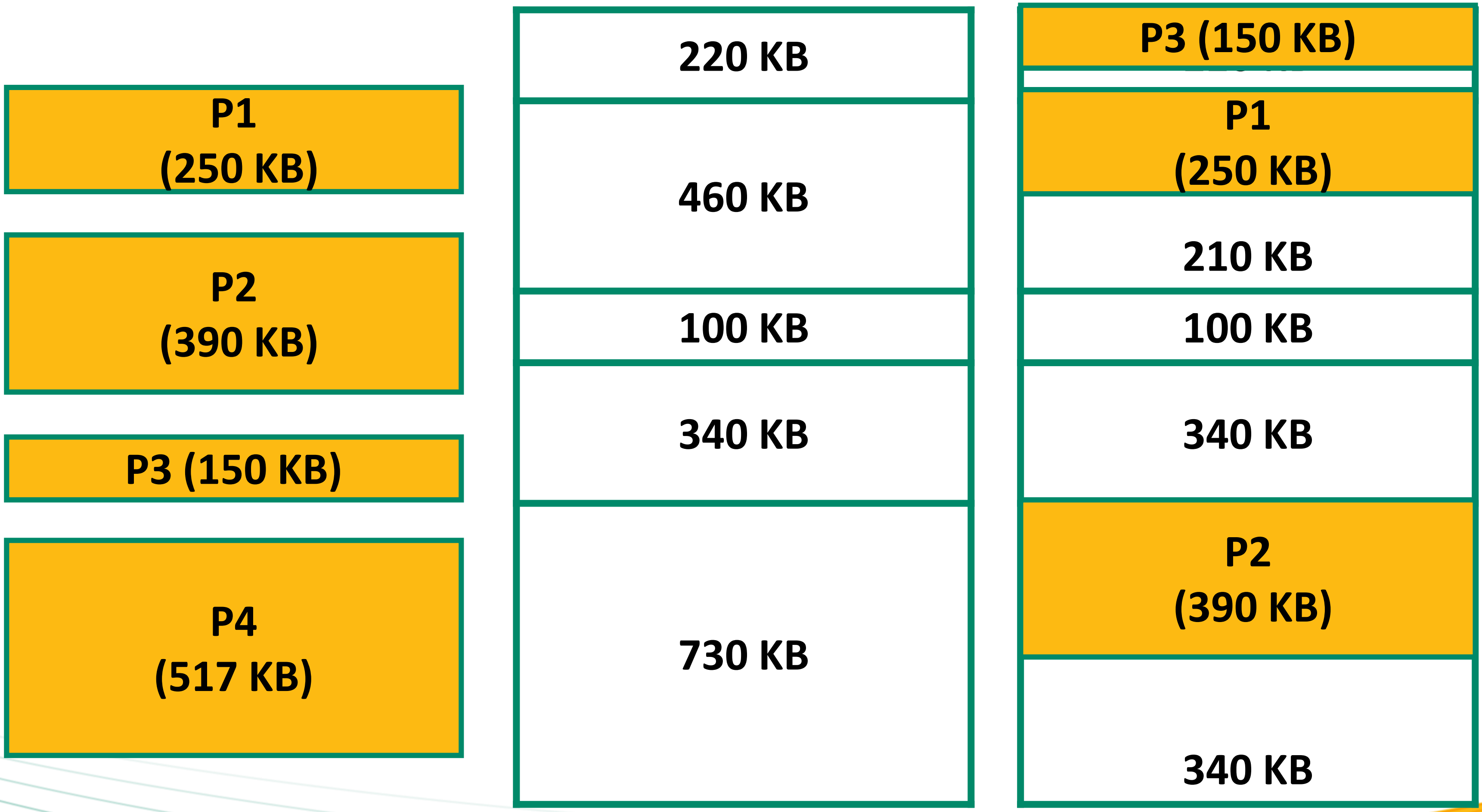
100 KB

340 KB

730 KB

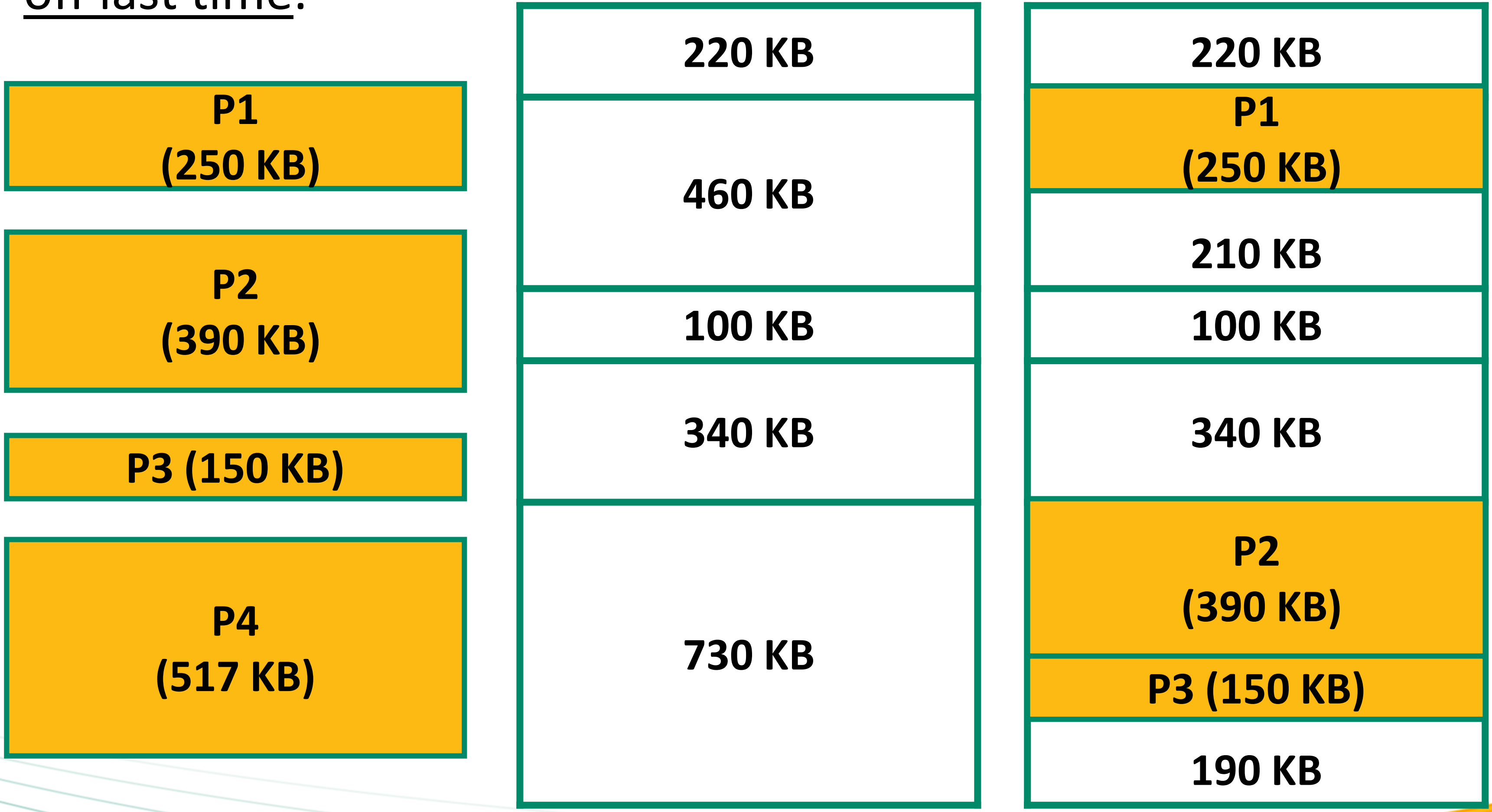
First Fit Memory Allocation Algorithm

- Allocate the **first hole that is big enough**



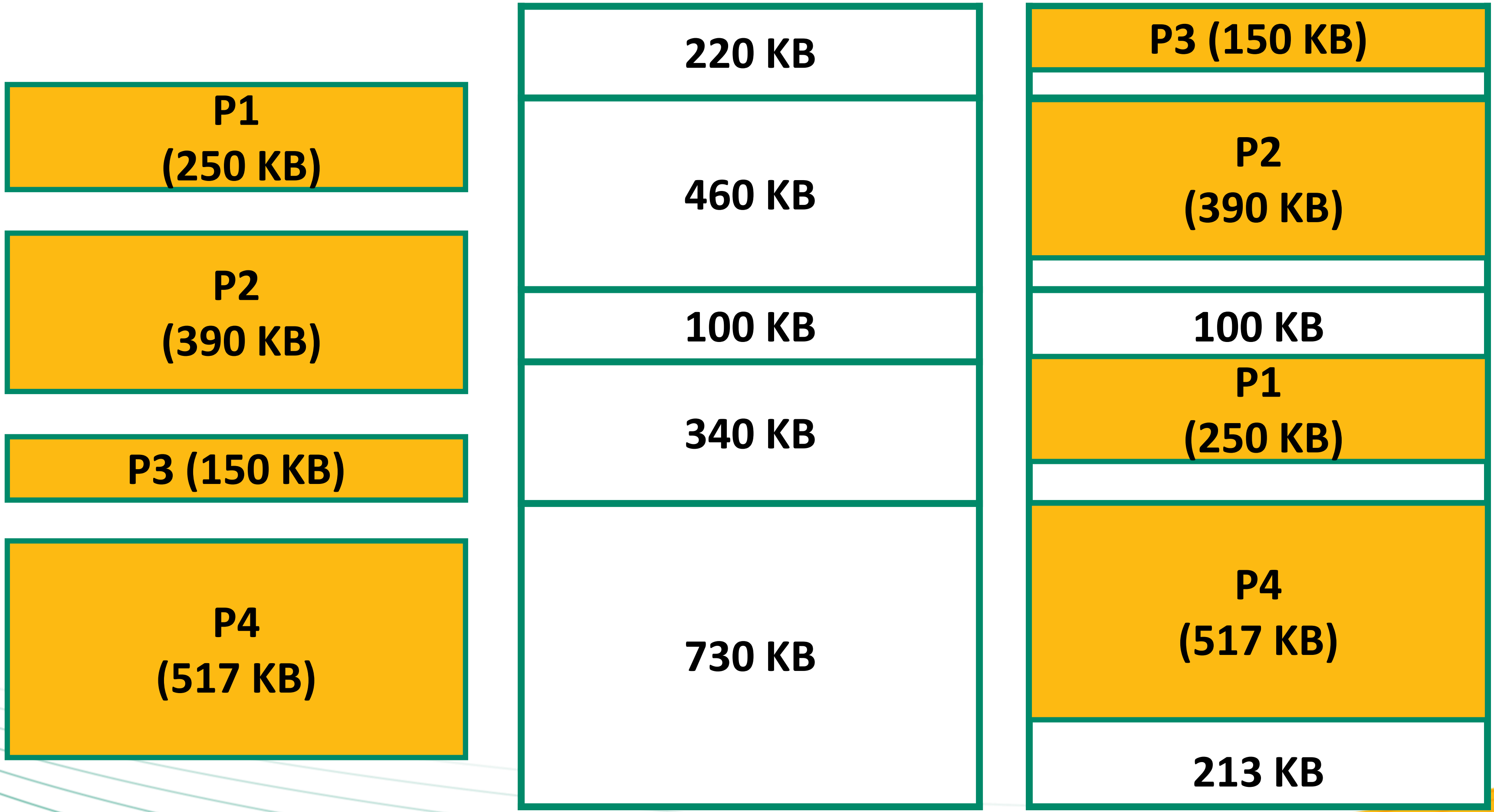
Next Fit Memory Allocation Algorithm

- Start the search for the first-fit hole from the place we left off last time.



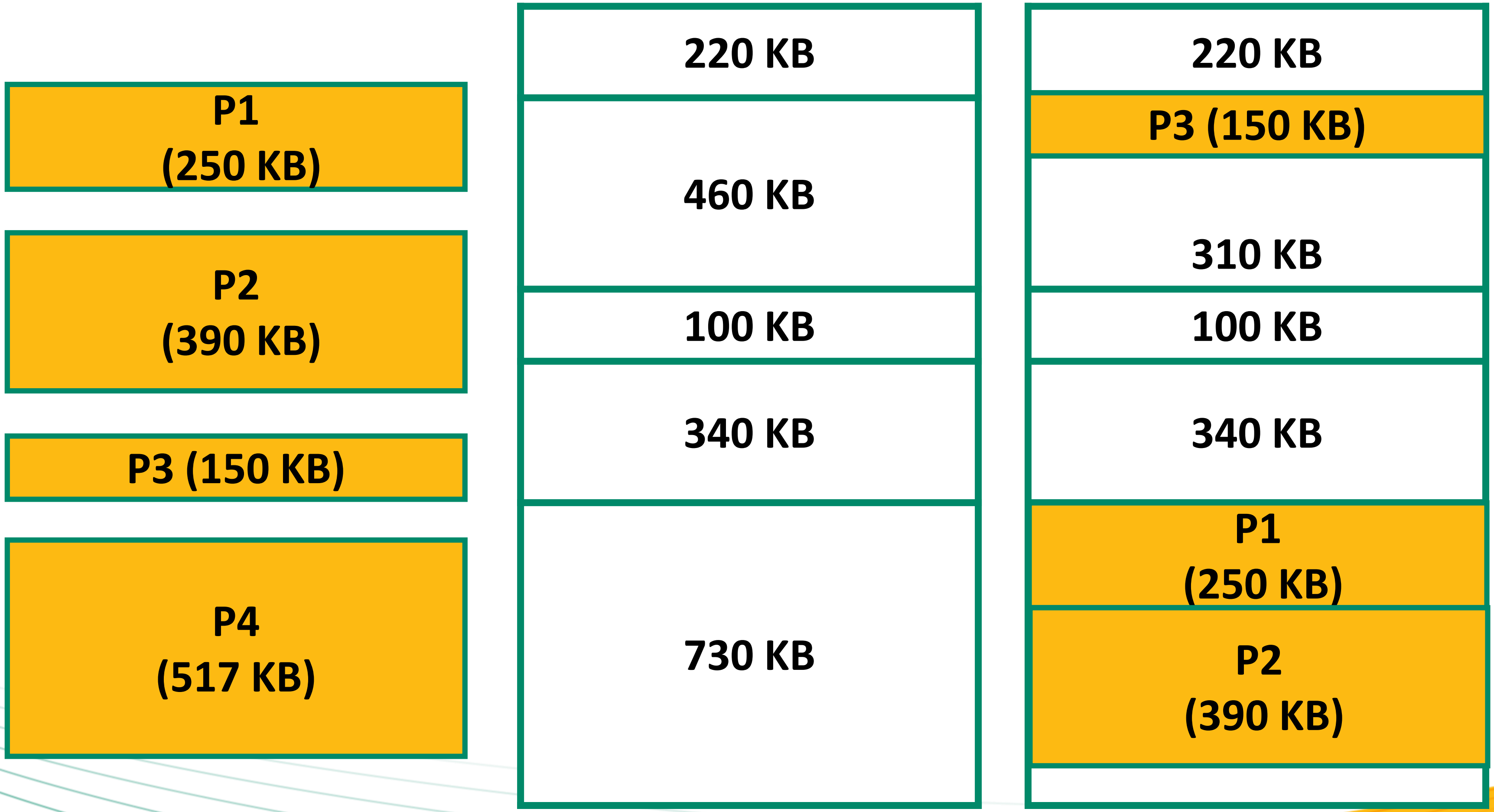
Best Fit Memory Allocation Algorithm

- Allocate the smallest hole that is big enough.
- Must search entire list, unless ordered by size
- Produces the smallest leftover hole



Worst Fit Memory Allocation Algorithm

- Allocate the largest hole; must also search entire list
- Produces the largest leftover hole



P1 (250 KB)

P2 (390 KB)

P3 (150 KB)

P4 (517 KB)

220 KB

460 KB

100 KB

340 KB

730 KB

220 KB

P3 (150 KB)

310 KB

100 KB

340 KB

P1 (250 KB)

P2 (390 KB)

Exercise

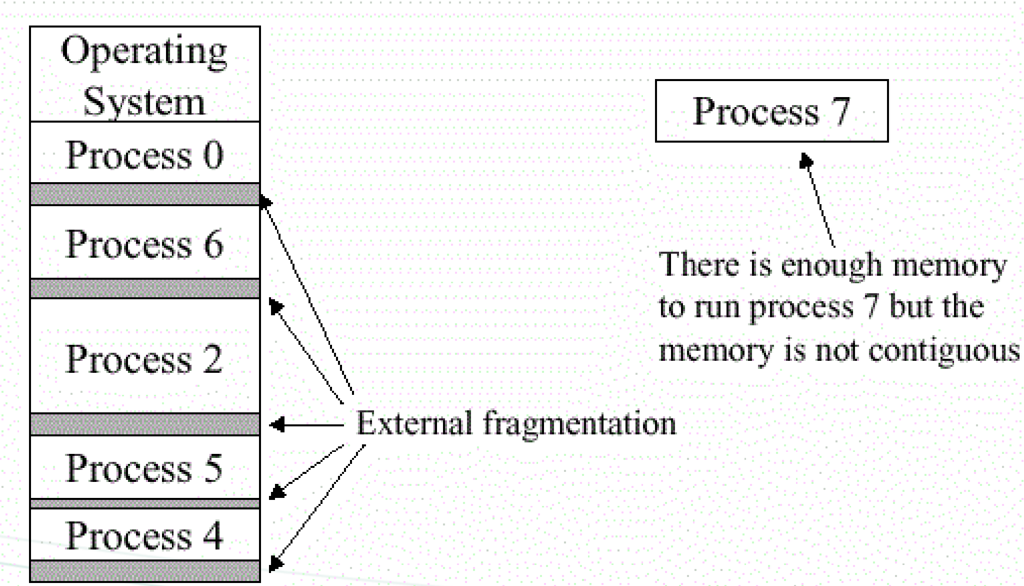
- Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order),
- how would the first-fit, next fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)?
- Rank the algorithms in terms of how efficiently they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

Exercise

- Given memory partitions of 100K, 500K, 200K, 300K, and 600K (in order),
- how would each of the First-fit, Next-Fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K, 112K, and 426K (in order)?
- Which algorithm makes the most efficient use of memory?

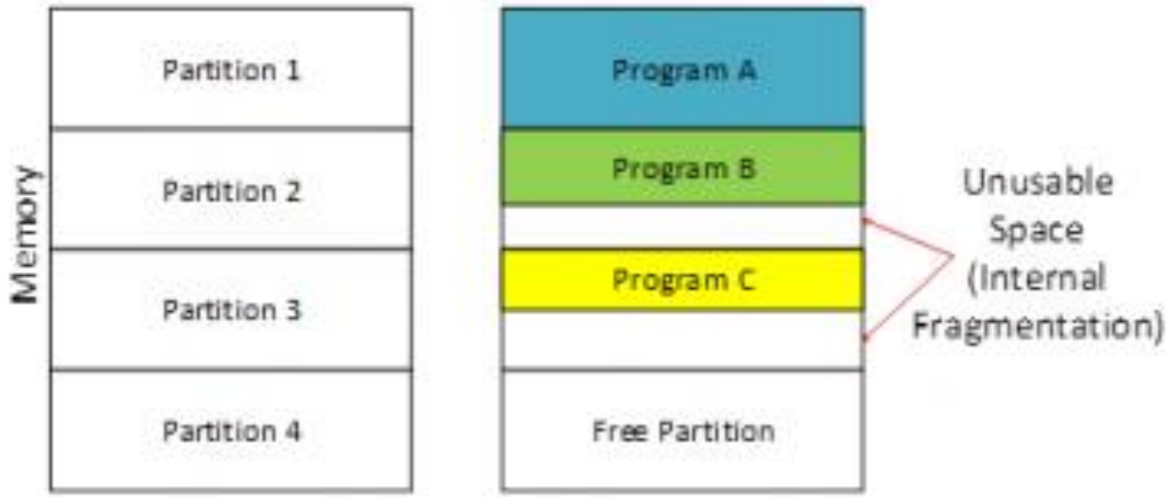
External Fragmentation

- **Total memory space exists to satisfy a request, but it is not contiguous**
- First fit analysis ➔ for every *N* blocks allocated, 0.5 *N* blocks lost to fragmentation ➔ **50-percent rule**
- $Unusable\ memory\ = \frac{0.5N}{N+0.5N} = \frac{1}{3}$



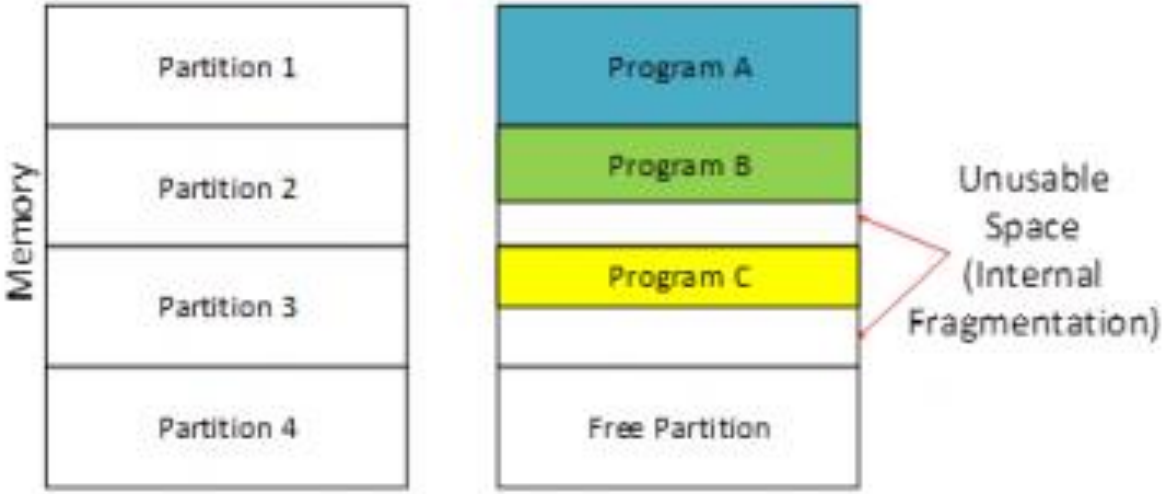
Internal Fragmentation

- Issue: Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes.
  - The overhead to keep track of this hole will be substantially larger than the hole itself.



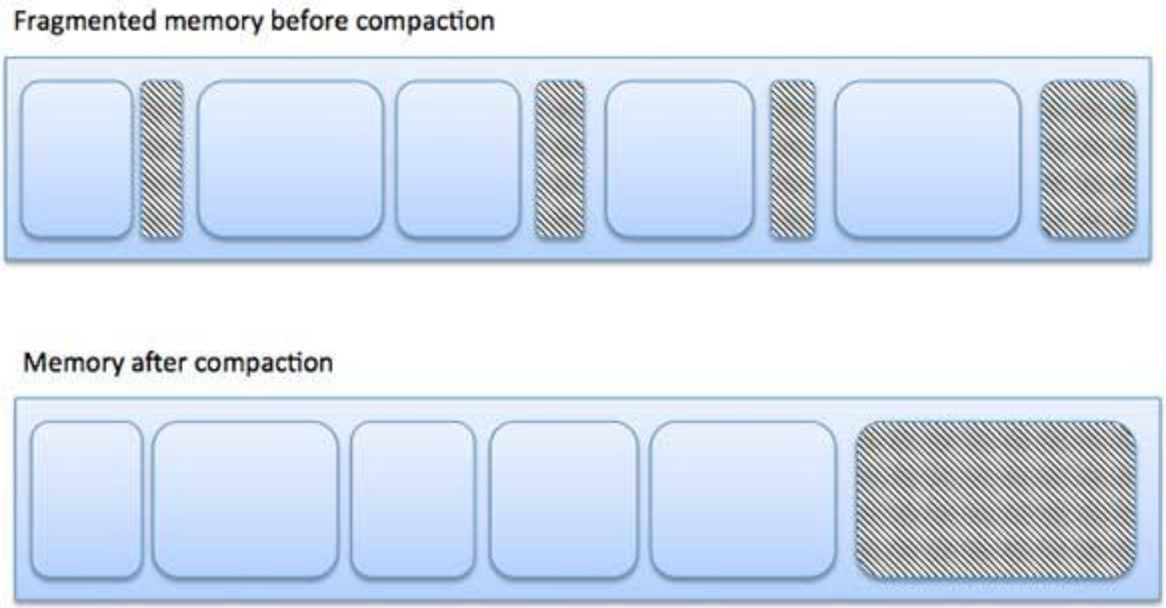
Internal Fragmentation

- **Solution**: *break the physical memory into **fixed-sized blocks** and allocate memory in units based on block size*.
- Allocated memory may be slightly larger than requested memory
  - This size difference is memory **internal** to a partition, but not being used



Fragmentation – Compaction

- 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**



Background

Contiguous Memory Allocation

Segmentation

Paging

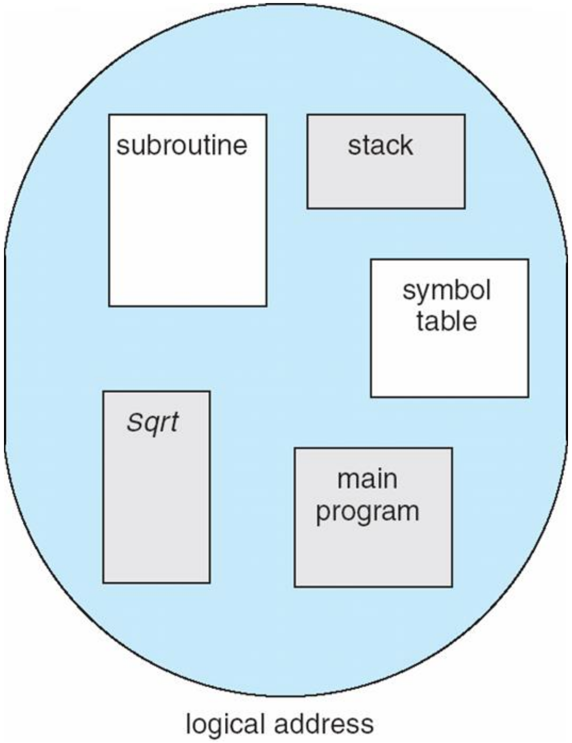
Structure of the Page Table

Swapping

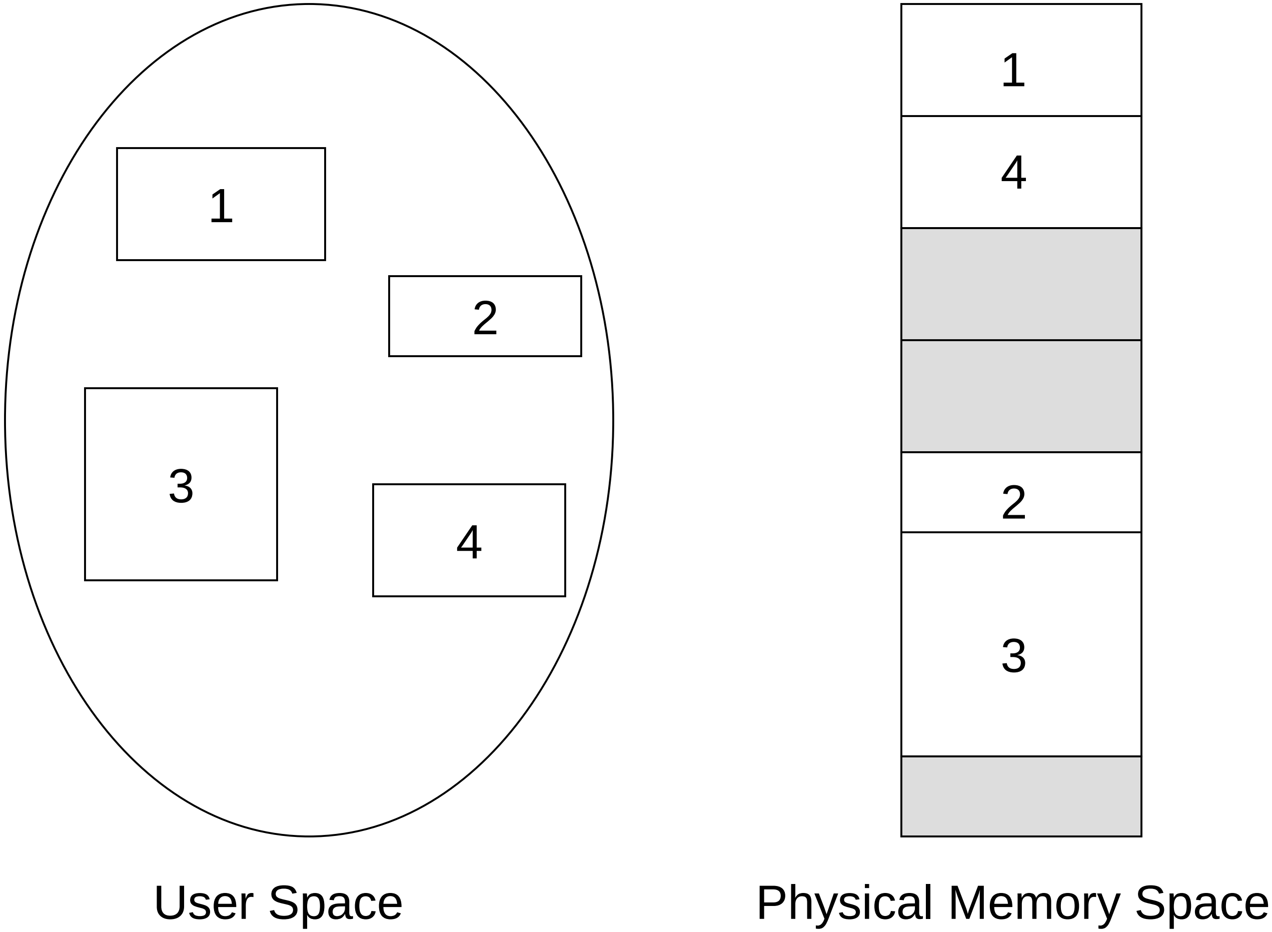
Segmentation

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

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



Logical View of Segmentation



Segmentation Architecture

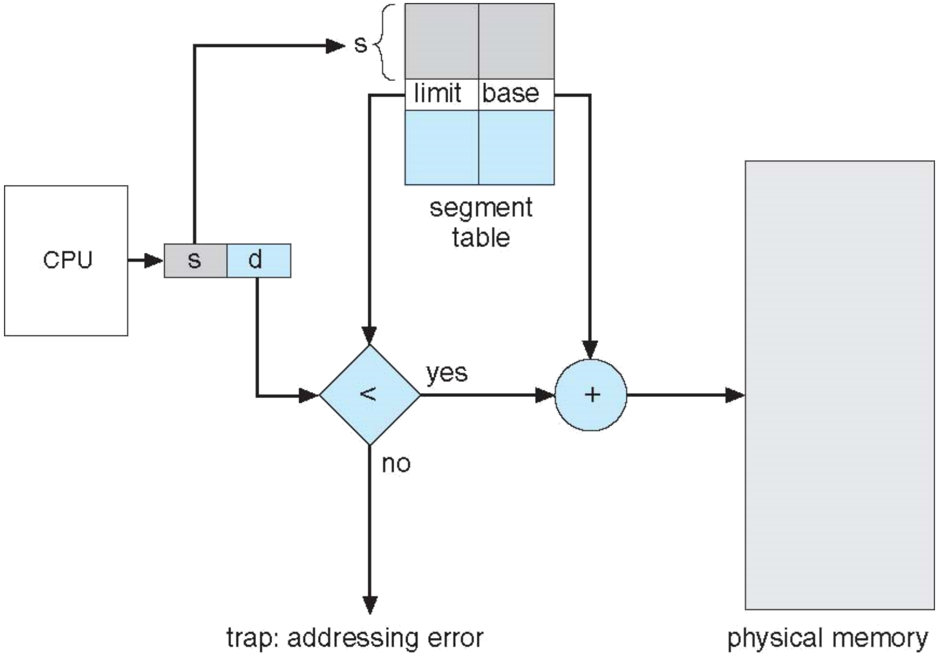
- Logical address consists of a two tuple:

  `<segment-number, offset>`

  - Programmer can now refer to objects in the program by a two-dimensional address
  - The actual **physical memory** is a one dimensional sequence of bytes.
- **How to map two-dimensional user-defined addresses into one-dimensional physical?**

Segmentation Hardware

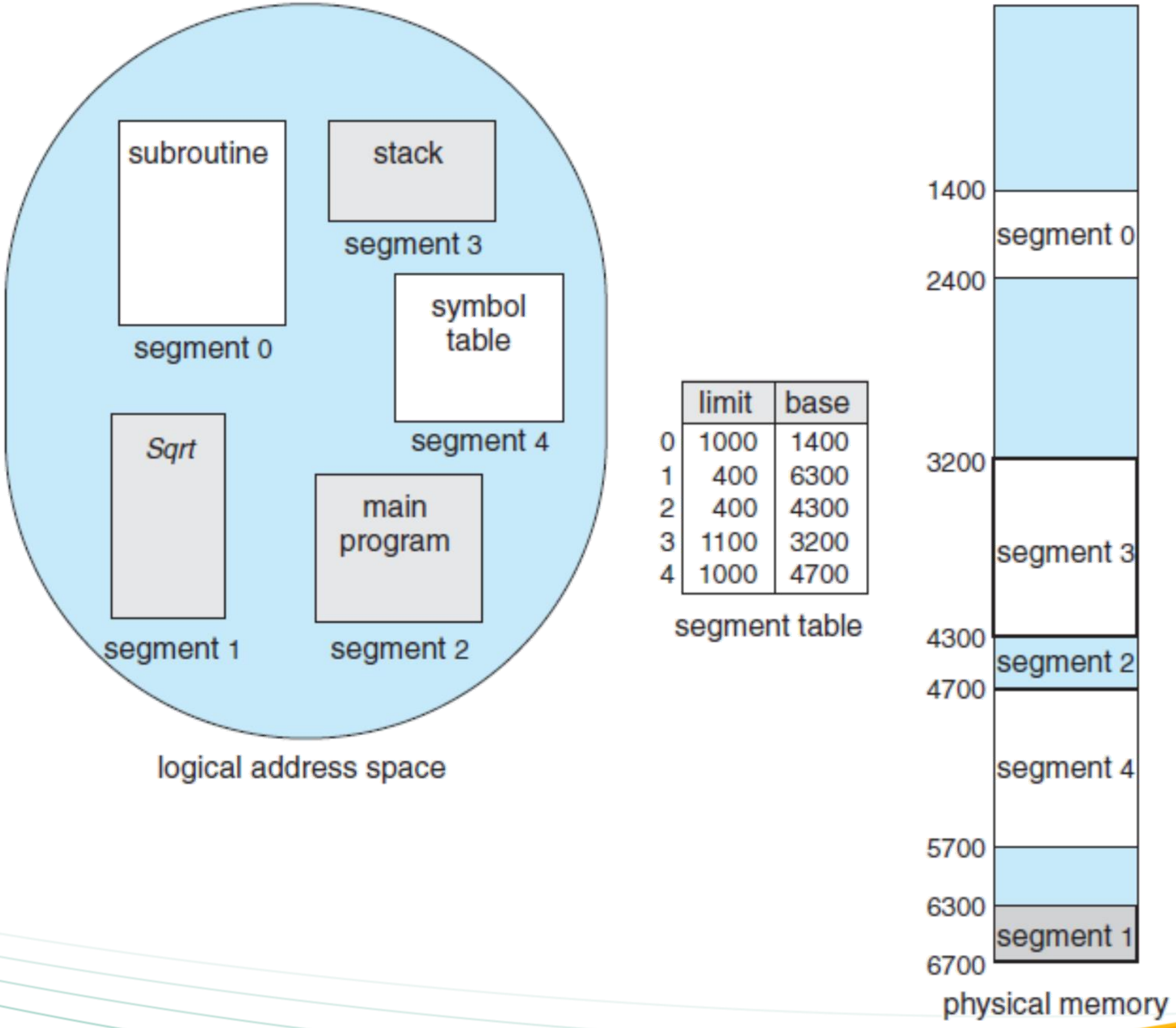
- **Segment table** maps 2D physical addresses; each table entry has:
  - **Base**: contains the starting physical address where the segments reside in memory
  - **Limit/length:** specifies the length of the segment



- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

segment number s is legal if s < STLR

Segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

Exercise

- Consider a simple segmentation system with the following segment table:

Segment #	Starting address	Length(bytes)
0	660	248
1	1752	422
2	222	198
3	996	604

- For each of the following logical addresses, determine the corresponding physical address or indicate if an interrupt is generated.
  - 0, 198
  - 2, 156
  - 1, 530
  - 3, 444
  - 0,222

Background

Contiguous Memory Allocation

Segmentation

Paging

Structure of the Page Table

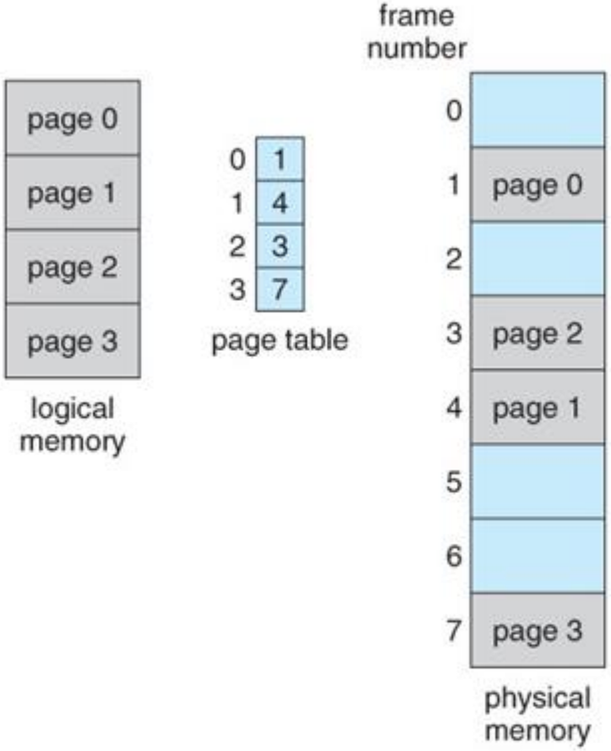
Swapping

Paging - Introduction

- Memory Management Scheme
- Advantages
  - **Non-contiguous** physical address space (same as segmentation)
  - **Avoids external fragmentation**
  - **Avoids problem of varying sized memory chunks**
- Implemented through **cooperation** between the operating system and the computer hardware

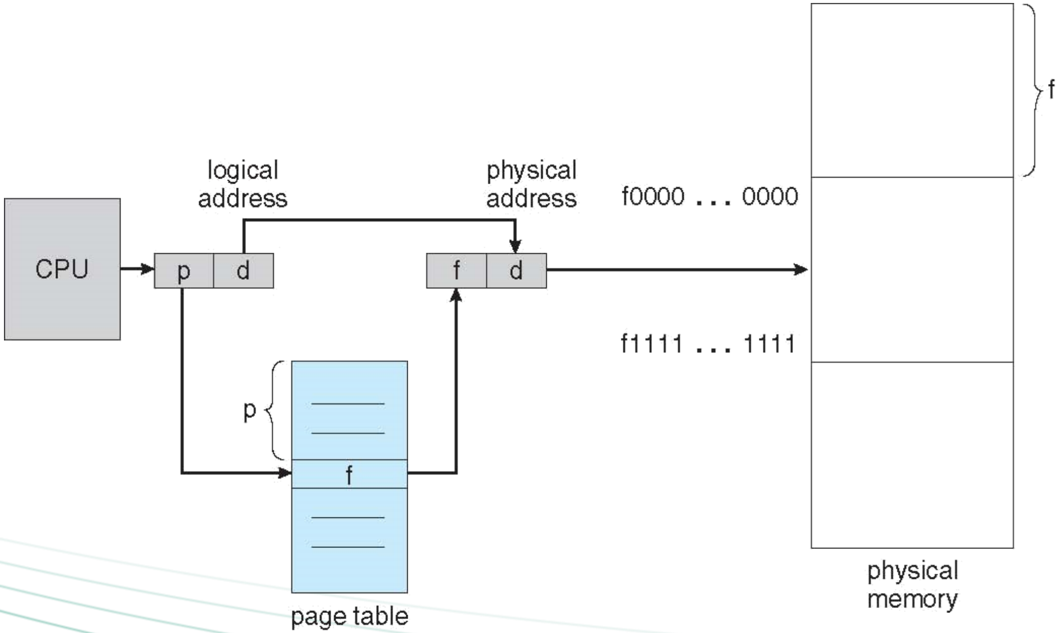
Paging

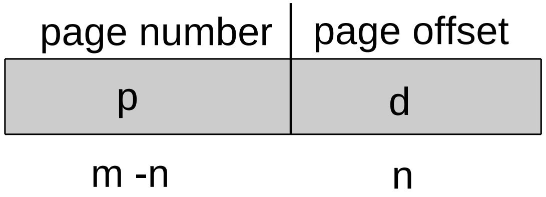
- Divide physical memory into fixed-sized blocks called **Frames**
  - ○ Size = 2^n, $512\ bytes \le 2^n \le 1\ GB$
- Divide logical memory into blocks of same size called **Pages**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to **translate logical to physical** addresses
- **Backing store** likewise split into pages
- Still have Internal fragmentation



Address Translation Scheme

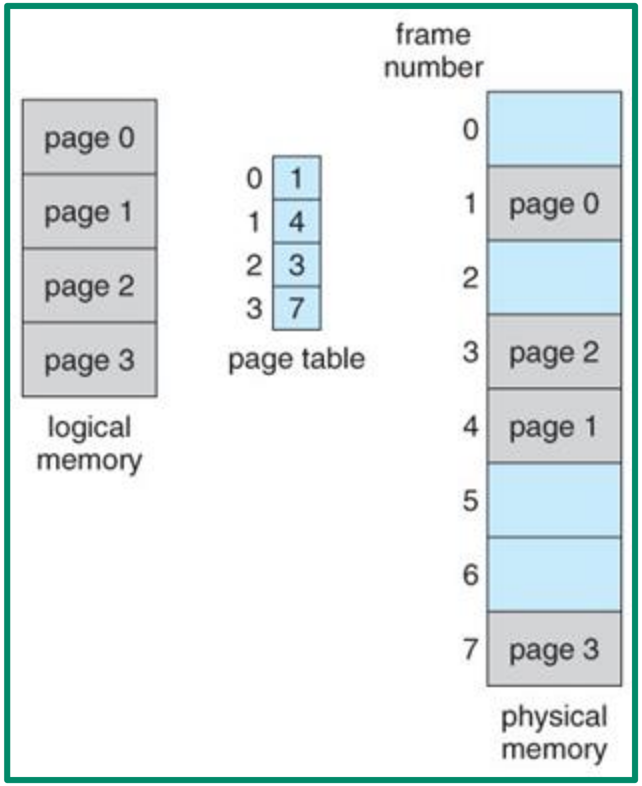
- Address generated by CPU is divided into:
  - **Page number (p)**: used as an **index into a page table** which contains base address of each page in physical memory
  - **Page offset (d)** : combined with base address to define the physical memory address that is sent to the memory unit

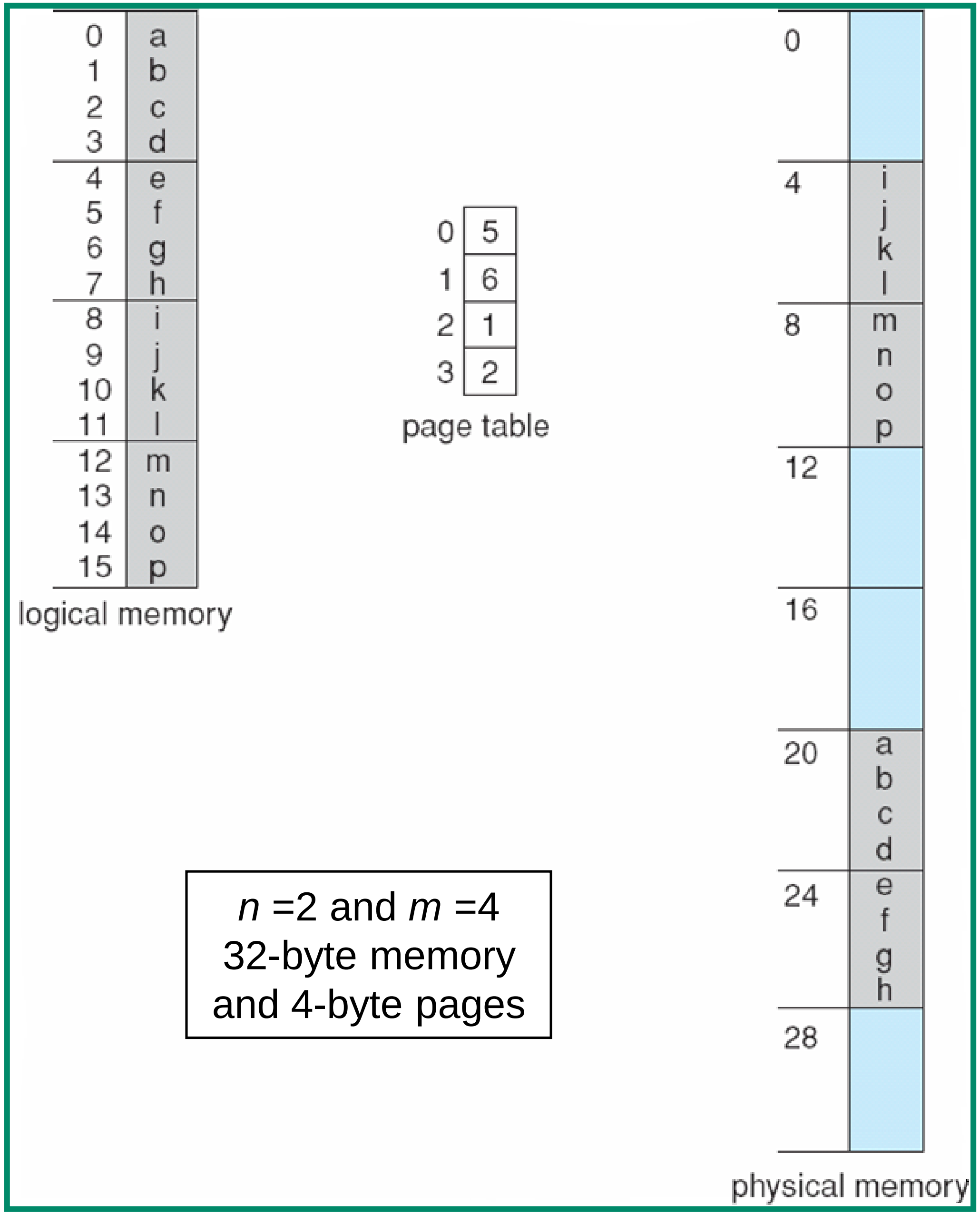




For given logical address space 2^m and page size 2^n

Paging Example



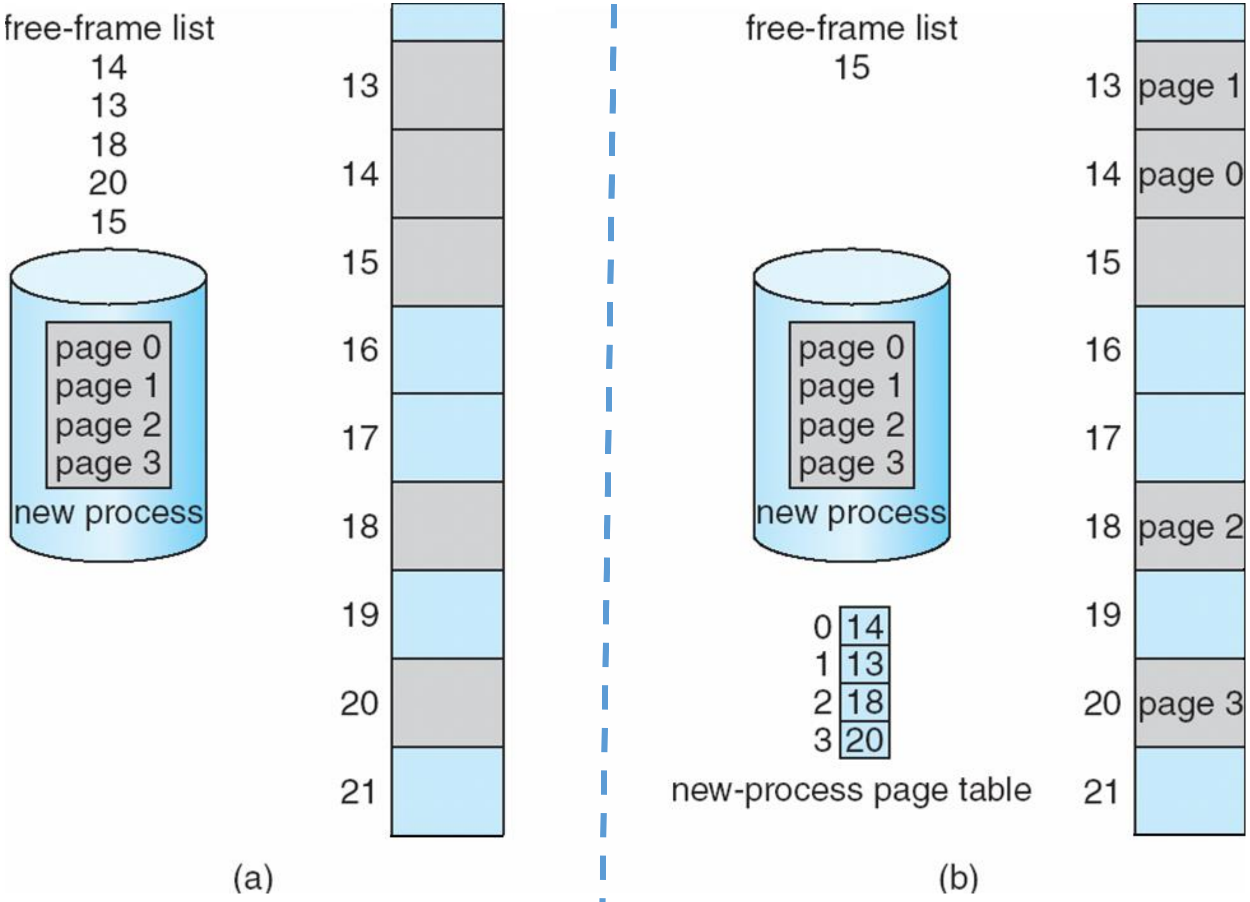


n =2 and *m* =4
32-byte memory
and 4-byte pages

Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
- So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB

Free Frames



Before allocation

After allocation

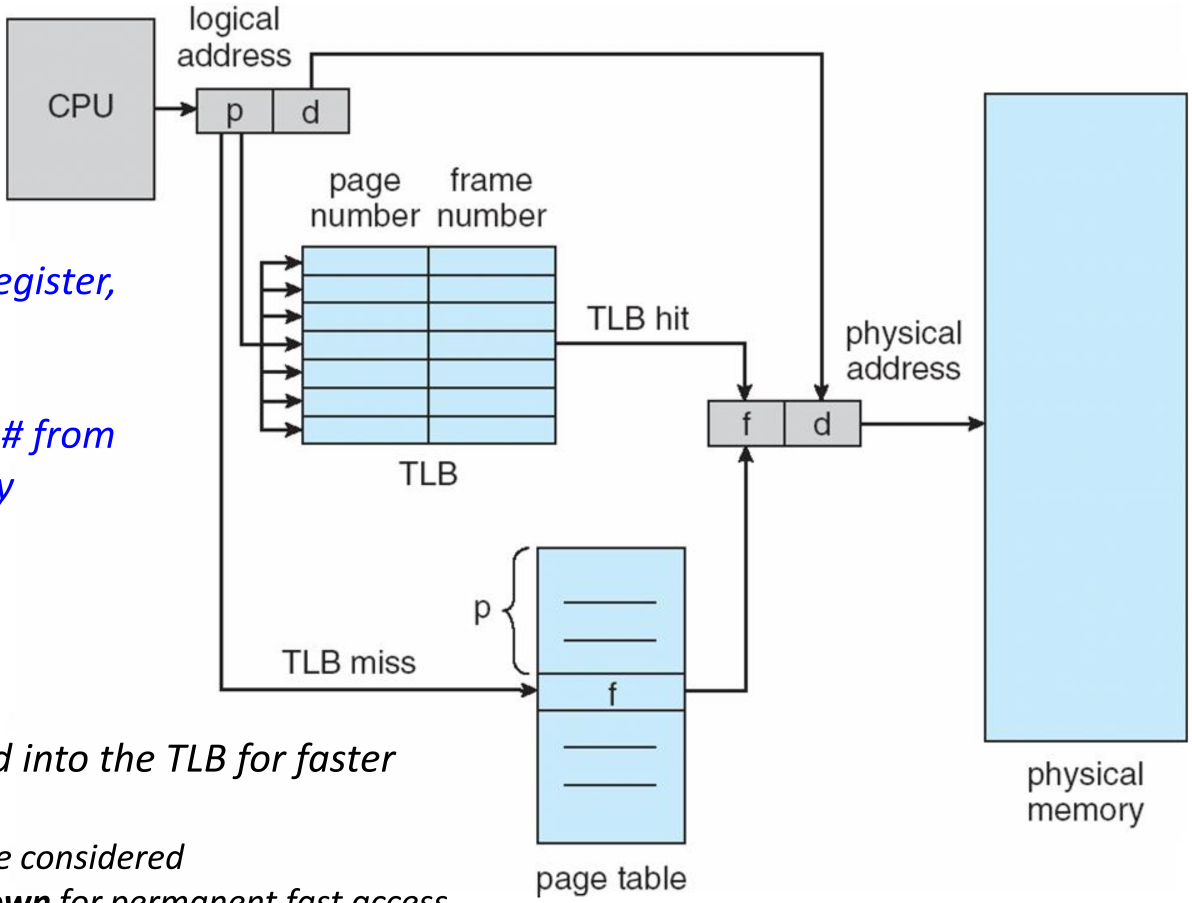
Implementation of Page Table

- 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
- **“Two memory access”** problem can be solved by the use of a special fast-lookup hardware **cache** called **Translation Look-aside Buffers** (**TLBs**)

Implementation of Page Table (Cont.)

- TLB is associative, high speed memory
- **Each entry in a TLB** consists of a **key** and a **value**
- Search for an item
  - **Compare** the item with **all keys simultaneously**.
  - If the item is found, the corresponding value field is returned.
- The search is fast
- TLBs typically small (64 to 1,024 entries)

Paging Hardware With TLB

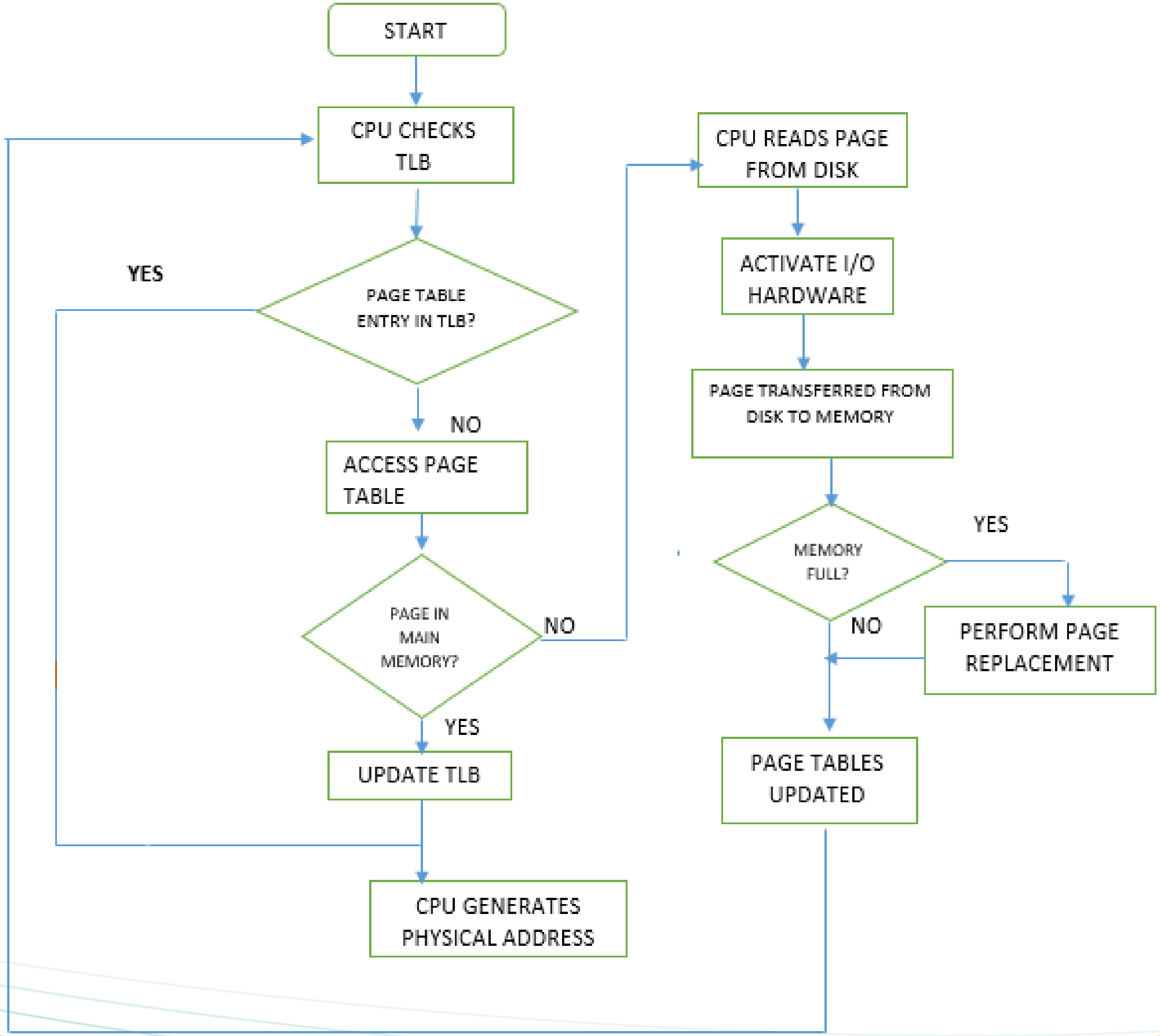


If p is in associative register, get frame # out

Otherwise get frame # from page table in memory

- *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*

Paging Hardware With TLB



Effective Access Time

- Associative Lookup = ϵ
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio : percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- If t is the memory access time what is the equation of **Effective Access Time (EAT)** in terms of $\boldsymbol{t}$, $\boldsymbol{\varepsilon}$ and $\boldsymbol{\alpha}$?

Effective Access Time (cont'd)

- Effective Access Time (EAT)

$$EAT = (t + \epsilon)\,\alpha + (2t + \epsilon)(1 - \alpha)$$

- Consider $\alpha = 80\%, \epsilon = 20ns$ for TLB search, $100ns$ for memory access
  - ○ Calculate EAT

- Consider more realistic hit ratio ➔ $\alpha = 99\%, \varepsilon = 20ns$ for TLB search, $100ns$ for memory access
  - ○ Calculate EAT

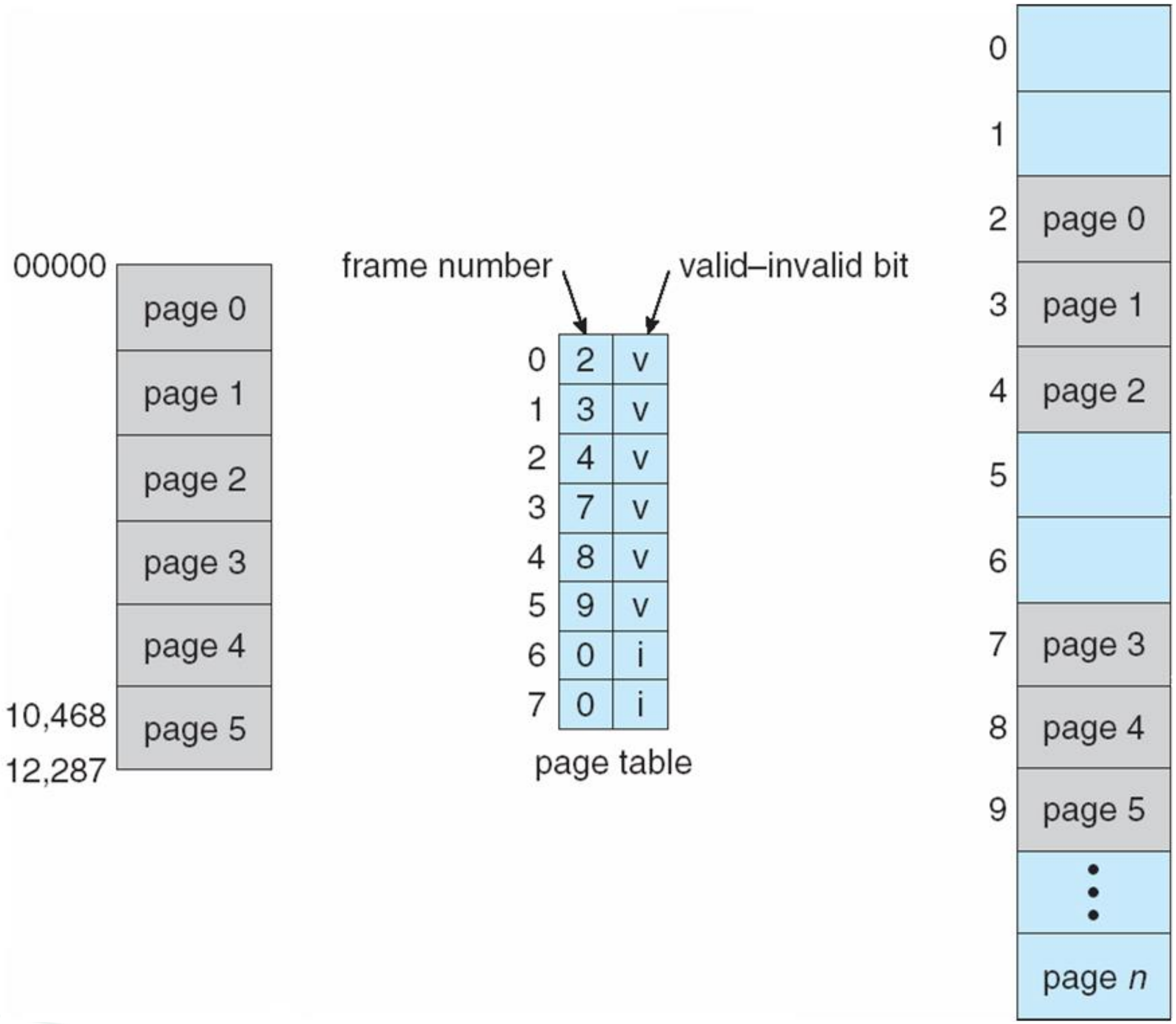
Paging Hardware With TLB (cont’d)

- CPUs today may provide multiple levels of TLBs. Calculating memory access times in modern CPUs is therefore much more complicated than the previous example.
- For instance, the Intel Core i7 CPU has a 128-entry L1 instruction TLB and a 64-entry L1 data TLB. In the case of a miss at L1, it takes the CPU six cycles to check for the entry in the L2 512-entry TLB.
- A miss in L2 means that the CPU must either walk through the page-table entries in memory to find the associated frame address, which can take hundreds of cycles, or interrupt to the operating system to have it do the work.

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



	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 Entry Format

- Physical page Number
- Valid/Invalid bit
- Protection bits (Read / Write / Execute)
- Modified bit (set on a write/store to a page)
  - Useful for page write-backs on a page-replacement.
- Referenced bit (set on each read/write to a page).
  - Will look at how this is used a little later.
- Disable caching
  - Useful for I/O devices that are memory-mapped.

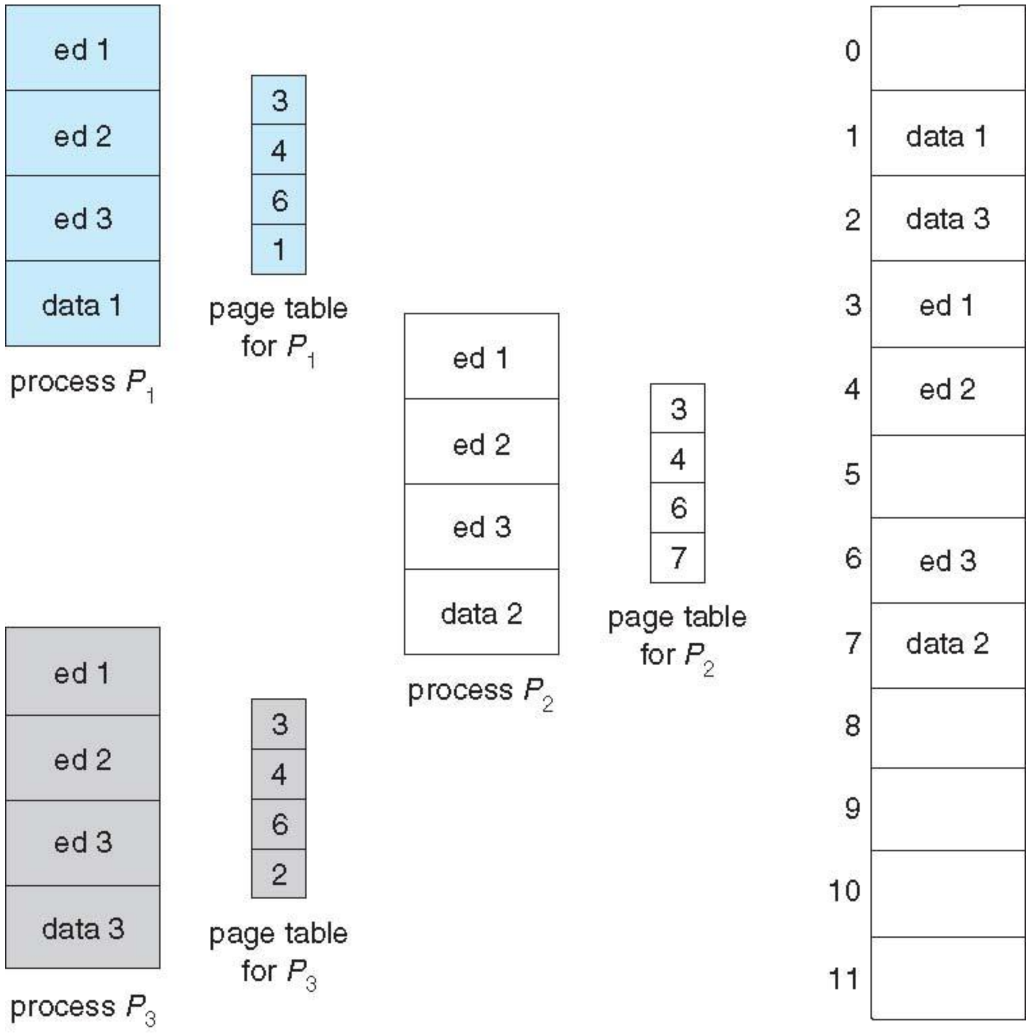
Shared Pages - Exercise

- Consider a system that supports 40 users, each of whom executes a text editor. The text editor consists of 150 KB of code and 50 KB of data space. What is the memory required to support the 40 users?

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
  - Useful for inter-process communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process keeps a **separate copy of the code and data**
  - Pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example



Shared Pages – Exercise – Revisited

- Consider a system that supports 40 users, each of whom executes a text editor. The text editor consists of 150 KB of code and 50 KB of data space. What is the memory required to support the 40 users, if shared pages are used?

Background

Contiguous Memory Allocation

Segmentation

Paging

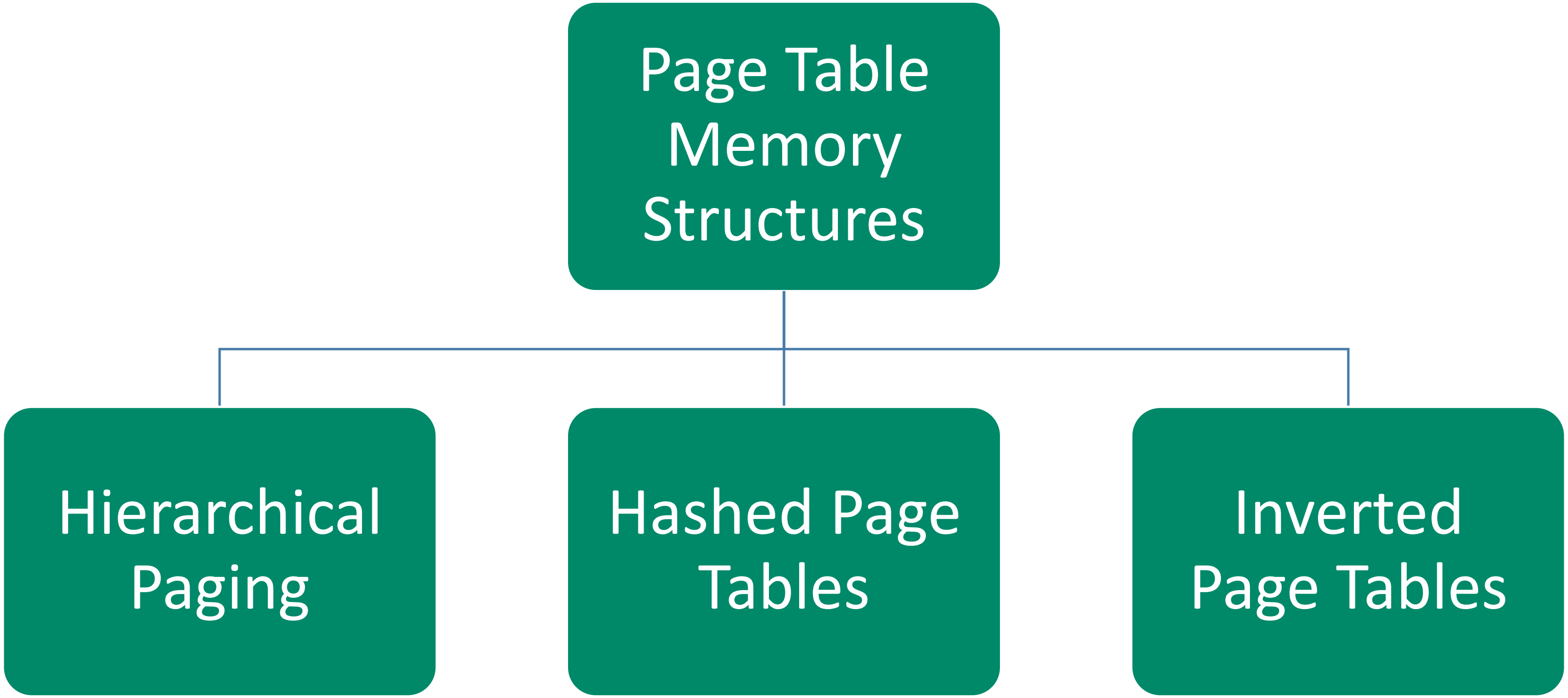
Structure of the Page Table

Swapping

Structure of the Page Table

- **Memory structures for paging** can get **huge** using straightforward methods
  - Consider a 32-bit logical address space
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes ➔ 4 MB of physical address space / memory for page table alone
    - High Cost
    - Don’t want to allocate that contiguously in main memory

Structure of the Page Table



Page Table Memory Structures

Hierarchical Paging

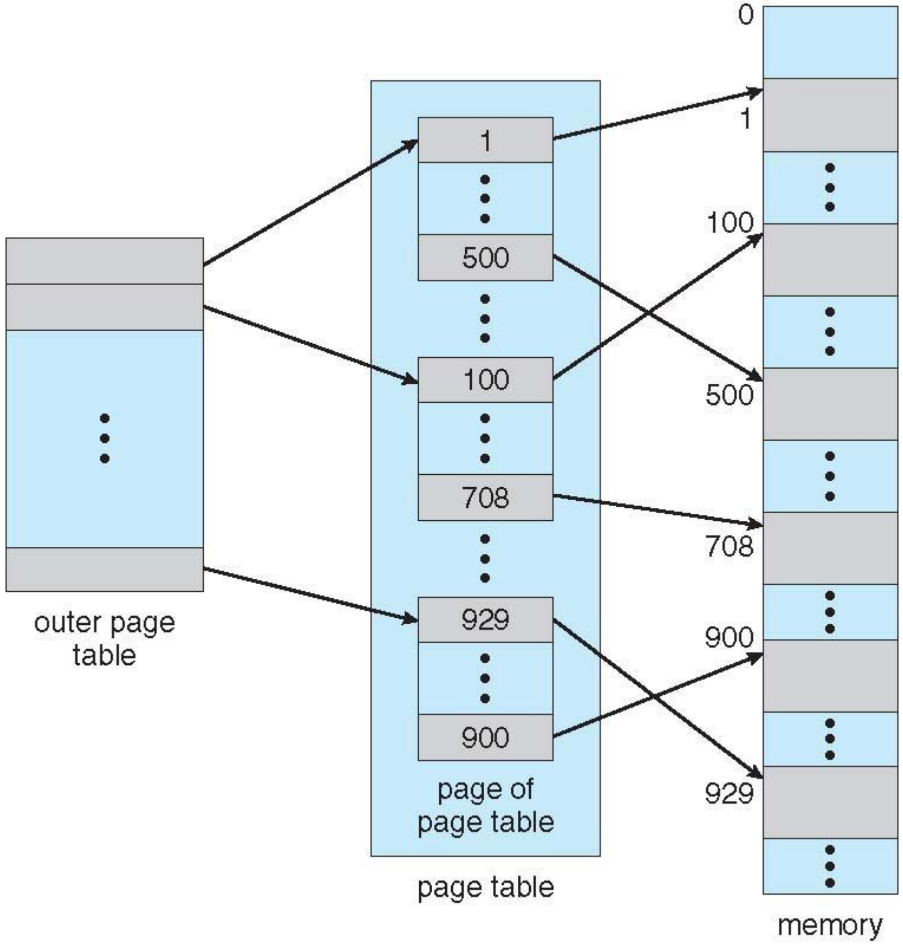
Hashed Page Tables

Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a **two-level page table** => page the page table

Two-Level Page-Table Scheme

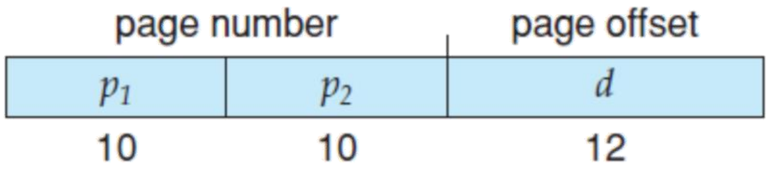


Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - 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

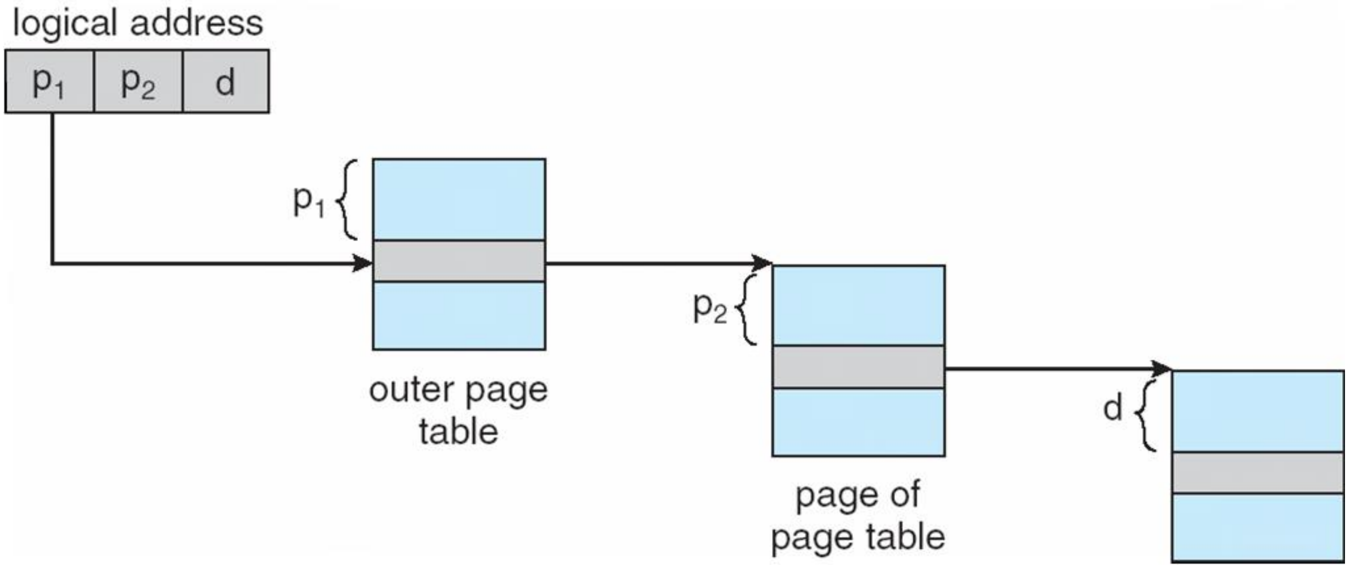
- Thus, a logical address is as follows:



page number		page offset
p_1	p_2	d
10	10	12

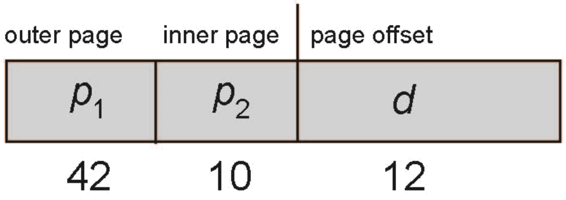
  - p_1 is an index into the outer page table
  - p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation Scheme



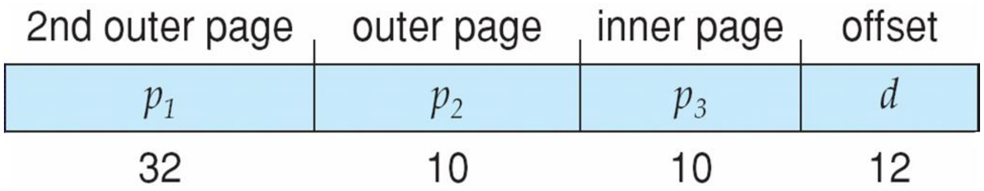
64-bit Logical Address Space

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



outer page	inner page	page offset
p_1	p_2	d
42	10	12

  - Outer page table has 2^{42} entries or 2^{44} bytes
  - Solution: add a 2^{nd} outer page table

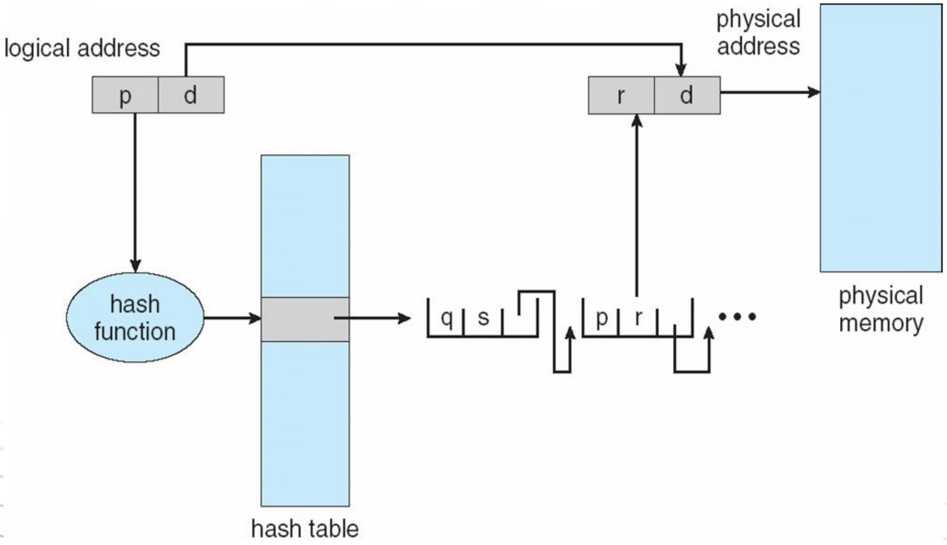


2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

  - But in the following example the 2^{nd} outer page table is still 2^{34} bytes (16GB) in size
    - And possibly 4 memory access to get to one physical memory location

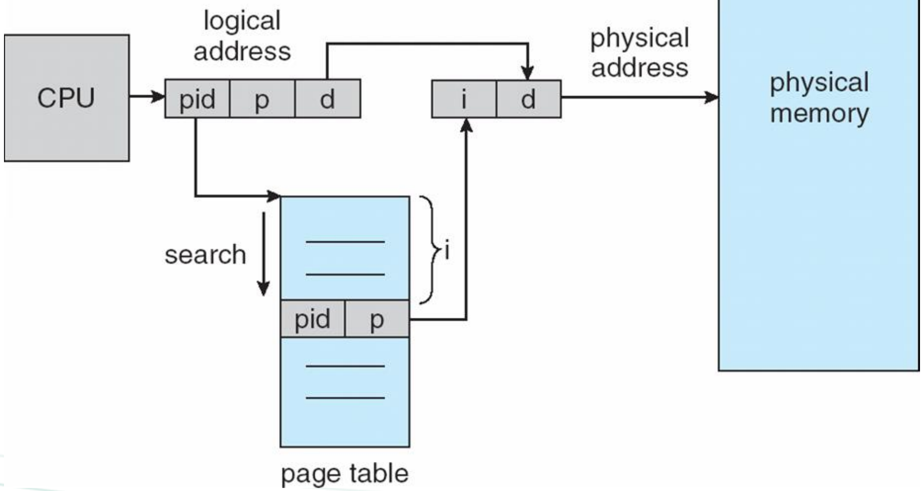
Hashed Page Tables

- 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
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted



Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, **track all physical pages**
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page



Inverted Page Table (cont'd)

- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
  - The inverted page table is sorted by physical address, but lookups occur on virtual addresses
    - the whole table might need to be searched before a match is found.
- **Solution:** Use hash table to limit the search to one (or at most a few) page-table entries
  - TLB can accelerate access

Background

Contiguous Memory Allocation

Segmentation

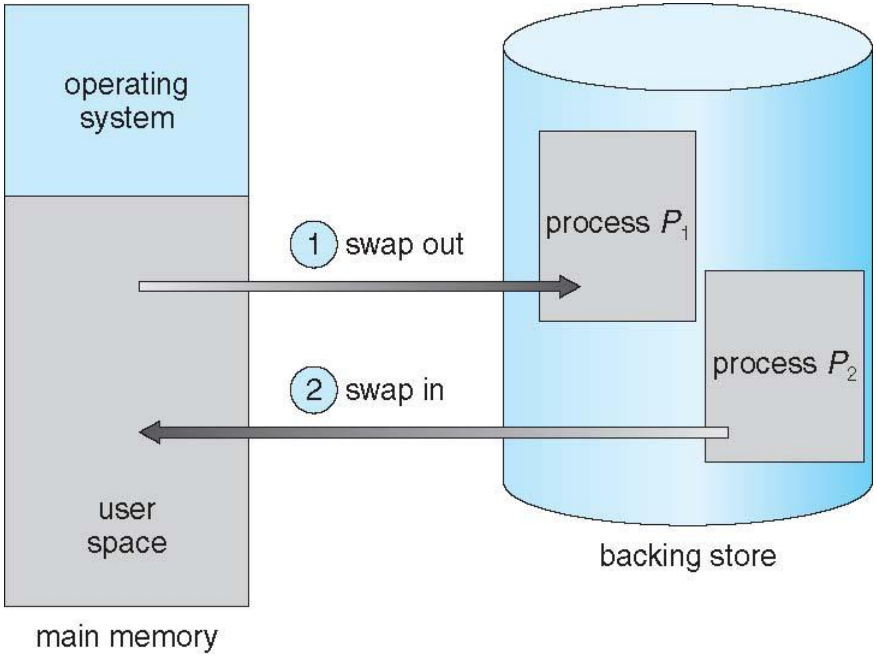
Paging

Structure of the Page Table

Swapping

Swapping

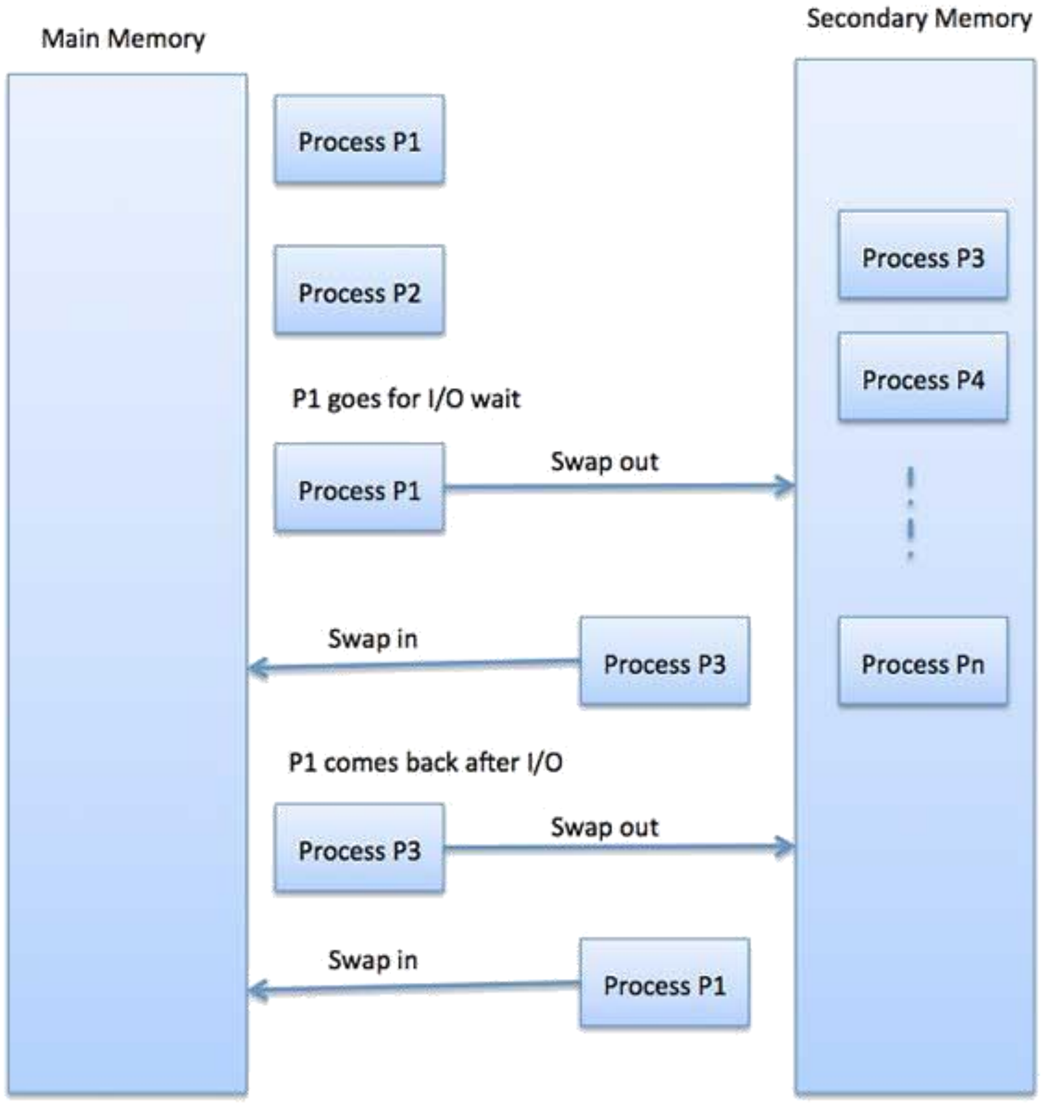
- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into (swapped in) memory for continued execution
  - *Total physical memory space of processes can exceed physical memory*



- **Backing store**: **fast** disk **large** enough to accommodate copies of all memory images for all users
  - Must provide direct access to these memory images

Swapping (cont'd)

- Major part of swap time is transfer time
  - Total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk



Swapping (cont.)

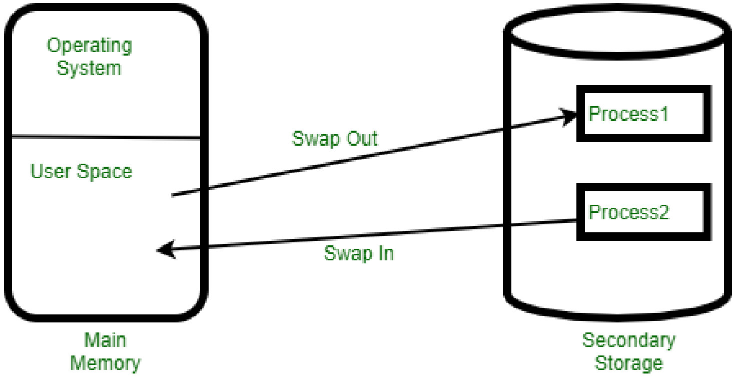
- **Q:** Does the swapped out process need to swap back in to **same physical addresses**?
- **A:** Depends on address binding method
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - **Swapping** normally disabled
  - Started if more than threshold amount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

Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory:
  1. Swap out a process and
  2. Swap in target process
- Context switch time can then be very high

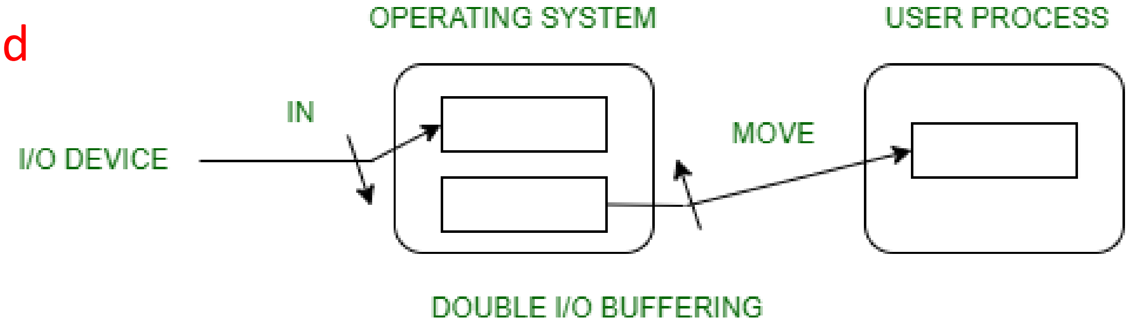
Example:

- 100MB process swapping to hard disk with transfer rate of 50MB/sec
- Swap out time of 2000 ms
- Plus swap in of same sized process
- Total context switch swapping component time of 4000ms (4 seconds)



Context Switch Time and Swapping (Cont.)

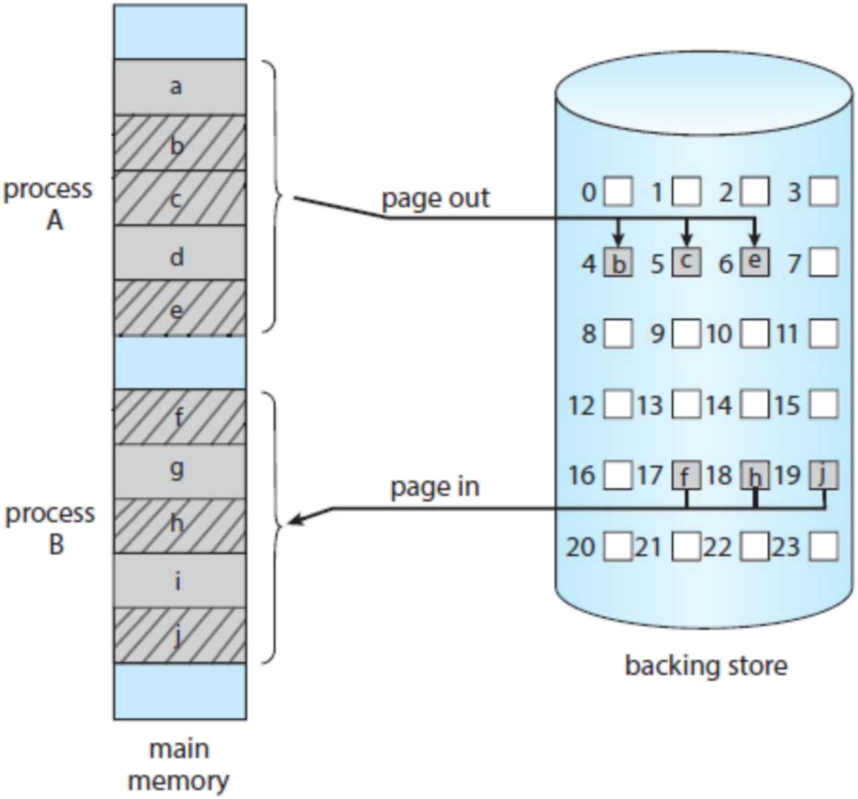
- Other constraints as well on swapping
  - **Problem**: Pending I/O
    - Solution 1: Never swap a process with pending I/O
    - Solution 2: Execute I/O operations into OS buffers (kernel space) then to I/O device
      - Known as **double buffering**
      - BUT it adds overhead



- Standard swapping not used in modern operating systems **or** Swap only when free memory extremely low

Swapping and Paging

- Most systems, including Linux and Windows, now use a variation of swapping
- Swapping Pages of a process (rather than an entire process)
- *Page in* and *Page out* steps
- Only a small number of pages will be involved in swapping
- **Paging = Swapping + Paging**



process A
a
b
c
d
e
process B
f
g
h
i
j
main memory
page out
page in
0
1
2
3
4 b
5 c
6 e
7
8
9
10
11
12
13
14
15
16
17 f
18 h
19 j
20
21
22
23
backing store