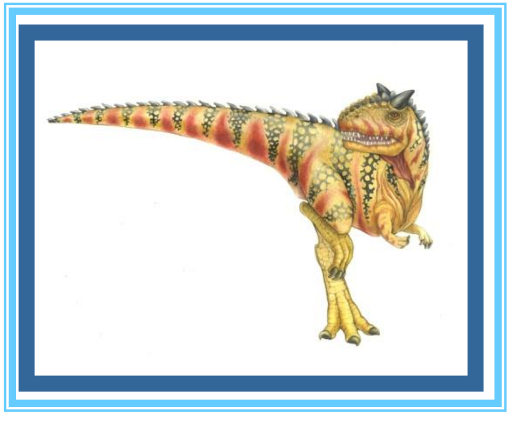
Fall 2015 - COMP3511
Review



Outline

- Monitor
- Deadlock and Banker Algorithm
- Paging and Segmentation
- Page Replacement Algorithms and Working-set Model
- File Allocation
- Disk Scheduling

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

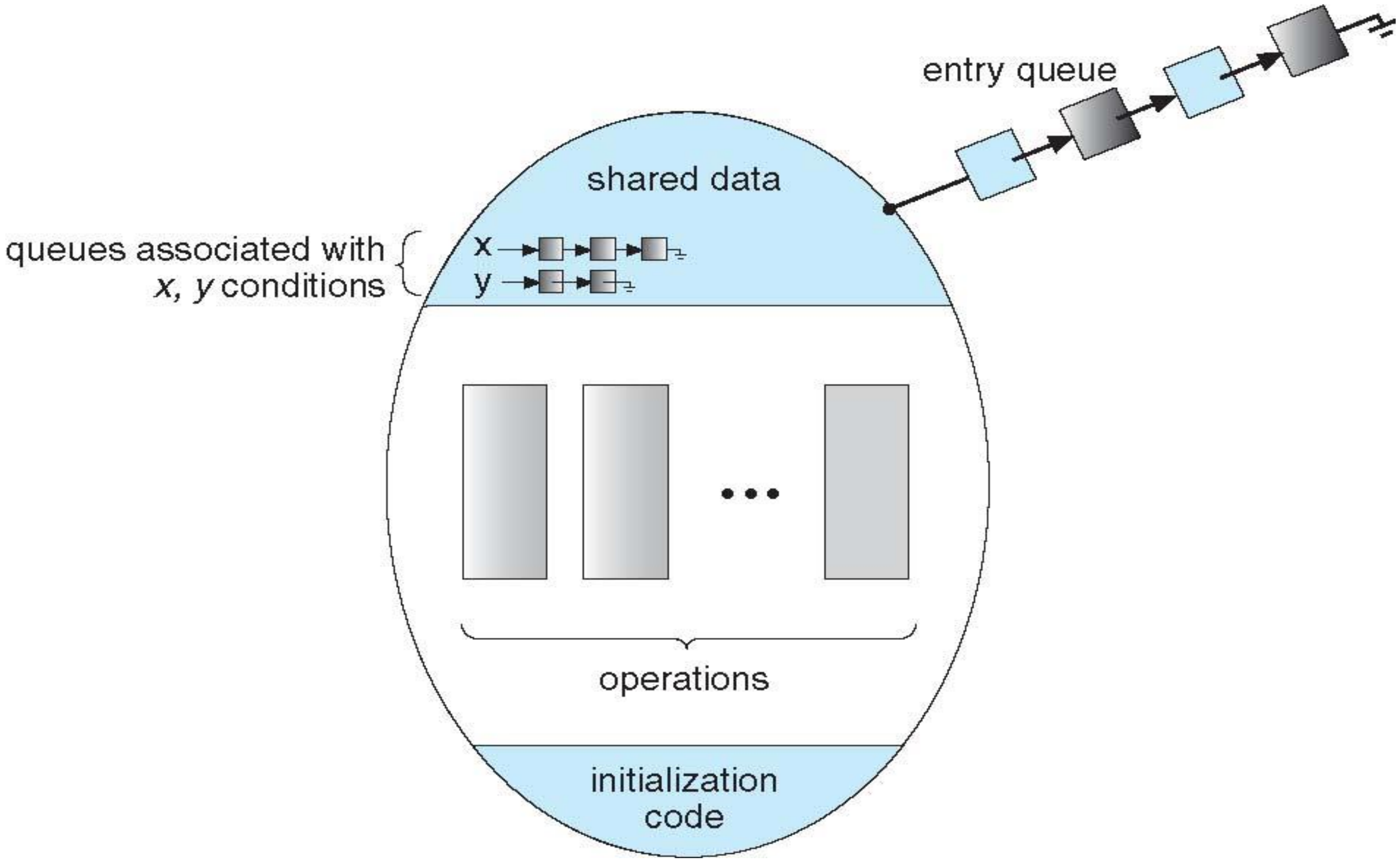
```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (…) { …. }

    procedure Pn (…) {……}

    Initialization code (…) { … }
    }
}
```

Condition Variables

- condition x, y;

- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended until x.signal ()
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()
    - If no x.wait () on the variable, then it has no effect on the variable

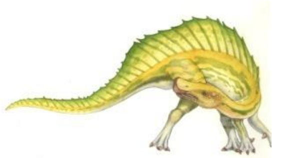
Monitor with Condition Variables



Deadlock Characterization

- Deadlock can arise if four conditions hold simultaneously

- Mutual exclusion
  - only one process at a time can use a resource.
- Hold and wait
  - a process holding at least one resource is waiting to acquire additional resources held by other processes.
- No preemption
  - a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- Circular wait
  - there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.



Resource-Allocation Graph

- A set of vertices *V* and a set of edges *E*.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows: request, use, release
- Request edge – directed edge $P_i \rightarrow R_j$
- Assignment edge – directed edge $R_j \rightarrow P_i$

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in safe state if there exists a safe sequence of all processes:
  - Sequence $<P_1, P_2, \ldots, P_n>$ is safe if for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j<i$
    - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
    - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
    - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Banker's Algorithm

- Each resource can have multiple instances.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.

Let n = number of processes, and m = number of resources types.

- *Available:* Vector of length m. If available [j] = k, there are k instances of resource type R_j available.
- *Max:* n x m matrix. If *Max* [i,j] = k, then process P_i may request at most k instances of resource type R_j.
- *Allocation:* n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j.
- *Need:* n x m matrix. If *Need*[i,j] = k, then P_i may need k more instances of R_j to complete its task

$$Need\,[i,j] = Max[i,j] - Allocation\,[i,j].$$

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

   Work = Available

   Finish [*i*] = *false* for *i* - 1,3, …, *n.*

2. Find and *i* such that both:

   (a) *Finish* [*i*] = *false*

   (b) $Need_i \leq Work$

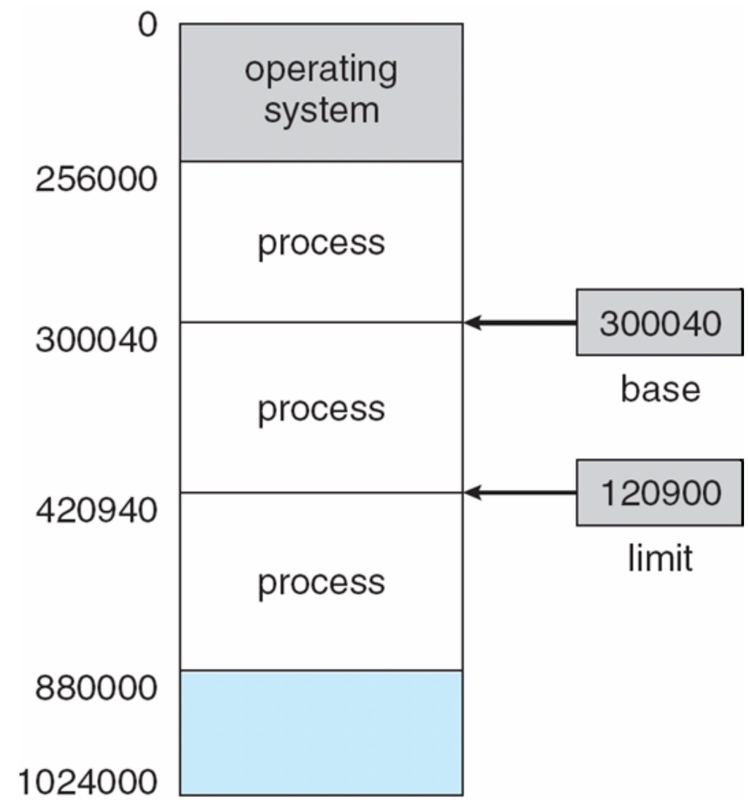
   If no such *i* exists, go to step 4

3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state

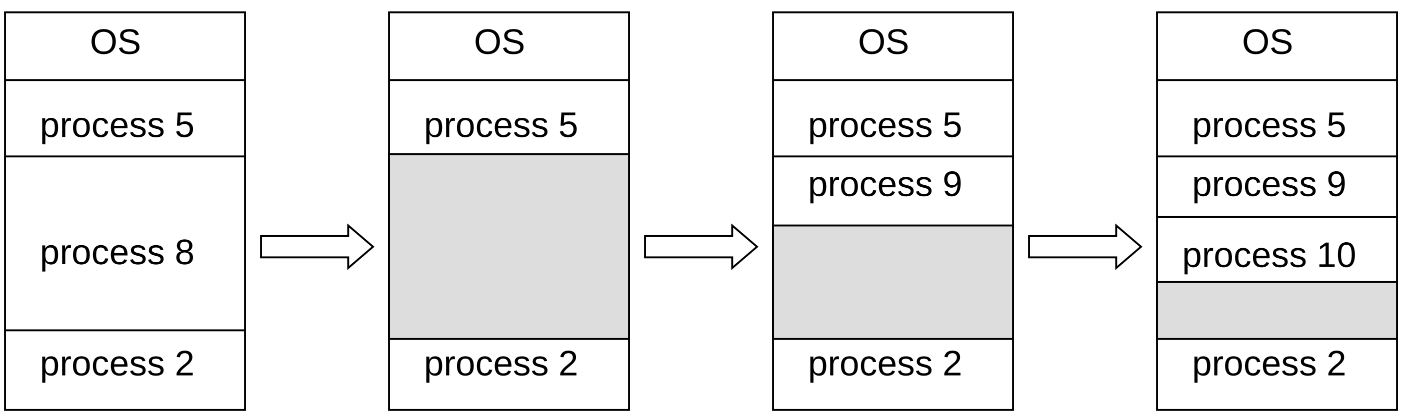
Base and Limit Registers

- Two special registers, **base** and **limit** are used to prevent user from straying outside the designated area
- During context switch, OS loads new base and limit register from PCB
- User is NOT allowed to change the base and limit registers (privileged instructions)



Contiguous Memory Allocation

- Each process is contained in a single contiguous section of memory
  - 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
  - Operating system maintains information about:
    a) allocated partitions   b) free partitions (hole)

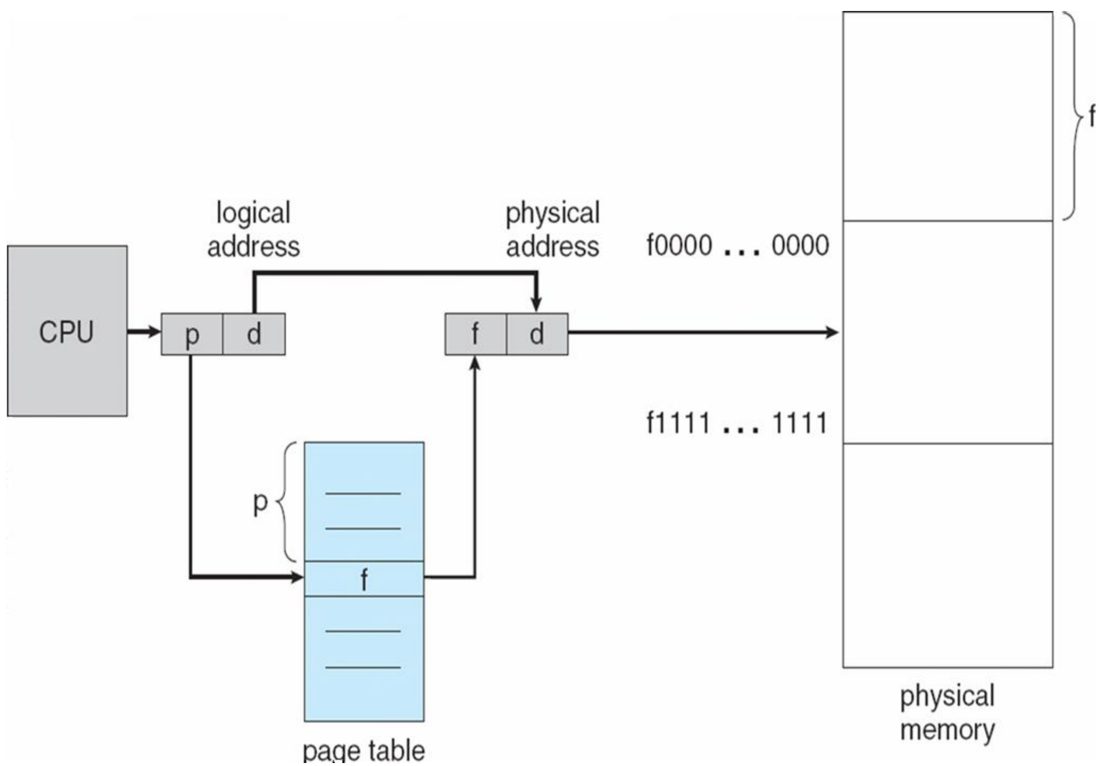


Paging

- Physical address space of a process can be *noncontiguous*
  - Divide physical memory into fixed-sized blocks called **frames,**
  - Divide logical memory into blocks of same size called **pages**.
  - Keep track of all free frames
- Set up a page table to translate logical to physical addresses

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



Page Table Implementation

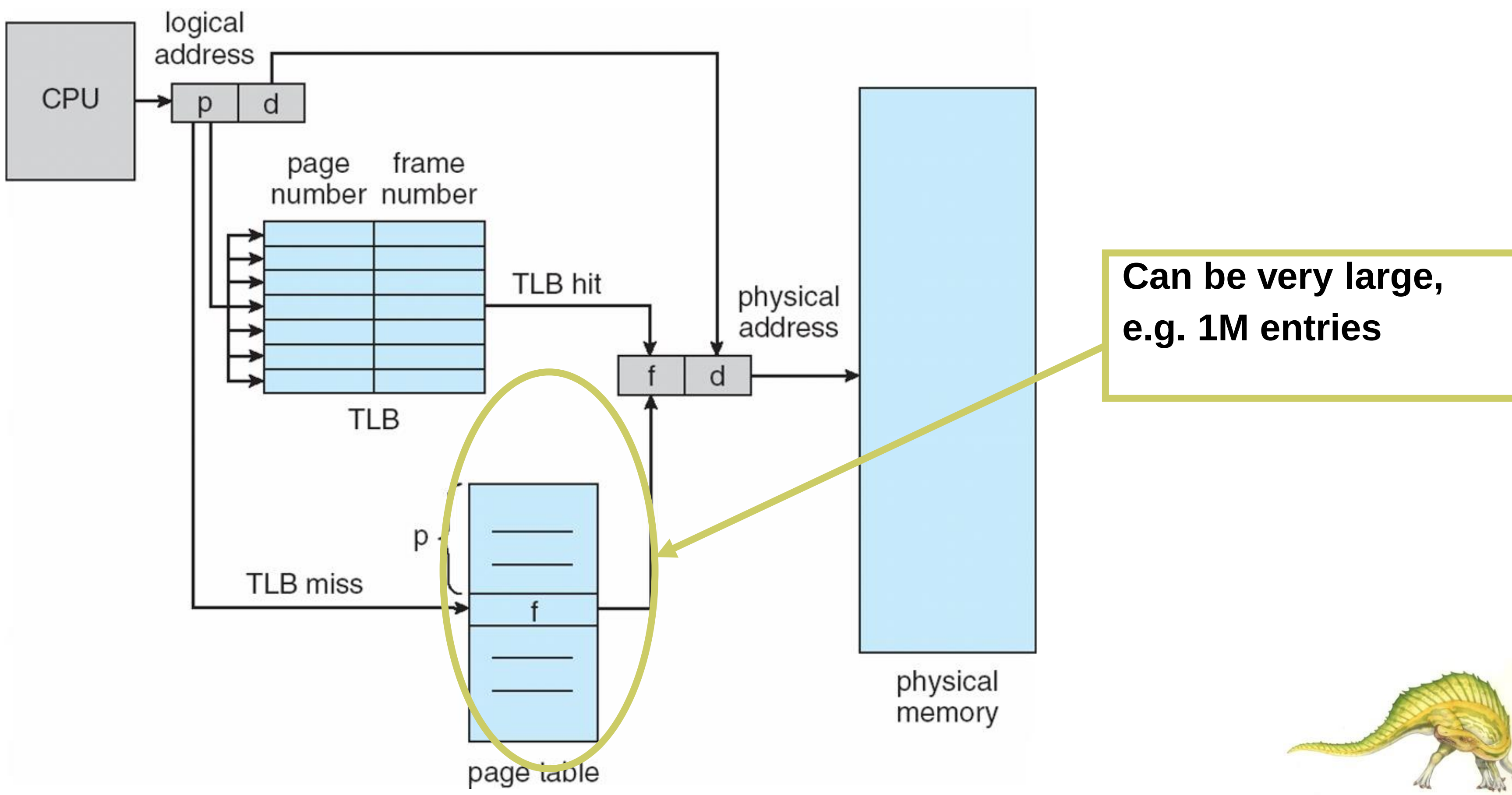
- Implementation of Page Table
  - Page table is kept in main memory
  - *Page-table base register* (PTBR) points to the page table
  - *Page-table length register* (PRLR) indicates size of the page table
  - In this scheme every data/instruction access requires two memory accesses.
    - One for the page table and one for the data/instruction

TLB

- The two memory access problem can be solved by using TLB (translation look-aside buffer)
  - a special, small, fast-lookup hardware cache
  - each entry in the TLB consists of a key (or tag) and a value
  - page number is presented to the TLB, if found, its frame number is immediately available to access memory
  - fast but expensive

Paging Hardware With TLB

- The two memory access problem can be solved by using TLB (translation look-aside buffer)



TLB miss and Hit ratio

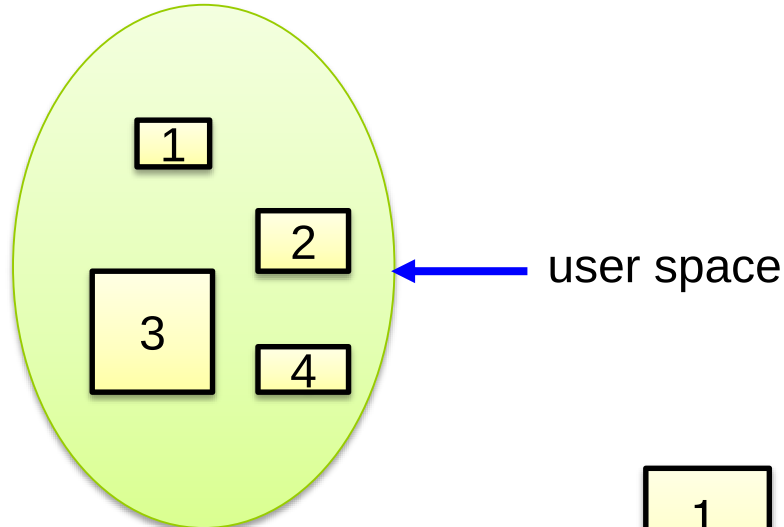
- **TLB miss**:
  - If the page number is not in the TLB, a memory reference to the page table must be made
- **Hit ratio**:
  - percentage of times that a page number is found in the TLB.
- For example:
  - Assume TLB search takes 20ns; memory access takes 100ns
  - TLB hit → 1 memory access; TLB miss → 2 memory accesses

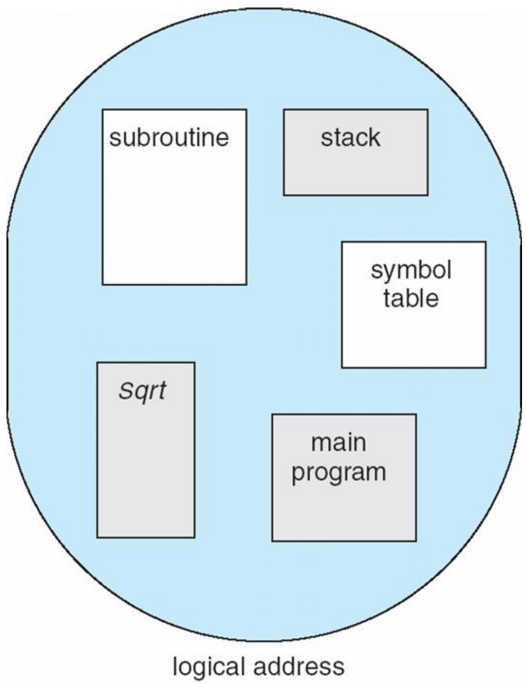
Effective Access Time (EAT)

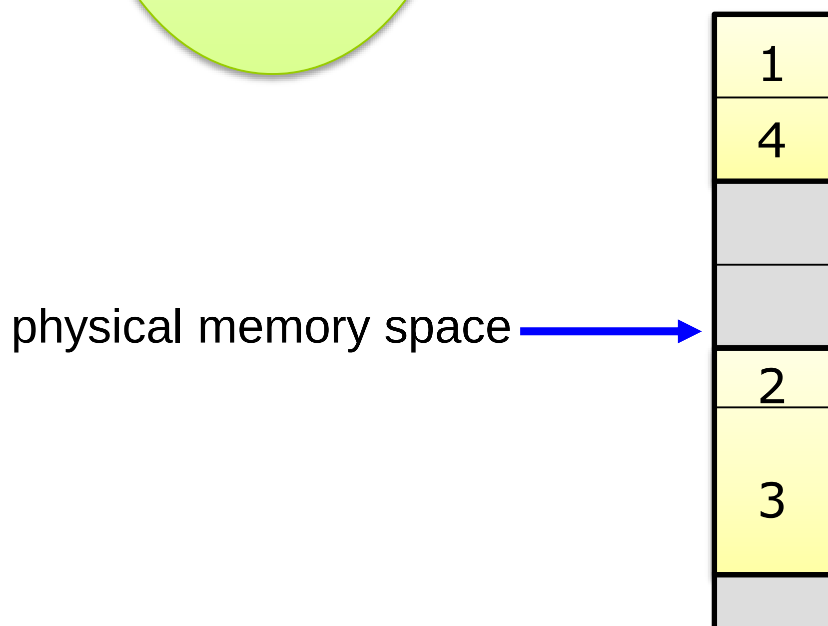
- If Hit ratio = 80%
  - EAT = (20 + 100) * 0.8 + (20 + 200) * 0.2 = 140ns
- If Hit ratio = 98%
  - EAT = (20 + 100) * 0.98 + (20 + 200) * 0.02 = 122ns

Segmentation

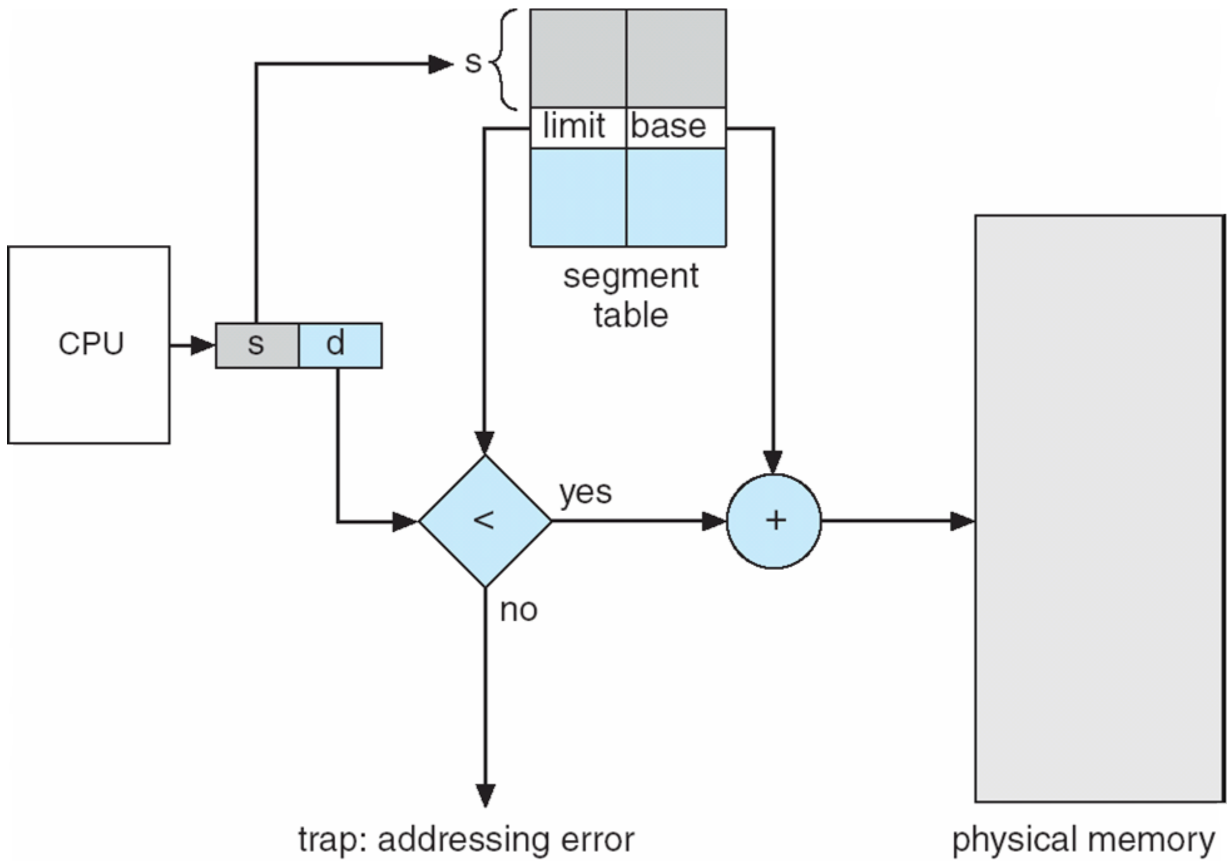
- Memory-management scheme that supports user view of memory
- A program is a collection of segments of different sizes
- A segment is a logical unit



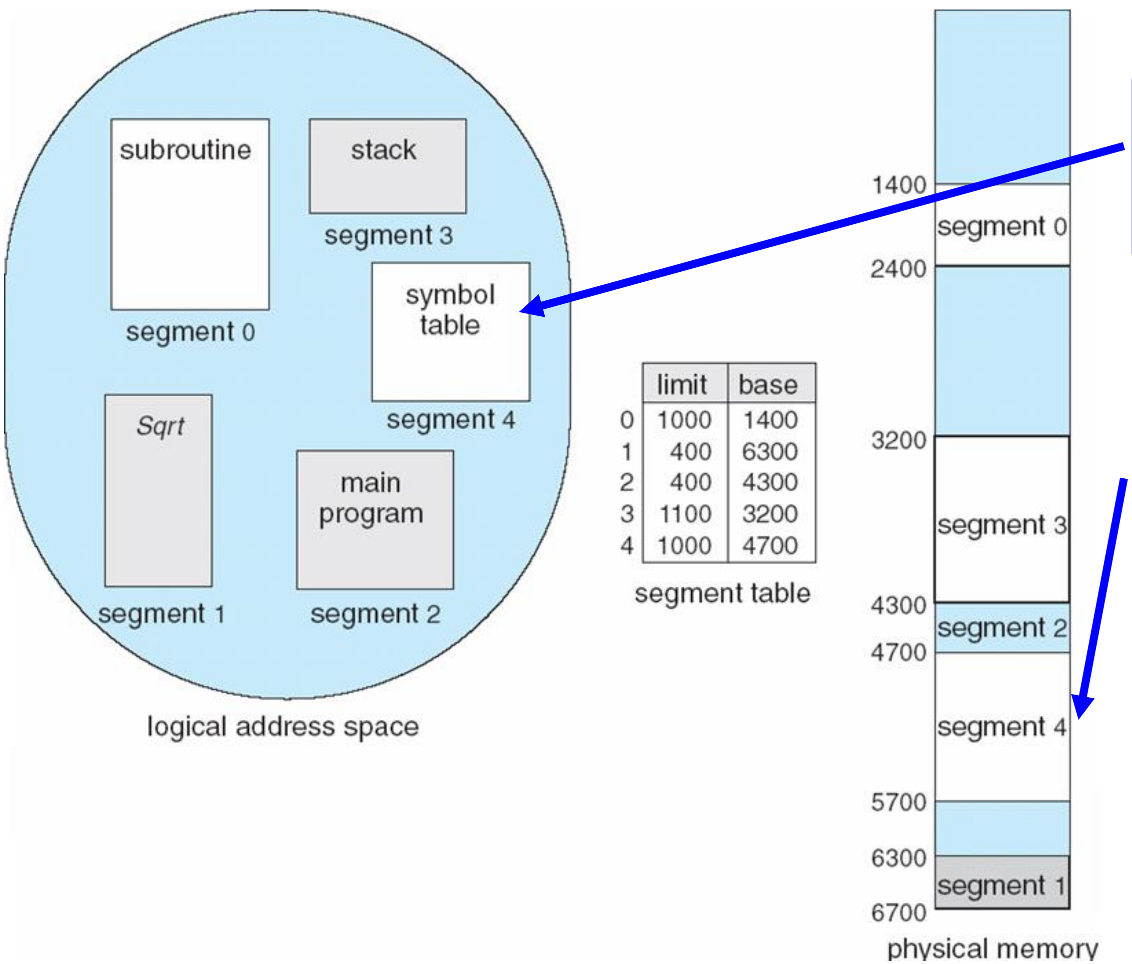




Address Translation



Example of Segmentation



Logical view: multiple separate segments

Each segment is allocated with a contiguous memory

External fragmentation

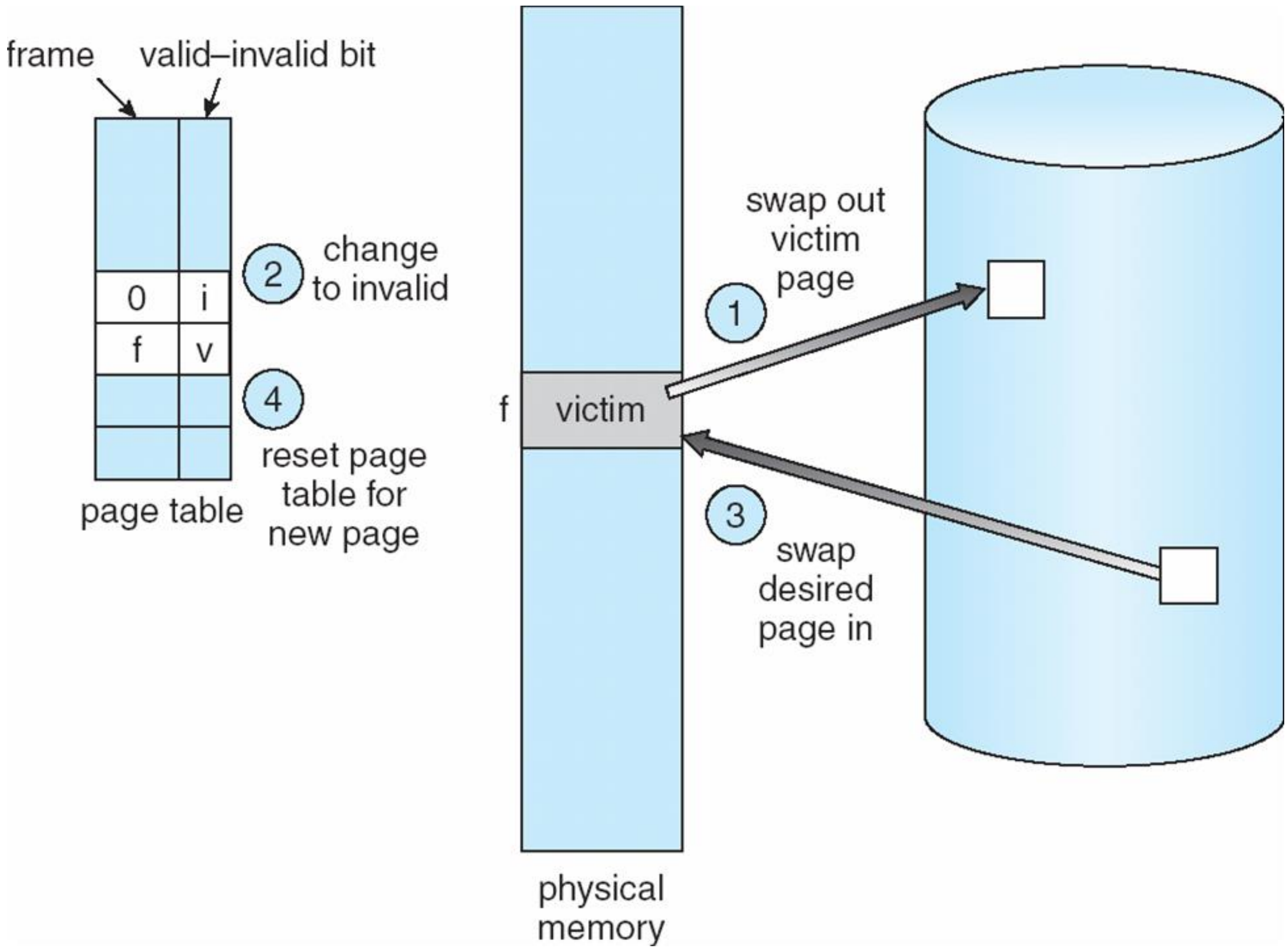
Motivation of virtual memory

- Should an entire process be in memory before it can execute?
  - In fact, real programs show that, in many cases, the entire program is not needed
  - Even in those cases where the entire program is needed, it may not all be needed at the same time
  - *More programs could run concurrently, increasing CPU utilization and throughput*
  - *Less I/O would be needed to load or swap each user program into memory, so each user program would run faster*
  - *Allow processes to share files easily and to implement shared memory*

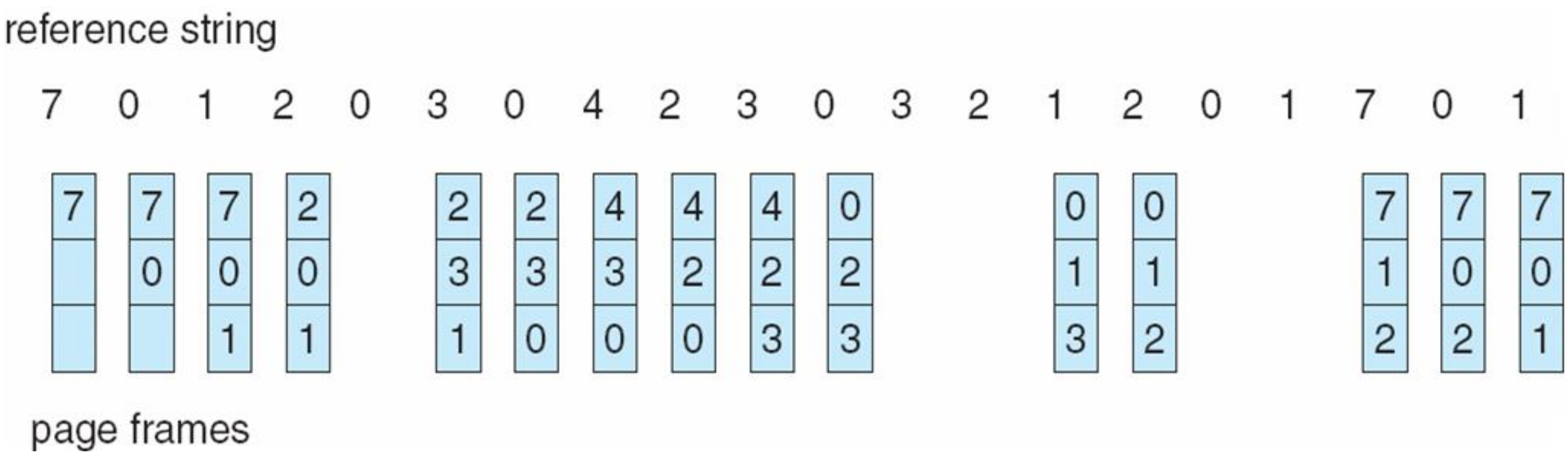
Page Replacement

- If there is no free frame
- Page replacement – find some page in memory, but not really in use, swap it out
  - Replacement algorithm
  - Performance – want an algorithm which will result in minimum number of page faults
  - Same page may be brought into and out of memory several times

Page Replacement



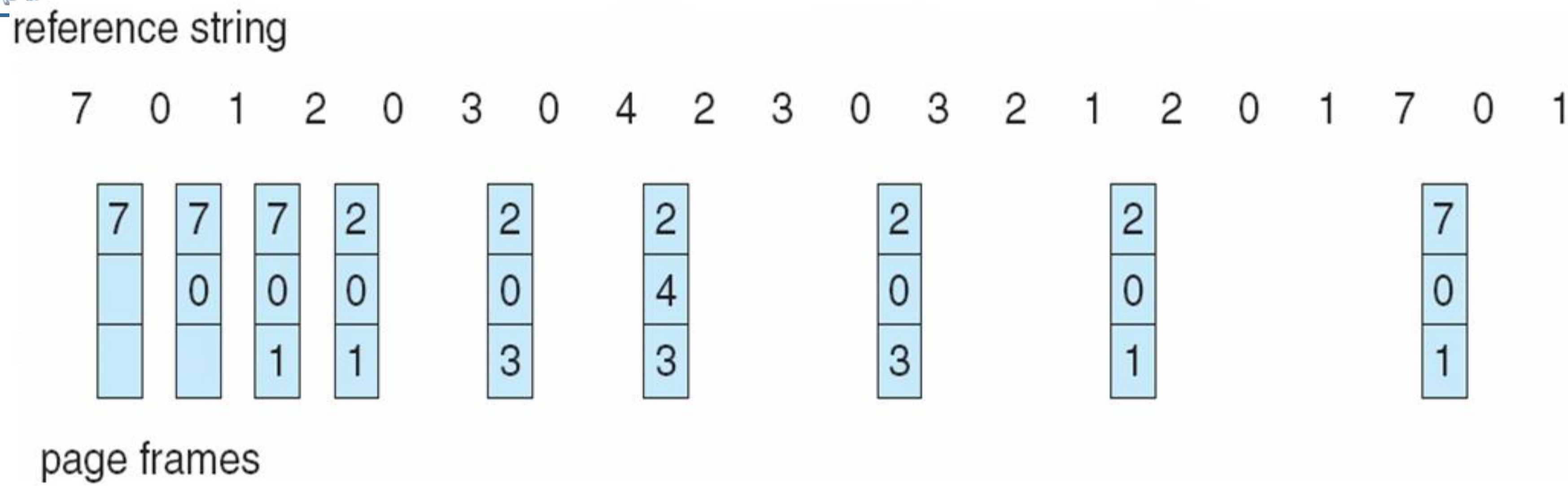
FIFO Page Replacement



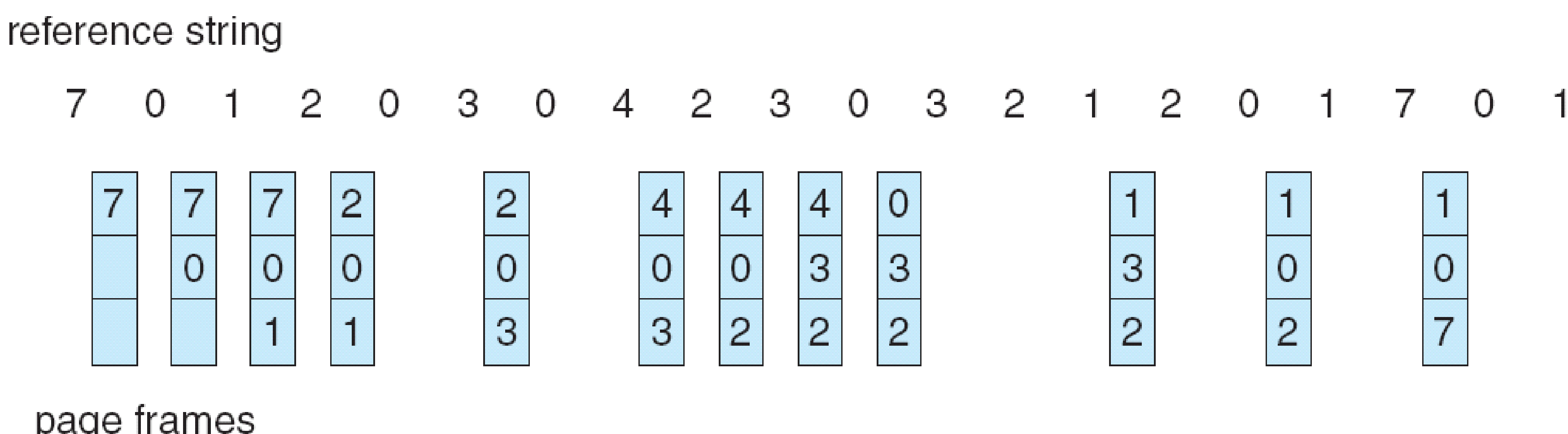
Algorithms for approximating optimal page replacement

- **LRU** (Least Recently Used) algorithm
  - Use the recent past as an approximation of the near future
    - Replace the page that has not been used for the longest period of time
  - Considered to be good, but how to implement
    - Few computer systems provide sufficient hardware support for true LRU
    - LRU-approximation: Reference bits, Second chance

- Optimal page replacement (9 page faults)



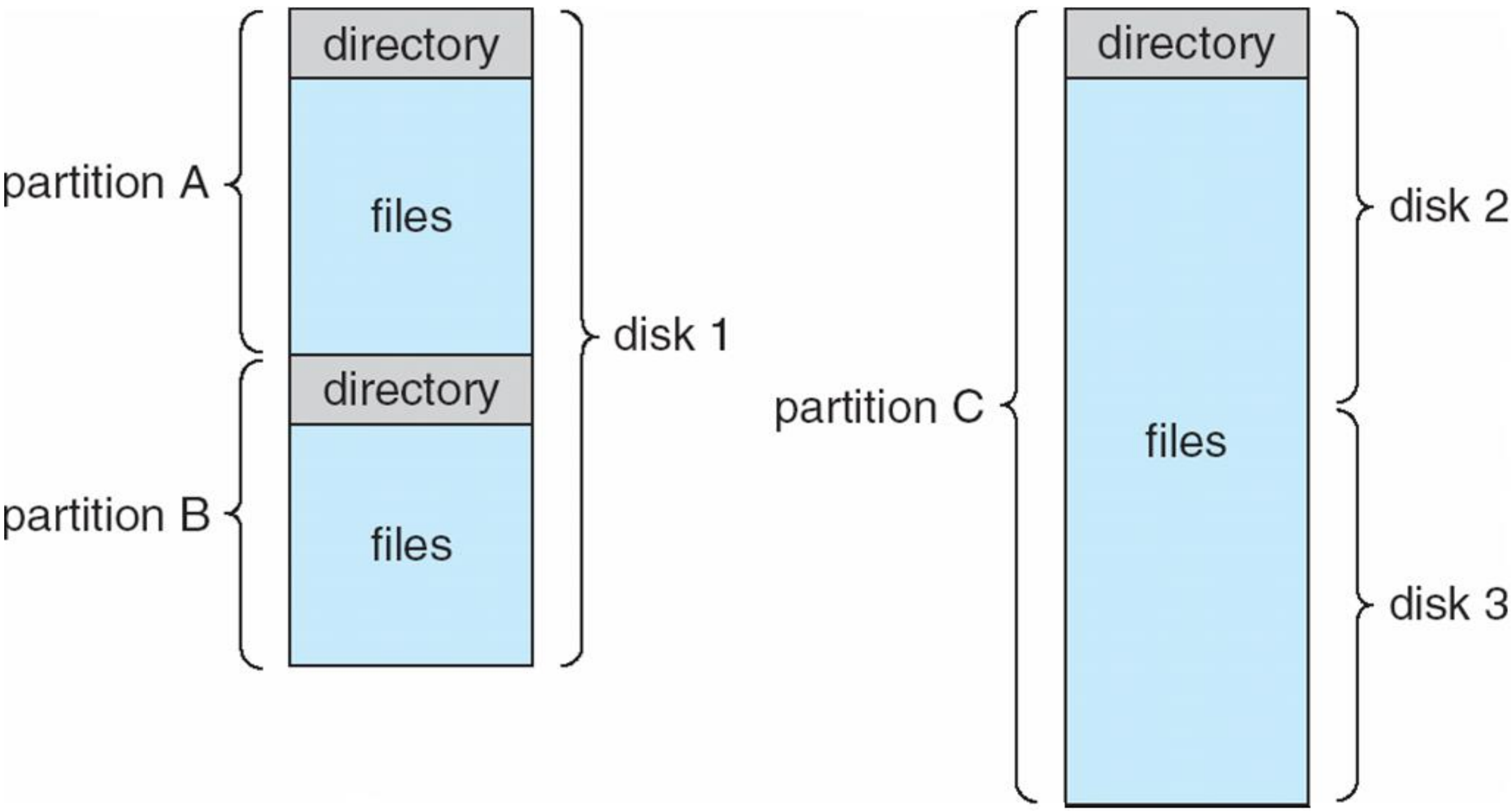
- LRU page replacement (12 page faults)



Working-Set Model

- Working-Set model is based on the locality
- $\Delta \equiv$ 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
- $D = \Sigma\, WSS_i \equiv$ total demand for frames (by all processes)
  - if $D > m \Rightarrow$ Thrashing (m is the available frames)
  - Policy if D > m, then suspend one of the processes; the process pages are swapped out, and its frames are re-allocated to other processes. The suspended process can be re-started later

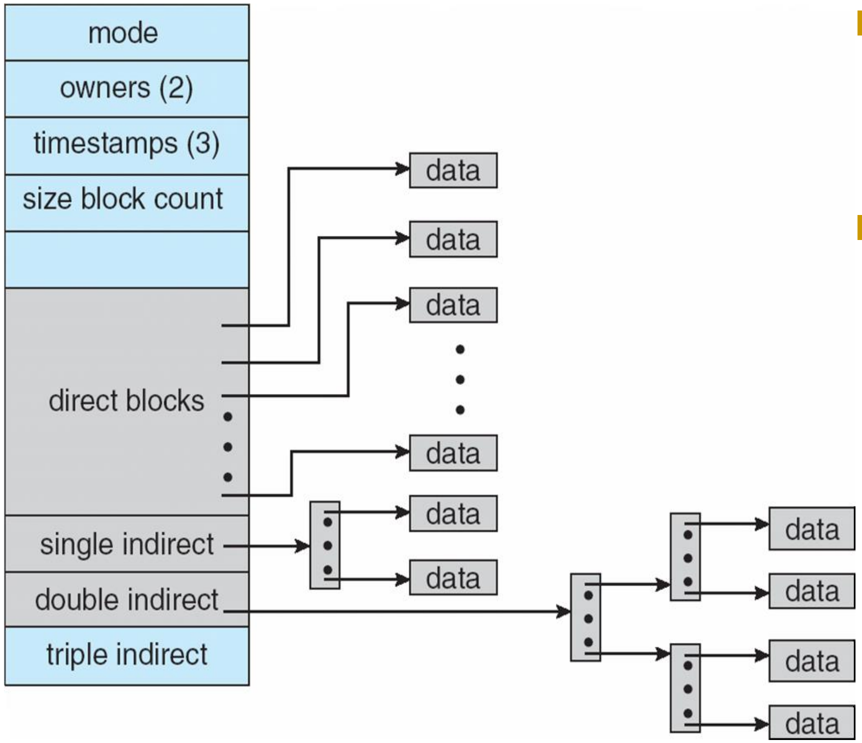
A Typical File-system Organization



Allocation Methods

- An allocation method refers to how disk blocks are allocated for files – Objectives:
  - Maximize sequential performance
  - Easy random access to file
  - Easy management of file (growth, truncation, and etc)
- Contiguous allocation
- Linked allocation
- Indexed allocation

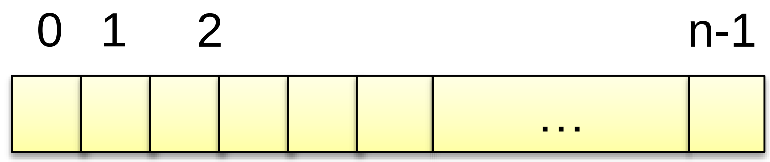
Combined Scheme: UNIX (4K bytes per block)



- Multi-level index file, key idea:
  - Efficient for small files, still allow large files
- File header format are:
  - First 10 pointers are to data blocks
  - Pointer 11 points to “indirect block” containing 256 block pointers
  - Pointer 12 points to “doubly indirect block” containing 256 indirect block pointers for total of 64K blocks
  - Pointer 13 points to a triply indirect block (16M blocks)
- Pointers get filled in dynamically

Free-Space Management

- Bit vector (*n* blocks)



$$\text{bit}[i] = \begin{cases} 1 \Rightarrow \text{block}[i] \text{ free} \\ 0 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

- Linked list (free list) (previous block contains a pointer to the next free block)
  - Cannot get contiguous space easily
  - No waste of space
- Grouping (stores the addresses of n free blocks in the first free block)
- Counting
  - Several contiguous blocks may be allocated and freed simultaneously <first free block, number of free contiguous blocks>

Disk Scheduling

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth.
- Access time has two major components
  - **Seek time** is the time for the disk are to move the heads to the cylinder (tracks) containing the desired sector.
  - **Rotational latency** is the additional time waiting for the disk to rotate the desired sector to the disk head.
- Minimize seek time
- Seek time $\approx$ seek distance
- The disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

Disk Scheduling

- When a process needs I/O to or from a disk, it issues a system call to the OS containing the following pieces of information
  - Whether the operation is input or output
  - What the disk address for the transfer is
  - What memory address for the transfer is
  - What the number of sectors to be transferred is
- Under multiprogramming system with many processes, the request may be placed in a disk queue waiting unless the desired disk drive and the controller are available
- The question is, when one request is completed, the OS needs to choose which pending requests to service next? How does the OS make this choice?
- We need *disk scheduling algorithms*
  - FCFS, SSTF, SCAN, LOOK, C-SCAN, C-LOOK