Operating Systems
(1DT020 & 1TT802)

Lecture 10
Memory Management:
Demand paging & page replacement

File system: Interface

May 07, 2008

Léon Mugwaneza

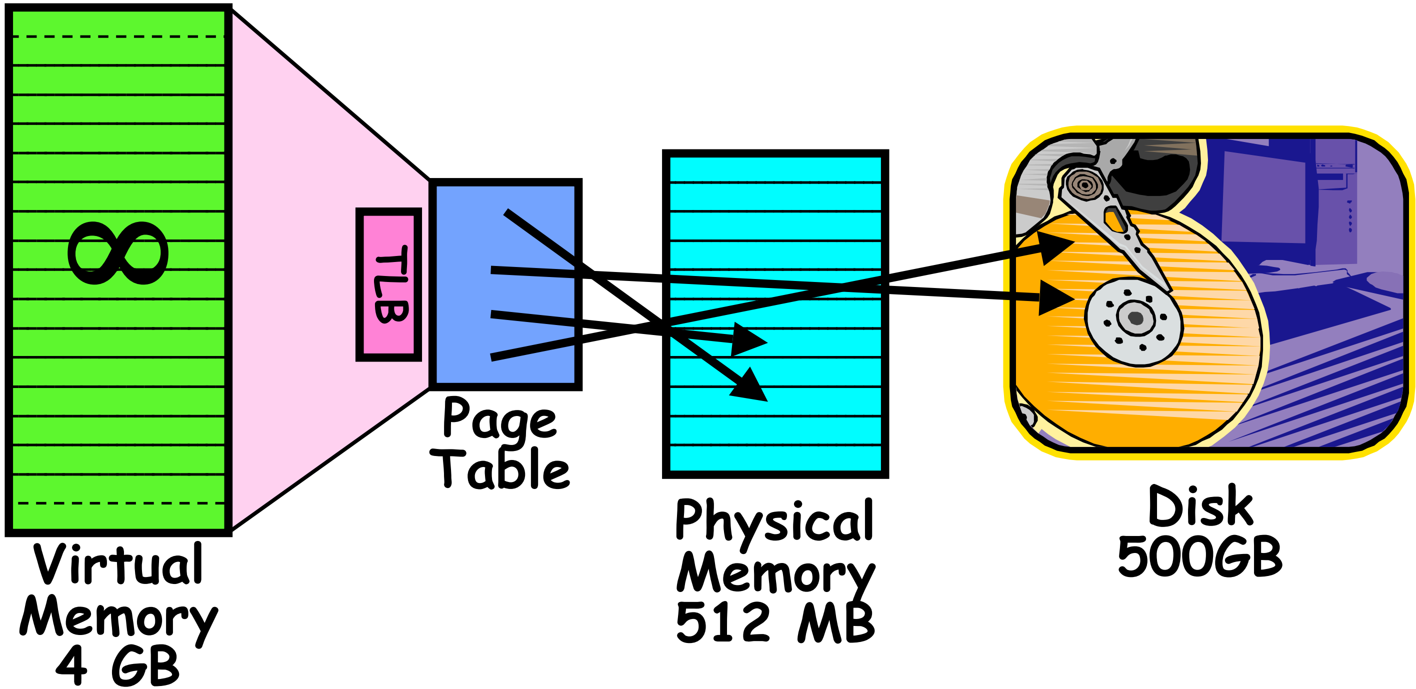
http://www.it.uu.se/edu/course/homepage/os/vt08

Goals for Today

- **Page replacement policies**
- **Page frame allocation**
- **File system interface**

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne, others from Kubiatowicz - CS162 ©UCB Fall 2007 (University of California at Berkeley)

Review: Demand paging and Illusion of "Infinite Memory"



- **Disk is larger than physical memory ⇒**
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory
    - More programs fit into memory, allowing more concurrency
- **Principle: Transparent Level of Indirection (page table)**
  - Supports flexible placement of physical data
    - Data could be on disk or somewhere across network
  - Variable location of data transparent to user program
    - Performance issue, not correctness issue

Review: Demand Paging Mechanisms

- **PTE helps us implement demand paging**
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ Page not in memory; use info in PTE to find it on disk when necessary
- **Suppose user references page with invalid PTE?**
  - Memory Management Unit (MMU) traps to OS
    - Resulting trap is a "Page Fault"
  - What does OS do on a Page Fault?:
    - Choose an old page to replace
    - If old page modified ("D=1"), write contents back to disk
    - Change its PTE and any cached TLB to be invalid
    - Load new page into memory from disk
    - Update page table entry, invalidate TLB for new entry
    - Continue thread from original faulting location
  - TLB for new page will be loaded when thread continued!
  - While pulling pages off disk for one process, OS runs another process from ready queue
    - Suspended process sits on wait queue
- **What if an instruction has side-effects?**
  - Unwind side-effects (easy to restart) or Finish off side-effects (messy!)
  - Example 1: `mov (sp)+,10`
    - What if page fault occurs when write to stack pointer?
    - Did `sp` get incremented before or after the page fault?

Cache

Demand Paging Example

- Since Demand Paging like caching, can compute average access time! ("Effective Access Time")
  - EAT = Hit Rate x Hit Time + Miss Rate x Miss Time
- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - Suppose p = Probability of miss, 1-p = Probably of hit
  - Then, we can compute EAT as follows:

$$\begin{aligned} \text{EAT} &= (1 - p) \times 200\text{ns} + p \times 8\text{ ms} \\ &= (1 - p) \times 200\text{ns} + p \times 8{,}000{,}000\text{ns} \\ &= 200\text{ns} + p \times 7{,}999{,}800\text{ns} \end{aligned}$$

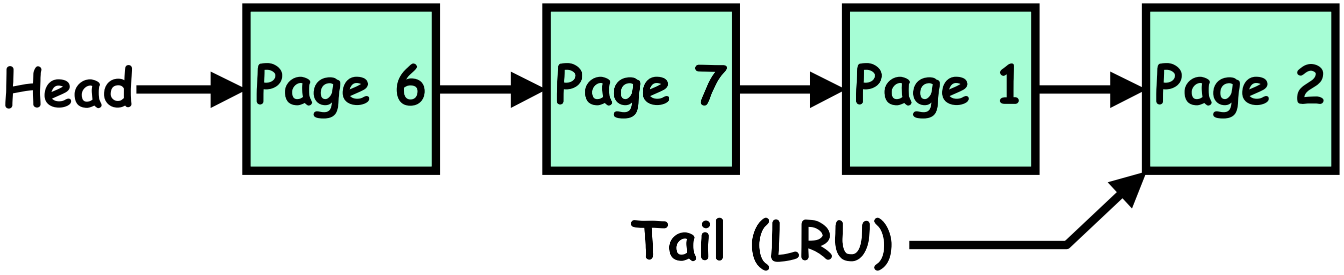
- If one access out of 1,000 causes a page fault, then EAT = 8.2 μs:
  - This is a slowdown by a factor of 40!
- What if want slowdown by less than 10%?
  - $200\text{ns} \times 1.1 > \text{EAT} \Rightarrow p < 2.5 \times 10^{-6}$
  - This is about 1 page fault in 400000!

Page Replacement Policies

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
  - Particularly important with pages
    - The cost of being wrong is high: must go to disk
    - Must keep important pages in memory, not toss them out
- FIFO (First In, First Out)
  - Throw out oldest page. Be fair – let every page live in memory for same amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages
- MIN (Minimum):
  - Replace page that won't be used for the longest time
  - Great, but can't really know future…
  - Makes good comparison case, however
- RANDOM:
  - Pick random page for every replacement
  - Typical solution for TLB's. Simple hardware
  - Pretty unpredictable – makes it hard to make real-time guarantees

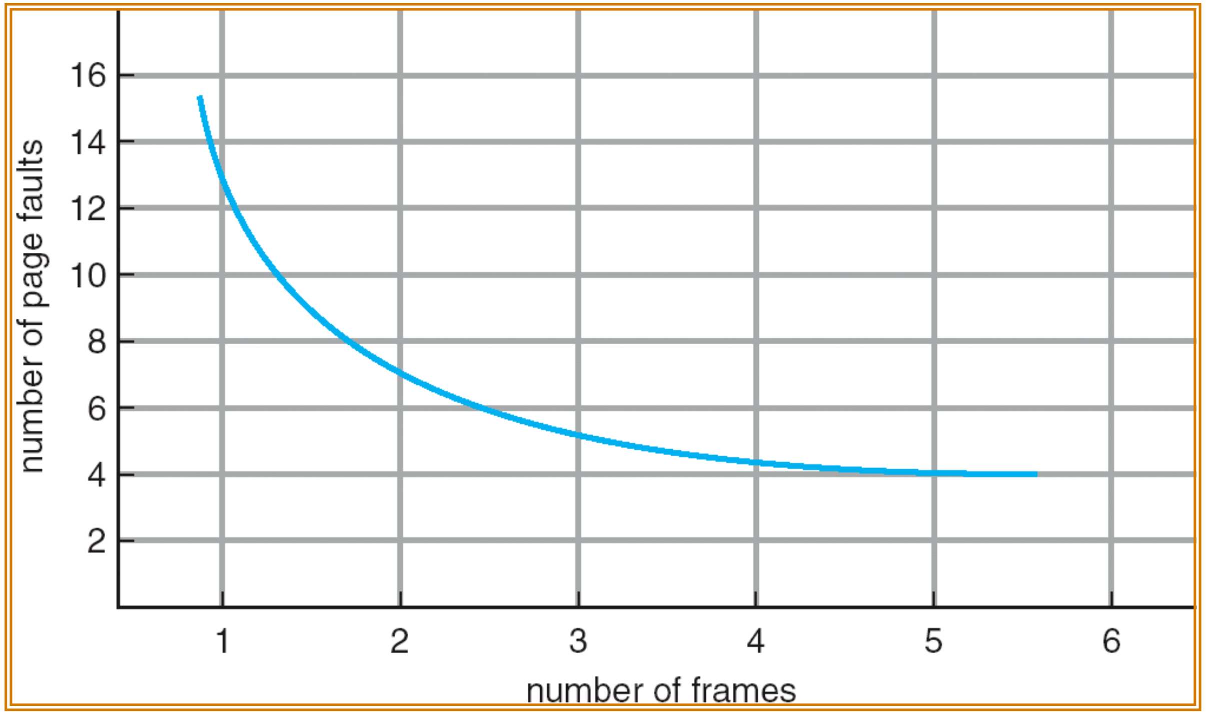
Replacement Policies (Con't)

- **LRU (Least Recently Used):**
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
- **How to implement LRU? Use a list!**



  - On each use, remove page from list and place at head
  - LRU page is at tail
- **Problems with this scheme for paging?**
  - Need to know immediately when each page used so that can change position in list…
  - Many instructions for each hardware access
- **In practice, people approximate LRU (more later)**

Graph of Page Faults Versus The Number of Frames



- **One desirable property: When you add memory the miss rate goes down**
  - **Does this always happen?**
  - **Seems like it should, right?**
- **No: BeLady’s anomaly**
  - **Certain replacement algorithms (FIFO) don’t have this obvious property!**

Adding Memory Doesn’t Always Help Fault Rate

- **Does adding memory reduce number of page faults?**
  - Yes for LRU and MIN
  - Not necessarily for FIFO! (Called Belady’s anomaly)

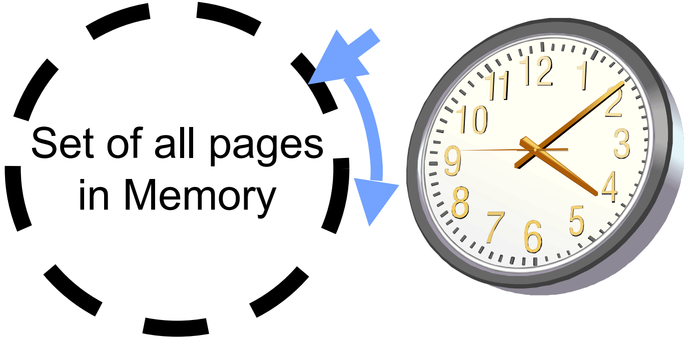
Ref: Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					
2		B			A					C		
3			C			B					D	

Ref: Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A						E				D	
2		B						A				E
3			C						B			
4				D						C		

- **After adding memory:**
  - With FIFO, number of fault increased (10 for 4 frames vs 9 for 3 frames)
  - In contrast, with LRU or MIN, set of pages in memory with X frames is a subset of set of pages in memory with X+1 frames

Implementing LRU

- Perfect:
  - Timestamp page on each reference
  - Keep list of pages ordered by time of reference
  - Too expensive to implement in reality for many reasons
- Clock Algorithm: Arrange physical pages in circle with single clock hand
  - Approximate LRU (approx to approx to MIN)
  - Replace an old page, not the oldest page

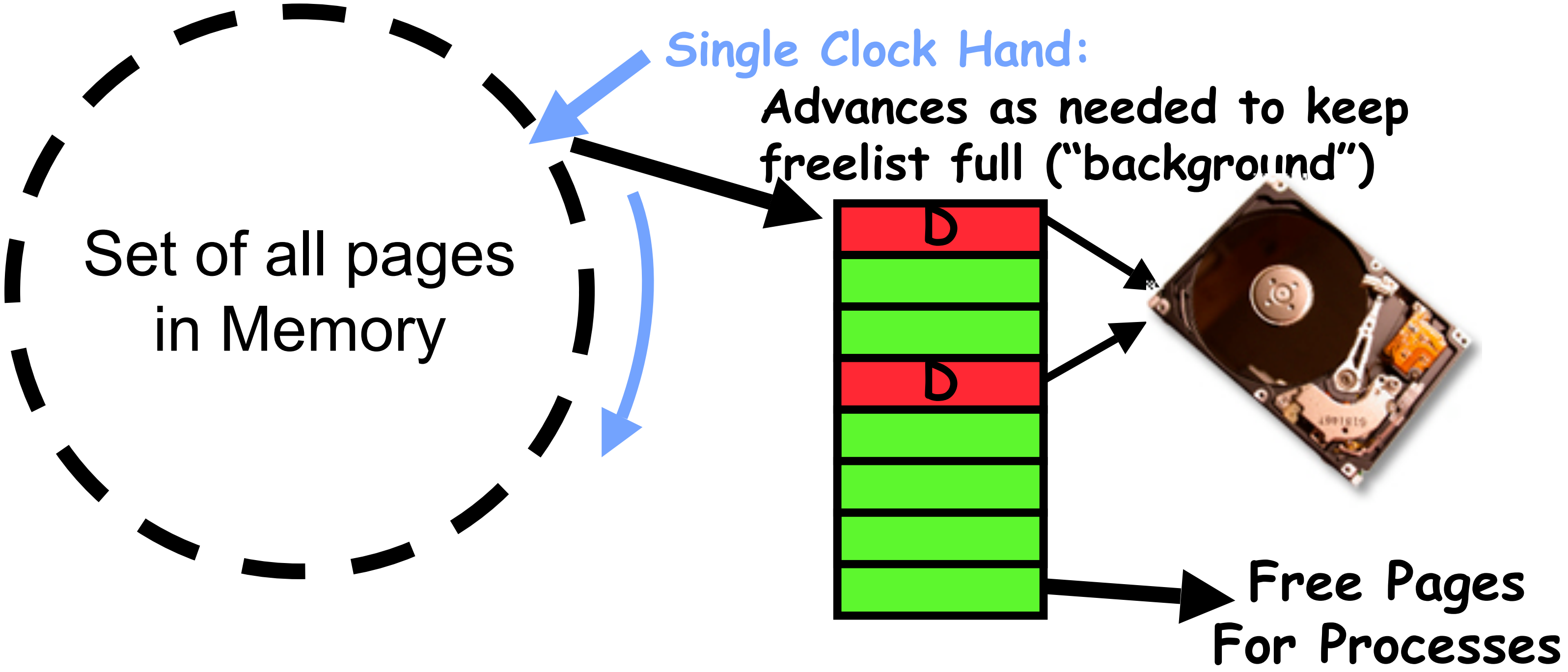


- Details:
  - Hardware "use" bit per physical page:
    - » Hardware sets use bit on each reference
    - » If use bit isn't set, means not referenced in a long time
    - » hardware sets use bit in the TLB; use bit copied back to page table when TLB entry gets replaced
  - On page fault:
    - » Advance clock hand (not real time)
    - » Check use bit: 1→used recently; clear and leave alone
      0→selected candidate for replacement
  - Will always find a page or loop forever?
    - » Even if all use bits set, will eventually loop around⇒FIFO
- One way to view clock algorithm:
  - Crude partitioning of pages into two groups: young and old
  - Why not partition into more than 2 groups?

N^th Chance version of Clock Algorithm

- **N^th chance algorithm:** Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    - » 1⇒clear use and also clear counter (used in last sweep)
    - » 0⇒increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced
- How do we pick N?
  - Why pick large N? Better approx to LRU
    - » If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    - » Otherwise might have to look a long way to find free page
- What about dirty pages?
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - Common approach:
    - » Clean pages, use N=1
    - » Dirty pages, use N=2 (and write back to disk when N=1)

Free List



- Keep set of free pages ready for use in demand paging
  - Free list filled in background by Clock algorithm or other technique ("Pageout demon")
  - Dirty pages start copying back to disk when enter list
  - If page needed before reused, just return to active set
- Advantage: Faster for page fault
  - Can always use page (or pages) immediately on fault

Allocation of Page Frames (Memory Pages)

- How do we allocate memory among different processes?
  - Does every process get the same fraction of memory? Different fractions?
  - Should we completely swap some processes out of memory?
- Each process needs *minimum* number of pages
  - Want to make sure that all processes that are loaded into memory can make forward progress
  - Example: IBM 370 – 6 pages to handle SS MOVE instruction:
    - » instruction is 6 bytes, might span 2 pages
    - » 2 pages to handle *from*
    - » 2 pages to handle *to*
- Possible Replacement Scopes:
  - Global replacement – process selects replacement frame from set of all frames; one process can take a frame from another
  - Local replacement – each process selects from only its own set of allocated frames

Fixed/Priority Allocation

- **Equal allocation** (Fixed Scheme):
  - Every process gets same amount of memory
  - Example: 100 frames, 5 processes⇒process gets 20 frames
- **Proportional allocation** (Fixed Scheme)
  - Allocate according to the size of process
  - Computation proceeds as follows:

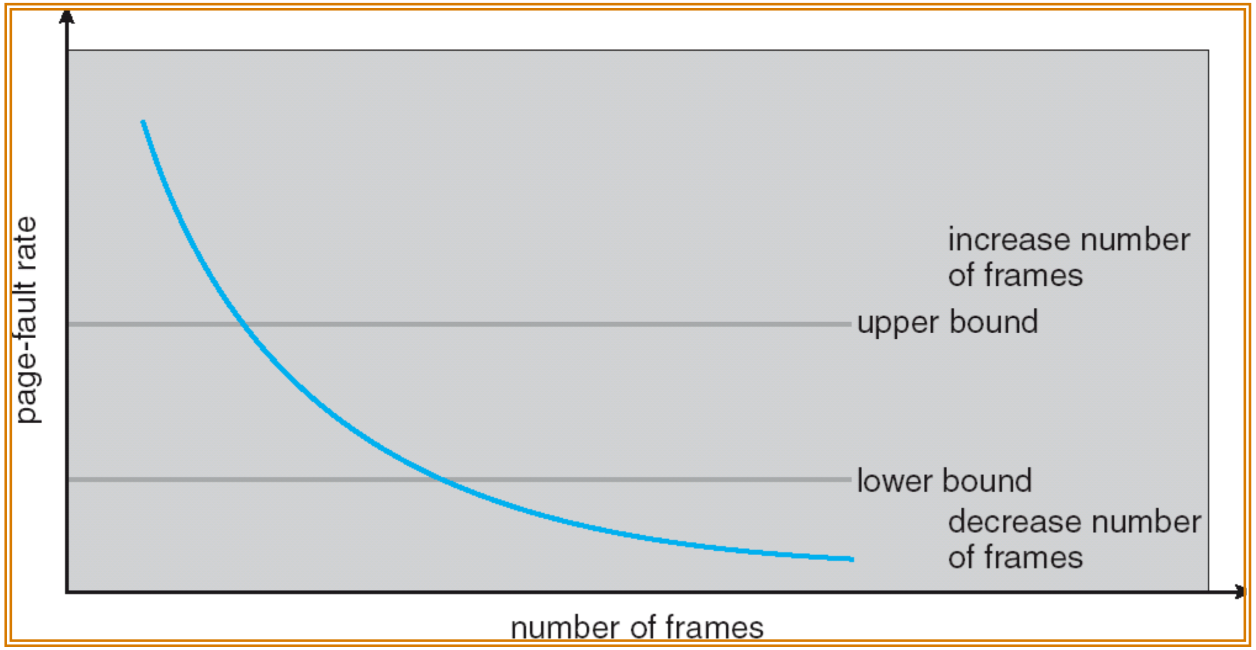
    s_i = size of process p_i and $S = \Sigma s_i$

    m = total number of frames

    a_i = allocation for p_i = $\dfrac{s_i}{S} \times m$
- **Priority Allocation:**
  - Proportional scheme using priorities rather than size
    - » Same type of computation as previous scheme
  - Possible behavior: If process p_i generates a page fault, select for replacement a frame from a process with lower priority number
- Perhaps we should use an adaptive scheme instead???
  - What if some application just needs more memory?

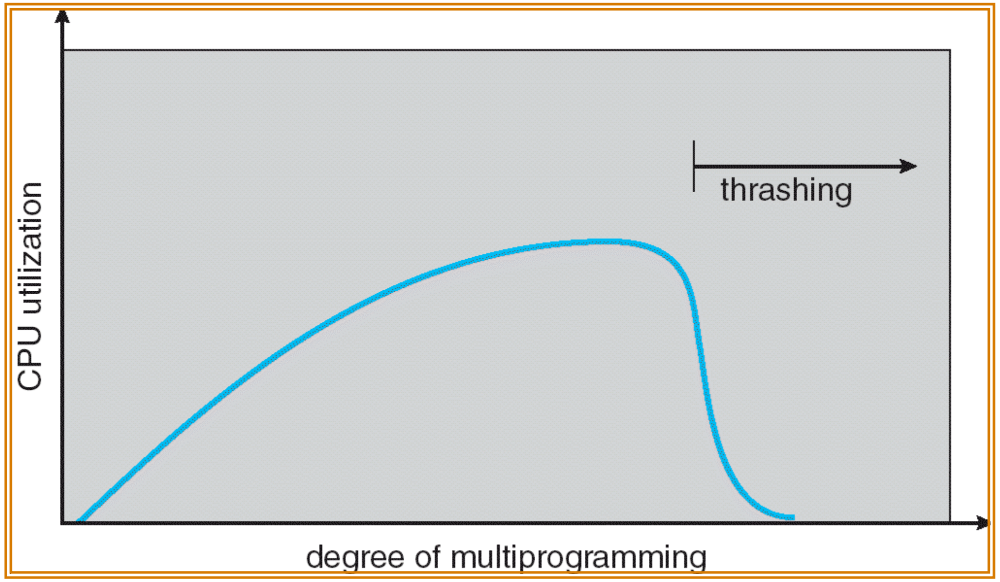
Page-Fault Frequency Allocation

- Can we reduce Capacity misses by dynamically changing the number of pages/application?



- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
- Question: What if we just don’t have enough memory?

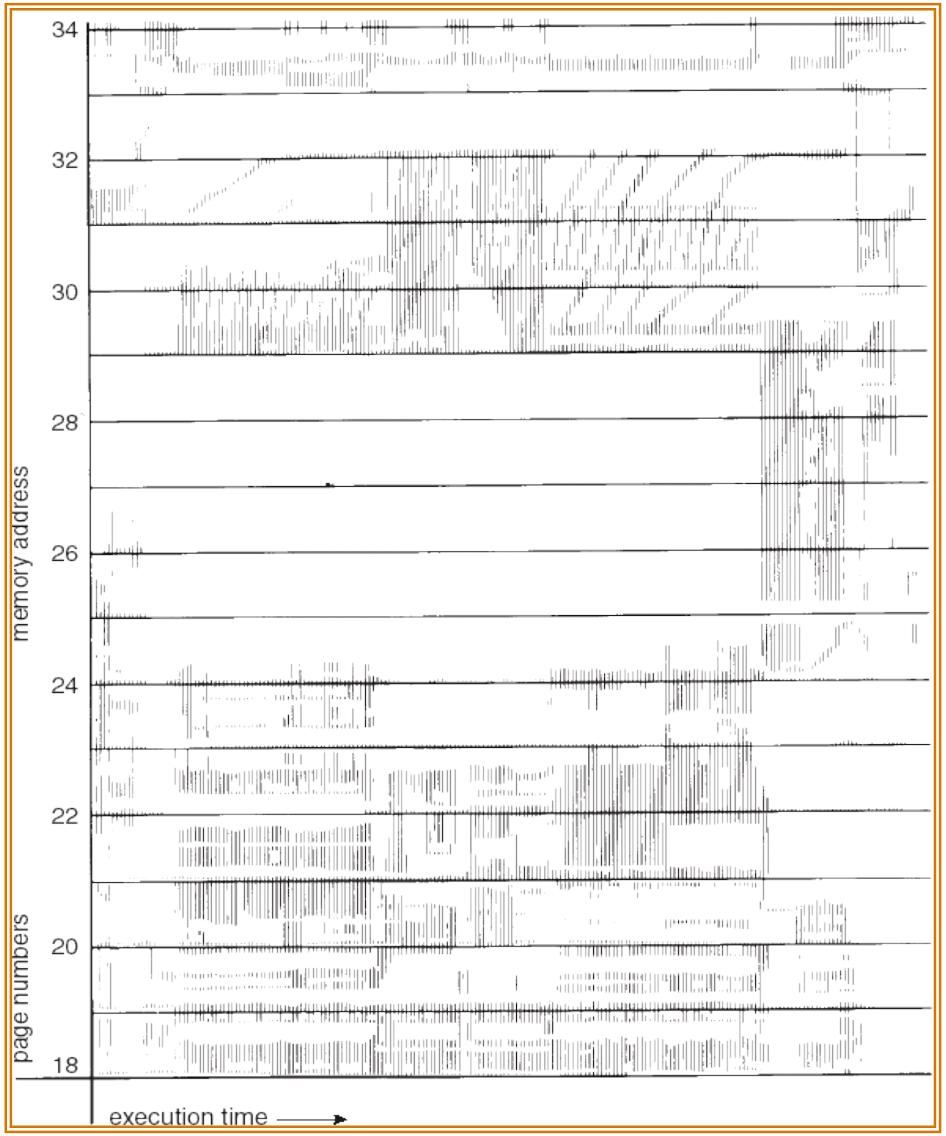
Thrashing



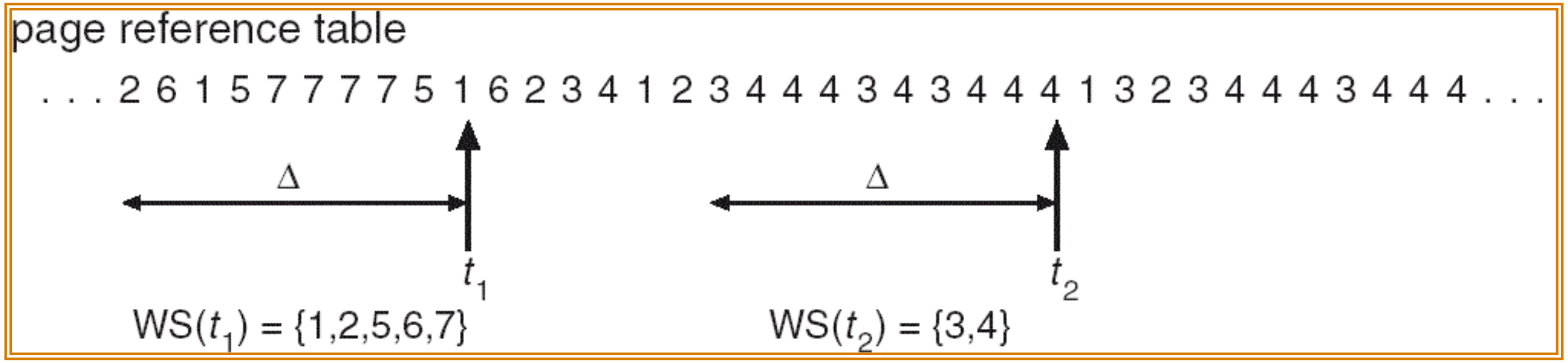
- **If a process does not have "enough" pages, the page-fault rate is very high. This leads to:**
  - **low CPU utilization**
  - **operating system spends most of its time swapping to disk**
- **Thrashing ≡ a process is busy swapping pages in and out**
- **Questions:**
  - **How do we detect Thrashing?**
  - **What is best response to Thrashing?**

Locality In A Memory-Reference Pattern

- Program Memory Access Patterns have temporal and spatial locality
  - Group of Pages accessed along a given time slice called the “Working Set”
  - Working Set defines minimum number of pages needed for process to behave well
- Not enough memory for Working Set⇒Thrashing
  - Better to swap out process?



Working-Set Model



- $\Delta \equiv$ working-set window $\equiv$ fixed number of page references
  - Example: 10,000 instructions
- WS_i (working set of Process P_i) = total set 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
- $D = \Sigma|WS_i| \equiv$ total demand frames
- if m is total number of frames, $D > m \Rightarrow$ Thrashing
  - Policy: if D > m, then suspend/swap out processes
  - This can improve overall system behavior by a lot!

Reducing Compulsory page faults by prepaging

- Compulsory page faults are faults that occur the first time that a page is see
  - Pages that are touched for the first time
  - Pages that are touched after process is swapped out/swapped back in
- Clustering:
  - On a page-fault, bring in multiple pages “around” the faulting page
  - Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages
- Working Set Tracking:
  - Use algorithm to try to track working set of application
  - When swapping process back in, swap in working set

Paging Summary

- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: Replace page that will be used farthest in future
  - LRU: Replace page used farthest in past
- Clock Algorithm: Approximation to LRU
  - Arrange all pages in circular list
  - Sweep through them, marking as not “in use”
  - If page not “in use” for one pass, than can replace
- Nth-chance clock algorithm: Another approx LRU
  - Give pages multiple passes of clock hand before replacing
- List of free page frames makes page fault handling faster
  - Filled in background by pageout demon
- Working Set:
  - Set of pages touched by a process recently
- Thrashing: a process is busy swapping pages in and out
  - Process will thrash if working set doesn’t fit in memory
  - Need to swap out a process

The file concept

- Collection of related information stored on a secondary storage (cf. logical secondary storage)
  - data files, program files (also, source, object, executable, ...).
- File Structure:
  - none (sequence of bytes), lines, more complex...
- Attributes:
  - name, size, last update, owner, … (try `ls -la`)
- File Operations:
  - open, close, create, read, write, delete, ...

Building a File System

- **File System:** Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.
- File System Components
  - Disk Management: collecting disk blocks into files
  - Naming: Interface to find files by name, not by blocks
  - Protection: Layers to keep data secure
  - Reliability/Durability: Keeping of files durable despite crashes, media failures, attacks, etc
- User vs. System View of a File
  - User's view:
    - » Durable Data Structures
  - System's view (system call interface):
    - » Collection of Bytes (UNIX)
    - » Doesn't matter to system what kind of data structures you want to store on disk!
  - System's view (inside OS):
    - » Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)
    - » Block size ≥ sector size; in UNIX, block size is 4KB

How do we actually access files?

- All information about a file contained in its file header
  - UNIX calls this an "inode"
    - Inodes are global resources identified by index ("inumber")
  - Once you load the header structure, all the other blocks of the file are locatable
- Question: how does the user ask for a particular file?
  - One option: user specifies an inode by a number (index).
    - Imagine: open("14553344")
  - Better option: specify by textual name
    - Have to map name→inumber
  - Another option: Icon
    - This is how Apple made its money. Graphical user interfaces. Point to a file and click.
- **Naming:** The process by which a system translates from user-visible names to system resources
  - In the case of files, need to translate from strings (textual names) or icons to inumbers/inodes
  - For global file systems, data may be spread over globe⇒need to translate from strings or icons to some combination of physical server location and inumber

Directories

- **Directory**: a relation used for naming
  - Just a table of (file name, inumber) pairs

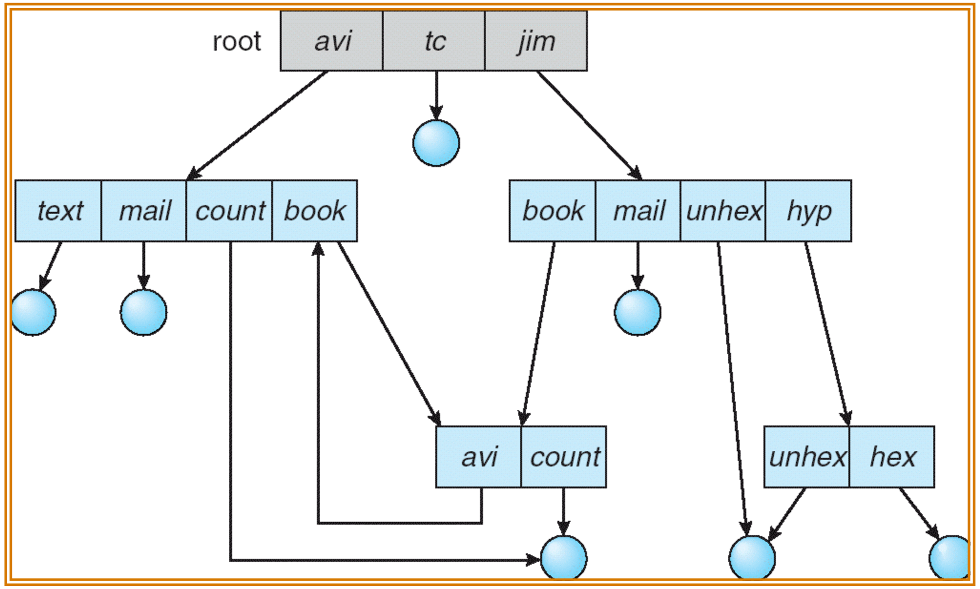
- How are directories constructed?
  - Directories often stored in files
    - Reuse of existing mechanism
    - Directory named by inode/inumber like other files
  - Needs to be quickly searchable
    - Options: Simple list or Hashtable
    - Can be cached into memory in easier form to search

- How are directories modified?
  - Originally, direct read/write of special file
  - System calls for manipulation: `mkdir`, `rmdir`
  - Ties to file creation/destruction
    - On creating a file by name, new inode grabbed and associated with new file in particular directory

Directory Organization

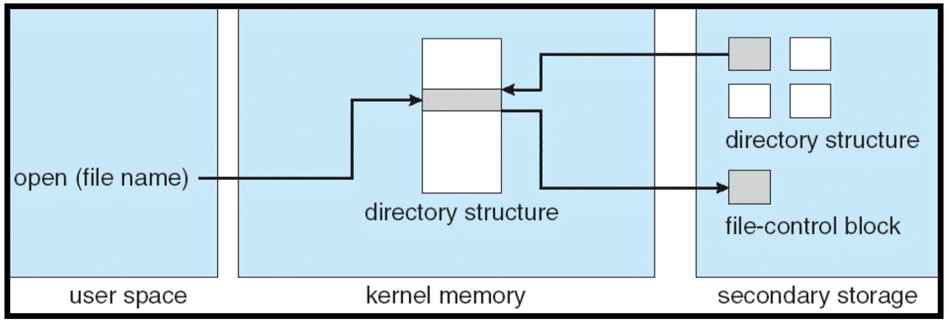
- Directories organized into a hierarchical structure
  - Seems standard, but in early 70’s it wasn’t
  - Permits much easier organization of data structures
- Entries in directory can be either files or directories
- Files named by ordered set (e.g., /programs/p/list)

Directory Structure

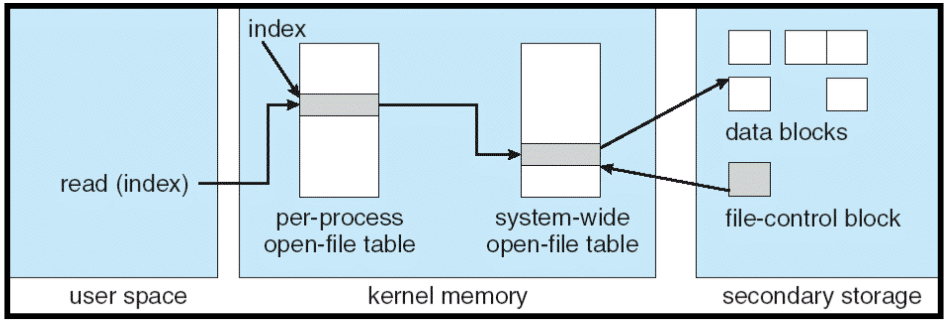


- Not really a hierarchy!
  - Many systems allow directory structure to be organized as an acyclic graph or even a (potentially) cyclic graph
  - Hard Links: different names for the same file
    - Multiple directory entries point at the same file
  - Soft Links: “shortcut” pointers to other files
    - Implemented by storing the logical name of actual file
- **Name Resolution:** The process of converting a logical name into a physical resource (like a file)
  - Traverse succession of directories until reach target file
  - Global file system: May be spread across the network

In-Memory File System Data Structures

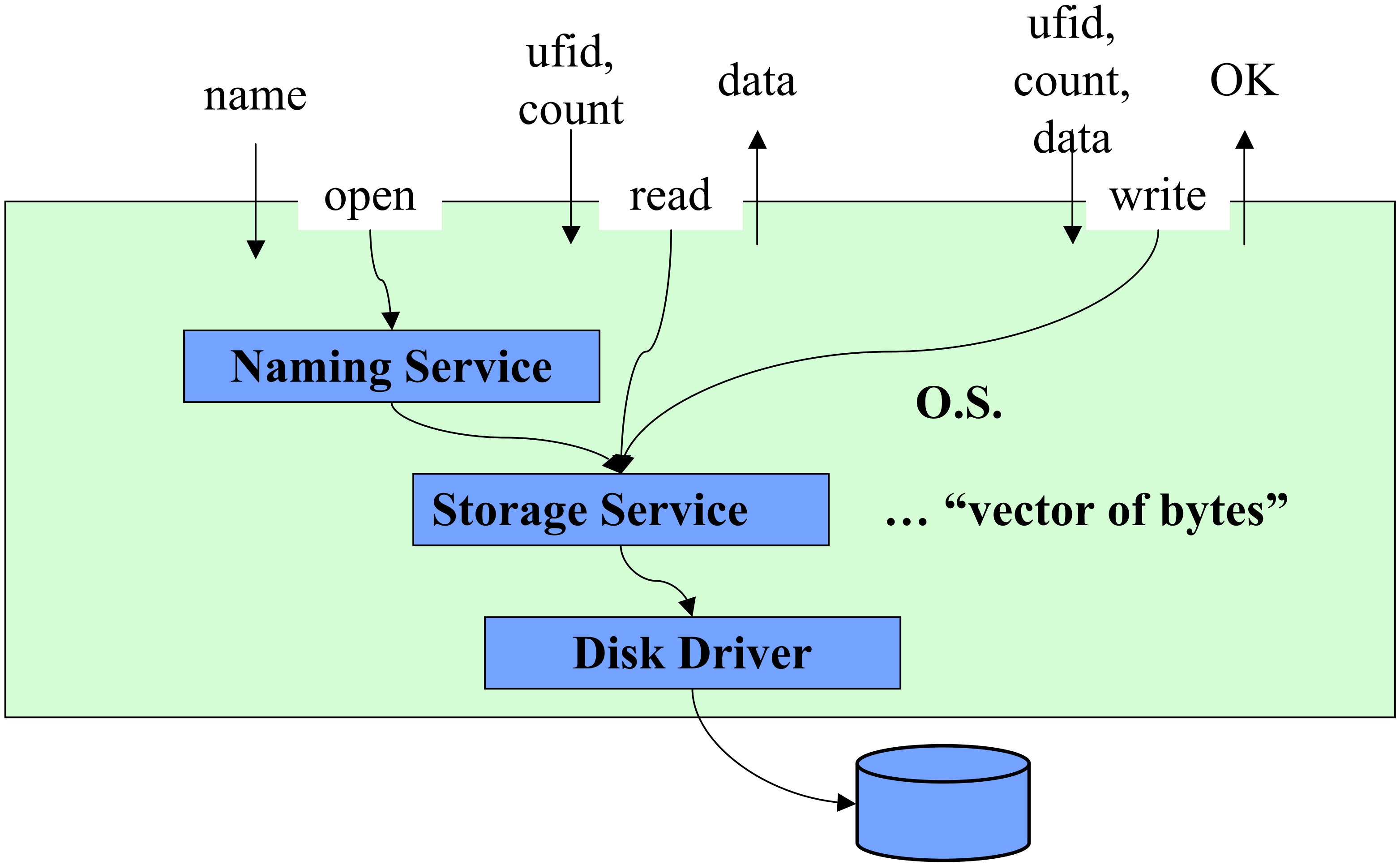


- Open system call:
  - Resolves file name, finds file control block (inode)
  - Makes entries in per-process and system-wide tables
  - Returns index (called “file handle”) in open-file table



- Read/write system calls:
  - Use file handle to locate inode
  - Perform appropriate reads or writes

File System is Layered



5/7/08

Protection and Concurrency

- Any application can generate names independent of username
  - /etc/password
  - /lib/libc.a
  - /boot/vmlinuz-2.2.1
- Protection must be applied independently of naming
  - File owner should be able to control
    - what can be done and by whom.
  - Types of access (eg, Unix: owner, group, public)
- Concurrency : how should multiple accesses be coordinated?
  - E.g., allow:
    - either one writer
    - or many readers

Existence Control

- **File may have multiple names:**
  - **/etc/sendmail**
  - **/usr/bin/mailq**
  - **/root/bin/newaliases**
- **Any name may be deleted from directory**

- **When should file storage space be released?**

File system interface summary

- A file is a collection of related information stored on a secondary storage (cf. logical secondary storage)
  - Attributes (name, size, last update, owner, …)
  - File Operations (open, close, create, read, write, delete, …)
- naming service (how do users select files?)
  - Directories are used for naming
  - A file can have several names
- Protection, concurrency control
  - from unauthorised access: all users are not equal!)
  - File sharing control
- existence control
  - When is the file storage space released?