G22.2250-001
Operating Systems

Lecture 9

Virtual Memory (cont'd)

October 30, 2007

Outline

- Announcements
  - Lab 4 due next Friday, November 9th
- Virtual Memory
  - Demand paging
  - Page replacement
  - Frame allocation
  - Read Sections 9.7 – 9.10
    - Kernel memory allocation, Various practical issues
- File system interface
  - File concept, access methods
  - Directory structure

[Silberschatz/Galvin/Gagne: Sections 9.2 - 9.6, Sections 10.1 – 10.3]

(Review)

VM Support (1): Demand Paging

- Key mechanism for supporting virtual memory
  - Paging-based, but similar scheme can also be developed for segments
- The idea
  - Allocate (physical) frames only for the (logical) pages being used
  - Some parts of the storage reside in memory and the rest on disk
    - For now, ignore how we choose which pages reside where
  - De-allocate frames when not used
- Implementation steps (must be completely transparent to the program)
  - Identifying an absent page
  - Invoking an OS action upon accesses to such pages
    - To bring in the page

(Review)

Demand Paging: Identifying Absent Pages

- Goal: Determine when a page is not present in physical memory

- Extend the interpretation of valid/invalid bits in a page-table entry
  - *valid*: the page being accessed is in the logical address space and is present in a (physical) frame
  - *invalid*: the page being accessed is either not in the logical address space or is currently not in active (physical) memory
    - An additional check (of the protection bits) is required to resolve these choices

- The (hardware) memory mapping mechanism
  1. Detects accesses to pages marked invalid
     - Runs on each memory access: instruction fetch, loads, stores
  2. Causes a trap to the OS: a page fault
     - As part of the trap processing, the OS loads the accessed page

(Review)

Page Fault Handling

On a page fault, the OS

1. Determines if the address is legal
   - Details are maintained in the PCB regarding address ranges
2. If illegal, "informs" the program (in Unix: a "signal")
3. Otherwise, allocates a frame
   - May involve "stealing" a frame from another page
4. Reads the requested page into the frame
   - Involves a disk operation
   - CPU can be context-switched to another process
5. Updates the page table
   - Frame information
6. Resumes the process
   - Re-executes the instruction causing the trap

Interrupting and Restarting

- Must make sure that it is possible to redo the side-effects of an instruction
  - Requires hardware support for precise exceptions
  - Note that page faults are only detected during instruction execution
    - An instruction can cause multiple page faults
- Some subtleties
  - Some architectures support primitive "block copying" instructions
    - Consider what happens if there is a page fault during the copy
    - Need to handle the situation where source and destination blocks overlap
  - What does it mean for the instruction to restart?
- See text book for other pathological cases that must be handled

Uses of Demand Paging

- Process creation
  - Load executable from disk on demand
  - UNIX fork semantics: child process gets a copy of parent address space
    - fork often followed by exec: explicit copying is wasteful
    - Demand-paging + page-protection bits enable copy-on-write
      - Child gets copy of parent's page table, with every page tagged read-only
      - When a write is attempted to this page, trap to the OS
        - » Allocate frame to hold (child's copy of) the page, copy contents, permit write
- Process execution
  - Frames occupied by unused data structures will eventually be reclaimed
    - Available for use by this and other processes
  - memcpy optimization: uses copy-on-write technique above
- Efficient I/O (Memory-mapped I/O)
  - Map files to virtual memory
  - Disk operations only initiated for accessed portions of the file

Cost of Demand Paging

- The cost of accessing memory
  - effective access time = *(1 – p).ma + p.pa*
  - where
    - *ma* is the memory access time when there is no page fault
    - *pf* is the page fault time
    - *p* is the probability of a page fault occurring
  - typical values
    - *p* is usually estimated empirically (and grossly) for the system
    - *ma* is 5-6 orders of magnitude smaller than *pf* (order of tens of milliseconds)

- disk access time
  +
- trapping the OS and saving user state
- checking legality of page reference
- context switch
- when disk read is complete, interrupt existing user and save state
- updating page table
- restarting interrupted user process

Controlling Demand Paging Costs

Three degrees of freedom

- Program structure
  - Selection of data structures and programming structures

```
var A: array [1..128] of array [1..128] of integer;
```

```
for j := 1 to 128                    for k := 1 to 128
  for k := 1 to 128                    for j := 1 to 128
    A[k][j] := 0;                        A[k][j] := 0;
```

- Page replacement
  - Given an allocation of frames to a process, how are these frames managed?
  - Algorithm must ensure that pages likely to be accessed are in memory
- Frame allocation
  - More frames allocated to a process → fewer page faults
  - How should the OS allocate frames to processes?

VM Support (2): Page Replacement

- In a fully-loaded system, all frames would be in use

- In general, page allocation involves
  - Selecting a page to "evict"
  - Writing it to disk (if it was modified)
  - Reading the new page from disk

- Objectives of page replacement/eviction policy
  - Remove a page with the least overall impact on system performance
    - (from the process' perspective)
      Minimize number of page faults
    - (from the system's perspective)
      Minimize disk activity

Page Replacement Algorithms: Components

- Reference strings: the sequence of page numbers being accessed
  - Example
    - A logical address sequence 0400, 0612, 0235, 0811, …
    - Will yield the reference string 4, 6, 2, 8, … (for 100-byte pages)
- Hardware support
  - Extra bits associated with the frames to store information about page use
    - Different from the bits stored in each page table entry
  - Commonly available: a page-referenced bit and a page-modified bit
  - Restriction: Must incur very low overhead to maintain
    - Potentially updated on every memory access
- Algorithms
  - FIFO algorithms
  - OPT (Clairvoyant) scheme
  - LRU algorithms and approximations

Page Replacement: FIFO

- Evict the page that was brought in the earliest

- Pro: Simple to implement
  - OS can maintain a FIFO queue and evict the one at the beginning
- Con: Assumes that a page brought in a long time ago has low utility
  - Obviously not true in general (e.g., much-used library routines)

- How does FIFO perform?
  - Consider reference string (length 12)

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

(with 3 frames) ↑ ↑ ↑ ↑1 ↑2 ↑3 ↑4 ↑1 ↑2 (9)

(with 4 frames) ↑ ↑ ↑ ↑ ↑1 ↑2 ↑3 ↑4 ↑5 ↑1 (10)

Belady's anomaly
Algorithms that don't exhibit this behavior are known as stack algorithms

Page Replacement: What is the Best Algorithm?

- For read-only pages (discounting clean-page preference issues), it can be proven that the optimal algorithm (OPT) is
  - Replace the page whose next use is the farthest

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

(with 3 frames)  (7)

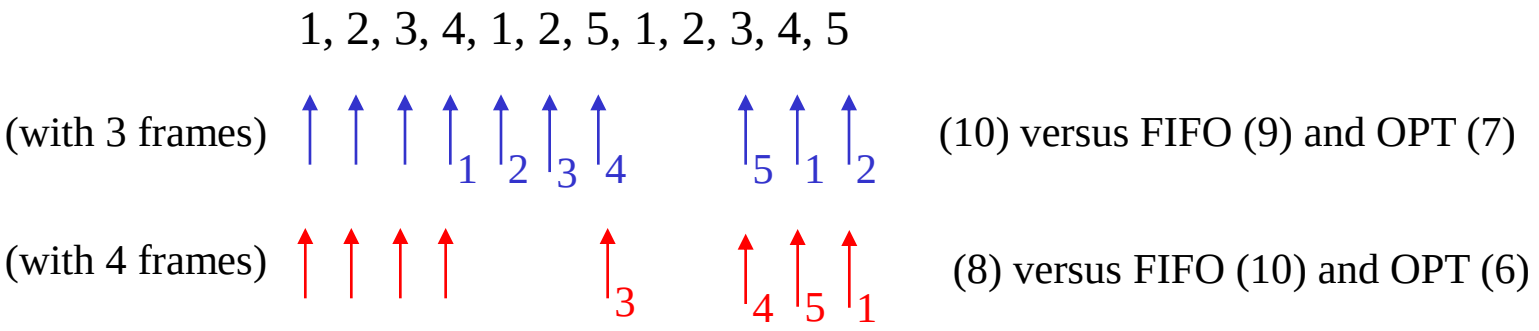
(with 4 frames)  (6)

- Optimality stems from the fact that
  - The page replaced will cause a page fault far away
  - Any other page will cause a fault at least as quickly

- How do you prove that OPT does not suffer from Belady's anomaly?

Page Replacement: LRU

- Problem with OPT: Clairvoyance is generally not possible
  - But sometimes possible to analyze deterministic algorithms
  - In any case, a good baseline to compare other policies against
- LRU (least recently used) is a good approximation of OPT
  - Assumes that recent past behavior is indicative of near future behavior
    - A phenomenon called locality which is exploited repeatedly in virtual memory
- Main idea: Evict the page that has not been used for the longest time

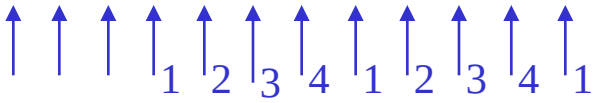


Page Replacement: LRU (cont'd)

- LRU works reasonably well in simulations
  - "real" program traces exhibit locality
  - but, some pathological access patterns

1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, …

(with 3 frames)



- Main problem with LRU: How does one maintain an active "history" of page usage?
  - Counters
  - Stack

Page Replacement: Implementing LRU

- Counters
  - Attach to each frame, a counter that serves as a logical clock
    - Updated by the hardware on every reference
  - Page replacement: choose page in frame with smallest counter value
    - Counter is reset when a new page is loaded
  - Problems: Elaborate hardware, Search time
  - Largely of theoretical value

- Stack
  - Maintain a stack of page numbers
    - On each access, hardware moves the page# to the top of the stack
  - Page replacement: the LRU page is at the bottom of the stack
  - Typical implementation: microcoded doubly linked list
    - Used by one of the earlier CDC machines
  - Still too high a hardware cost

Page Replacement: LRU Approximations

- Page reference bit
  - Stored with the frame containing the page
  - Bit is set whenever the page is accessed
  - Periodically, the OS (or hardware) resets all reference bits
  - Page replacement: Choose an unreferenced page

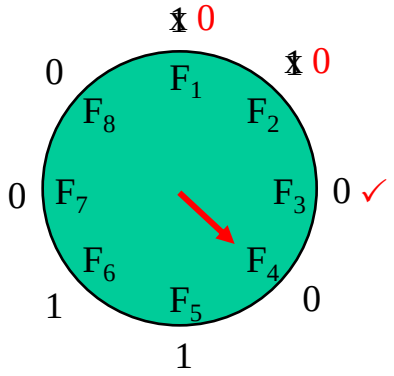
- Additional reference bits
  - For each page *p*, OS maintains an n-bit last-reference-time *lrt[p]*
  - Periodically, OS (or hardware)
    - Shifts right *lrt[p]*, adds current reference bit as MSB, and resets reference bit
  - Note that the additional bits can be maintained in software

  - Page selected is the one with the lowest *lrt*

    lrt[p1] = 11000100 has been used more recently than *lrt[p2]* = 01110111

Page Replacement: LRU Approximations (cont'd)

- Second-chance Algorithm (also known as Clock)
  - Only uses single-bit page reference information
  - Maintains a list of frames as a circular list
  - Maintains a pointer into the list
  - Replacement: search for a page with reference bit zero
    - If there is a page with reference bit 1
      - Set the bit to 0, and continue searching
    - Each page gets a second chance before being evicted



- Enhanced second-chance algorithm
  - Make decision using two bits: page reference and page modify
    - (0, 0): neither recently used nor modified: *best candidate*
    - (0, 1): not recently used but modified
    - (1, 0): recently used, but not modified
    - (1, 1): recently used and modified: *worst candidate*
  - Used in the Macintosh

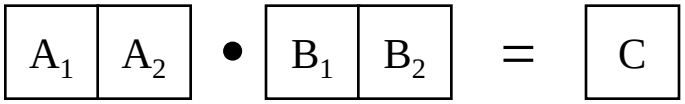
Page Replacement: Performance Enhancements

- Maintain a pool of free frames
  - Buffered (delayed) writes
    - Frame allocation precedes deallocation
    - Allocate immediately from pool, replace later
  - Rapid frame and page reclaim
    - Keep track of which page was in which frame
    - Reclaim pages from free pool if referenced before re-use
      - Can be used as an enhancement to FIFO schemes
- Background updates of writes to secondary store
  - Whenever the disk update mechanism is free
    - Write out a page whose modified bit is set and then reset the bit
- Delayed write (copy-on-write)
  - Create a *lazy* copy (on the first write): defer allocation
    - Used to optimize Unix fork, memcpy

VM Support (3): Frame Allocation

- We have discussed how OS can manage frames allocated to a process Control is also possible in how we allocate frames to processes
- Naïve single-user system
  - Keep a list of free frames
  - Allocate from this list
  - Use eviction (replacement) algorithm when list exhausted
- Problem: Multiprogrammed systems
  - How many frames for each process?
  - Performance varies dramatically with the number of frames
  - E.g., vector dot-product (c := A.B)
    - Vectors of length 32, 4-byte words
    - A page size of 64 bytes (each vector fits into 2 pages)
  - Lets examine number of page faults with 1 – 5 frames …

Vector Dot-Product Example




```
for (i = 0; i<N; i++)
  c += a_i × b_i;
```

Memory reference stream:

$A_1, B_1, C,$	$A_1, B_1, C,$	$A_1, B_1, C,$
$A_1, B_1, C,$	$A_1, B_1, C,$	$A_1, B_1, C,$
$A_1, B_1, C,$	$A_1, B_1, C,$	$A_1, B_1, C,$
…, (16 elements)	…,	…,
$A_2, B_2, C,$	$A_2, B_2, C,$	$A_2, B_2, C,$
$A_2, B_2, C,$	$A_2, B_2, C,$	$A_2, B_2, C,$
$A_2, B_2, C,$	$A_2, B_2, C,$	$A_2, B_2, C,$
… (16 elements)	…	…

- With 5 available frames: 5 page faults (1 for each page)
- With 3 available frames: 5 page faults
- With 2 available frames: 96 page faults OPT: 52 page faults
- With 1 available frame: 3×32 = 96 faults

Frame Allocation: Two Critical Questions

- How many frames to assign to each process?
  - Fixed
  - Variable (from a global pool)
  - Is there a minimum (critical) number of frames that must be allocated?

- How are they assigned?
  - When a new process needs more frames, do we
    - Take away uniformly from a given process
    - Or do we assign frames back and forth between processes?

Frame Allocation Algorithms: How Many?

- Static approach
  - Allocate once and stays fixed during the process' lifetime
- Uniform approach
  - Given *m* frames and *n* processes, allocate *m/n* per process
  - Very simple, but can lead to a lot of wasted frame usage since the *size of the process' virtual space is not considered*
- Proportional allocation
  - Let S be the sum of all the virtual memory "needs" across processes where *si* is the virtual memory need of process *i*
    - Allocate *(si / S)*m* frames to process *I*
  - Problems:
    - Does not distinguish between process priorities
    - Does not distinguish between process behaviors

Frame Allocation: Scope of Replacement

- How are additional requests for frame allocation satisfied?

- Local replacement
  - New frames are allocated to pages from a fixed set associated with the process
  - Number does not change with time

- Global replacement
  - New frames can be selected from a variable pool that is shared by the whole system
  - The performance due to page faults of any one process is dependent on the behavior and demands of others using this approach

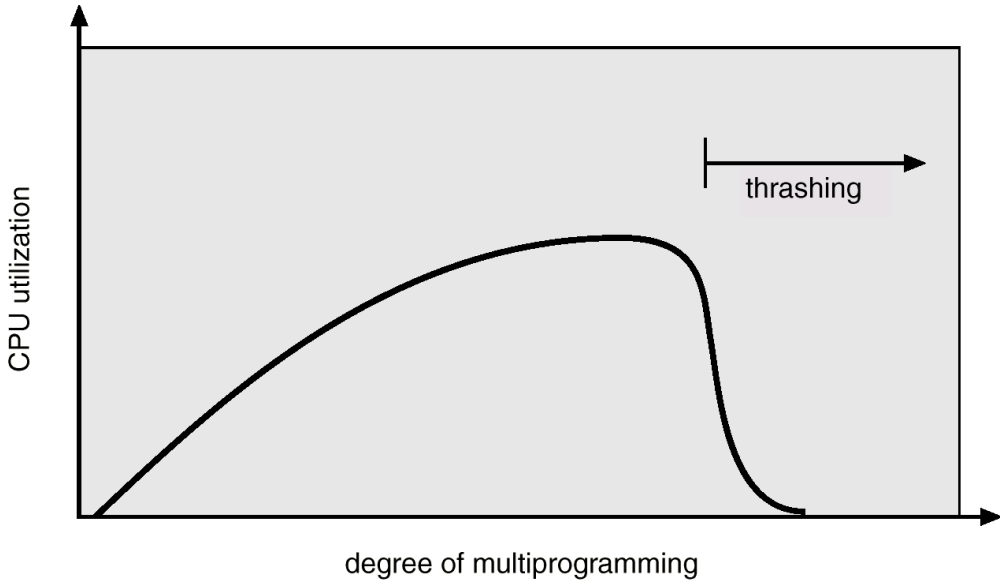
Frame Allocation: Constraints on Number of Frames

- Hardware: Determined by page fault induced instruction restarts
  - Need frames to store all the needs of a single instruction
  - Could be more than one page
    - CISC instruction may straddle page boundary
    - Data may straddle page boundary
    - Indirect addressing may straddle page boundary

- Software: Clearly there is a constraint
  - If a process gets too few frames, it spends all its time demand paging
  - This phenomenon is called thrashing
  - Formally,
    - Over any time window and summed over all processes, let T be the time spent by the process in computing and P be the time spent in page faults
    - A characterization of thrashing in a time window is when $T < P$
  - We can define it, but can we do anything to reduce it?

Thrashing

- Not enough memory for all processes
  - Processes spend their time page-faulting



Dealing with Thrashing

- The idea
  - Exploit the fact that programs demonstrate temporally localized behavior in terms of their memory access
  - Over each "time window"
    - Monitor the behavior of active processes
    - Estimate how many pages each process needs
    - Adjust the frame allocation (and multiprogramming level) accordingly

- The working set of a process over time window W is the set of pages it accesses within W
  - Use of the working set
    - Choose a parameter W
    - Over a time window of size W, estimate the size $|w_i|$ of the working set of each process i
    - Do not activate more processes if the current sum of the $|w_i|$ together with the set $|w_j|$ of the new process j exceeds available memory

Working Set Model

- Examine the most recent Δ page references
  - This defines the process working set
    - If a page is in active use it will be in the process working set
    - Otherwise, it will drop from the working set Δ units after its last reference

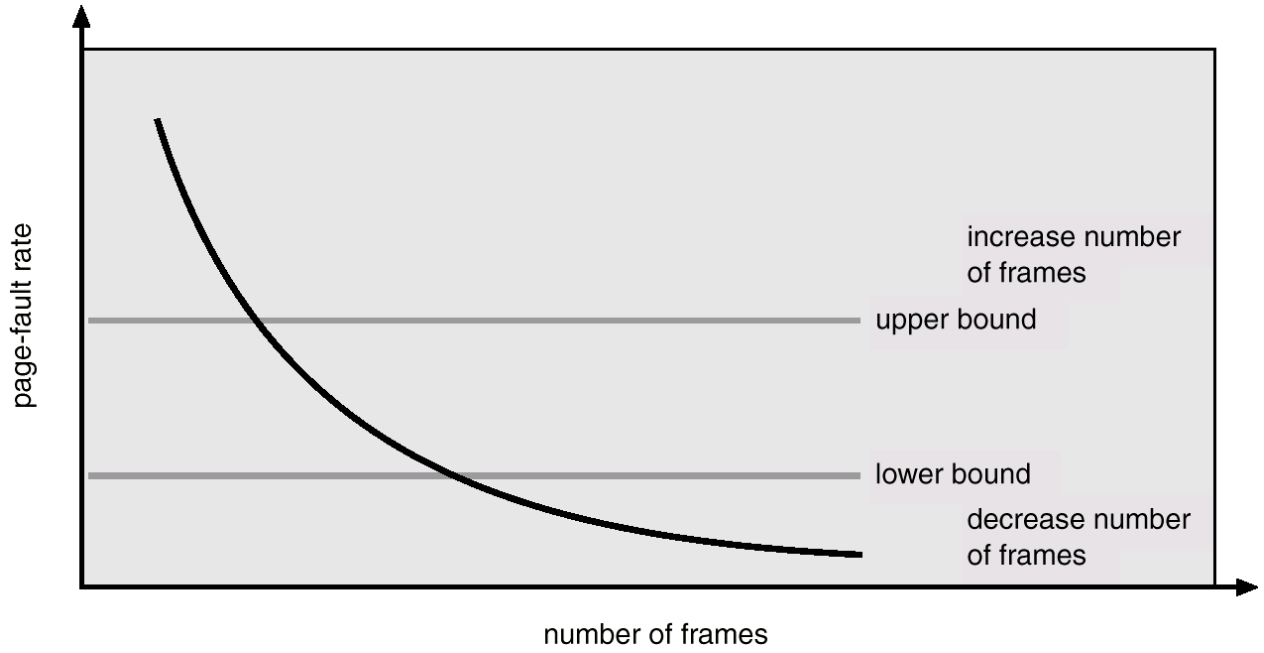
… 2 6 1 5 7 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 4 3 4 4 4 4 1 3 2 3 4 4 3 4 4 4 …

WS = {1,2,5,6,7} WS = {3,4}

- Working set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible
  - Lots of empirical evidence
- Difficulty: Keeping track of the working set
  - Approximated using a fixed interval timer interrupt and a reference bit
    - Periodically, write out reference bits into a structure

Page-Fault Frequency

- More direct approach for controlling thrashing
- Keep track of the page-fault rate of a process
  - When too high: process needs more frames
  - When too low: process might have too many frames
  - Keep each process' page-fault rate within a upper and a lower bound



Demand Paging: Other Issues

- I/O interlocking
  - Need to ensure that I/O does not refer to pages that are swapped out
  - Two common solutions
    - Use kernel buffers to receive I/O responses
    - "pin-down" (or lock) the concerned pages
- Prepaging (warm start)
  - Initial working set is brought in as a block
  - Advantageous when the cost of bringing in a block is lower than that of generating page faults to bring in the subset of the working set that is used
- Choice of page size
  - Large pages: smaller tables, smaller I/O costs, fewer page faults
  - Small pages: less external fragmentation, less overall I/O
  - Trend towards larger page sizes
    - Limiting factor is reducing the number of page faults (disks are slow)

Outline

- Announcements
  - Lab 4 due next Friday, November 9th
- Virtual Memory
  - Demand paging
  - Page replacement
  - Frame allocation
  - Read Sections 9.7 – 9.10
    - Kernel memory allocation, Various practical issues
- File system interface
  - File concept, access methods
  - Directory structure

[Silberschatz/Galvin/Gagne: Sections 9.2 - 9.6, Sections 10.1 – 10.3]

Course Review and Topics to Come

- Protected kernels
  - Supervisor/user mode (how OS code is isolated from user code)
  - Exception and trap handling (how the OS gets control)
- Processes
  - Run-time representation of programs, granularity at which the OS allocates resources, manages protection
  - Synchronization (locks, semaphores, condition variables, monitors, …)
  - Deadlock handling
- Resource handling
  - CPU scheduling
    - Advanced topic: Lottery Scheduling
  - Memory management and virtual memory
  - File systems, secondary storage
    - Advanced topic: Virtual Machines
    - Advanced topic: Log-structured File Systems
- Protection and security

File Systems

- Organization of non-volatile memory
  - disks, tapes, CDROMs, etc.
- Data is regarded as a set of files
  - a file is an abstract data-type containing
    - a logical set of data
    - descriptive information about the file (attributes)
- Files are grouped into directories
  - enable file identification & retrieval

File Structure and Attributes

- Consists of a number of records
  - a record is the finest granularity for accessing data
    - can be a byte or a "complicated object"
  - a file is usually one of
    - a sequence of records (UNIX: sequence of bytes)
    - a map from keys to records (UNIX: offset used as key)

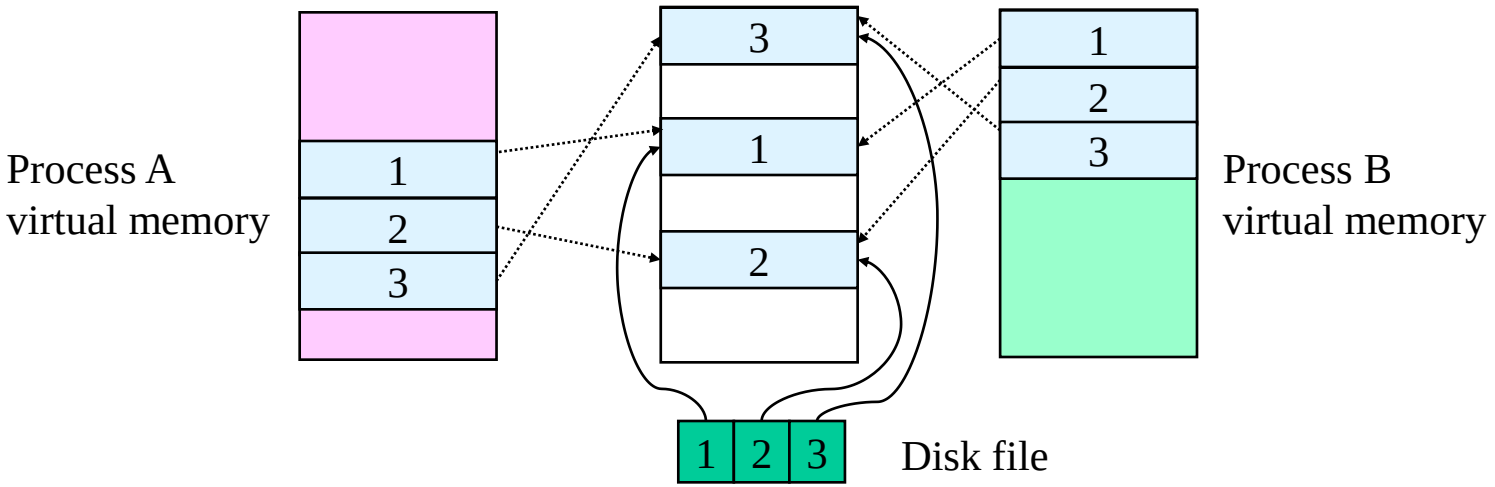
- Attributes: Used to simplify file identification and access
  - **name**
  - **type**: e.g. executable module, ASCII text, ...
  - **location**: pointer to where the data is stored
  - **size**
  - **protection**: e.g. read/write privileges
  - **time & dates**: e.g. of creation, last modification & use
  - **user**: "owner" of file

File Operations (OS System Calls)

- Create and Delete
  - allocate and deallocate space
  - add/delete the file to/from the directory (to be discussed)
- Open and Close
  - reduce directory searching
  - open
    - locates a file in the directory structure
    - stores this information in the OS open file table
    - returns a handle to the file
      - usable for reads, writes, etc.
  - close
    - removes the file table entry
- Read, Write, and Reposition
  - depends on the file structure
  - sequential operations
    - OS maintains a file position pointer
    - read reads from this position
    - write overwrites starting at this position
    - reposition changes the file position pointer
- Other operations
  - append to an existing file
  - copy a file

Files as Segments

- Files can be mapped into memory
  - e.g. OpenAsSegment (file-name)
  - Memory operations are treated as file reads and writes
  - Closing the file removes the file from the virtual address space
- In virtual memory systems
  - several processes can share a file
  - the OS keeps only one copy of the actual file



File Types

- Provide means of interpreting file contents
  - can be left completely to the application program
  - OS typically provides support for a minimal set of types

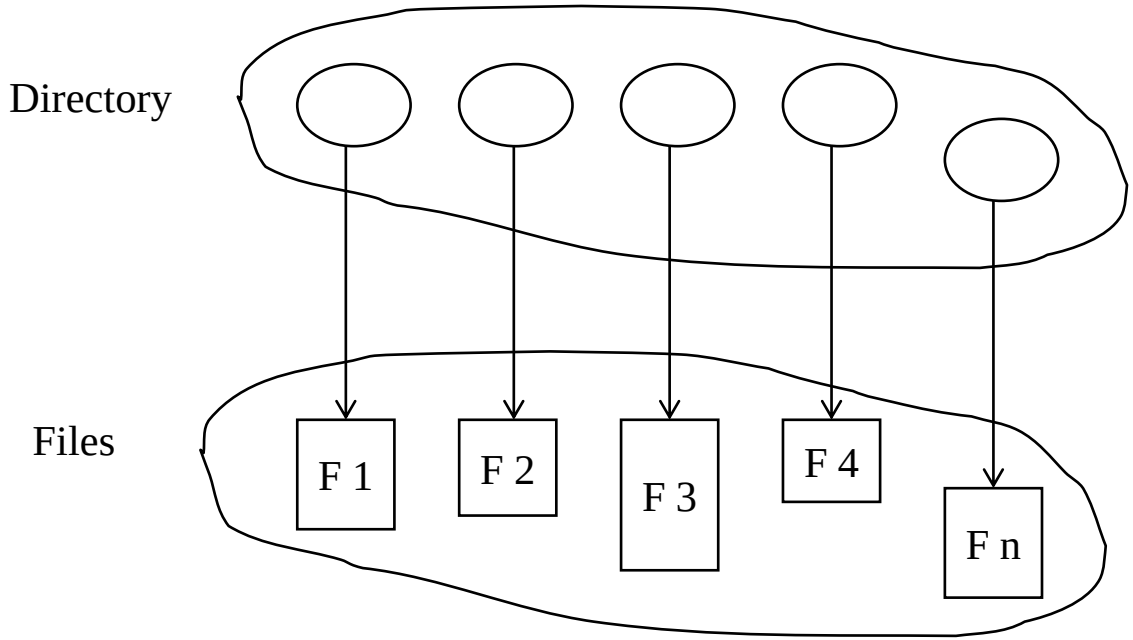
- Three ways of associating file types
  - OS maintains file type information
    - e.g. Unix executable which indicates a binary object module
  - names indicate types
    - typically expressed via an extension (e.g. file-name.type)
    - convenient for user and application programs
    - OS can associate programs with file extensions (e.g., Word with .doc)
  - internal to the file
    - e.g. a magic numbers convention

File Storage and Access Models

- Files are stored in (persistent) secondary storage
  - as data chunks called blocks
    - sectors on disks
  - blocks are usually fixed size
- File access models
  - sequential: read/write (fid, buf)
  - direct: read/write (fid, record#, buf)
  - indexed: read/write (fid, record-key, buf)
- Mapping
  - user program/library is responsible for
    - mapping record numbers/keys into blocks
  - OS is responsible for
    - hiding details of physical structure
    - packing records into blocks and handling multi-block records

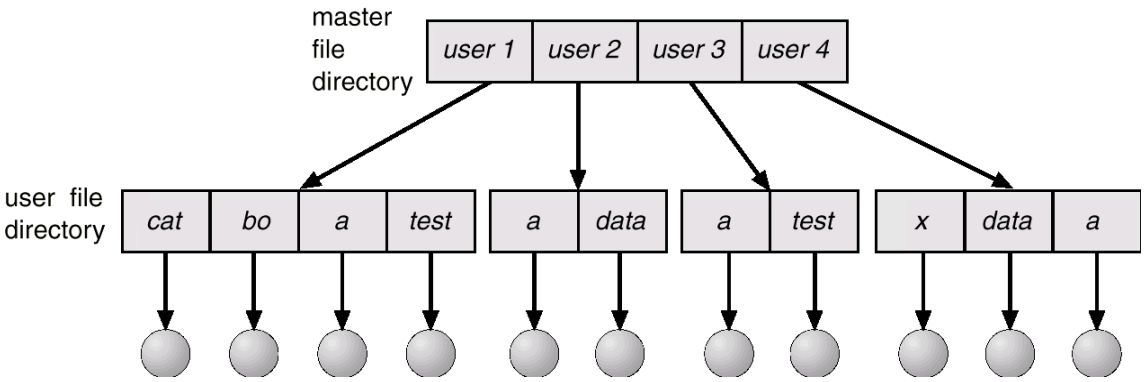
Directories

- A logical way of organizing the files
  - a "symbol table" for translating file names into its attributes
  - to expedite access to files
  - to implement access privileges easily



Directory Operations and Structure

- Directory operations
  - create and delete directories
  - list/search for files
  - rename and relocate files
  - rename and move directories
- Directory structure
  - early systems
    - single directory, named files (at the level of partitions ≡ virtual disks)
    - two-level, users at second level



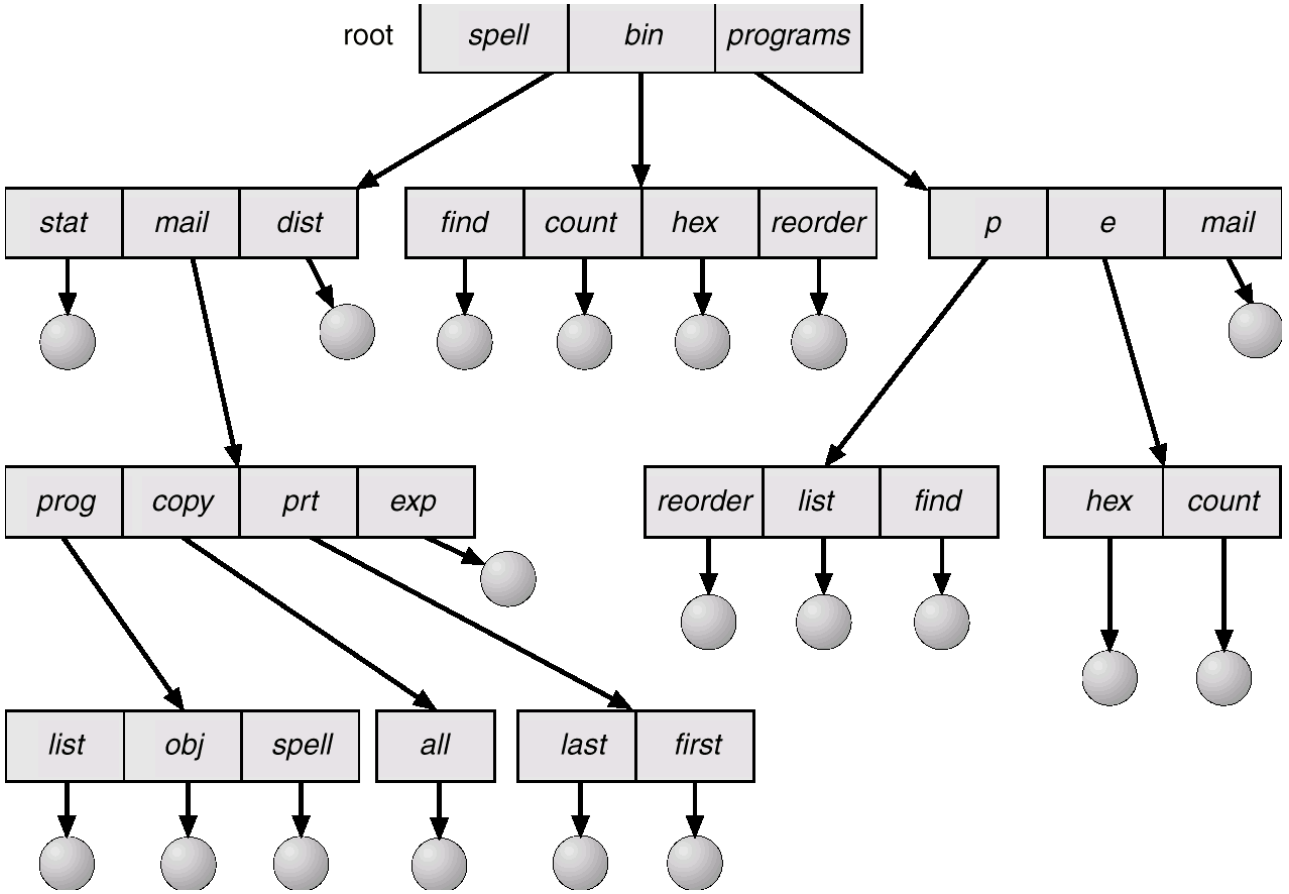
Tree-structured Directories

- The directory is a tree of unlimited depth
  - each node is either a file or a (sub)directory
  - each file/directory is identified by a path-name
    - from the root (an absolute pathname)
    - from a specified directory (a relative pathname)
- Protection information specified at each node in the path
  - for files: readable, writeable, executable
  - for directories: visible, searchable, writeable
  - UNIX scheme:
    - Three fields (each with three bits): owner, group, universe

```
drwxr-xr-x    2 vijayk   None         0 Apr  8 13:03 adirectory
-rw-r--r--    1 vijayk   None         0 Apr  8 13:03 afile
```

  - discussed at length in Lecture 12

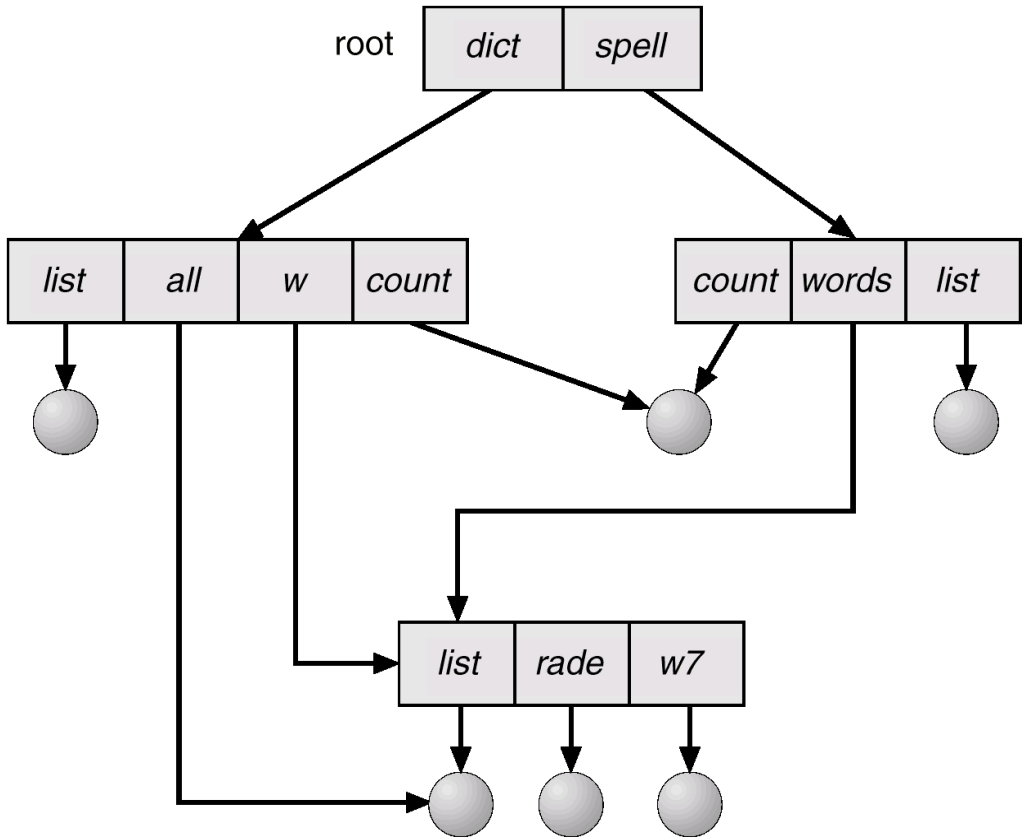
Tree Structured Directories: Example



Non-tree Directory Structures

- Limitations of tree-structured directories: Restricted file sharing
  - each file has a single name and belongs in a single directory
- Generalizations of trees
  - directed acyclic graphs (DAGs)
  - (unrestricted) directed graphs
- Basic idea
  - allow a file or directory to be in multiple directories
  - consequence
    - multiple path names
    - problems with consistency

Non-tree Structured Directories: Example



Non-tree Directory Structures: Issues

- Problem: The same file may be referred to by multiple names
  - How do we maintain consistency of the directory information?
    - E.g., file size, update time, etc.

Two approaches

- Maintain a single copy of the file
  - each parent directory contains pointers called links in Unix
    - hard links: a new (replicated) directory entry
      - OS keeps these entries consistent
    - soft links: a directory entry that points to the original
  - pointers must be transparent to application programs

- Maintain explicit copies
  - OS must update all copies to reflect the latest state

Directories: Other Problems

- Shared files
  - problem with deleting shared files
  - solutions
    - leave links dangling (e.g. soft links in Unix)
      - the links are checked (and generate errors) when accessed
    - maintain a reference count of links to a file/directory
      - delete only when count is zero (e.g., hard links in Unix)

- Directory cycles
  - adding links can create cycles
    - can happen unless each directory creation is checked
  - problems with directory search
    - also, reference counts no longer work
  - solutions
    - do nothing (Unix)
    - make sure that directory traversal operations check for cycles