

**V22.0202-001**  
**Computer Systems Organization II (Honors)**  
(Introductory Operating Systems)

Lecture 18  
File Systems

April 11, 2005

# Outline

---

- Announcements
  - Lab 4 demos should be done this week
    - Will provide a solution to this lab tomorrow/Wednesday
  - Lab 5 due **April 20th**
- Virtual memory (cont'd)
  - Thrashing and the working set model
  - Read Sections 9.7 – 9.10
    - Kernel memory allocation
    - Various practical issues
- File system interface
  - File concept, access methods
  - Directory structure
  - File system mounting
  - File sharing

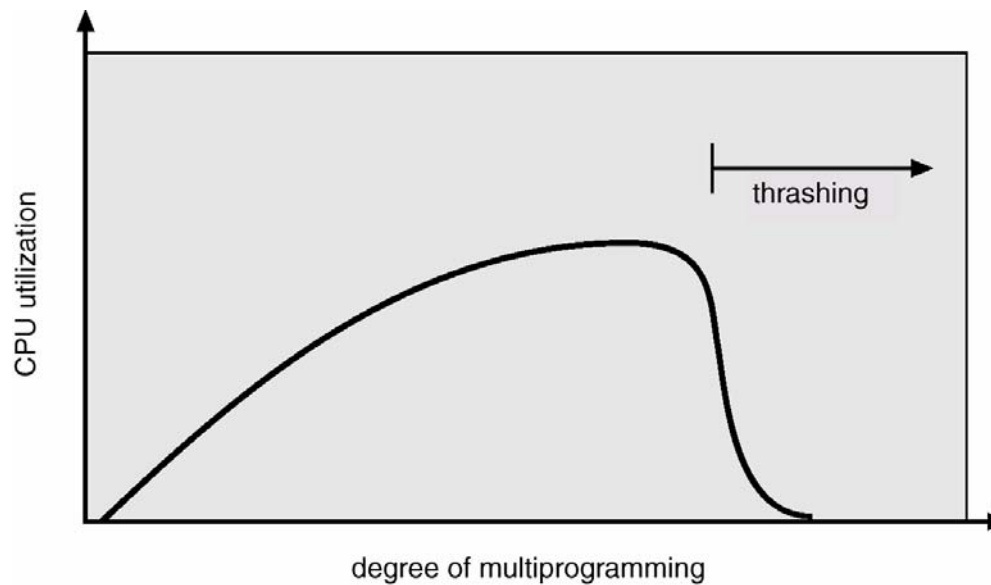
*[ Silberschatz/Galvin/Gagne: Section 9.6, Chapter 10 ]*

4/11/2005

# (Review) Thrashing

---

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



# (Review) Dealing with Thrashing

---

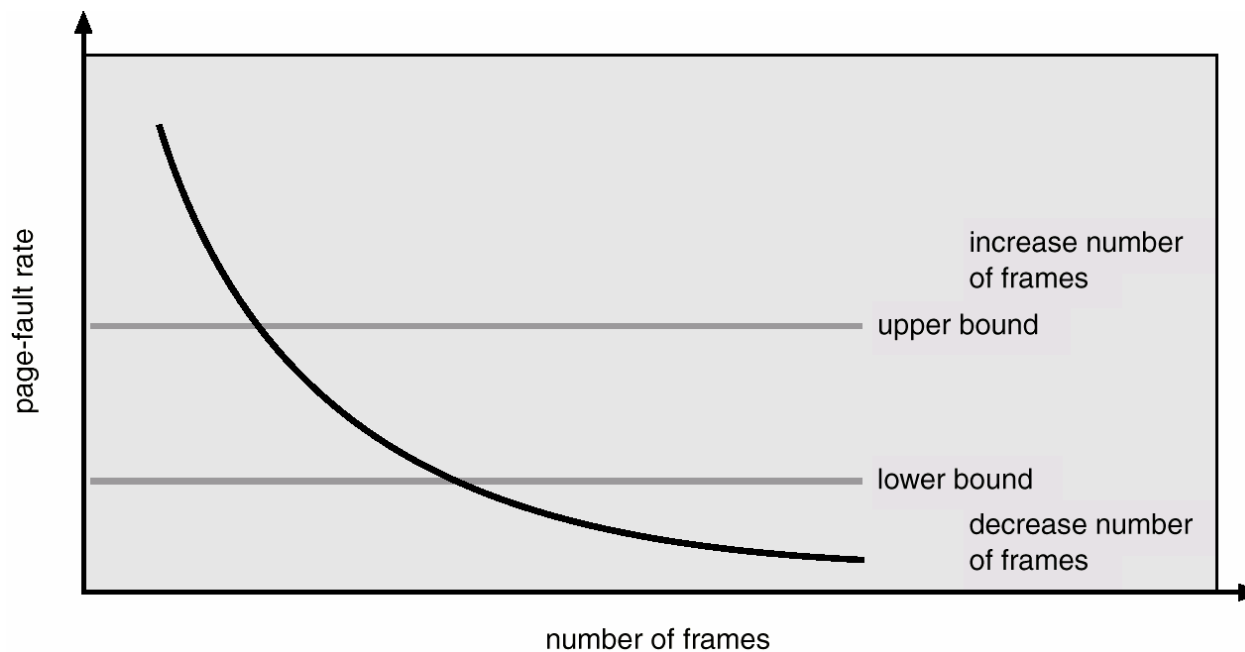
- The idea
  - Exploit the fact that programs demonstrate **temporally localized behavior** in terms of their memory access
  - Over each “time window”
    - **Monitor** the behavior of active processes
    - **Estimate** how many pages each process needs
    - **Adjust** the frame allocation (and multiprogramming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



# 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 an 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 demos should be done this week
    - Will provide a solution to this lab tomorrow/Wednesday
  - Lab 5 due April 20th
- Virtual memory (cont'd)
  - Thrashing and the working set model
  - Read Sections 9.7 – 9.10
    - Kernel memory allocation
    - Various practical issues
- File system interface
  - File concept, access methods
  - Directory structure
  - File system mounting
  - File sharing

*[ Silberschatz/Galvin/Gagne: Section 9.6, Chapter 10 ]*

4/11/2005

# 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
  - Memory management and virtual memory
  - File systems, secondary storage
- Protection and security

# File Systems

---

- Organization of non-volatile memory
  - disks, tapes, CDRROMs, 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)

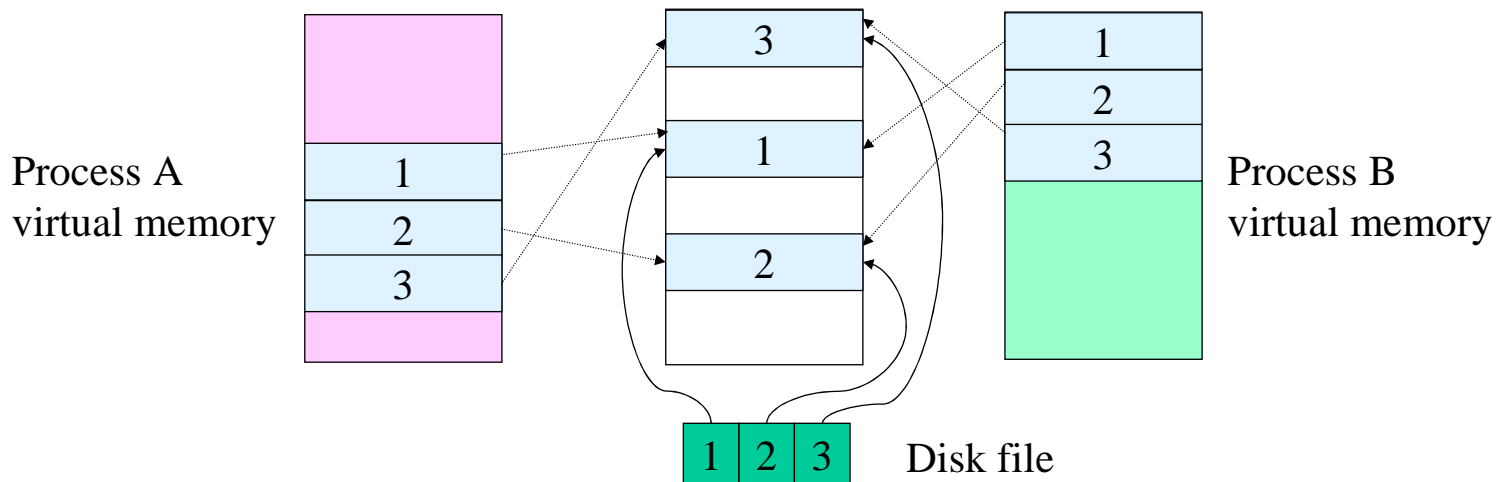
---

- **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

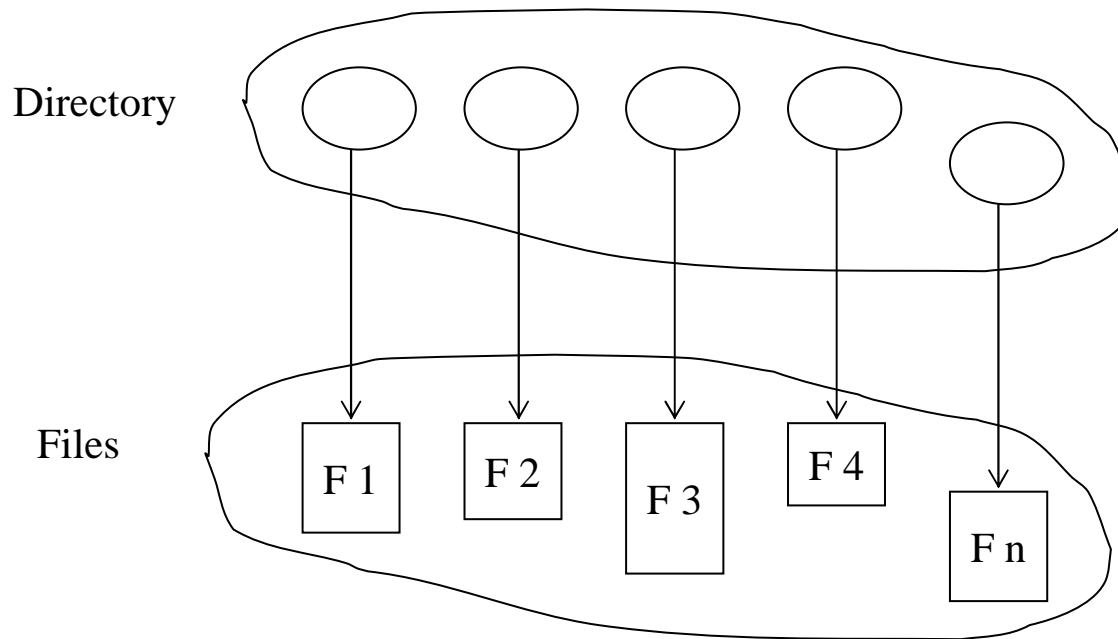
---

- 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

---

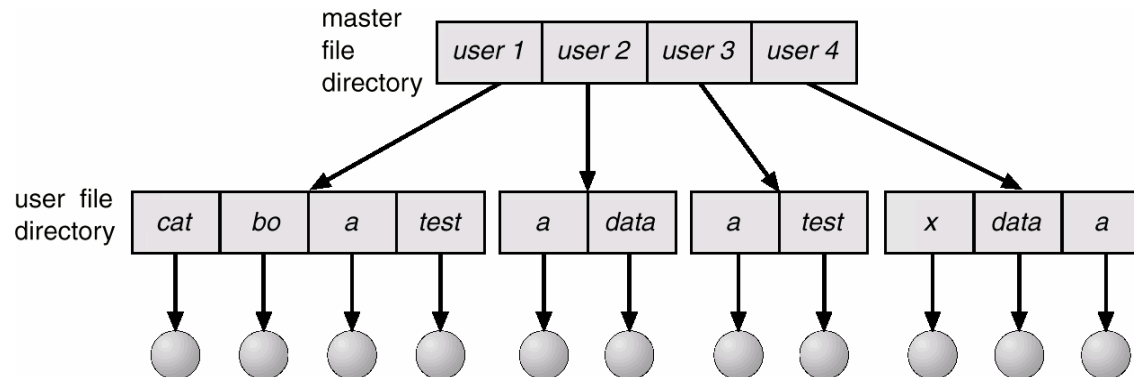
- A logical way of organizing the files
  - a “**symbol table**” for translating file names into its attributes
  - to expedite access to files
  - to implement access privileges easily



# 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  $\equiv$  **virtual disks**)
    - two-level, users at second level



# Tree-structured Directories

---

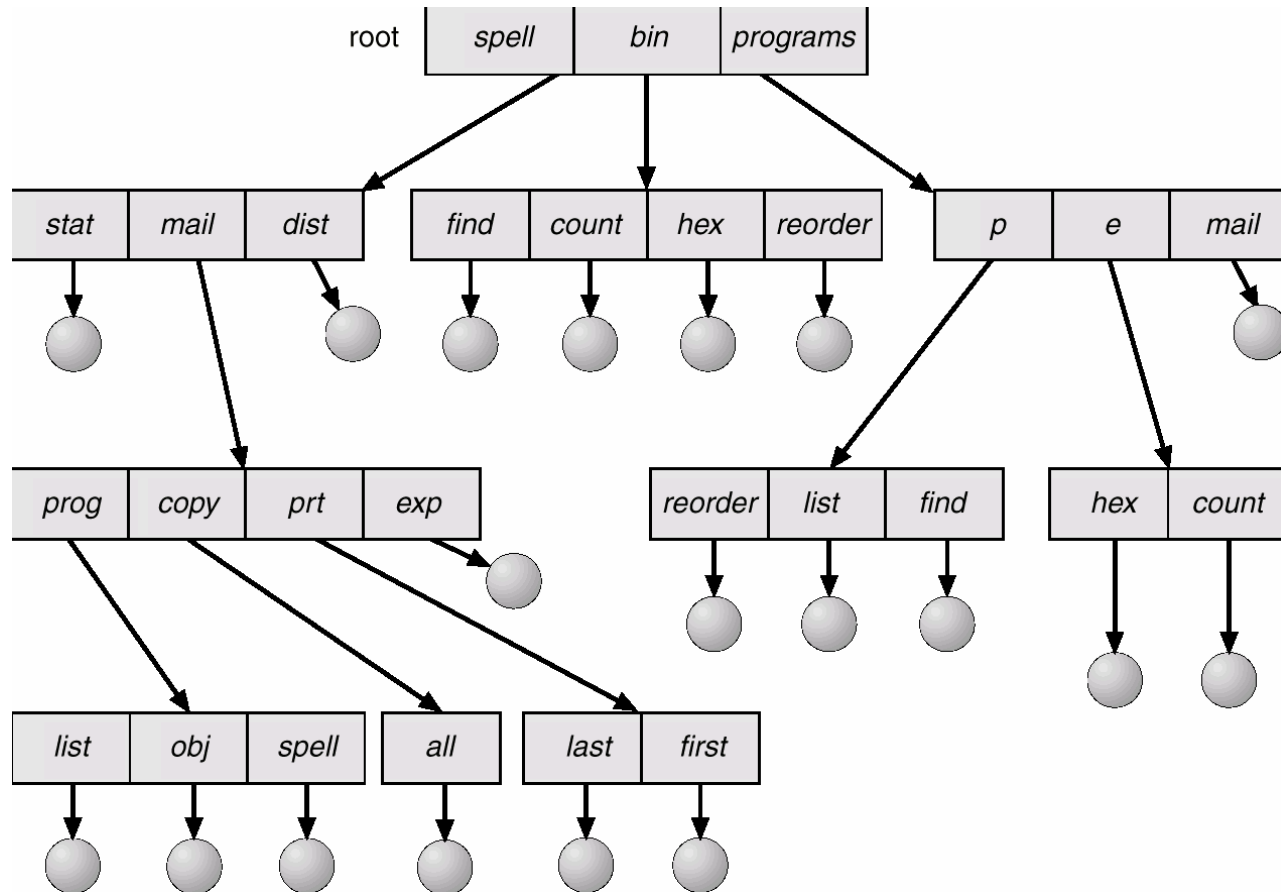
- The directory is a tree of unlimited depth
  - each node is either a **file** or a **(sub)directory**
  - each file/directory is identified by a path-name
    - from the root (an **absolute** pathname)
    - from a specified directory (a **relative** pathname)
- Protection information specified at each node in the path
  - for files: **r**eadable, **w**riteable, **e**xecutable
  - for directories: visible, searchable, writeable
  - UNIX scheme:
    - Three fields (each with three bits): **o**wner, **g**roup, **u**niverse

```
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 20

# Tree Structured Directories: Example

---



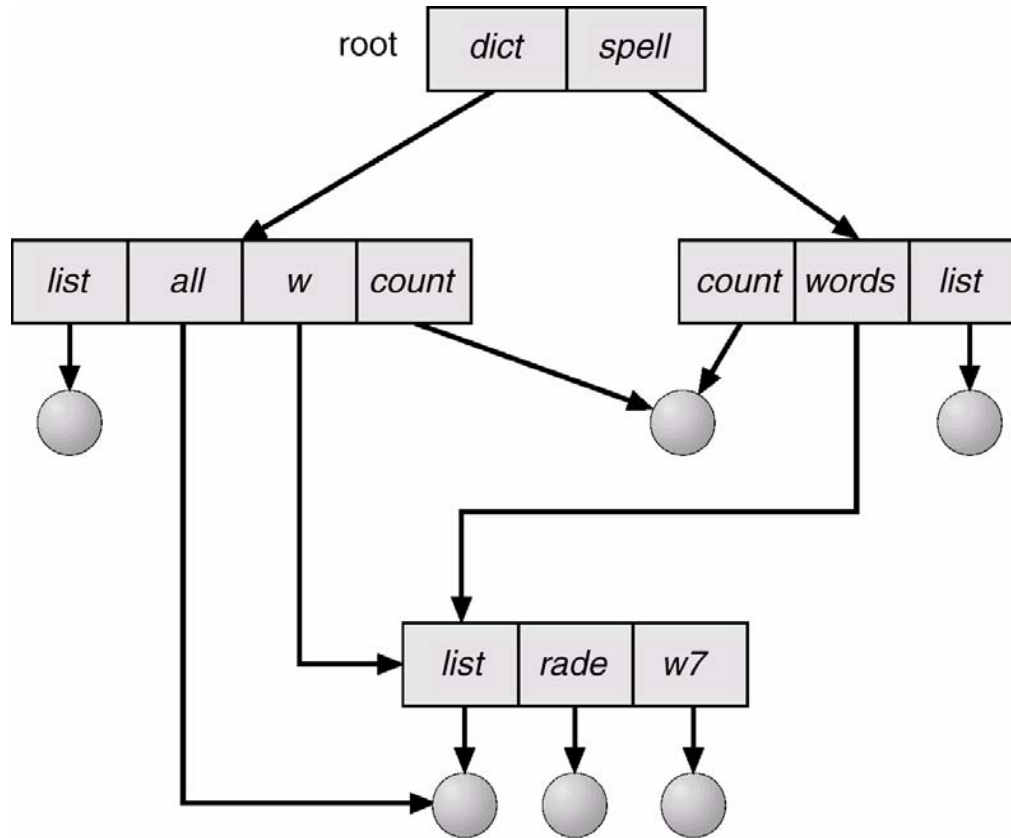
# Non-tree Directory Structures

---

- 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

# Mounting File Systems

---

- Similar to opening files: Need to mount a file system before its use
- Mount parameters
  - the device on which the file system is resident
  - the location in the file system: called the **mount point**
- The OS
  - verifies that the device does contain the file system of interest
  - indicates in its directory structure that a file system is mounted
- Policy issues
  - What happens to existing files?
  - Can a file system be mounted multiple times?
- Mounting is usually automatic or script-driven
  - at start-up time (e.g., with hard disks)
  - on first access (e.g., with floppy disks)

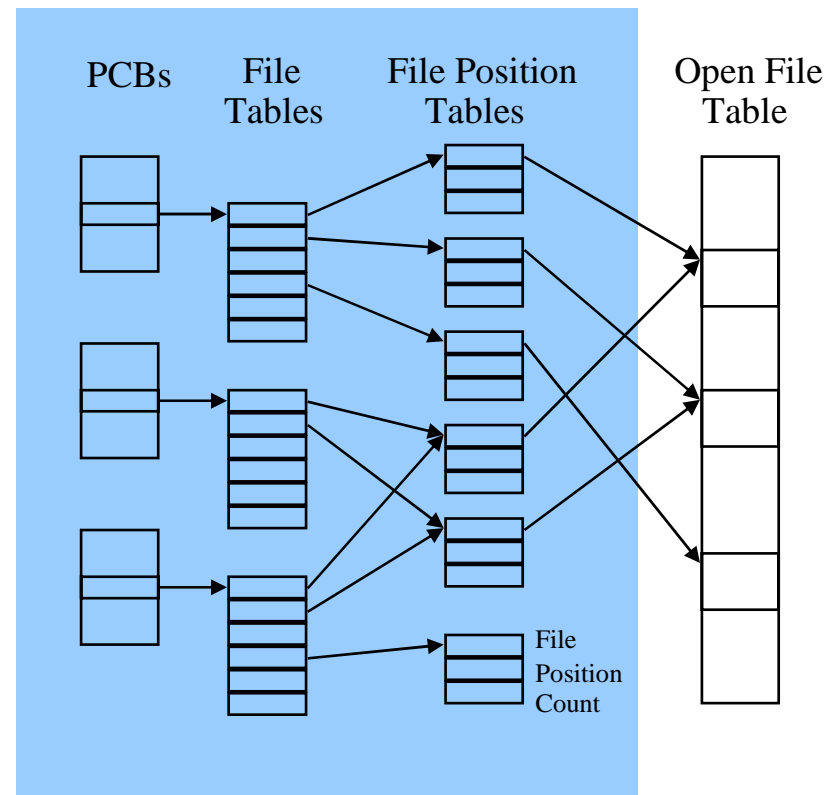
# Sharing Files

---

- Multiple processes may access the same file
  - for reading
  - for writing
- Multiple readers are no problem
  - each reader process keeps its own pointer
- Concurrent writing is a problem
  - common solution
    - OS provides a **lock** operation
    - concurrency control handled by the processes themselves
  - Semantics
    - **UNIX**: writes are immediately visible to all readers
    - **Session**: writes are visible to subsequent open's **after** writer close's the file

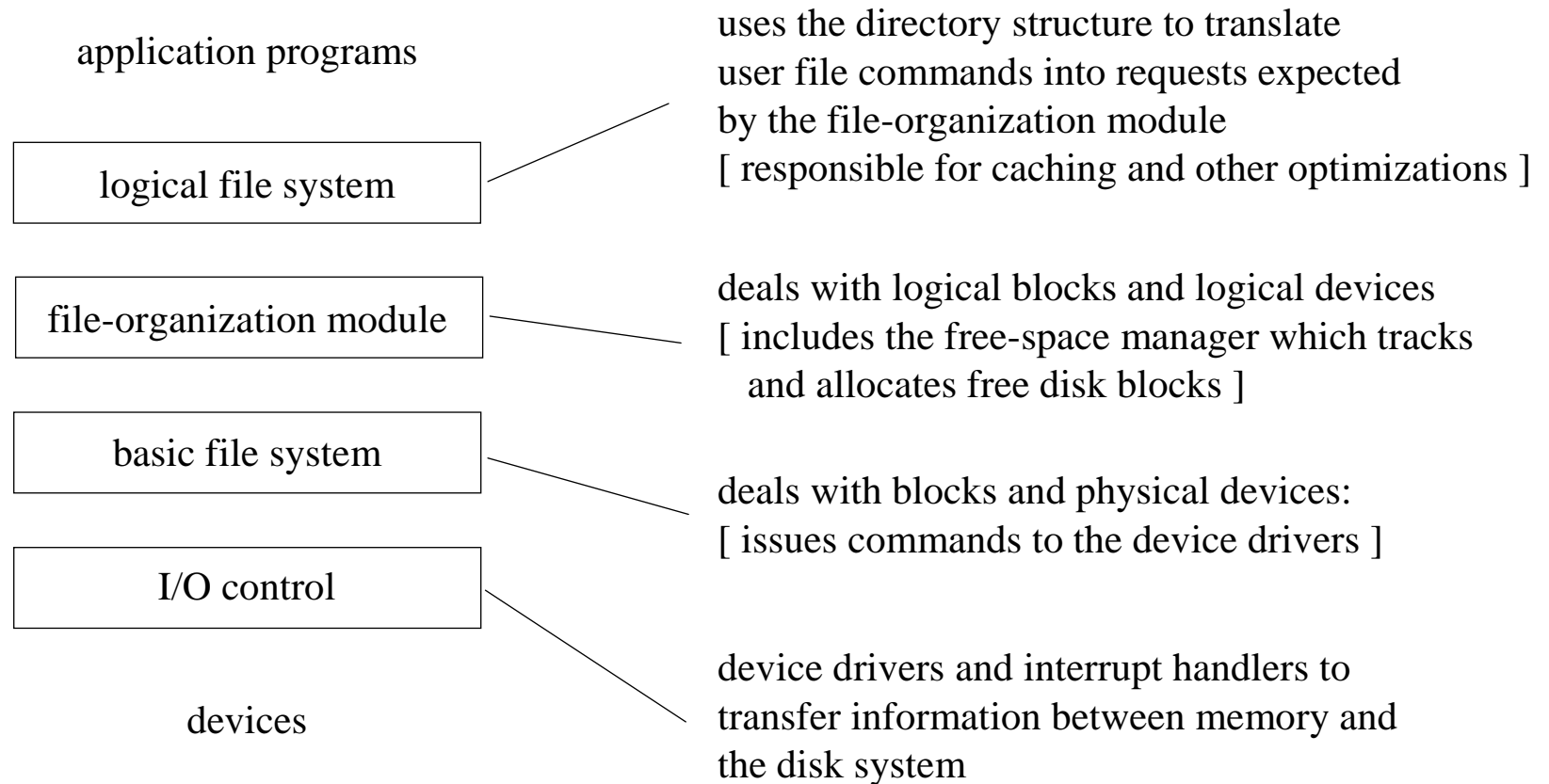
# File Sharing in Unix

- Any process can open any file
  - subject to access constraints
  - no concurrency control check
- Unix maintains one entry in open file table
  - to maintain OS consistency
  - keeps reference count, deallocates when zero
- Processes can share file pointers
  - e.g. after fork; used by shells



# File System Implementation: Overall Structure

---



# Issue 1: Allocation of Disk Space

---

- Three types of information stored on the disk
  - (millions of) **data blocks**: *~thousands of bytes*
  - (thousands of) **file descriptors**: *~tens of bytes*
    - name, access privileges, usage times, ...
  - **storage allocation information** (sometimes called a **superblock**)
- Disk block allocation
  - disk blocks are “numbered”
    - usually a simple function of <platter #, surface #, cylinder #, sector #>
  - sometimes allocated in multiples to reduce management overhead
  - commonly used schemes
    - **contiguous** allocation
    - **linked** allocation
    - **indexed** allocation

# Scheme 1: Contiguous Allocation

---

- Similar to contiguous memory allocation
  - files occupy a **sequence** of "blocks" of disk
    - a file occupies blocks  $b, b+1, \dots, b+(n-1)$
  - the file descriptor stores  $b$  and its length (in bytes)
  - mapping from logical to physical: Logical address/Block Size
    - Quotient: block number
    - Remainder: block offset
- Finding disk space
  - methods such as first-fit, best-fit, worst-fit
- Problems
  - external fragmentation
    - compaction would work but involves high-overhead disk operations
  - need a good estimate of the file size at allocation time
    - too small: potentially expensive copying is needed to **relocate** it as it grows
    - too large: internal fragmentation

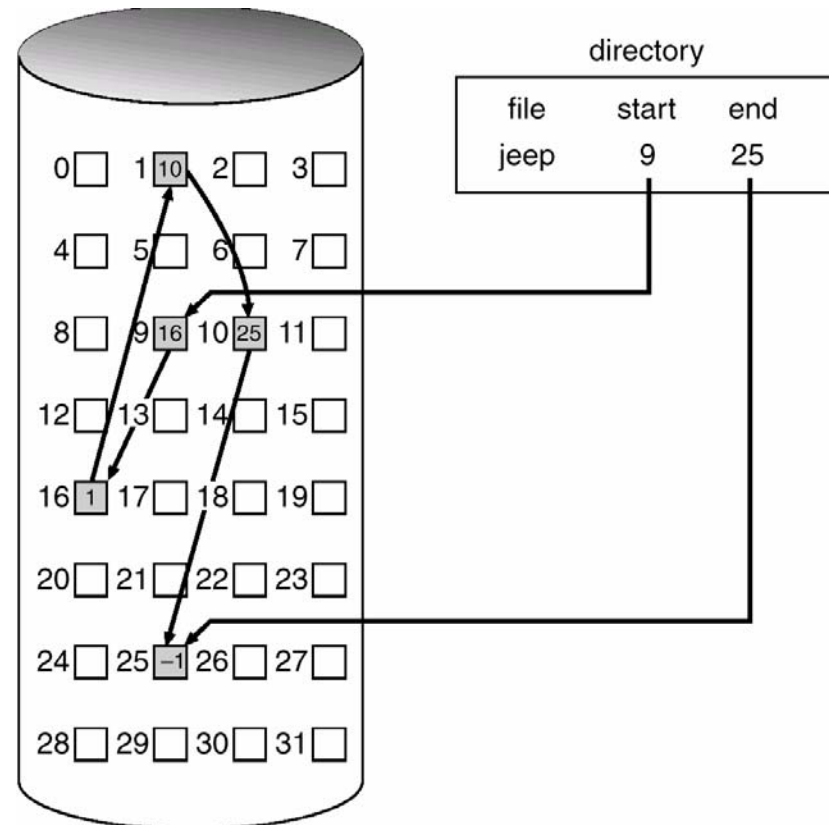
# Extents

---

- A compromise scheme
  - start with an initial allocation
  - extend if necessary with a **non-contiguous** segment
    - called an **extent**
    - the file descriptor contains
      - the size
      - starting point
      - a pointer to the beginning of the next extent
  - in some cases, the size of the extent might be under user control
  - mapping is managed by the logical file system

# Scheme 2: Linked Allocation

- The file consists of
  - arbitrary **sequence of disk blocks**
  - a **set of pointers** from each block to the next
- Initially, the file descriptor
  - contains a **null** pointer when the file is empty
- Extending the file is done by
  - adding **any free** block
  - modifying the pointers
- No problem with external fragmentation
- Where are the pointers stored?



# Pointer Storage for Linked Allocation

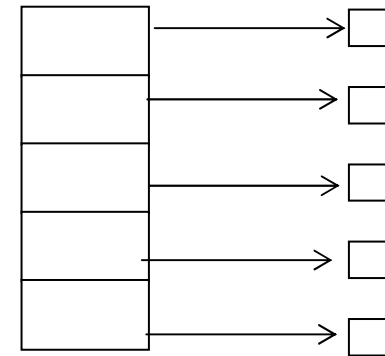
---

- Store the pointer in the blocks
  - very slow random/direct access
- Store the pointers separately
  - e.g. MSDOS File Allocation Table (FAT)
    - uses a contiguous array  $A$  of indices (stored in a fixed place on the disk)
    - if block  $i$  is followed by block  $j$  in a file  $A[i] = j$
    - free blocks have a special entry (usually zero) in them
    - array  $A$  can be stored in memory during use
      - or partially stored (cached)
    - allows (slow) random access to file by scanning array
- Reliability
  - requires integrity of pointer structure
  - some redundancy is sometimes used

# Scheme 3: Indexed Allocation

---

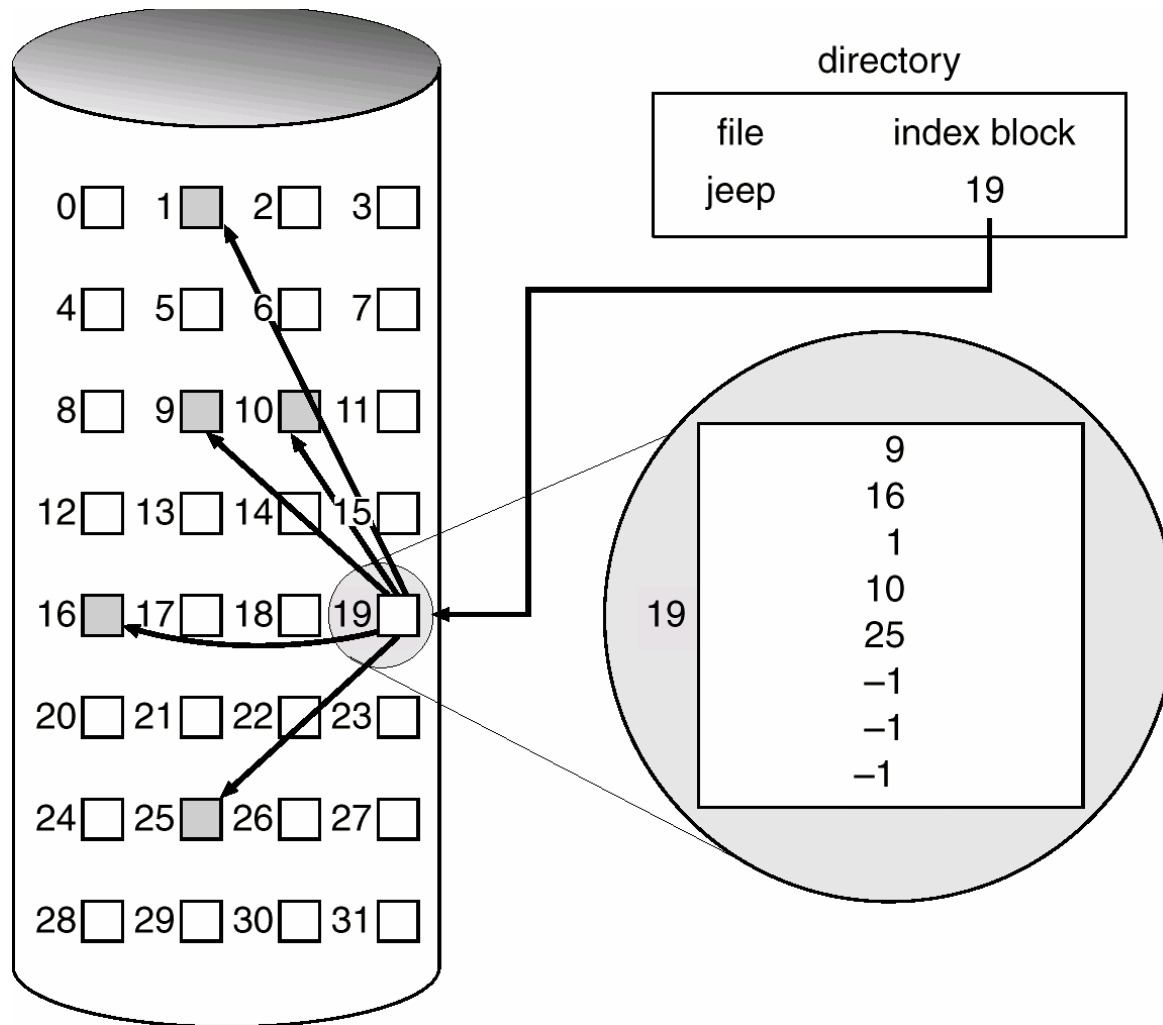
- Each file has an index table A
  - the  $i^{\text{th}}$  block of the file is in block A[i]
  - can be cached for rapid access
- Pros
  - supports direct access
  - has no external fragmentation
- Cons
  - for small files, index block space may be wasted



index table

# Example of Indexed Allocation

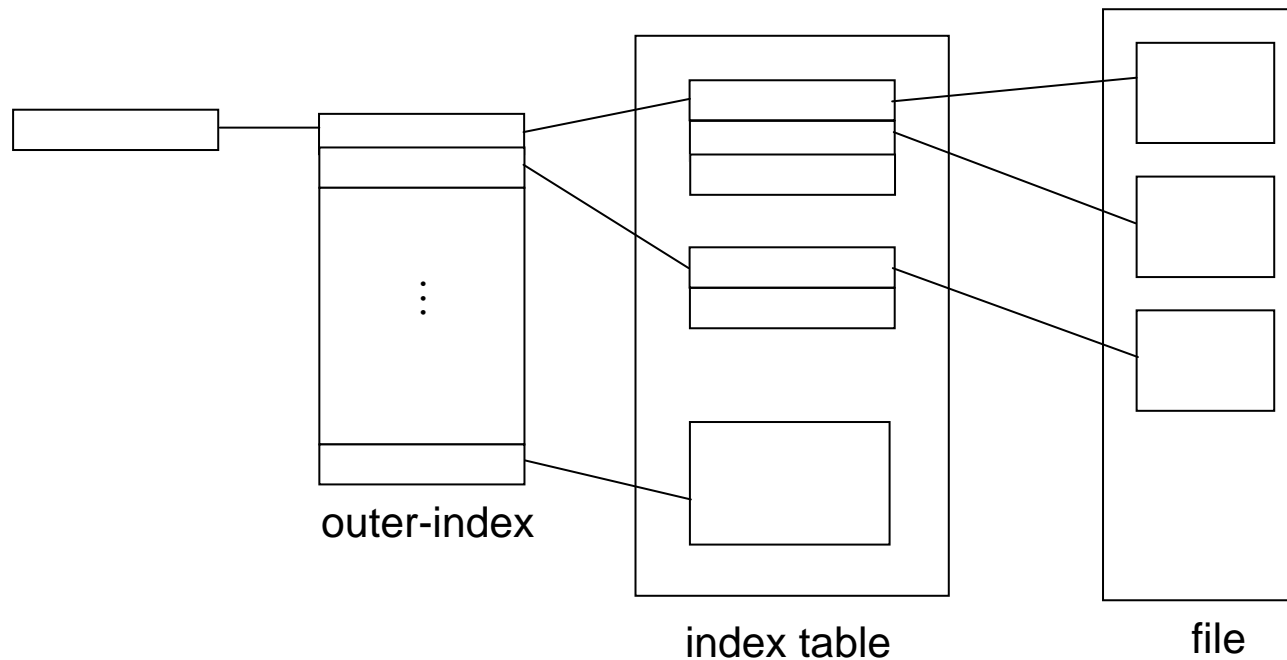
---



# General Indexed Allocation

---

- A General Scheme: **Linked indexing**
  - **flat** indexing: index is a linked list of index sub-blocks
  - **multilevel** indexing: use a hierarchy of index tables
    - table at level  $i$  is an index into table at level  $i+1$
    - why is this useful?
    - What is the downside?



# Asymmetrical Index Trees

- Used in BSD UNIX to provide **fast access to small files**
  - File descriptor is called an i-node

