+ swap page in

+ restart overhead)

- **Example**:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - EAT = (1 -p) x 200 + p (8 milliseconds)

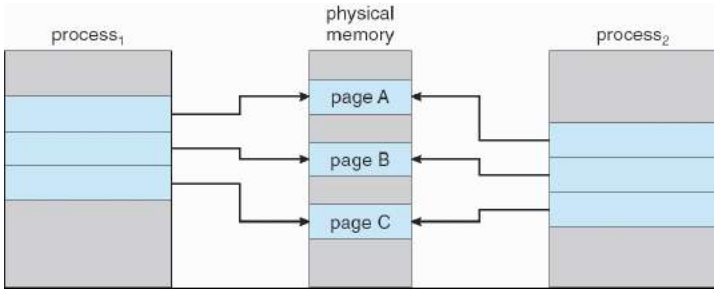
    = (1 -p x 200 + p x 8,000,000

    = 200 + p x 7,999,800
  - If one access out of 1,000 causes a page fault, then

    EAT = 8.2 microseconds.
  - This is a slowdown by a factor of 40!!

Copy-on-Write

- p.369 TB
- Virtual memory allows other benefits during **process creation**:
  - Copy-on-Write
  - Memory-Mapped Files (later)
- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- Free pages are allocated from a **pool** of zeroed-out pages
- Before process 1 modifies page C



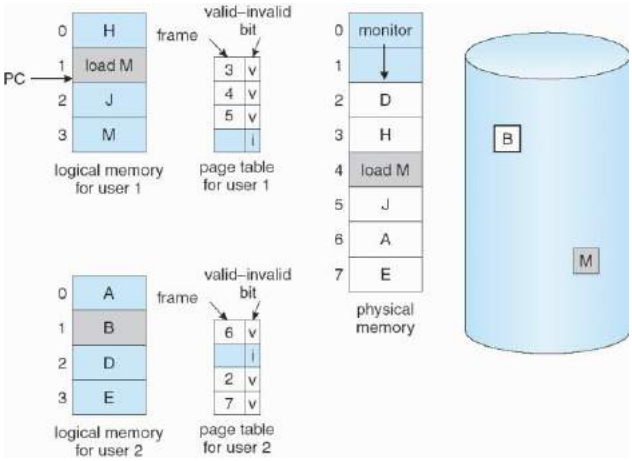
- After process 1 modifies page C there will also be a Copy of page C in physical memory (p.368 bottom)
- **If there is no free frame, the following happens**:
  - Page replacement - find some page in memory, but not really in use, swap it out
    - algorithm
    - performance - want an algorithm which will result in minimum number of page faults
  - Same page may be brought into memory several times

Page Replacement

- If we increase our degree of multiprogramming, we are over-allocating memory:
  - While a process is executing, a page fault occurs

- The hardware traps to the OS, which checks its internal tables to see that this page fault is a genuine one
- The OS determines where the desired page is residing on disk, but then finds **no free frames** on the free- frame list
- The OS then could:
    - Terminate the user process (Not a good idea)
    - Swap out a process, freeing all its frames, and reducing the level of multiprogramming
    - Perform page replacement

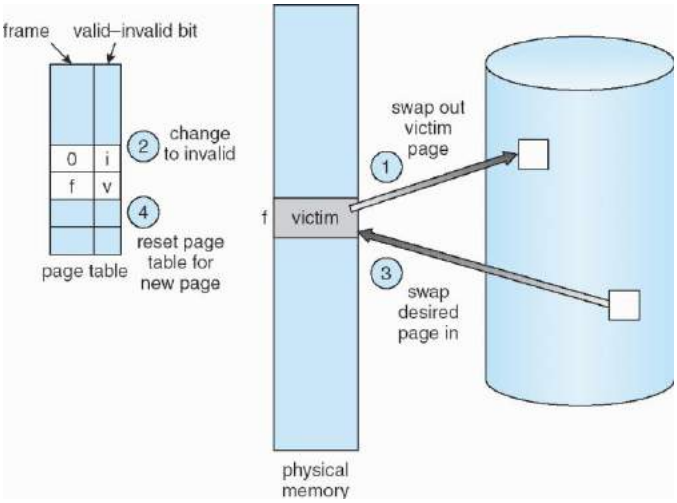
- The need for page replacement arises:



- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use **modify (dirty) bit** to reduce overhead of page transfers -only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory -large virtual memory can be provided on a smaller physical memory

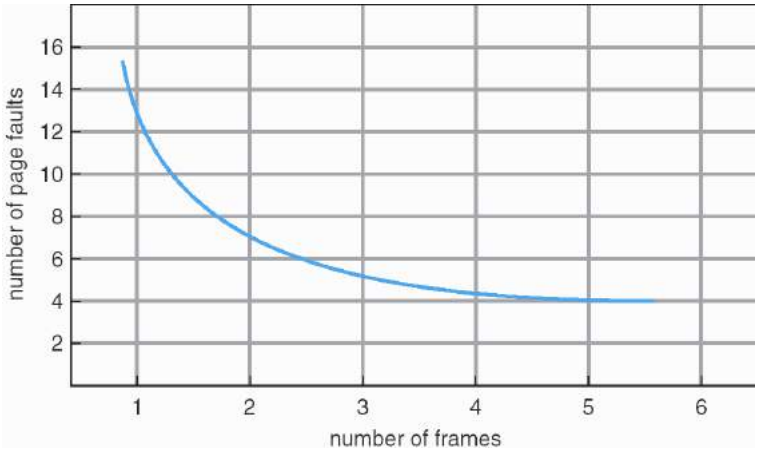
Basic Page Replacement

- Basic page replacement approach:
    - If no frame is free, we find one that is not being used and free it
- Page replacement takes the following steps:
    - Find the location of the desired page on the disk
    - Find a free frame:
        - If there is a free frame, use it, else
        - Select a **victim frame** with a page-replacement algorithm
        - Write the victim page to the disk and change the page & frame tables accordingly
    - Read the desired page into the (newly) free frame and change the page & frame tables
    - Restart the user process



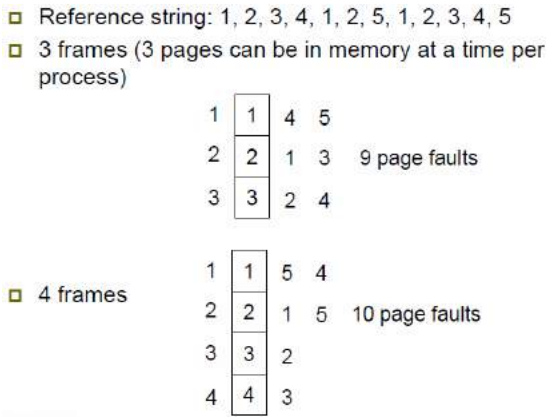
- **Note**: if no frames are free, two page transfers (one out & one in) are required, which doubles the page-fault service time and increases the effective access time accordingly
- We can reduce this overhead by using a **modify / dirty bit**:
  - When a page is modified, its modify bit is set
  - If the bit is set, the page must be written to disk
  - If the bit is not set, you don't need to write it to disk since it is already there, which reduces I/O time
- We must solve two major problems to implement demand paging:
  - Develop a frame-allocation algorithm
    - If we have multiple processes in memory, we must decide how many frames to allocate to each process
  - Develop a page-replacement algorithm
    - When page replacement is required, we must select the frames that are to be replaced
- When selecting a particular algorithm, we want the one with the lowest page-fault rate
- To evaluate an algorithm, run it on a reference string (a string of memory references) and compute the number of page faults
  - Want lowest page-fault rate
  - Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - In all our examples, the reference string is

    1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- You can **generate** reference strings or trace a **given system** and record the address of each memory reference
- Graph of page faults versus the number of frames:

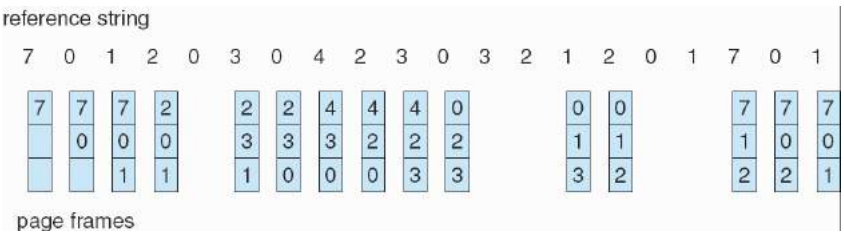


FIFO Page Replacement

- p.373 TB
- The simplest page-replacement algorithm is the first-in, first-out (FIFO) algorithm
    - A FIFO replacement algorithm associates with each page the time when that page was brought into memory
    - When a page must be replaced, the oldest page is chosen
    - Notice it is not strictly necessary to record the time when a page is brought in
    - We can create a FIFO queue to hold all pages in memory
    - We replace the page at the head of the queue
    - When a page is brought into memory, we insert it at the tail of the queue
- Easy to understand and implement
- **Example**:

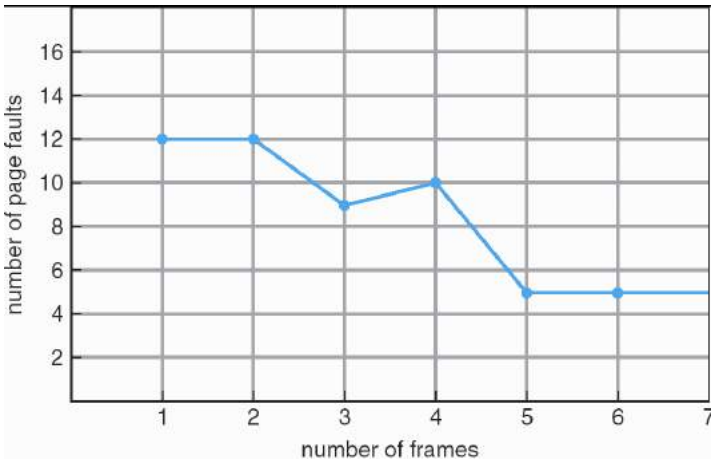


- FIFO page replacement algorithm:



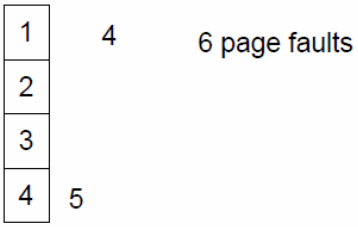
    - Yields 15 page faults
    -

- **Belady's anomaly**:
    - For some algorithms, the page-fault rate may **increase** as the number of allocated frames increases
- **FIFO illustration of Belady's Anomaly**:

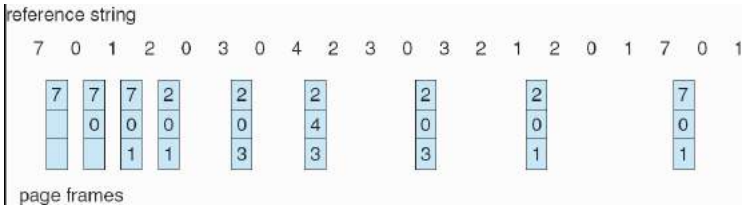


Optimal Page Replacement

- Optimal page replacement was found as a result of Belady's anomaly
- Guarantees the lowest possible page-fault rate for a fixed number of frames
- This algorithm exists and is called either OPT or MIN
- Difficult to implement because we require future knowledge of the reference string
    - Used mainly for comparative studies
- **Algorithm**:
    - Replace the page that **<u>will</u> not be used** for the longest period
- 4 frames **example:**
    - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



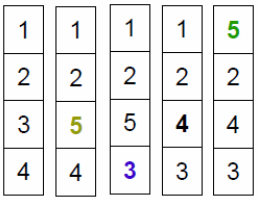
- How do you know this?
- Used for measuring how well your algorithm performs
- Difficult to implement, because you need future knowledge of the reference string
- Optimal Page replacement:



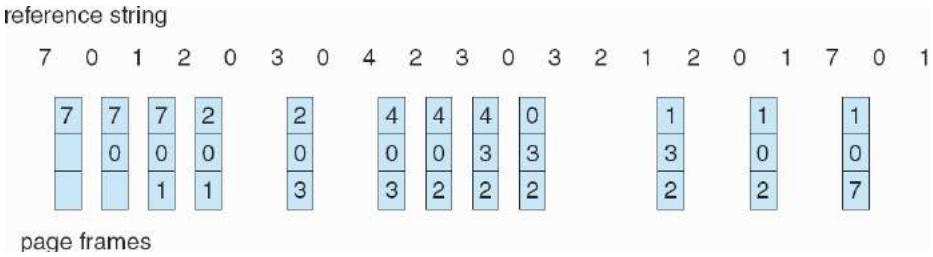
- Yields 9 page faults

LRU (least recently-used) Page Replacement

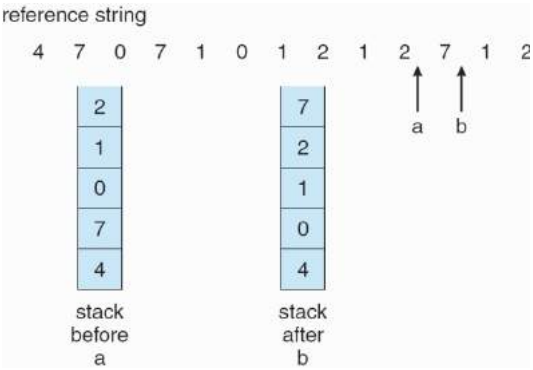
- An approximation of the optimal page replacement algorithm
- We use the recent past as an approximation of the near future
- Replace the page that **<u>has</u> not been used** for the longest period
  - Think of this algorithm as the backward looking optimal page-replacement algorithm
- **Example**:
  - Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**



- LRU Page Replacement:



  - Yields 12 page faults
- Two ways to determine an order for the frames defined by the time of last use:
  - Counters:
    - Each page-table entry has a time-of-use field and the CPU gets a logical clock / counter
    - Whenever a page is referenced, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page
  - Stack:
    - Whenever a page is referenced, it is removed from the stack and put on top
    - The bottom of the stack is the LRU page
- Use of a stack to record the most recent page references:



- **Neither optimal replacement nor LRU replacement suffers from Belady's anomaly**

LRU-Approximation Page Replacement

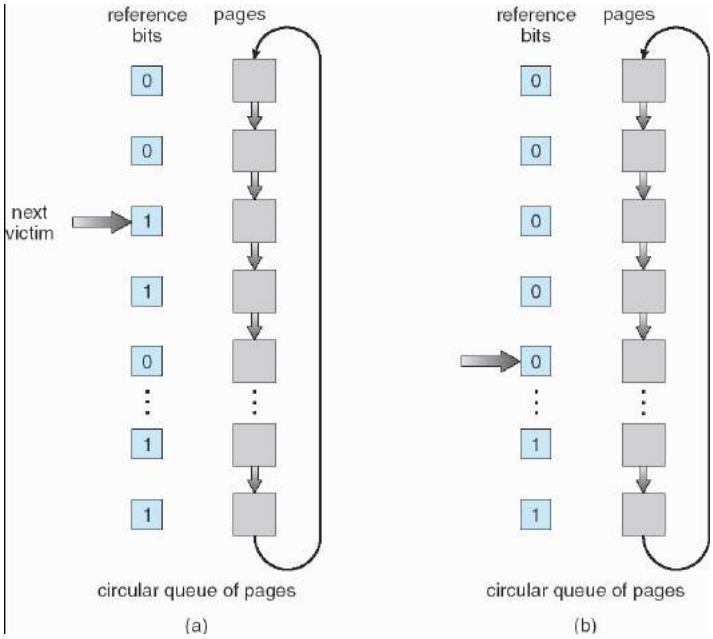
- **Reference bit:**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1 by hardware
  - Replace the one which is 0 (if one exists)
    - We do not know the order, however

Additional-Reference-Bits Algorithm

- You can gain additional ordering info by recording reference bits at regular intervals
- A 8-bit byte is used for each page in a table in memory
- This register is a right-shift register
  - Every 100 ms the pages are referenced and if the page was used then a 1 is moved into the MSB of the byte
- **Examples**:
  - 00000000 - This page has not been used in the last 8 time units (800 ms)
  - 11111111 - Page has been used every time unit in the past 8 time units
  - 11000100 has been used more recently than 01110111
  - These can be treated as unsigned integers and the page with the lowest value is the LRU page
  - If numbers are equal, FCFS is used

Second-Chance Algorithm

- Like the FIFO replacement algorithm, but you inspect the reference bit and replace the page if the value is 0
- If the reference bit is 1, that page gets a second chance, its reference bit is cleared (0), and its arrival time is reset
- A page that has been given a second chance will not be replaced until all other pages are replaced
- If a page is used often enough to keep its reference bit set, it will never be replaced
- **Second chance:**
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules
- Second-Chance (clock) Page-Replacement Algorithm:



Enhanced Second-Chance Algorithm

- Consider both the reference bit and the modify bit as an ordered pair:
  - (0,0) neither recently used nor modified – best to replace
  - (0,1) not recently used but modified – not quite as good
  - (1,0) recently used but clean – probably used again soon
  - (1,1) recently used and modified – probably used again soon and will need to be written to disk before being replaced

Counting-Based Page Replacement

- Keep a counter of the number of references to each page
- **LFU (least frequently used)** page-replacement algorithm
  - The page with the smallest count is replaced
- **MFU (most frequently used)** page-replacement algorithm
  - The page with the smallest count was probably just brought in and has yet to be used

Page-Buffering Algorithms

- If a page is read into a free frame from the pool **before** a victim frame is written out, the process can restart as soon as possible, without waiting for the victim page to be written out
- Whenever the paging device is idle, a modified page is selected and written to disk, increasing the probability that a page will be clean when selected for replacement
- You can **remember** which page was in each frame from the pool and reuse old pages directly if needed, before the frame is reused

Applications and Page Replacement

- Sometimes applications processing data knows better how to handle their own data then the general-purpose use of page replacement used by the OS
  - **Example**:
    - Databases

- Data warehouses perform massive sequential disk reads, followed by computations and writes
    - MFU would be more efficient than LFU

Allocation of Frames

- p.382 TB
- Each process needs *minimum* number of pages
- 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*
- Two major allocation schemes
    - fixed allocation
    - priority allocation

Minimum Number of Frames

- The instruction-set architecture defines the minimum number of frames that must be allocated
    - The maximum number is defined by the amount of available memory

Allocation Algorithms

- Equal allocation
    - Every process is given an equal share of frames
- Proportional allocation
    - Allocate memory to each process according to its size

Global versus Local Allocation

- Global replacement
    - A process can select a replacement frame from the whole set
    - A process may even select only frames allocated to **other** processes, increasing the number of frames allocated to it
    - Problem: A process can't control its own page-fault rate
    - Global replacement is the more common method since it results in greater system throughput
- Local replacement
    - A process selects only from its own set of allocated frames
    - The number of frames allocated to a process does not change

Non-Uniform Memory Access

- p.385 TB

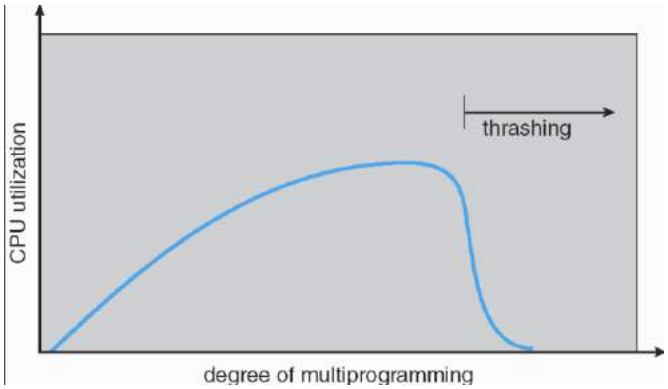
Thrashing

- If a process does not have "enough"pages, the page-fault rate is very high. This leads to:
    - low CPU utilization
    - operating system thinks that it needs to increase the degree of multiprogramming
    - another process added to the system

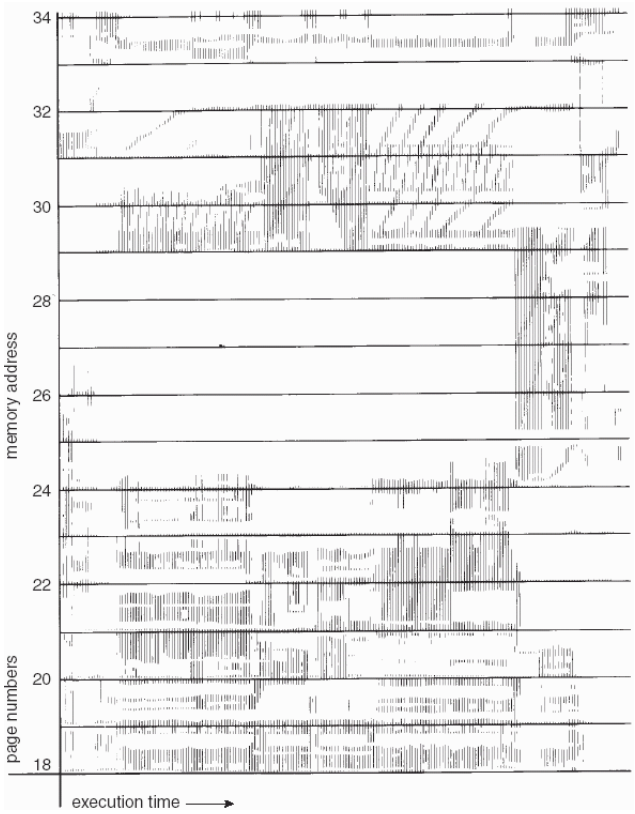
- If the number of frames allocated to a low-priority process falls below the minimum number required, it must be suspended
- A process is **thrashing** if it is spending more time paging than executing (E.g. its pages are **all in use**, and it must replace a page that will be needed again right away)

Cause of Thrashing

- The thrashing phenomenon:
  - As processes keep faulting, they queue up for the paging device, so CPU utilization decreases
  - The CPU scheduler sees the decreasing CPU utilization and **increases** the degree of multiprogramming as a result
  - The new process causes even more page faults and a longer queue!



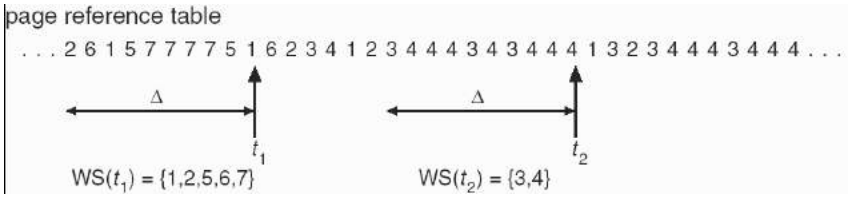
- We can limit the effects of thrashing by using a local replacement algorithm:
  - If one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash also
  - Pages are replaced with regard to the process of which they are a part
  - However, if processes are thrashing, the effective access time will increase even for a process that is not thrashing
- To prevent thrashing, we must give a process **enough frames**:
  - The **locality model** of process execution:
    - As a process executes, it moves from locality to locality
      - (A locality is a set of pages that are actively used together – a program may have overlapping localities)
    - If we allocate enough frames to a process to **accommodate its current locality**, it will fault for pages in its locality until they are all in memory, and it won't fault again until it changes localities
- Locality in a memory-reference pattern:



Working-Set Model

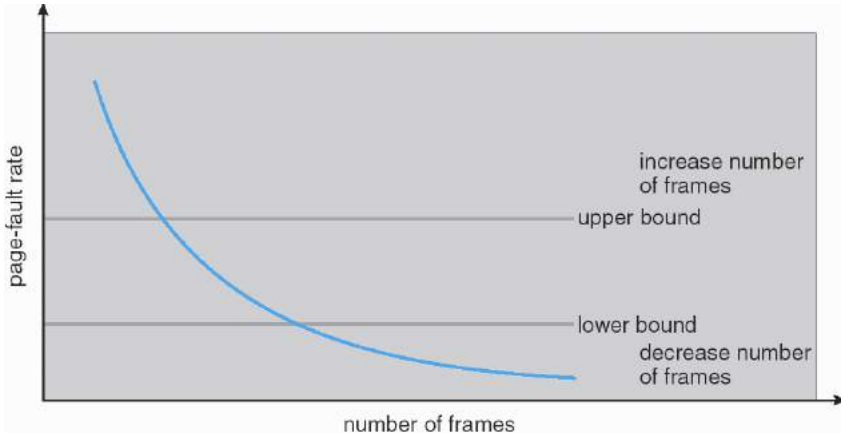
- Based on the assumption of locality
- A parameter, Δ, defines the working-set window
- Working set = set of pages in the most recent Δ page references
- If a page is in use, it will be in the working set, else it will drop from the working set Δ time units after its last reference
- The accuracy of the working set depends on the selection of Δ
    - If Δ is too small, it won't encompass the entire locality
    - If Δ is too large, it may overlap several localities
- The working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible
- **Working-Set Model**:
    - Δ ≡working-set window ≡a fixed number of page references

      Example: 10,000 instruction
    - *WSSi* (working set of Process *Pi*) = 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 Δ = ∞⇒ will encompass entire program
    - *D* = Σ *WSSi* ≡total demand frames
    - if *D* > *m* ⇒ Thrashing
    - Policy if *D* > m, then suspend one of the processes



Page-Fault Frequency

- This takes a more direct approach than the working-set model
- To prevent thrashing, control the page-fault rate:
    - When it is too high, we know the process needs more frames
    - When it is too low, the process has too many frames
- Establish upper & lower bounds on the desired page-fault rate
    - Allocate / remove frames if needed
- If the page-fault rate increases and no frames are available, select a process to suspend and re-distribute its freed frames



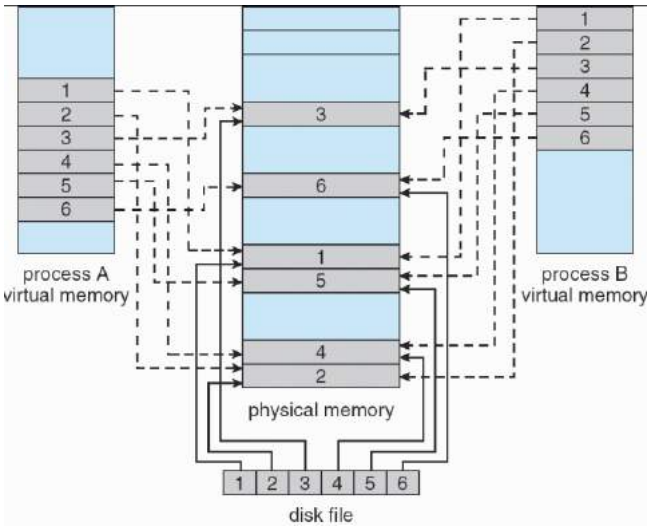
Memory-Mapped Files

- Memory-mapping a file allows a part of the virtual address space to be logically associated with a file
- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging
    - A page-sized portion of the file is read from the file system into a physical page
    - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies file access by treating file I/O through memory rather than `read()` `write()` system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared

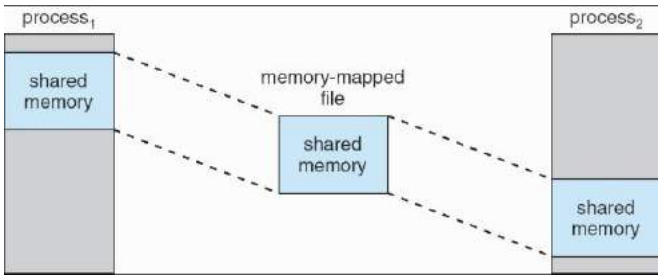
Basic Mechanism

- p.392 TB
- A disk block is mapped to a page in memory
- Initial access to the file proceeds using ordinary demand paging
- Page-sized portions of the file are read from the file system into physical pages
- Subsequent reads & writes to the files are handled as routine memory accesses, simplifying file access and usage

- File manipulation through memory incurs **less overhead** than read() and write() system calls
- Closing the file results in all the memory-mapped data being written back to disk and removed from the virtual memory
- **Memory Mapped files**:



- In many ways, the sharing of memory mapped files is similar to shared memory
- Processes can communicate using shared memory by having the communicating processes memory-map the same file into their virtual address spaces



Shared Memory in the Win32 API

- p.393 - 395 TB

Memory-Mapped I/O

- p.395 - 396 TB

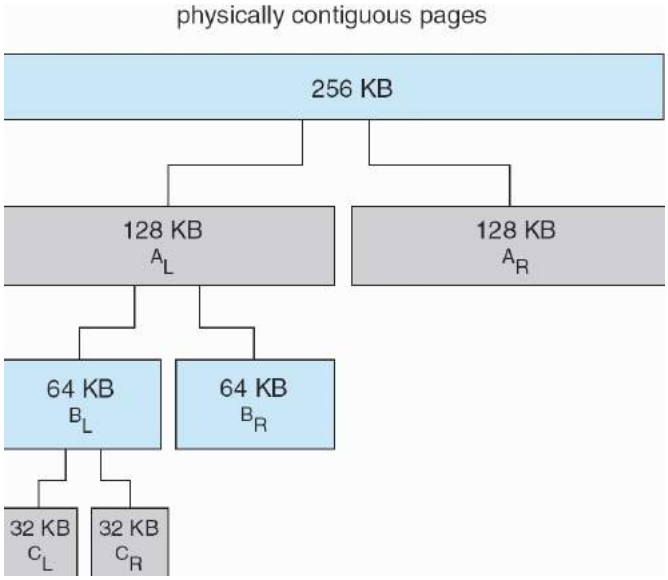
Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
    1. Kernel requests memory for structures of varying sizes
    2. Some kernel memory needs to be contiguous

Buggy System

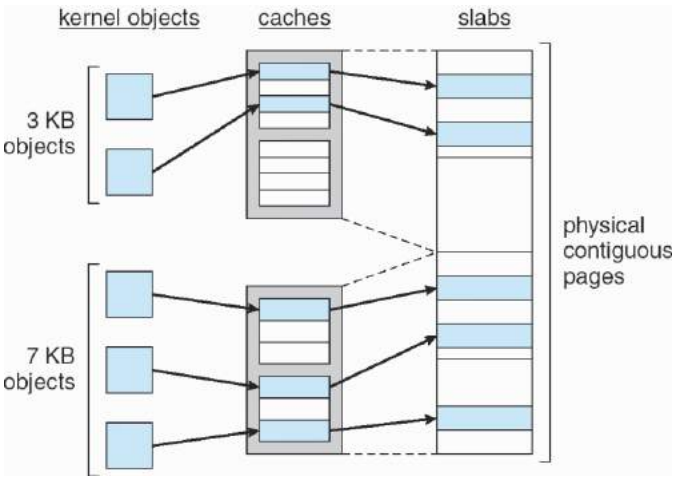
- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
    - Satisfies requests in units sized as power of 2
    - Request rounded up to next highest power of 2

- When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available



Slab Allocation

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
    - Each cache filled with **objects**-instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
    - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction



Other Considerations

Prepaging

- p.400 mid TB
- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume *s* pages are prepaged and *α* of the pages is used
    - Is cost of *s * α* save pages faults > or < than the cost of prepaging *s * (1-α)* unnecessary pages*?*
    - *α* near zero ⇒ prepaging loses

Page Size

Large page size	**Small page size**
A large page size decreases the number of pages – desirable since each active process must have its own copy of the page table	Memory is better utilized with smaller pages, since it minimizes internal fragmentation
With I/O transfer, latency and seek time dwarf transfer time, so a larger page size minimizes I/O time	Total I/O should be reduced, since locality will be improved
We must allocate and transfer not only what is needed, but also anything else in the page	Better resolution, allowing us to isolate only the memory that is actually needed, resulting in less I/O & less total allocated memory

- The trend is towards **larger** page sizes
- Page size selection must take into consideration:
    - fragmentation
    - table size
    - I/O overhead
    - locality

TLB Reach

- Hit ratio for the TLB = the % of virtual address translations that are resolved in the TLB rather than the page table
- To increase the hit ratio, increase the number of TLB entries
- However, this is both expensive and power-hungry
- TLB reach = the amount of memory accessible from the TLB (= TLB size x the page size)
- Ideally, the working set for a process is stored in the TLB
- If not, the process will spend a considerable amount of time resolving memory reference in the page table rather than TLB
- To increase the TLB reach, you can
    - increase the size of the page
        - May lead to an increase in fragmentation

- provide multiple page sizes
    - Requires the OS, not hardware, to manage the TLB
    - Managing the TBL in software raises performance costs

Inverted Page Tables

- Create a table that has one entry per physical memory page, indexed by the pair <process-id, page-number>
- Because they keep info about which virtual-memory page is stored in each physical frame, inverted page tables reduce the amount of physical memory needed to store this information
- The inverted page table no longer contains complete info about a process' logical address space, which demand paging requires
- For this information to be available, an external page table (one per process) must be kept
    - These tables are referenced only when a page fault occurs, so they don't need to be available quickly
    - They are paged in and out of memory as necessary

Program Structure

- Demand paging is designed to be transparent to the user program
- Sometimes, system performance can be improved if the user has an awareness of the underlying demand paging
- Careful selection of data structures and programming structures can increase locality and lower the page-fault rate
- E.g. a stack has good locality and a hash table has bad locality
    - The choice of programming language can also affect paging: C++ uses pointers which randomize access to memory a bad locality
- **Program Structure**:
    - `int[128,128] data;`
    - Each row is stored in one page
    - **Program 1**

```
for (j = 0; j <128; j++)
for (i = 0; i < 128; i++)
data[i,j] = 0;
```

        - 128 x 128 = 16,384 page faults
    - **Program 2**

```
for (i = 0; i < 128; i++)
for (j = 0; j < 128; j++)
data[i,j] = 0;
```

        - 128 page faults

I/O Interlock

- A **lock bit** is associated with every frame
- I/O pages are locked, and unlocked when the I/O is complete
- This is because I/O must not be paged out until end of transfer

- Another use for a lock bit involves normal page replacement:
    - To prevent replacing a newly brought-in page until it can be used at least once, it can be locked until used

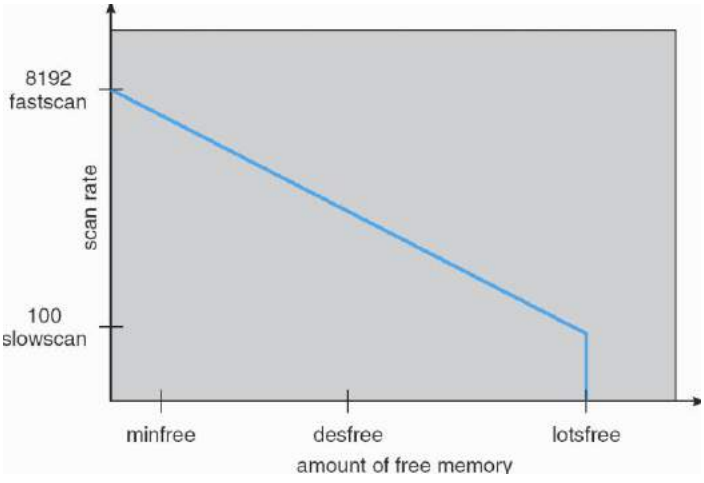
Operating-System Examples

Windows XP

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page.
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming**is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

Solaris

- Maintains a list of free pages to assign faulting processes
- *Lotsfree* - threshold parameter (amount of free memory) to begin paging
- *Desfree* - threshold parameter to increasing paging
- *Minfree* - threshold parameter to being swapping
- Paging is performed by *pageout* process
- Pageout scans pages using modified clock algorithm
- *Scanrate* is the rate at which pages are scanned
    - This ranges from *slowscan* to *fastscan*
- Pageout is called more frequently depending upon the amount of free memory available
- **Solaris 2 Page Scanner**:



Summary

Storage Management

- OS provides uniform, logical view of information storage
    - Abstracts physical properties to logical storage unit -**file**

- Each medium is controlled by device (i.e., disk drive, tape drive)
    - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)

File-System Management

- Computers can store info on several different types of physical media (e.g. magnetic disk, optical disk…)
- Each medium is controlled by a device (e.g. disk drive)
- The OS maps files onto physical media, and accesses these files via the storage devices
- File = a collection of related information
- The OS implements the abstract concept of a file by managing mass storage media and the devices that control them
- The OS is responsible for these file management activities:
    - Creating and deleting files
    - Creating and deleting directories
    - Supporting primitives for manipulating files & directories
    - Mapping files onto secondary storage
    - Backing up files on stable (non-volatile) storage media

PART FIVE: STORAGE MANAGEMENT

Chapter 10: File System

Objectives:

- To explain the function of file systems
- To describe the interfaces to file systems
- To discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures
- To explore file-system protection

File Concept

- File = a named collection of related info on secondary storage
- Data can't be written to secondary storage unless its in a file
- A file has a certain defined **structure** according to its type
    - **Text file**: sequence of characters organized into lines
    - **Source file**: sequence of subroutines & functions
    - **Object file**: sequence of bytes organized into blocks
    - **Executable file**: series of code sections
- Contiguous logical address space
- **Types**:
    - **Data**
        - numeric
        - character
        - binary

- **Program**

File Attributes

- **Name**: The only info kept in human-readable form
- **Identifier**: Unique tag
- **Type**: Info needed for those systems that support different types
- **Location**: Pointer to a device and to the location of the file
- **Size**: Current size (in bytes, words, or blocks)
- **Protection**: Access-control info
- **Time, date, & user id**: Useful for protection & usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk

File Operations

- **Creating a file**:
  - First, space in the file system must be found for the file
  - Then, an entry for the file must be made in the directory
- **Writing a file**:
  - Make a system call specifying both the name of the file and the info to be written to the file
  - The system must keep a write pointer to the location in the file where the next write is to take place
- **Reading a file**:
  - Use a system call that specifies the name of the file and where in memory the next block of the file should be put
  - Once the read has taken place, the read pointer is updated
- **Repositioning within a file**:
  - The directory is searched for the appropriate entry and the current-file-position is set to a given value
- **Deleting a file**:
  - Search the directory for the named file and release all file space and erase the directory entry
- **Truncating a file**:
  - The contents of a file are erased but its attributes stay
- Most of these file operations involve searching the directory for the entry associated with the named file
- To avoid this constant searching, many systems require that an 'open' system call be used before that file is first used
- The OS keeps a small table containing info about all open files
- When a file operation is requested, the file is specified via an index into the open-file table, so no searching is required
- When the file is no longer actively used, it is closed by the process and the OS removes its entry in the open-file table
- Some systems implicitly open a file when the first reference is made to it, and close it automatically when the program ends

- Most systems require that the programmer open a file explicitly with the 'open' system call before that file can be used
- A **per-process** table tracks all files that a process has open and includes access rights to the file & accounting info
- Each entry in the per-process table in turn points to a **system-wide** open-file table, which contains process- independent info, such as the file's disk location, access dates, and file size
- Information associated with an **open file**:
    - **File pointer**:
        - For the system to track the last read-write location
    - **File open count**:
        - A counter tracks the number of opens & closes and reaches zero on the last close
    - **Disk location of the file**:
        - Location info is kept in memory to avoid having to read it from disk for each operation
    - **Access rights**:
        - Each process opens a file in an access mode
- **Open file locking**:
    - Provided by some operating systems and file systems
    - Mediates access to a file
    - **Mandatory or advisory**:
        - **Mandatory**–access is denied depending on locks held and requested
        - **Advisory**–processes can find status of locks and decide what to do

File Types

- If an OS recognizes the type of a file, it can operate on the file in reasonable ways
- A common technique for implementing file types is to include the type as part of the file name
    - Name split into 2 parts:
        - name
        - extension

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

- The system uses the extension to indicate the type of the file and the type of operations that can be done on that file
    - Example:
        - Only a file with a .com, .exe, or .bat extension can be executed
        - .com and .exe are two forms of binary executable files
        - .bat file is a batch file containing, in ASCII format, commands to the operating system
- **MS-DOS** only recognizes a few files but application programs also use extensions to indicate file types they are interested in
    - Because application extensions are not supported by the operating system, they can be considered as "hints" to the applications that operate on them
- The **TOPS-20** operating system will automatically recompile an object program if the source code was modified or edited
    - This way the user always runs with an up-to-date object file
    - For this purpose the operating system must be able to discriminate the source file from the object file, to determine if and when it was modified
    - File extensions are used for this purpose
- In the **Mac OS X** operating system every file has a type, such as TEXT or APPL
    - Each file has a creator attribute that contain the name of the program that created it, set by the operating during the create() call - use is enforced by the system
- **UNIX** uses a crude magic number stored at the beginning of some files to indicate roughly the type of file, executable program; batch file (shell script), PostScript; etc
    - Not all files have magic numbers so system features cannot be based solely on this information
    - Does not record the name of the creating program
    - File extensions are meant mostly to aid users in determining what type of contents a file contains
    - Extensions can be used or ignored by applications
        - Up to the applications programmer

File Structure

- File types can indicate the internal structure of the file
- Disadvantage of supporting multiple file structures: large size
- All OSs must support at least one structure: an executable file
- The Mac OS file structure:
    - Resource fork (contains info of interest to the user)
    - Data fork (contains program code / data)
- Too few structures make programming inconvenient
- Too many structures cause OS bloat and programmer confusion

Internal File Structure

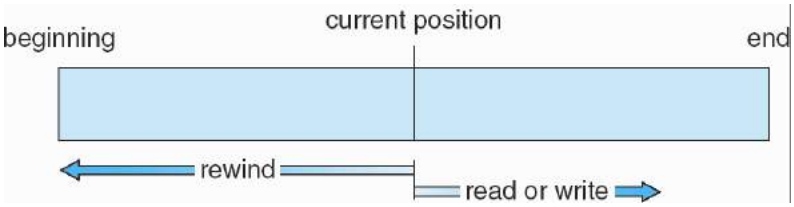
- **p.430 mid bot (make note on this piece)**
- Locating an offset within a file can be complicated for the OS
- Disk systems typically have a well-defined block size
- It is unlikely that the physical record size will exactly match the length of the desired logical record

- Packing a number of logical records into physical blocks is a common solution to this problem
- The logical record size, physical block size, and packing technique determine how many logical records are in each physical block
- The packing can be done either by the user's program or OS

Access Methods

Sequential Access

- Information in the file is processed in order
- The most common access method (e.g. editors & compilers)
- **Read**: reads the next file portion and advances a file pointer
- **Write**: appends to the end of file and advances to the new end



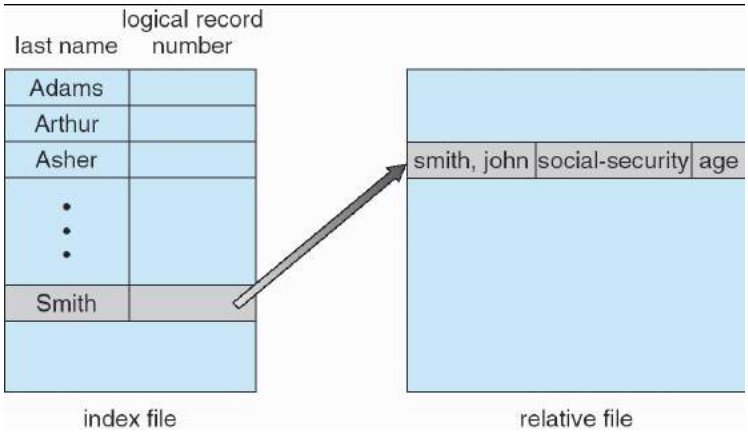
Direct Access

- A file is made up of fixed-length logical records that allow you to read & write records rapidly in no particular order
- File operations include a relative block number as parameter, which is an index relative to the beginning of the file
- The use of relative block numbers
    - allows the OS to decide where the file should be placed and
    - helps to prevent the user from accessing portions of the file system that may not be part of his file

sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

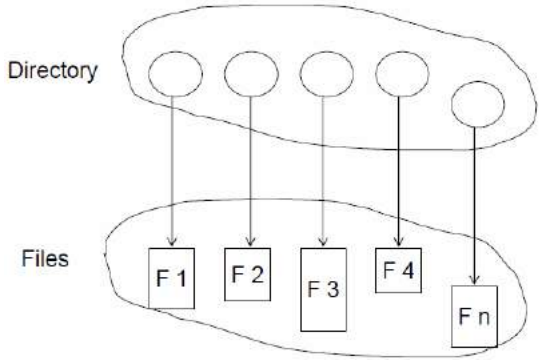
Other Access Methods

- These methods generally involve constructing a **file index**
- The index contains pointers to the various blocks
- To find a record in the file
    - First search the index
    - Then use the pointer to access the file directly and to find the desired record
- With large files, the index file itself may become too large to be kept in memory
- Solution: create an index for the index file

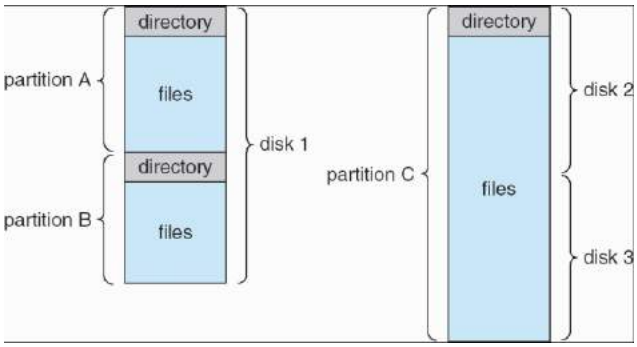


Directory and Disk Structure

- A collection of nodes containing information about all files



- Both the directory structure and the files reside on disk
- Backups of these two structures are kept on tapes
- **Typical File-System Organization**:



Storage Structure

- **p434 (make sure)**
- Organizing a lot of data can be done in two parts:
  - Disks are split into one or more partitions
  - Each partition contains info about files within it (e.g. name, location, size, type…) in a device directory

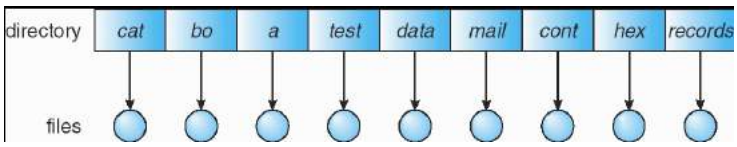
Directory Overview

- **Very Important p.435 (make sure)**
- Operations that can be performed on a directory:

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

Single-Level Directory

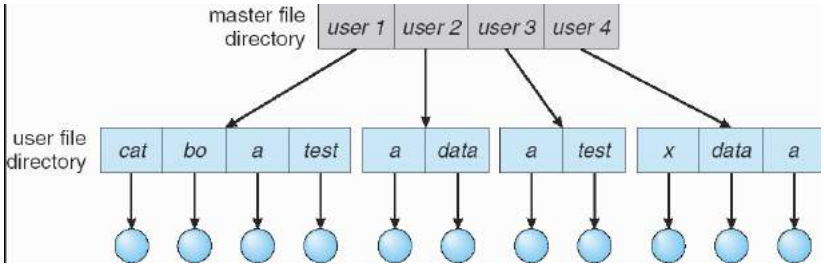
- All files are contained in the same directory
- Limitations because all files must have unique names
- A single directory for all users



- Naming problem
- Grouping problem

Two-Level Directory

- Separate directory for each user (UFD = user file directory)
- Each entry in the MFD (master file directory) points to a UFD
- Advantage: No filename-collision among different users
- Disadvantage: Users are isolated from one another and can't cooperate on the same task
- System files (e.g. compilers, loaders, libraries…) are contained in a special user directory (e.g. user 0) that is searched if the OS doesn't find the file in the local UFD
- Search path = directory sequence searched when a file is named



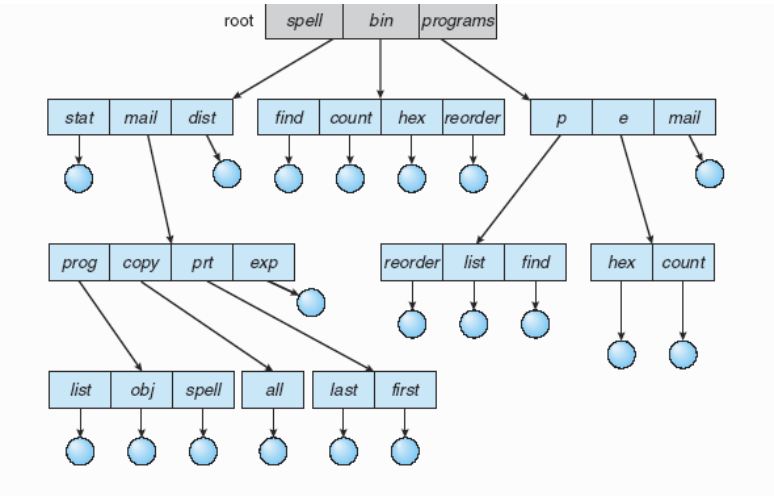
- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability

Tree-Structured Directories

- Users can create their own subdirectories and organize files
- **Absolute path names**: begin at the root
- **Relative path names**: define a path from the current directory
- To delete an empty directory:

- Just delete it

- To delete a non-empty directory:
  - First delete all files in the directory, or
  - Delete all that directory's files and subdirectories
- Efficient searching
- Grouping Capability
- Current directory (working directory)
  - cd /spell/mail/prog
  - type list

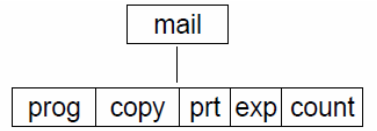


- **Absolute** or **relative** path name
- Creating a new file is done in current directory
- Delete a file

  rm <file-name>
- Creating a new subdirectory is done in current directory

  mkdir <dir-name>
- Example: if in current directory /mail

  mkdir count

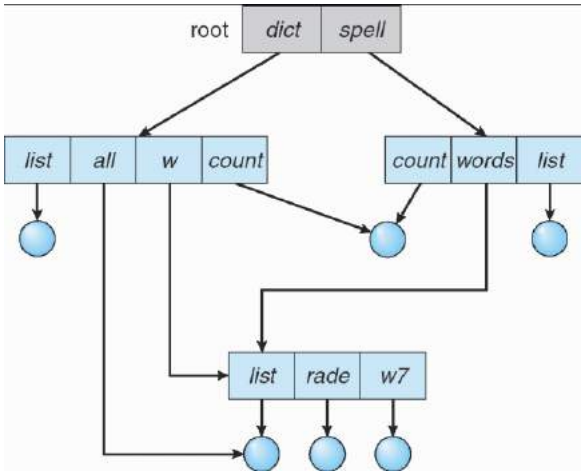


- Deleting "mail" ⇒ deleting the entire subtree rooted by "mail"

Acyclic-Graph Directories

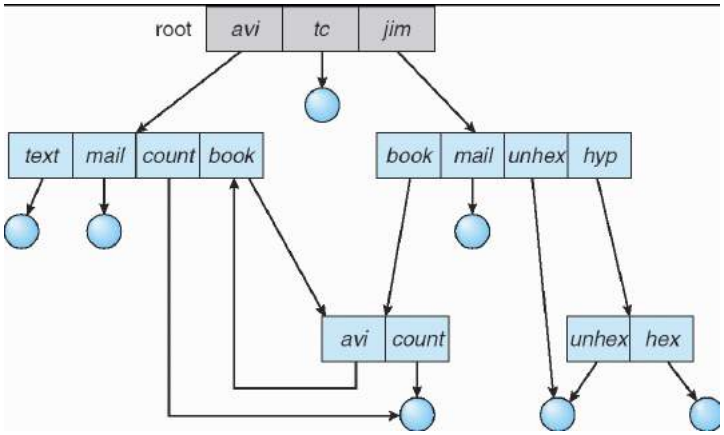
- Directories can have **shared** subdirectories and files
- Advantage: simple algorithms to traverse the graph
- Only one file exists, so changes made by one person are immediately visible to the other
- Ways to implement shared files / subdirectories:
  - Create a new directory entry called a **link**, which is a pointer to another file / subdirectory

    - Duplicate all info about shared files in both sharing directories
- Problems:
    - A file may now have multiple absolute path names
    - Deletion may leave dangling pointers to a non-existent file
- Solutions to deletion problems:
    - Backpointers, so we can delete all pointers
    - Variable size records a problem
    - Backpointers using a daisy chain organization
    - Entry-hold-count solution
- Approaches to deletion:
    - With symbolic links, remove only the link, not the file. If the file itself is deleted, the links are left dangling and can be removed / left until an attempt is made to use them
    - Preserve the file until all references to it are deleted. A mechanism is needed to determine that the last reference to the file has been deleted. Problem: potentially large size of the file- reference list
- New directory entry type
- **Link**-another name (pointer) to an existing file
- **Resolve the link**-follow pointer to locate the file



General Graph Directory

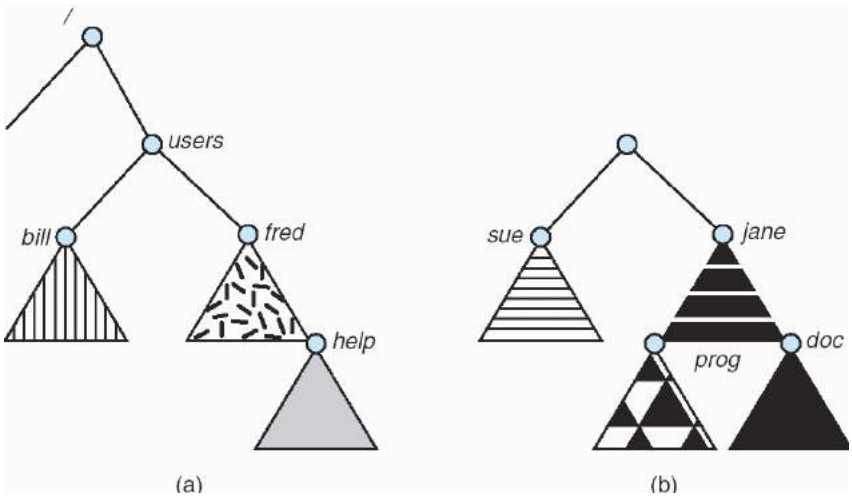
- Can have cycles: links are added to an **existing** directory
- If there are cycles, we want to avoid searching components twice
    - Solution: limit the no of directories accessed in a search
- Similar problem when determining when a file can be deleted:
    - With cycles, the reference count may be nonzero, even when it is no longer possible to refer to a directory / file (This anomaly results from the possibility of self-referencing in the directory structure)
    - **Garbage collection** is needed to determine when the last reference has been deleted, only because of possible cycles



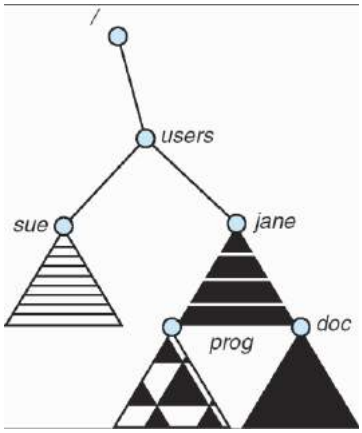
- How do we guarantee no cycles?
  - Allow only links to file not subdirectories
  - Garbage collection
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK

File-System Mounting

- A file system must be mounted before it can be available to processes on the system
- Mount procedure:
  - The OS is given the name of the device and location within the file structure at which to attach the file system
  - The OS verifies that the device contains a valid file system
  - The OS notes in its directory structure that a file system is mounted at the specified mount point
- **Existing (a) ; Unmounted Partition (b)** :



- **Mount Point**:



File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection** scheme
- On distributed systems, files may be shared across a network
- Network File System (NFS) is a common distributed file-sharing method

Multiple Users

- File sharing
    - The system can allow a user to access the files of other users by default, or
    - It may require that a user specifically grant access
- To implement sharing & protection, the system must maintain more file & directory attributes than on a single-user system
- Most systems use concepts of file/directory owner and group
- Owner = the user who may change attributes, grant access, and has the most control over the file / directory
    - Most systems implement owner attributes by managing a list of user names and user IDs
- Group = the attribute of a file that is used to define a subset of users who may share access to the file
    - Group functionality can be implemented as a system-wide list of group names and group IDs
- The owner and group IDs of a file / directory are stored with the other file attributes, and can be used to allow / deny ops

Remote File Systems

- Uses networking to allow file system access between systems
    - Manually via programs like FTP
    - Automatically, seamlessly using **distributed file systems**
    - Semi automatically via the **world wide web**

The Client-Server Model

- The server specifies which resources (files) are available to which clients
- **Client-server** model allows clients to mount remote file systems from servers
    - Server can serve multiple clients

- Client and user-on-client identification is insecure or complicated
- **NFS** is standard UNIX client-server file sharing protocol
- **CIFS** is standard Windows protocol
- Standard operating system file calls are translated into remote calls

Distributed Information Systems

- Provide unified access to info needed for remote computing
- Distributed Information Systems **(distributed naming services)** such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing
- DNS provides host-name-to-network-address translations

Failure Modes

- RAID can prevent the loss of a disk
- Remote file systems have more failure modes because the network can be interrupted between two hosts
- Protocols can enforce delaying of file-system operations to remote hosts, for when the host becomes available again
- Remote file systems add new failure modes, due to network failure, server failure
- Recovery from failure can involve state information about status of each remote request
- Stateless protocols such as NFS include all information in each request, allowing easy recovery but less security

Consistency Semantics

- **Consistency semantics** represent an important criterion of evaluating file systems that supports file sharing
    - These semantics specify how multiple users of a system are to access a shared file simultaneously
    - In particular, they specify when modifications of data by one user will be observed by other users
    - These semantics are typically implemented as code with the file system
- Consistency semantics are directly related to the process-synchronization algorithms of chapter 6
    - However, the complex algorithms of that chapter tend not to be implemented in the case of file I/O because of the great latencies and slow transfer rates of disks and networks
    - For example, performing an atomic transaction to a remote disk could involve several network communications, several disk reads and writes, or both
    - Systems that attempt such a full set of functionalities tend to perform poorly
    - A successful implementation of complex sharing semantics can be found in the Andrew file system
- For the following discussion, we assume that a series of file accesses (that is reads and writes) attempted by a user to the same file is always enclosed between the open() and close() operations
    - The series of accesses between the open() and close() operations make up a file session
    - To illustrate the concept, we sketch several prominent examples of consistency semantics

UNIX Semantics

- Unix file system (UFS) uses the following consistency semantics:
    - Writes to an open file by a user are visible immediately to other users who have this file open
    - One mode of sharing allows users to share the pointer of current location into a file
        - Thus, the advancing of the pointer by one user affects all sharing users

        - Here, a file has a single image that interleaves all accesses, regardless of their origin
- In the UNIX semantics, a file is associated with a single physical image that is accessed as an exclusive resource
- Contention for this single image causes delays in user processes

Session Semantics

- Andrew File System (AFS) uses the following consistency semantics
    - Writes to an open file by a user are not visible immediately to other users that have the same file open
    - Once a file is closed, the changes made to it are visible only in sessions starting later
        - Already open instances of the file do not reflect these changes
- According to these semantics, a file may be associated temporarily with several (possibly different) images at the same time
    - Consequently, multiple users are allowed to perform both read and write accesses concurrently on their images of the file, without delay
    - Almost no constraints are enforced on scheduling accesses

Immutable-Shared-Files Semantics

- A unique approach is that of immutable shared files
    - Once a file is declared as shared by its creator, it cannot be modified
    - An immutable file has two key properties:
        - Its name may not be reused
        - Its contents may not be altered
    - Thus, the name of an immutable file signifies that the contents of the file are fixed
- The implementation of these semantics in a distributed system is simple, because the sharing is disciplined (read- only)

Protection

- File owner/creator should be able to control:
    - what can be done
    - by whom

Types of Access

- Systems that don't permit access to other users' files don't need protection so protection can provided by prohibiting access
- This is too extreme, so controlled access is needed:
    - Limit the types of file access that can be made
    - You can control operations like Read, Write, Delete, List…
- Types of access:
    - **Read**
    - **Write**
    - **Execute**
    - **Append**

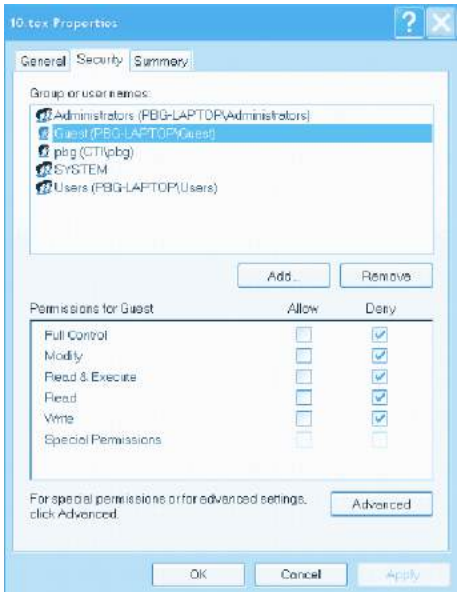
- **Delete**
- **List**

Access Control

- The most common approach to the protection problem is to make access dependent on the identify of the user
- Files can be associated with an access-control list (ACL) specifying the user name and the types of access allowed for each user
- Problems:
  - Constructing a list can be tedious
  - The directory entry now needs to be of variable size, resulting in more complicated space management
- These problems can be resolved by combining access control lists with an 'owner, group, universe' access-control scheme
  - To condense the length of the access-control list, many systems recognize three classifications of users in connection with each:
    - Owner
      - The user who created the file is the owner
    - Group
      - A set of users who are sharing the file and need similar access is a group, or work group
    - Universe
      - All other users in the system constitute the universe
  - **Samples**:

			RWX
a) **owner access**	7	⇒	1 1 1
b) **group access**	6	⇒	RWX 1 1 0
c) **public access**	1	⇒	RWX 0 0 1

  - E.g. rwx bits indicate which users have permission to read/write/execute
- **Windows XP Access-control list management**:



- **A Sample UNIX directory listing**:

-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

Other Protection Approaches

- A password can be associated with each file
- Disadvantages:
  - The no of passwords you need to remember may become large
  - If only one password is used for all the files, then all files are accessible if it is discovered
  - Commonly, only one password is associated with all of the user's files, so protection is all-or-nothing
- In a multilevel directory structure, we need to provide a mechanism for **directory protection**
- The directory operations that must be protected are different from the file operations:
  - Control creation & deletion of files in a directory
  - Control whether a user can determine the existence of a file in a directory (i.e. the 'dir' command in DOS)

Summary

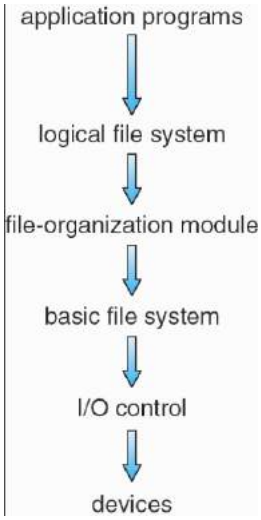
Chapter 11: Implementing File System

Objectives:

- To describe the details of implementing local file systems and directory structures
- To describe the implementation of remote file systems
- To discuss block allocation and free-block algorithms and trade-offs

File-System Structure

- Disks provide the bulk of secondary storage on which a file system is maintained
    - They have two characteristics that make them a convenient medium for storing multiple files:
        - They can be rewritten in place
            - Can read a block from disk, modify the block, and write it back into the same place
        - A disk can access directly any block of information it contains
            - Thus, it is simple to access any file either sequentially or randomly , and switching from one file to another requires only moving the read-write heads and waiting for the disk to rotate
- To improve I/O efficiency, I/O transfers between memory and disk are performed in units of **blocks**
    - Each block has one or more sectors
    - Depending on the disk drive, sector size varies from 32 bytes to 4096 bytes
    - The usual size is 512 bytes
- File systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily
- Design problems of file systems:
    - Defining how the file system should look to the user
    - Creating algorithms & data structures to map the logical file system onto the physical secondary-storage devices
- The file system itself is generally composed of many different levels
    - Every level in the design uses the features of lower levels to create new features for use by higher levels
- **A layered File System**:



- Levels of the file system:
    - **I/O Control** (lowest level)
        - Consists of **device drivers** & interrupt handlers to transfer info between main memory & disk
    - **Basic file system**

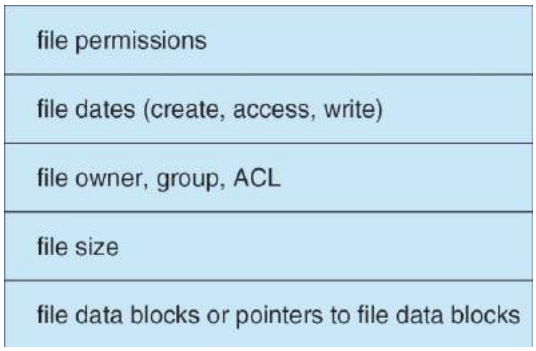
        - Need only issue generic commands to the appropriate device driver to read & write blocks on the disk
    - **File-organization module**
        - Knows about files and their logical & physical blocks
        - Translates logical block address to physical ones
    - **Logical file system**
        - Manages metadata information
        - Manages the directory structure
        - Maintains file structure via **file control blocks (FCB)**
    - **Application programs**

File-System Implementation

- Here we delve into the structures and operations used to implement file-system operations, like the open() and close() operations

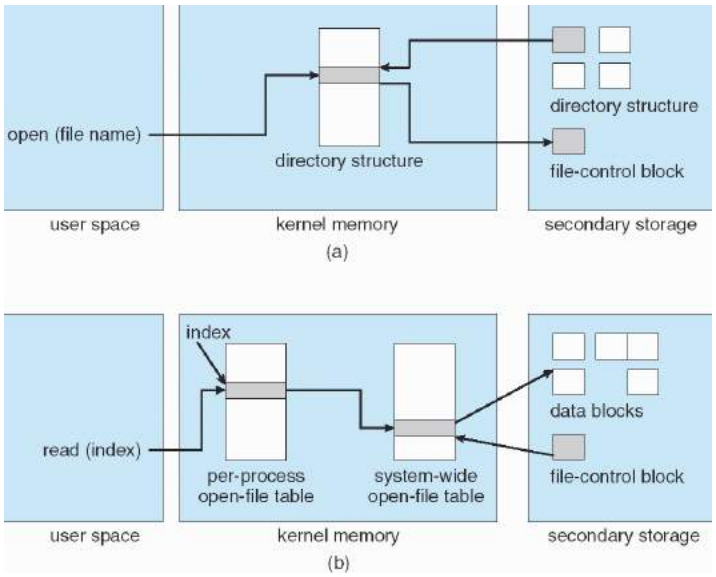
Overview

- On-disk & in-memory structures used to implement a file system:
- On-disk structures include:
    - A **boot control block**
        - Contains info needed to boot an OS from that partition
    - A **partition control block**
        - Contains partition details, like the no of blocks in the partition, size of the blocks, free-block count…
    - A **directory structure**
        - Used to organize the files
    - An **FCB (file control block)**
        - Contains file details, e.g. ownership, size…
- **A Typical File Control Block**:



- In-memory structures is used for both file-system management and performance improvement via caching:
    - An **in-memory mount table**
        - Contains information about each mounted partition
    - An **in-memory directory structure**

        - Holds directory info of recently accessed directories
    - The **system-wide open-file table**
        - Contains a copy of the FCB of each open file
    - The **per-process open-file table**
        - Contains a pointer to the appropriate entry in the system-wide open-file table
    - **Buffers** hold file-system blocks when they are being read from disk or written to disk
- **In-Memory File System Structure: (a) File open, (b) File read**



- To create a new file, a program calls the logical file system (LFS)
- The 'LFS' knows the format of the directory structures
- To **create** a new file, it
    - Allocates a new FCB
    - Reads the appropriate directory into memory
    - Updates it with the new file name and FCB
    - Writes it back to the disk
- After a file has been created, it can be used for I/O
    - First the file must be opened
    - FCB: copied to a system-wide open-file table in memory
    - An entry is made in the **per-process** open-file table, with a pointer to the entry in the **system-wide** open- file table
    - The open call returns a pointer to the appropriate entry in the per-process file-system table
    - All file operations are then performed via this pointer
    - When a process closes the file
        - The per-process table entry is removed
        - The system-wide entry's open count is decremented

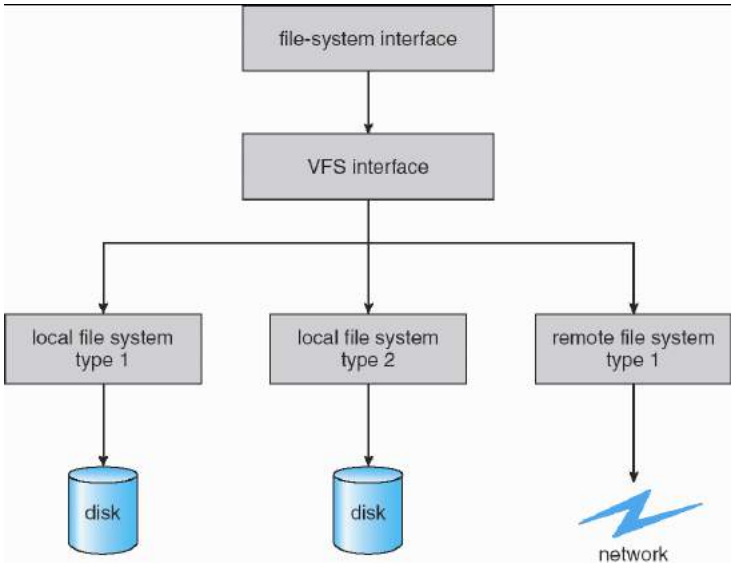
Partitions and Mounting

- Disk layouts:

    - A disk can be sliced into multiple partitions, or
    - A partition can span multiple disks (RAID)
- Each partition can either be:
    - Raw (containing no file system), or
    - Cooked (containing a file system)
- Boot info can be stored in a **separate partition** with its own format, since at boot time the system doesn't have file- system device drivers loaded and can't interpret the file-system format
    - Boot info format:
        - A sequential **series of blocks**, loaded as an **image** into memory, and execution of the image starts at a predefined location, such as the first byte
- The root partition containing the kernel is mounted at boot time
- Other partitions can be mounted at boot time, or later, manually
- As part of a successful mount operation, the OS verifies that the device contains a valid file system
- The OS notes in its in-memory mount table structure that a file system is mounted, and the type of the file system

Virtual File Systems

- The OS allows **multiple types** of file systems to be integrated into a directory structure (and be distinguished by the VFS!)
- The file-system implementation consists of 3 major layers:
    - File-system interface
    - Virtual File System (VFS) interface (serves 2 functions):
        - Separates file-system-generic operations from their implementation by defining a clean VFS interface
        - The VFS is based on a file-representation structure (vnode) that contains a numerical designator for a network-wide unique file, to support NFS
    - Local file system
- Schematic view of a virtual file system:



Directory Implementation

- The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system
- Here we look at the trade-offs involved in choosing one of these algorithms

Linear List

- The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks
    - A linear list of file names has pointers to the data blocks
    - Requires a linear search to find a particular entry
    - Simple to program, but time-consuming to execute (linear search)
- A cache can store the most recently used directory information
    - A sorted list allows a binary search and decreases search times

Hash Table

- A linear list stores the directory entries, but a hash data structure is also used
- The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list
    - Some provision must be made for collisions
        - Situation in which two file names hash to the same location
- **Disadvantages**:
    - Fixed size of hash table and the dependence of the hash function on that size

Allocation Methods

- The direct-access nature of disks allows us flexibility in the implementation of files
    - In almost every case, many files are stored on the same disk
    - The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly
    - Three major methods of allocating disk space are in wide use:
        - **Contiguous**
        - **Linked**
        - **Indexed**
    - Each method has advantages and disadvantages
    - Some systems support all three (Data General's RDOS for its Nova line of computers)
    - More commonly, a system uses one method for all files within a file-system type

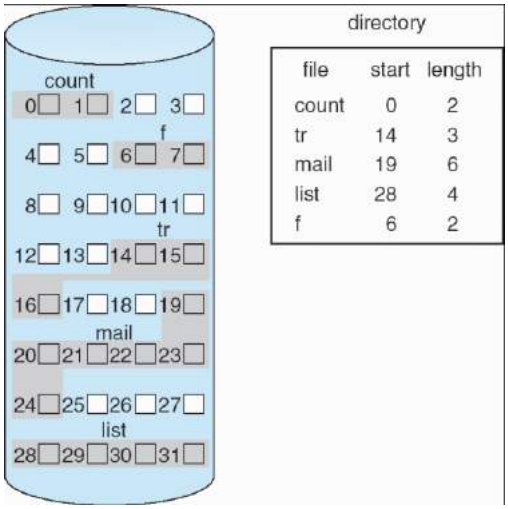
Contiguous Allocation

See also: Memory Allocation

Read p.472 mid NB!!!!!

- Each file occupies a set of contiguous blocks on the disk
- Disk addresses define a linear ordering on the disk
- The number of disk seeks required for accessing contiguously allocated files is minimal, as is seek time

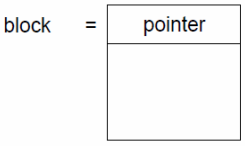
- Both sequential and direct access can be supported
- Problems with contiguous allocation:
    - Finding space for a new file
        - External fragmentation can occur
    - Determining how much space is needed for a file
        - If you allocate too little space, it can't be extended
        - If you allocate too much space, it may go unused
    - To minimize these drawbacks:
        - A contiguous chunk of space can be allocated initially, and then when that amount is not large enough, another chunk of contiguous space (an '**extent**') is added
- **Contiguous Allocation of Disk Space**:



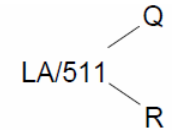
- An **extent** is a contiguous block of disks
    - Extents are allocated for file allocation
    - A file consists of one or more extents

Linked Allocation

- Each file is a linked list of disk blocks (scattered anywhere)



- The **directory contains a pointer to the first & last blocks**
- Each block contains a pointer to the next block
    - **Mapping**:



    - Block to be accessed is the Qth block in the linked chain of blocks representing the file
    - Displacement into block = R + 1