

# Storage Management (Ch 10-13 of S&G)

# Disk Scheduling

- Algorithm Objective
  - minimize seek time
- Assumptions
  - single disk module
  - requests for equal sized blocks
  - random distribution of locations on disk
  - one movable arm
  - seek time proportional to tracks crossed
  - no delay due to controller
  - read/write times are equal

- measures of response
  - mean wait time
  - variance of wait time
  - throughput
- measures of load
  - number of requests per second
  - average queue length

# Policies

- FCFS
  - fair
  - OK for small loads, saturated quickly
  - high mean wait time
  - low variance
  - leads to wide swings in seeks leading to high mean wait time

- SSTF (Shortest seek time first)
  - analogous to SJF for CPU scheduling
  - good throughput
  - saturates slowest
  - high variance
  - possibility of starvation

- SCAN (elevator algorithm)
  - start at one end, move toward the other end
  - service requests on the way
  - reverse and continue scanning
  - lower variance than SSTF, similar mean waiting time
  - requests just in front versus behind the direction of head motion?

- C-SCAN
  - treat disk as if it were circular
  - scan in one direction, retract, and resume scan
  - leads to more uniform waiting time than SCAN
- Look
  - same as SCAN but reverse direction if no more requests in the scan direction
  - leads to better performance than SCAN
- C-Look
  - circular look

- N-Step SCAN
  - two queues
    - *active* (N requests or less)
    - *latent*
  - service *active* queue
  - when no more in *active*, transfer N requests from *latent* to *active*
  - leads to lower variance compared to SCAN
  - worse than SCAN for mean waiting time

# Algorithm Selection

- based on load
- SCAN and C-SCAN best for heavy loads

# Other Considerations

- Minimize rotational latency
  - primarily for fixed head disks
    - swap disks
    - not very relevant today since seek times have shrunk due to the disk size shrink
  - policies
    - FCFS
    - SLTF (or *sector* queuing)
      - could also be used for regular disks on top of SCAN

- Minimize controller contention
  - assumption
    - one controller
    - many disk units
    - only one transfer at a time
  - policy
    - minimum latency scheduling (requires hardware position sensing)

# Storage Abstraction

- *process* is the OS abstraction for *processor*
- *data structures* is the language abstraction for *memory*
- consider a *program*
  - *text*, and *data structures* are meaningful for the life of the program (i.e. when it is executing as a *process*)
  - what about *state* that a program wants to leave behind *after* execution as a process *terminates*?
    - *magnetic/optical* store on the hardware side
    - an *abstraction* of this hardware presented by the OS

# I/O Buffer Management

- User or OS?
- User
  - pros
    - avoid extra copying
    - application has better control since it knows number and size of I/O buffers
  - cons
    - complexity to application software
    - OS more appropriate for global resource management
    - OS may be able to avoid copying by *locking* pages in memory

# Device independent I/O

- What abstraction should the OS provide?
  - virtual devices with generic API
    - open, seek, read, write, close
  - advantages
    - programming API is device independent
    - easy to add new devices
- step towards such a device independent I/O
  - file systems...

# File System

- File: A Collection of info with attributes
  - data (the actual user info)
  - attributes: name, access rights, physical representation, logical representation, accounting info, ...
- where are the attributes?
  - can be part of the file, and/or
  - physical representation kept in directory

- file names
  - single level (Univac EXEC 8, circa late 1970's)
    - unique name for each file...restrictive
    - single directory to search...expensive
  - two-level (DEC TOPS-10, circa early 80's)
    - top level: user/project
      - directory tells how to get to an individual user/project
    - second level: file name unique to user/project
  - hierarchical (Multics, Unix, NT)
    - a file name made of several parts (/users/r/rama/foo)
    - each part unique with respect to previous parts
    - directory?
      - just another file that contains info about files at next lower levels

- other naming issues
  - aliases
    - what is common and what is distinct?
      - data? attributes?
    - deletion of a file?
    - recursive aliases
  - file name extensions
    - mandatory (TOPS 10)
    - optional (Unix, NT)
  - version numbers
    - write implicitly creates new versions
    - write over-writes previous version (Unix, NT)

# Access Rights

- permissions on a file
  - read, write, execute, delete, change permissions
  - when set?
- change permissions
  - who can? Who should?
- storing access rights
  - one per user per file: feasible?
  - Unix?
    - restrictive
  - NT?
    - most general

# Physical Representation of File

- Goals
  - fast sequential access (SA)
  - fast random access (RA)
  - ability to grow the file
  - easy allocation of storage
- data structure in the file system
  - a table that maps file name to disk address
  - free list of disk blocks
  - where should these be?
    - contrast this to VM which also needs a mapping of VM page to disk address

# Allocation Strategies

- 1. Fixed contiguous regions
  - regions
    - one track
    - one cylinder
    - contiguous range of cylinders
  - characteristics
    - data structures (on the disk at a well-known place)
      - free list: bit map of free cylinders
      - directory: mapping table (file name to cylinder address)
    - SA and RA quick; allocation is quick as well
    - cannot grow file size above allocaton :-(

- 2. Contiguous regions with overflow areas
  - secondary area for growth (also contiguous)
  - characteristics
    - RA requires some computation; SA as fast as (1)
- 3. Linked allocation
  - file divided into (sector sized) blocks
  - each block points to next one; disk becomes a giant linked list!
  - characteristics
    - data structures (on the disk at a well-known place)
      - *free-list* (a linked list); mapping table (file name to starting disk block)
    - SA slow due to seek time; RA: pointer chasing
    - growth easy; allocation is expensive

- 4. File Allocation Table (FAT) MS-DOS
  - at the beginning of each partition, a table that contains one entry for each disk block in that partition (0 indicates it is free)
  - a file occupies a number of entries in the FAT
  - each entry points to the next entry in the FAT for SA (-1 indicates it is the last block)
  - characteristics
    - FAT is the data structure; efficient for allocation; less chance of screwing up as in (3)
    - SA requires FAT lookup (can be alleviated by caching the FAT)
    - RA requires some computation
    - limit on size of partition (this is BAD!); growth easy

- 5. Indexed Allocation
  - file represented by an index block (on the disk), a table of disk blocks for the file
  - characteristics
    - data structures
      - mapping table (file name to index block)
      - free list (can be a bit map of available disk blocks)
    - limit on the size of the file
- 6. Multilevel Indexed Allocation
  - file represented by an index block (on the disk), an indirection table of disk blocks for that file
    - one level indirection
    - two level indirection
    - triple indirection

- 7. Hybrid (BSD Unix)
  - combination of (5) and (6)
  - each file represented by an i-node (index node)
    - index to first ‘n’ disk blocks, plus
    - a single indirect index, a double indirect index, a triple indirect index
  - characteristics
    - SA requires i-node reference
      - overhead reduced by in-memory cache
    - RA requires some computation but much quicker than (3)
    - growth is easy
    - allocation overhead same as (3)

# Logical Representation

- FS may treat a file as an un-interpreted stream of bytes (Unix approach)
- structured files of user-defined record (not in Unix or NT)
  - only feasible if FS implemented at user level
- what happens when you edit a file?
  - Growth of a file is it always at the end of the file?
  - How does the storage system handle insertions in the middle of a file?

# Robustness

- How to handle machine crashes?
  - duplicate critical blocks (i-nodes) on disk
  - check for agreement on restart (*fsck* in Unix)
  - on write
    - prepare new blocks on disk
    - prepare intention to change i-node on disk
    - change i-node on the disk
    - get rid of intention list from disk
  - backup on tape system