Lecture 13: File Systems

CSE 120: Principles of Operating Systems
Alex C. Snoeren

HW 3 Due Now, Lab 2 Due 5/22
In a multiprogramming system, we need a way to allocate memory to competing processes. Problem: How to determine how much memory to give to each process?

- Fixed space algorithms
  - Each process is given a limit of pages it can use
  - When it reaches the limit, it replaces from its own pages
  - Local replacement
    - Some processes may do well while others suffer

- Variable space algorithms
  - Process’ set of pages grows and shrinks dynamically
  - Global replacement
    - One process can ruin it for the rest
A working set of a process is used to model the dynamic locality of its memory usage
- Defined by Peter Denning in 60s

Definition
- \( WS(t,w) = \{\text{pages P such that P was referenced in the time interval } (t, \ t-w)\} \)
- \( t \) – time, \( w \) – working set window (measured in page refs)

A page is in the working set (WS) only if it was referenced in the last \( w \) references
Working Set Size

- The working set size is the number of pages in the working set
  - The number of pages referenced in the interval \((t, t-w)\)
- The working set size changes with program locality
  - During periods of poor locality, you reference more pages
  - Within that period of time, the working set size is larger
- Intuitively, want the working set to be the set of pages a process needs in memory to prevent heavy faulting
  - Each process has a parameter \(w\) that determines a working set with few faults
  - Denning: Don’t run a process unless working set is in memory
Working Set Problems

- Problems
  - How do we determine \( w \)?
  - How do we know when the working set changes?

- Too hard to answer
  - So, working set is not used in practice as a page replacement algorithm

- However, it is still used as an abstraction
  - The intuition is still valid
  - When people ask, “How much memory does Firefox need?”, they are in effect asking for the size of Firefox’s working set
Page Fault Frequency (PFF)

- Page Fault Frequency (PFF) is a variable space algorithm that uses a more ad-hoc approach
  - Monitor the fault rate for each process
  - If the fault rate is above a high threshold, give it more memory
    » So that it faults less
    » But not always (FIFO, Belady’s Anomaly)
  - If the fault rate is below a low threshold, take away memory
    » Should fault more
    » But not always

- Hard to use PFF to distinguish between changes in locality and changes in size of working set
Page replacement algorithms avoid thrashing

- When most of the time is spent by the OS in paging data back and forth from disk
- No time spent doing useful work (making progress)
- In this situation, the system is overcommitted
  - No idea which pages should be in memory to reduce faults
  - Could just be that there isn’t enough physical memory for all of the processes in the system
  - Ex: Running Windows XP with 64 MB of memory… (or Vista with 1 GB…)

Possible solutions
- Swapping – write out all pages of a process
- Buy more memory
Demand paging summary

- Page replacement algorithms
  - Belady’s – optimal replacement (minimum # of faults)
  - FIFO – replace page loaded furthest in past
  - LRU – replace page referenced furthest in past
    » Approximate using PTE reference bit
  - LRU Clock – replace page that is “old enough”
  - Working Set – keep the set of pages in memory that has minimal fault rate (the “working set”)
  - Page Fault Frequency – grow/shrink page set as a function of fault rate

- Multiprogramming
  - Should a process replace its own page, or that of another?
First we’ll discuss properties of physical disks
   - Structure
   - Performance
   - Scheduling

Then we’ll discuss how we build file systems on them
   - Files
   - Directories
   - Sharing
   - Protection
   - File System Layouts
   - File Buffer Cache
   - Read Ahead
Disks and the OS

- Disks are messy physical devices:
  - Errors, bad blocks, missed seeks, etc.

- The job of the OS is to hide this mess from higher level software
  - Low-level device control (initiate a disk read, etc.)
  - Higher-level abstractions (files, databases, etc.)

- The OS may provide different levels of disk access to different clients
  - Physical disk (surface, cylinder, sector)
  - Logical disk (disk block #)
  - Logical file (file block, record, or byte #)
Physical Disk Structure

- Disk components
  - Platters
  - Surfaces
  - Tracks
  - Sectors
  - Cylinders
  - Arm
  - Heads

- Logically, disk broken down into sectors
  - Addressed by cylinder, head, sector (CHS)
Specifying disk requests requires a lot of info:
- Cylinder #, surface #, track #, sector #, transfer size…

Older disks required the OS to specify all of this
- The OS needed to know all disk parameters

Modern disks are more complicated
- Not all sectors are the same size, sectors are remapped, etc.

Current disks provide a higher-level interface (SCSI)
- The disk exports its data as a logical array of blocks [0…N]
  » Disk maps logical blocks to cylinder/surface/track/sector
- Only need to specify the logical block # to read/write
- But now the disk parameters are hidden from the OS
Disk Parameters (2008)

- Seagate Barracuda ES.2 (largest SATA disc available)
  - Form factor: 3.5"
  - Capacity: 1 TB
  - Rotation rate: 7,200 RPM
  - Platters: 4
  - Surfaces: 8
  - Sector size: 512 bytes
  - Cache: 32 MB
  - Transfer rate: 53 MB/s (inner) – 104 MB/s (outer)
  - Average seek: 9.5 ms
Disk Performance

- Disk request performance depends upon a number of steps
  - Seek – moving the disk arm to the correct cylinder
    » Depends on how fast disk arm can move (increasing very slowly)
  - Rotation – waiting for the sector to rotate under the head
    » Depends on rotation rate of disk (increasing, but slowly)
  - Transfer – transferring data from surface into disk controller electronics, sending it back to the host
    » Depends on density (increasing quickly)

- When the OS uses the disk, it tries to minimize the cost of all of these steps
  - Particularly seeks and rotation
Because seeks are so expensive (milliseconds!), it helps to schedule disk requests that are queued waiting for the disk

- **FCFS (do nothing)**
  - Reasonable when load is low
  - Long waiting times for long request queues
- **SSTF (shortest seek time first)**
  - Minimize arm movement (seek time), maximize request rate
  - Favors middle blocks
- **SCAN (elevator)**
  - Service requests in one direction until done, then reverse
- **C-SCAN**
  - Like SCAN, but only go in one direction (typewriter)
In general, unless there are request queues, disk scheduling does not have much impact
- Important for servers, less so for PCs

Modern disks often do the disk scheduling themselves
- Disks know their layout better than OS, can optimize better
- Ignores, undoes any scheduling done by OS
File Systems

- File systems
  - Implement an abstraction (**files**) for secondary storage
  - Organize files logically (**directories**)
  - Permit sharing of data between processes, people, and machines
  - Protect data from unwanted access (security)
Files

- A file is data with some properties
  - Contents, size, owner, last read/write time, protection, etc.

- A file can also have a type
  - Understood by the file system
    - Block, character, device, portal, link, etc.
  - Understood by other parts of the OS or runtime libraries
    - Executable, dll, source, object, text, etc.

- A file’s type can be encoded in its name or contents
  - Windows encodes type in name
    - .com, .exe, .bat, .dll, .jpg, etc.
  - Unix encodes type in contents
    - Magic numbers, initial characters (e.g., #! for shell scripts)
Basic File Operations

Unix
- creat(name)
- open(name, how)
- read(fd, buf, len)
- write(fd, buf, len)
- sync(fd)
- seek(fd, pos)
- close(fd)
- unlink(name)

NT
- CreateFile(name, CREATE)
- CreateFile(name, OPEN)
- ReadFile(handle, …)
- WriteFile(handle, …)
- FlushFileBuffers(handle, …)
- SetFilePointer(handle, …)
- CloseHandle(handle, …)
- DeleteFile(name)
- CopyFile(name)
- MoveFile(name)
Some file systems provide different access methods that specify different ways for accessing data in a file:

- Sequential access – read bytes one at a time, in order
- Direct access – random access given block/byte number
- Record access – file is array of fixed- or variable-length records, read/written sequentially or randomly by record #
- Indexed access – file system contains an index to a particular field of each record in a file, reads specify a value for that field and the system finds the record via the index (DBs)

What file access method does Unix, NT provide?

Older systems provide more complicated methods.
Directories

- Directories serve two purposes
  - For users, they provide a structured way to organize files
  - For the file system, they provide a convenient naming interface that allows the implementation to separate logical file organization from physical file placement on the disk

- Most file systems support multi-level directories
  - Naming hierarchies (/, /usr, /usr/local/, …)

- Most file systems support the notion of a current directory
  - Relative names specified with respect to current directory
  - Absolute names start from the root of directory tree
A directory is a list of entries

- \(<\text{name, location}>\)
- Name is just the name of the file or directory
- Location depends upon how file is represented on disk

List is usually unordered (effectively random)

- Entries usually sorted by program that reads directory

Directories typically stored in files

- Only need to manage one kind of secondary storage unit
Basic Directory Operations

Unix
- Directories implemented in files
  - Use file ops to create dirs
- C runtime library provides a higher-level abstraction for reading directories
  - opendir(name)
  - readdir(DIR)
  - seekdir(DIR)
  - closedir(DIR)

NT
- Explicit dir operations
  - CreateDirectory(name)
  - RemoveDirectory(name)
- Very different method for reading directory entries
  - FindFirstFile(pattern)
  - FindNextFile()
Path Name Translation

- Let’s say you want to open “/one/two/three”
- What does the file system do?
  - Open directory “/” (well known, can always find)
  - Search for the entry “one”, get location of “one” (in dir entry)
  - Open directory “one”, search for “two”, get location of “two”
  - Open directory “two”, search for “three”, get location of “three”
  - Open file “three”
- Systems spend a lot of time walking directory paths
  - This is why open is separate from read/write
  - OS will cache prefix lookups for performance
    » /a/b, /a/bb, /a/bbb, etc., all share “/a” prefix
Next time…

- Read Chapters 11.8, 12.7
- Lab Due Thursday before midnight