CS162
Operating Systems and
Systems Programming
Lecture 18

Disk scheduling & File Systems

October 31st, 2018
Prof. Ion Stoica
http://cs162.eecs.Berkeley.edu
FROM LAST LECTURE
Recal: A Little Queuing Theory: Some Results (2/2)

• Parameters that describe our system:
  – \( \lambda \): mean number of arriving customers/second
  – \( T_{\text{ser}} \): mean time to service a customer
  – \( C \): squared coefficient of variance
  – \( \mu \): service rate = \( 1/T_{\text{ser}} \)
  – \( u \): server utilization (0 \( \leq \) \( u \) \( \leq \) 1):
    \( u = \lambda \times T_{\text{ser}} \)

• Parameters we wish to compute:
  – \( T_q \): Time spent in queue
  – \( L_q \): Length of queue = \( \lambda \times T_q \)

• Results (M: Poisson arrival process, 1 server):
  – Memoryless service time distribution (C = 1):
    Called an M/M/1 queue
    \[ T_q = T_{\text{ser}} \times \frac{u}{1 - u} \]
  – General service time distribution (no restrictions):
    \[ T_q = T_{\text{ser}} \times \frac{1}{2} (1 + C) \times \frac{u}{1 - u} \]

Why does the response/queueing delay grow unboundedly even though the utilization is < 1%?
Why unbounded response time?

• Assume deterministic arrival process and service time
  – Possible to sustain utilization $= 1$ with bounded response time!
Why unbounded response time?

- Assume stochastic arrival process (and service time)
  - No longer possible to achieve 
    utilization = 1

This wasted time can never be reclaimed!
So cannot achieve u = 1!
Optimize I/O Performance

Response Time = Queue + I/O device service time

- How to improve performance?
  - Make everything faster 😊
  - More decoupled (parallelism) systems
  - Do other useful work while waiting
    > Multiple independent buses or controllers
  - Optimize the bottleneck to increase service rate
    > Use the queue to optimize the service
- Queues absorb bursts and smooth the flow
- Add admission control (finite queues)
  - Limits delays, but may introduce unfairness and livelock
When is Disk Performance Highest?

• When there are big sequential reads, or
• When there is so much work to do that they can be piggy backed (reordering queues—one moment)

• OK to be inefficient when things are mostly idle
• Bursts are both a threat and an opportunity
• <your idea for optimization goes here>
  – Waste space for speed?

• Other techniques:
  – Reduce overhead through user level drivers
  – Reduce the impact of I/O delays by doing other useful work in the meantime
Disk Scheduling (1/2)

- Disk can do only one request at a time; What order do you choose to do queued requests?

  - **FIFO Order**: Fair among requesters, but order of arrival may be to random spots on the disk $\Rightarrow$ Very long seeks

  - **SSTF: Shortest seek time first**
    - Pick the request that's closest on the disk
    - Although called SSTF, today must include rotational delay in calculation, since rotation can be as long as seek
    - **Con**: SSTF good at reducing seeks, but may lead to starvation
Disk Scheduling (2/2)

• Disk can do only one request at a time; What order do you choose to do queued requests?

  User Requests

  ![Diagram of disk scheduling](image)

• SCAN: Implements an Elevator Algorithm: take the closest request in the direction of travel
  – No starvation, but retains flavor of SSTF
Disk Scheduling (2/2)

- Disk can do only one request at a time; What order do you choose to do queued requests?

- C-SCAN: Circular-Scan: only goes in one direction
  - Skips any requests on the way back
  - Fairer than SCAN, not biased towards pages in middle
Recall: How do we Hide I/O Latency?

• **Blocking Interface:** “Wait”
  – When request data (e.g., read() system call), put process to sleep until data is ready
  – When write data (e.g., write() system call), put process to sleep until device is ready for data

• **Non-blocking Interface:** “Don’t Wait”
  – Returns quickly from read or write request with count of bytes successfully transferred to kernel
  – Read may return nothing, write may write nothing

• **Asynchronous Interface:** “Tell Me Later”
  – When requesting data, take pointer to user’s buffer, return immediately; later kernel fills buffer and notifies user
  – When sending data, take pointer to user’s buffer, return immediately; later kernel takes data and notifies user
I/O & Storage Layers

Operations, Entities and Interface

Application / Service

High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

Commands and Data Transfers

Disks, Flash, Controllers, DMA

streams

handles

registers

file_open, file_read, ... on struct file * & void *

descriptors

we are here ...

10/31/18 CS162 ©UCB Fall 2018 Lec 18.12
Recall: C Low level I/O

- Operations on File Descriptors – as OS object representing the state of a file
  - User has a “handle” on the descriptor

#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

int open (const char *filename, int flags [, mode_t mode])
int create (const char *filename, mode_t mode)
int close (int filedes)

Bit vector of:
- Access modes (Rd, Wr, …)
- Open Flags (Create, …)
- Operating modes (Appends, …)

Bit vector of Permission Bits:
- User|Group|Other X R|W|X

Recall: C Low Level Operations

ssize_t read (int filedes, void *buffer, size_t maxsize)
  - returns bytes read, 0 => EOF, -1 => error
ssize_t write (int filedes, const void *buffer, size_t size)
  - returns bytes written
off_t lseek (int filedes, off_t offset, int whence)
  - set the file offset
    * if whence == SEEK_SET: set file offset to “offset”
    * if whence == SEEK_CRT: set file offset to crt location + “offset”
    * if whence == SEEK_END: set file offset to file size + “offset”
int fsync (int fildes)
  - wait for i/o of filedes to finish and commit to disk
void sync (void) - wait for ALL to finish and commit to disk

• When write returns, data is on its way to disk and can be read,
  but it may not actually be permanent!
Building a File System

- **File System**: Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.

- File System Components
  - **Naming**: Interface to find files by name, not by blocks
  - **Disk Management**: collecting disk blocks into files
  - **Protection**: Layers to keep data secure
  - **Reliability/Durability**: Keeping of files durable despite crashes, media failures, attacks, etc.
Recall: User vs. System View of a File

- **User’s view:**
  - Durable Data Structures

- **System’s view (system call interface):**
  - Collection of Bytes (UNIX)
  - Doesn’t matter to system what kind of data structures you want to store on disk!

- **System’s view (inside OS):**
  - Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)
  - Block size $\geq$ sector size; in UNIX, block size is 4KB
Recall: Translating from User to System View

- What happens if user says: give me bytes 2—12?
  - Fetch block corresponding to those bytes
  - Return just the correct portion of the block
- What about: write bytes 2—12?
  - Fetch block
  - Modify portion
  - Write out Block
- Everything inside File System is in whole size blocks
  - For example, `getc()`, `putc()` \(\Rightarrow\) buffers something like 4096 bytes, even if interface is one byte at a time
- From now on, file is a collection of blocks
Disk Management Policies (1/2)

• Basic entities on a disk:
  – File: user-visible group of blocks arranged sequentially in logical space
  – Directory: user-visible index mapping names to files

• Access disk as linear array of sectors. Two Options:
  – Identify sectors as vectors [cylinder, surface, sector], sort in cylinder-major order, not used anymore
  – Logical Block Addressing (LBA): Every sector has integer address from zero up to max number of sectors
  – Controller translates from address ⇒ physical position
    » First case: OS/BIOS must deal with bad sectors
    » Second case: hardware shields OS from structure of disk
Recall: Disk Management Policies (2/2)

• Need way to track free disk blocks
  – Link free blocks together ⇒ too slow today
  – Use bitmap to represent free space on disk

• Need way to structure files: File Header
  – Track which blocks belong at which offsets within the logical file structure
  – Optimize placement of files’ disk blocks to match access and usage patterns
Designing a File System …

• What factors are critical to the design choices?
• Durable data store => it’s all on disk
• (Hard) Disks Performance !!!
  – Maximize sequential access, minimize seeks
• Open before Read/Write
  – Can perform protection checks and look up where the actual file resource are, in advance
• Size is determined as they are used !!!
  – Can write (or read zeros) to expand the file
  – Start small and grow, need to make room
• Organized into directories
  – What data structure (on disk) for that?
• Need to allocate / free blocks
  – Such that access remains efficient
Components of a File System

- File path
- Directory Structure
- File number structure ("inode")
- File Index Structure
- Data blocks

One Block = multiple sectors
Ex: 512 sector, 4K block
Components of a file system

- Open performs **Name Resolution**
  - Translates pathname into a “file number”
    » Used as an “index” to locate the blocks
  - Creates a file descriptor in PCB within kernel
  - Returns a “handle” (another integer) to user process

- Read, Write, Seek, and Sync operate on handle
  - Mapped to file descriptor and to blocks
Directories
Directory

• Basically a hierarchical structure

• Each directory entry is a collection of
  – Files
  – Directories
    » A link to another entries

• Each has a name and attributes
  – Files have data

• Links (hard links) make it a DAG, not just a tree
  – Softlinks (aliases) are another name for an entry
• 5 additional points to recognize the difficulty of the exam

• Regrade requests deadline is next Monday Nov 5 at 11:59PM
BREAK
I/O & Storage Layers

Application / Service

High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

streams

handles

registers

descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA

Directory Structure

Data blocks

#4 - handle
File

- Named permanent storage

- Contains
  - Data
    » Blocks on disk somewhere
  - Metadata (Attributes)
    » Owner, size, last opened, …
  - Access rights
    • R, W, X
    • Owner, Group, Other (in Unix systems)
    • Access control list in Windows system
• **Open system call:**
  – Resolves file name, finds file control block (inode)
  – Makes entries in per-process and system-wide tables
  – Returns index (called “file handle”) in open-file table
In-Memory File System Structures

- Read/write system calls:
  - Use file handle to locate inode
  - Perform appropriate reads or writes
Our first filesystem: FAT (File Allocation Table)

• The most commonly used filesystem in the world!

• Assume (for now) we have a way to translate a path to a “file number”
  – i.e., a directory structure

• Disk Storage is a collection of Blocks
  – Just hold file data (offset o = < B, x >)

• Example: file_read 31, < 2, x >
  – Index into FAT with file number
  – Follow linked list to block
  – Read the block from disk into memory
FAT Properties

• File is collection of disk blocks

• FAT is linked list 1-1 with blocks

• File Number is index of root of block list for the file

• File offset ($o = <B, x>$)

• Follow list to get block #

• Unused blocks $\Leftrightarrow$ FAT free list
FAT Properties

- File is collection of disk blocks

- FAT is linked list 1-1 with blocks

- File Number is index of root of block list for the file

- File offset \( o = < B, x > \)

- Follow list to get block #

- Unused blocks \( \Leftrightarrow \) FAT free list

- Ex: `file_write(31, < 3, y >)`
  - Grab blocks from free list
  - Linking them into file

File 31, Block 0
File 31, Block 1
File 31, Block 2
File 31, Block 3
Disk Blocks

Memory

Lec 18.33
FAT Properties

• File is collection of disk blocks

• FAT is linked list 1-1 with blocks

• File Number is index of root of block list for the file

• Grow file by allocating free blocks and linking them in

• Ex: Create file, write, write

File is collection of disk blocks

- FAT is linked list 1-1 with blocks
- File Number is index of root of block list for the file
- Grow file by allocating free blocks and linking them in
- Ex: Create file, write, write
FAT Assessment

- **FAT32** (32 instead of 12 bits) used in Windows, USB drives, SD cards, …

- Where is FAT stored?
  - On Disk, on boot cache in memory, second (backup) copy on disk

- What happens when you format a disk?
  - Zero the blocks, link up the FAT free-list

- What happens when you quick format a disk?
  - Link up the FAT free-list

- Simple
  - Can implement in device firmware
FAT Assessment – Issues

- Time to find block (large files) ??
- Block layout for file ??
- Sequential Access ??
- Random Access ??
- Fragmentation ??
  – MSDOS defrag tool
- Small files ??
- Big files ??

File number

Free

File 2 number

Memory

Disk Blocks

File 31, Block 0
File 31, Block 1
File 63, Block 1
File 31, Block 3
File 63, Block 0
File 31, Block 2

FAT

0:

31:

63:

N-1:

N-1:

Lec 18.36

CS162 ©UCB Fall 2018
What about the Directory?

- Essentially a file containing `<file_name: file_number>` mappings
- Free space for new entries
- In FAT: file attributes are kept in directory (!!!)
- Each directory a linked list of entries
- Where do you find root directory ("/")?
Directory Structure (cont’d)

• How many disk accesses to resolve “/my/book/count”?
  – Read in file header for root (fixed spot on disk)
  – Read in first data block for root
    » Table of file name/index pairs. Search linearly – ok since directories typically very small
  – Read in file header for “my”
  – Read in first data block for “my”; search for “book”
  – Read in file header for “book”
  – Read in first data block for “book”; search for “count”
  – Read in file header for “count”

• Current working directory: Per-address-space pointer to a directory (inode) used for resolving file names
  – Allows user to specify relative filename instead of absolute path
    (say CWD=“/my/book” can resolve “count”)

10/31/18
Many Huge FAT Security Holes!

• FAT has no access rights

• FAT has no header in the file blocks

• Just gives an index into the FAT
  – (file number = block number)
Summary

• File System:
  – Transforms blocks into Files and Directories
  – Optimize for access and usage patterns
  – Maximize sequential access, allow efficient random access

• File (and directory) defined by header, called “inode”

• File Allocation Table (FAT) Scheme
  – Linked-list approach
  – Very widely used: Cameras, USB drives, SD cards
  – Simple to implement, but poor performance and no security

• Look at actual file access patterns – many small files, but large files take up all the space!
Happy Halloween