CS162
Operating Systems and
Systems Programming
Lecture 18

Queuing Theory,
File Systems

November 2\textsuperscript{nd}, 2015
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiatowicz at Berkeley, with few minor updates/changes. When slides are obtained from other sources, a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.
Recall: I/O Performance

Response Time = Queue + I/O device service time

- Performance of I/O subsystem
  - Metrics: Response Time, Throughput
  - Effective BW per op = transfer size / response time
    \[ \text{EffBW}(n) = \frac{n}{S + n/B} = \frac{B}{1 + SB/n} \]
  - Contributing factors to latency:
    - Software paths (can be loosely modeled by a queue)
    - Hardware controller
    - I/O device service time

- Queuing behavior:
  - Can lead to big increases of latency as utilization increases
  - Solutions?
A Little Queuing Theory: Some Results

- **Assumptions:**
  - System in equilibrium; No limit to the queue
  - Time between successive arrivals is random and memoryless

- **Parameters that describe our system:**
  - $\lambda$: mean number of arriving customers/second
  - $T_{	ext{ser}}$: mean time to service a customer ("m1")
  - $C$: squared coefficient of variance = $\sigma^2/m1^2$
  - $\mu$: service rate = $1/T_{\text{ser}}$
  - $u$: server utilization ($0 \leq u \leq 1$): $u = \lambda/\mu = \lambda \times T_{\text{ser}}$

- **Parameters we wish to compute:**
  - $T_q$: Time spent in queue
  - $L_q$: Length of queue = $\lambda \times T_q$ (by Little's law)

- **Results:**
  - **Memoryless service distribution ($C = 1$):**
    - Called $M/M/1$ queue: $T_q = T_{\text{ser}} \times u/(1 - u)$
  - **General service distribution (no restrictions), 1 server:**
    - Called $M/G/1$ queue: $T_q = T_{\text{ser}} \times \frac{1}{2}(1+C) \times u/(1 - u)$
When is the 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

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

- Scheduling algorithms:
  - FIFO
  - SSTF: Shortest seek time first
  - SCAN
  - C-SCAN
FIFO: First In First Out

- Schedule requests in the order they arrive in the queue

- Example:
  - Request queue: 2, 1, 3, 6, 2, 5
  - Scheduling order: 2, 1, 3, 6, 2, 5

- Pros: Fair among requesters

- Cons: Order of arrival may be to random spots on the disk ⇒ Very long seeks
SSTF: Shortest Seek Time First

- Pick the request that's closest to the head on the disk
  - Although called SSTF, include rotational delay in calculation, as rotation can be as long as seek

- Example:
  - Request queue: 2, 1, 3, 6, 2, 5
  - Scheduling order: 5, 6, 3, 2, 2, 1

- Pros: reduce seeks

- Cons: may lead to starvation
  - Greedy. Not optimal
• Implements an Elevator Algorithm: take the closest request in the direction of travel

• Example:
  - Request queue: 2, 1, 3, 6, 2, 5
  - Head is moving towards center
  - Scheduling order: 5, 3, 2, 2, 1, 6

• Pros:
  - No starvation
  - Low seek

• Cons: favors middle tracks
  - May spend time on sparse tracks while dense requests elsewhere
C-SCAN

- Like SCAN but only serves request in only one direction

- Example:
  - Request queue: 2, 1, 3, 6, 2, 5
  - Head only serves request on its way from center towards edge
  - Scheduling order: 5, 6, 1, 2, 2, 3

- Pros:
  - Fairer than SCAN
  - Accumulate work in remote region then go get it

- Cons: longer seeks on the way back
Review: Device Drivers

- **Device Driver**: Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the `ioctl()` system call

- **Device Drivers typically divided into two pieces**:
  - **Top half**: accessed in call path from system calls
    » implements a set of **standard, cross-device calls** like open(), close(), read(), write(), ioctl(), strategy()
    » This is the kernel's interface to the device driver
    » Top half will start I/O to device, may put thread to sleep until finished
  - **Bottom half**: run as interrupt routine
    » Gets input or transfers next block of output
    » May wake sleeping threads if I/O now complete
Kernel vs User-level I/O

- Both are popular/practical for different reasons:
  - **Kernel-level drivers** for critical devices that must keep running, e.g. display drivers.
    » Programming is a major effort, correct operation of the rest of the kernel depends on correct driver operation.
  - **User-level drivers** for devices that are non-threatening, e.g USB devices in Linux (libusb).
    » Provide higher-level primitives to the programmer, avoid every driver doing low-level I/O register tweaking.
    » The multitude of USB devices can be supported by Less-Than-Wizard programmers.
    » New drivers don’t have to be compiled for each version of the OS, and loaded into the kernel.
Kernel vs User-level Programming Styles

• Kernel-level drivers
  - Have a much more limited set of resources available:
    » Only a fraction of libc routines typically available.
    » Memory allocation (e.g. Linux kmalloc) much more limited in capacity and required to be physically contiguous.
    » Should avoid blocking calls.
    » Can use asynchrony with other kernel functions but tricky with user code.

• User-level drivers
  - Similar to other application programs but:
    » Will be called often - should do its work fast, or postpone it - or do it in the background.
    » Can use threads, blocking operations (usually much simpler) or non-blocking or asynchronous.
Performance: multiple outstanding requests

- Suppose each read takes 10 ms to service.
- If a process works for 100 ms after each read, what is the utilization of the disk?
  - \( U = \frac{10 \text{ ms}}{110 \text{ ms}} = 9\% \)
- What if there are two such processes?
  - \( U = \frac{10 \text{ ms} + 10 \text{ ms}}{110 \text{ ms}} = 18\% \)
- What if each of those processes have two such threads?
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

stream

data

handles

registers

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

descriptors

we are here ...

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

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

int open (const char *filename, int flags [, mode_t mode])
int creat (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)

int fsync (int filedes) – wait for i/o to finish
void sync (void) – wait for ALL to finish

• When write returns, data is on its way to disk and can be read, but it may not actually be permanent!
• **File System**: Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.

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

• **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 ≥ sector size; in UNIX, block size is 4KB
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() ⇒ buffers something like 4096 bytes, even if interface is one byte at a time

• From now on, file is a collection of blocks
So you are going to design a file system ...

- What factors are critical to the design choices?
- Durable data store => it’s all on disk
- 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
Disk Management Policies

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

- Access disk as linear array of sectors. Two Options:
  - Identify sectors as vectors [cylinder, surface, sector]. Sort in cylinder-major order. Not used much anymore.
  - Logical Block Addressing (LBA). Every sector has integer address from zero up to max number of sectors.
  - Controller translates from address \( \Rightarrow \) physical position
    - First case: OS/BIOS must deal with bad sectors
    - Second case: hardware shields OS from structure of disk

- Need way to track free disk blocks
  - Link free blocks together \( \Rightarrow \) 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
Components of a File System

File path

Directory Structure

File number

File Index Structure

Data blocks

File path

File number
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 int) to user process
- Read, Write, Seek, and Sync operate on handle
  - Mapped to 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
I/O & Storage Layers

Application / Service

High Level I/O
- streams
- handles
- registers

Low Level I/O
- descriptors
- Syscall
- commands and data transfers

File System

I/O Driver

Disks, Flash, Controllers, DMA

Data blocks

Directory Structure
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
Our first filesystem: FAT (File Allocation Table)

• 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
• Example: file_read 31, < 2, x >
  - Index into FAT with file number
  - Follow linked list to block
  - Read the block from disk into mem
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 ↔ FAT free list
• 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

FAT Properties

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

Disk Blocks

FAT

mem

N-1:

0:

31:

0:

N-1:
• 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
- Used in DOS, Windows, thumb drives, ...

- Where is FAT stored?
  - On Disk, restore on boot, copy in memory

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

- Simple
FAT Assessment

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

File 31, Block 0

File 31, Block 1

File 31, Block 2

File 63, Block 1

File 63, Block 0

File 63, Block 2

Disk Blocks

File 2 number

File number

Free

Mem

N-1:

0:
What about the Directory?

- Essentially a file containing `<file_name: file_number>` mappings
- Free space for new entries
- In FAT: attributes kept in directory (!!!)
- Each directory a linked list of entries
- Where do you find root directory ("/")?
Directory Structure (Con't)

• 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”)
Big FAT security holes

- FAT has no access rights
- FAT has no header in the file blocks
- Just gives and index into the FAT
  - (file number = block number)
Characteristics of Files

- Most files are small
- Most of the space is occupied by the rare big ones
So what about a “real” file system

• Meet the inode:

  - Inode Array
  - File Metadata
  - Direct Pointers
  - Indirect Pointer
    - Db1. Indirect Ptr.
    - Tripl. Indirect Ptr.
  - Triple Indirect Blocks
  - Double Indirect Blocks
  - Indirect Blocks
  - Data Blocks
Unix File System

• Original inode format appeared in BSD 4.1
  - Berkeley Standard Distribution Unix
  - Part of Berkeley heritage!
  - Similar structure for Linux Ext2/3

• File Number is index into inode arrays

• Multi-level index structure
  - Great for little and large files
  - Asymmetric tree with fixed sized blocks

• Metadata associated with the file
  - Rather than in the directory that points to it

• UNIX FFS: BSD 4.2: Locality Heuristics
  - Block group placement
  - Reserve space

• Scalable directory structure
An “almost real” file system

- Pintos: src/filesys/file.c, inode.c

```c
/* An open file. */
struct file {
    struct inode *inode;    /* File's inode. */
    off_t pos;              /* Current position. */
    bool deny_write;        /* Has file_deny_write() been called? */
};

/* In-memory inode. */
struct inode {
    struct list_elem elem;    /* Element in inode list. */
    block_sector_t sector;    /* Sector number of disk location. */
    int open_cnt;             /* Number of openers. */
    bool removed;             /* True if deleted, false otherwise. */
    int deny_write_cnt;       /* 0: writes ok, >0: deny writes. */
    struct inode_disk data;   /* Inode content. */
};

/* On-disk inode. Must be exactly BLOCK_SECTOR_SIZE bytes long. */
struct inode_disk {
    block_sector_t start;    /* First data sector. */
    off_t length;            /* File size in bytes. */
    unsigned magic;          /* Magic number. */
    uint32_t unused[125];    /* Not used. */
};
```
File Attributes

• Inode metadata

User
Group
9 basic access control bits
  - UGO x RWX
Setuid bit
  - execute at owner permissions
  - rather than user
Getgid bit
  - execute at group’s permissions
Data Storage

- Small files: 12 pointers direct to data blocks

Direct pointers

4kB blocks ⇒ sufficient
For files up to 48KB

Indirect pointers

File Metadata

Inode

Triple
Indirect
Blocks

Double
Indirect
Blocks

Indirect
Blocks

Data
Blocks

Fig. 2. Histograms of files by size.
Data Storage

- Large files: 1, 2, 3 level indirect pointers

Indirect pointers
- point to a disk block containing only pointers
- 4 kB blocks => 1024 ptrs
  => 4 MB @ level 2
  => 4 GB @ level 3
  => 4 TB @ level 4

A Five-Year Study of File-System Metadata

\[
\begin{array}{c}
\text{Direct Pointers} \\
\text{Indirect Pointers} \\
\text{Double Indirect Blocks} \\
\text{Triple Indirect Blocks} \\
\text{Data Blocks}
\end{array}
\]
Where are inodes stored?

- In early UNIX and DOS/Windows' FAT file system, headers stored in special array in outermost cylinders
  - Header not stored anywhere near the data blocks. To read a small file, seek to get header, seek back to data.
  - Fixed size, set when disk is formatted. At formatting time, a fixed number of inodes were created (They were each given a unique number, called an “inumber”)

Where are inodes stored?

• Later versions of UNIX moved the header information to be closer to the data blocks
  – Often, inode for file stored in same “cylinder group” as parent directory of the file (makes an ls of that directory run fast).
  – Pros:
    » UNIX BSD 4.2 puts a portion of the file header array on each of many cylinders. For small directories, can fit all data, file headers, etc. in same cylinder ⇒ no seeks!
    » File headers much smaller than whole block (a few hundred bytes), so multiple headers fetched from disk at same time
    » Reliability: whatever happens to the disk, you can find many of the files (even if directories disconnected)
  – Part of the Fast File System (FFS)
    » General optimization to avoid seeks
4.2 BSD Locality: Block Groups

- File system volume is divided into a set of block groups
  - Close set of tracks
- Data blocks, metadata, and free space interleaved within block group
  - Avoid huge seeks between user data and system structure
- Put directory and its files in common block group
- First-Free allocation of new file blocks
  - To expand file, first try successive blocks in bitmap, then choose new range of blocks
  - Few little holes at start, big sequential runs at end of group
  - Avoids fragmentation
  - Sequential layout for big files
- Important: keep 10% or more free!
  - Reserve space in the BG
File System Summary

• File System:
  - Transforms blocks into Files and Directories
  - Optimize for size, access and usage patterns
  - Maximize sequential access, allow efficient random access
  - Projects the OS protection and security regime (UGO vs ACL)

• File defined by header, called “inode”

• Naming: act of translating from user-visible names to actual system resources
  - Directories used for naming for local file systems
  - Linked or tree structure stored in files

• Multilevel Indexed Scheme
  - inode contains file info, direct pointers to blocks, indirect blocks, doubly indirect, etc..
  - NTFS uses variable extents, rather than fixed blocks, and tiny files data is in the header

• 4.2 BSD Multilevel index files
  - Inode contains pointers to actual blocks, indirect blocks, double indirect blocks, etc.
  - Optimizations for sequential access: start new files in open ranges of free blocks, rotational Optimization