Recall: I/O Performance

- Metrics: Response Time, Throughput
- Effective BW per op = transfer size / response time
  - \( \text{EffBW}(n) = n / (S + n/B) = B / (1 + SB/n) \)
- 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 Simple Deterministic World

- Assume requests arrive at regular intervals, take a fixed time to process, with plenty of time between …
- Service rate (\(\mu = 1/T_S\)) - operations per second
- Arrival rate: (\(\lambda = 1/T_A\)) - requests per second
- Utilization: \(U = \lambda/\mu\), where \(\lambda < \mu\)
- Average rate is the complete story

A Ideal Linear World

- What does the queue wait time look like during overload?
  - Grows unbounded at a rate ~ \((T_S/T_A)\) till request rate subsides
**Reality: A Bursty World**

- Requests arrive in a burst, must queue up till served
- Same average arrival time, but:
  - Almost all of the requests experience large queue delays
  - Even though average utilization is low!

**Queue Server**

<table>
<thead>
<tr>
<th>arrivals</th>
<th>Queue</th>
<th>Server</th>
<th>departures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Background: General Use of Random Distributions**

- Server spends variable time ($T$) with customers
  - Mean (Average) $m = \Sigma p(T) \times T$
  - Variance (stddev$^2$) $\sigma^2 = \Sigma p(T) \times (T-m)^2 = \Sigma p(T) \times T^2-m^2$
  - Squared coefficient of variance: $C = \sigma^2/m^2$
    Aggregate description of the distribution

- Important values of $C$:
  - No variance or deterministic $\Rightarrow C=0$
  - “Memoryless” or exponential $\Rightarrow C=1$
    » Past tells nothing about future
    » Poisson process – *purely* or *completely* random process
    » Many complex systems (or aggregates) are well described as memoryless
  - Disk response times $C \approx 1.5$ (majority seeks < average)

**Introduction to Queuing Theory**

- What about queuing time??
  - Let's apply some queuing theory
  - Queuing Theory applies to long term, steady state behavior $\Rightarrow$
    Arrival rate = Departure rate

- Arrivals characterized by some probabilistic distribution
- Departures characterized by some probabilistic distribution

**So how do we model the burstiness of arrival?**

- Elegant mathematical framework if you start with *exponential distribution*
  - Probability density function of a continuous random variable with a mean of $1/\lambda$
    $$ f(x) = \lambda e^{-\lambda x} $$
  - “Memoryless”

Likelihood of an event occurring is independent of how long we've been waiting

- Lots of short arrival intervals (i.e., high instantaneous rate)
- Few long gaps (i.e., low instantaneous rate)
Little's Law

- In any stable system
  - Average arrival rate = Average departure rate
- The average number of jobs/tasks in the system ($N$) is equal to arrival time / throughput ($\lambda$) times the response time ($L$)
  - $N \text{ (jobs)} = \lambda \text{ (jobs/s)} \times L \text{ (s)}$
- Regardless of structure, bursts of requests, variation in service
  - Instantaneous variations, but it washes out in the average
  - Overall, requests match departures

Example

- $\lambda = 1$
- $L = 5$
- Jobs

A: $N = \lambda \times L$
- E.g., $N = \lambda \times L = 5$

Little's Theorem: Proof Sketch

- What is the system occupancy, i.e., average number of jobs in the system?
Little's Theorem: Proof Sketch

1. Let $N(i)$ be the number of jobs in the system at time $t$.
2. Let $S(i) = L(i) \times 1 = L(i)$, where $L(i)$ is the response time of job $i$.
3. The total area $S$ under the curve is the sum of the areas $S(1), S(2), \ldots, S(k)$:
   
   $S = S(1) + S(2) + \ldots + S(k) = L(1) + L(2) + \ldots + L(k)$

4. The average occupancy $N_{avg}$ is given by:
   
   $N_{avg} = S/T = (L(1) + \ldots + L(k))/T$

5. The average number of jobs in the system is equal to the average occupancy times the throughput $\lambda$:

   $N_{avg} = (L(1) + \ldots + L(k))/T = (N_{total}/T)*(L(1) + \ldots + L(k))/N_{total}$
Little's Theorem: Proof Sketch

\[ L(i) = \text{response time of job } i \]
\[ N(t) = \text{number of jobs in system at time } t \]
\[ S(i) = L(i) \times 1 = L(i) \]

\[ N_{\text{avg}} = \frac{N_{\text{total}}}{T} \times (L(1) + \ldots + L(k))/N_{\text{total}} = \lambda_{\text{avg}} \times L_{\text{avg}} \]

Administrivia

• Midterm II: Too long! Sorry!
  – Yup, we misjudged that one
• Midterm II Statistics:
  – Mean: 58.9, STD: 12.7, Max: 94.0
• Solutions are up
  – Regrade requests close on Friday 4/12, 11:59pm

A Little Queuing Theory: Some Results

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

  \[ \lambda: \text{mean number of arriving customers/second} \]
  \[ T_{\text{ser}}: \text{mean time to service a customer ("m1")} \]
  \[ C: \text{squared coefficient of variance} = \sigma^2/m1^2 \]
  \[ \mu: \text{service rate} = 1/T_{\text{ser}} \]
  \[ u: \text{server utilization (0} \leq u \leq 1); \quad u = \lambda / \mu = \lambda \times T_{\text{ser}} \]

• Parameters we wish to compute:
  – \[ T_q: \text{Time spent in queue} \]
  – \[ L_q: \text{Length of queue} = \lambda \times T_q \text{ (by Little’s law)} \]

• Results:
  – Memoryless service distribution (C = 1): (an “M/M/1 queue”):
    \[ T_q = T_{\text{ser}} \times u(1 - u) \]
  – General service distribution (no restrictions), 1 server (an “M/G/1 queue”):
    \[ T_q = T_{\text{ser}} \times \frac{1}{2}(1 + C) \times u(1 - u) \]
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
  - $T_{ser}$: mean time to service a customer
  - $C$: squared coefficient of variation
  - $\mu$: service rate = $1/T_{ser}$
  - $u$: server utilization ($0 \leq u \leq 1$)

- 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 ($C = 1$): (an "M/M/1 queue"): $T_{q} = T_{ser} \times u/(1 – u)$
  - General service distribution (no restrictions), 1 server (an "M/G/1 queue"): $T_{q} = T_{ser} \times \frac{1}{2}(1+C) \times u/(1 – u)$

Why unbounded response time?

- Assume deterministic arrival process and service time
  - Possible to sustain utilization = 1 with bounded response time!

A Little Queuing Theory: An Example

- Example Usage Statistics:
  - User requests 10 x 8KB disk I/Os per second
  - Requests & service exponentially distributed ($C=1.0$)
  - Avg. service = 20 ms (From controller+seek+rot+trans)

- Questions:
  - How utilized is the disk?
    - Ans: server utilization, $u = \lambda \times T_{ser}$
  - What is the average time spent in the queue?
    - Ans: $T_{q}$
  - What is the number of requests in the queue?
    - Ans: $L_{q}$
  - What is the avg response time for disk request?
    - Ans: $T_{sys} = T_{q} + T_{ser}$

- Computation:
  - $\lambda$ (avg # arriving customers/s) = 10/s
  - $T_{ser}$ (avg time to service customer) = 20 ms (0.02s)
  - $u$ (server utilization) = $\lambda \times T_{ser}$ = 10/s x 0.02s = 0.2
  - $T_{q}$ (avg time/customer in queue) = $T_{ser} \times u/(1 – u)$ = 20 x 0.2/(1-0.2) = 20 x 0.25 = 5 ms (0.005s)
  - $L_{q}$ (avg length of queue) = $\lambda \times T_{q}$=10x5s x 0.005s = 0.05
  - $T_{sys}$ (avg time/customer in system) = $T_{q} + T_{ser}$ = 25 ms
Queuing Theory Resources

- Resources page contains Queueing Theory Resources (under Readings):
  - Scanned pages from Patterson and Hennessy book that gives further discussion and simple proof for general equation: https://cs162.eecs.berkeley.edu/static/readings/patterson_queue.pdf
  - A complete website full of resources: http://web2.uwindsor.ca/math/hlynka/qonline.html

- Some previous midterms with queueing theory questions

- Assume that Queueing Theory is fair game for Midterm III!

Optimize I/O Performance

- How to improve performance?
  - Make everything faster 😊
  - More Decoupled (Parallelism) systems
    » multiple independent buses or controllers
  - Optimize the bottleneck to increase service rate
    » Use the queue to optimize the service
  - Do other useful work while waiting

- Queues absorb bursts and smooth the flow
- Admissions control (finite queues)
  - Limits delays, but may introduce unfairness and livelock

Response Time = Queue + I/O device service time

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?

User Requests: 3, 2, 5, 2, 1, 3, 2, 1, 3

- FIFO Order
  - Fair among requesters, but 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 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?
  - SCAN: Implements an Elevator Algorithm: take the closest request in the direction of travel
    - No starvation, but retains flavor of SSTF

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
- streams
- handles
- registers
- file_open, file_read, ... on struct file *
- descriptors
- we are here ...
- Commands and Data Transfers
- Disks, Flash, Controllers, DMA
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 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 filedes)
  - 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 ≥ 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() ⇒ 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

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 Index Structure
- File number "inumber"
- One Block = multiple sectors
  Ex: 512 sector, 4K block
- Data blocks
- "inode"

Components of a file system

file name → file number → Storage block
offset

- 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

- 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: streams
- High Level I/O
- Low Level I/O

I/O Driver
- Syscall

File System
- Commands and Data Transfers
- Disks, Flash, Controllers, DMA

Data blocks

High Level I/O
- handles
- registers

Low Level I/O
- descriptors

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

File handle

File descriptor

- Inode (file object)
- Position

In-Memory File System Structures

- 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

- 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 \( o = \langle B, x \rangle \))
- Example: file_read 31, \(< 2, x >\)
  - Index into FAT with file number
  - Follow linked list to block
  - Read the block from disk into memory

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

FAT

Disk Blocks

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

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 \( o = \langle B, x \rangle \)
- Follow list to get block #
- Unused blocks ⇔ Marked free (no ordering, must scan to find)

Ex: file_write(31, \(< 3, y >\))
  - Grab free block
  - Linking them into file

File 31, Block 3
File 31, Block 2

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
- Grow file by allocating free blocks and linking them in
- Ex: Create file, write, write

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

memory
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, Mark FAT entries “free”
- What happens when you quick format a disk?
  - Mark all entries in FAT as free
- 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 ??

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”)
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

- Bursts & High Utilization introduce queuing delays
- Queuing Latency:
  - M/M/1 and M/G/1 queues: simplest to analyze
  - As utilization approaches 100%, latency $\to \infty$
  - $T_q = T_{ser} \times \frac{1}{2(1+C)} \times \frac{u}{1 - u}$
- 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!