Recall: I/O Performance

- Performance of I/O subsystem
  - Metrics: Response Time, Throughput
  - Effective BW per op = transfer size / response time
  » \( \text{EffBW(n)} = \frac{n}{(S + n/B)} = B \left( \frac{1}{1 + SB/n} \right) \)
  - 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 sec
- 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?
  - Grows unbounded at a rate ~ \( \left( \frac{T_S}{T_A} \right) \) till request rate subsides
A Bursty World

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

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)

Background: General Use of random distributions

- Server spends variable time with customers
  - Mean (Average) $m_1 = \sum p(T) \times T$
  - Variance $\sigma^2 = \sum p(T) \times (T - m_1)^2 = \sum p(T) \times T^2 - m_1^2$
  - Squared coefficient of variance: $C = \sigma^2/m_1^2$
    - 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
    - Many complex systems (or aggregates) well described as memoryless
  - Disk response times $C \approx 1.5$ (majority seeks < avg)

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
Little's Law

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

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
  - \( T_{\text{ser}} \): 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: \( T_q = T_{\text{ser}} \times \frac{1}{2}(1+C) \times u/(1 - u) \)

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, \( u = \lambda T_{\text{ser}} \)
  - 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: \( T_{\text{sys}} = T_q + T_{\text{ser}} \)
- **Computation:**
  - \( \lambda \) (avg # arriving customers/s) = 10/s
  - \( T_{\text{ser}} \) (avg time to service customer) = 20 ms (0.02s)
  - \( u \) (server utilization) = \( \lambda \times T_{\text{ser}} = 10/s \times .02s = 0.2 \)
  - \( T_q \) (avg time/customer in queue) = \( T_{\text{ser}} \times u/(1 - u) \) = \( 20 \times 0.2/(1-0.2) = 20 \times 0.25 = 5 \text{ ms (0.005s)} \)
  - \( L_q \) (avg length of queue) = \( \lambda \times T_q = 10/s \times 0.005s = 0.05 \)
  - \( T_{\text{sys}} \) (avg time/customer in system) = \( T_q + T_{\text{ser}} = 25 \text{ ms} \)

Queuing Theory Resources

- Handouts page contains Queuing Theory Resources:
  - Scanned pages from Patterson and Hennesey book that gives further discussion and simple proof for general eq.
  - A complete website full of resources
- Midterms with queueing theory questions:
  - Midterm IIs from previous years that I’ve taught
  - Assume that Queueing theory is fair game for Midterm II and/or the final!
**Administrivia**

- HW3 – Moved deadline to Wednesday (11/04)
  - Sorry about fact that server was down!
- Project 2 code due this Friday!
- Midterm I Regrade requests: Due this Wednesday
- Midterm II: Coming up in 3 weeks! (11/23)
  - 7-10PM, "here" (2040, 2050, 2060 VLSB)
  - Topics up to and including previous Wednesday
  - 2 pages of hand-written notes, both sides

---

**Quick Aside: Big Projects**

- What is a big project?
  - Time/work estimation is hard
  - Programmers are eternal optimistics (it will only take two days)
    » This is why we bug you about starting the project early
    » Had a grad student who used to say he just needed "10 minutes" to fix something. Two hours later...
- Can a project be efficiently partitioned?
  - Partitionable task decreases in time as you add people
  - But, if you require communication:
    » Time reaches a minimum bound
    » With complex interactions, time increases!
  - Mythical person-month problem:
    » You estimate how long a project will take
    » Starts to fall behind, so you add more people
    » Project takes even more time!

---

**Techniques for Partitioning Tasks**

- Functional
  - Person A implements threads, Person B implements semaphores, Person C implements locks...
  - Problem: Lots of communication across APIs
    » If B changes the API, A may need to make changes
    » Story: Large airline company spent $200 million on a new scheduling and booking system. Two teams "working together." After two years, went to merge software. Failed! Interfaces had changed (documented, but no one noticed). Result: would cost another $200 million to fix.
- Task
  - Person A designs, Person B writes code, Person C tests
  - May be difficult to find right balance, but can focus on each person's strengths (Theory vs systems hacker)
  - Since Debugging is hard, Microsoft has two testers for each programmer
- Most CS162 project teams are functional, but people have had success with task-based divisions

---

**Communication**

- More people mean more communication
  - Changes have to be propagated to more people
  - Think about person writing code for most fundamental component of system: everyone depends on them!
  - You should be meeting in person at least twice/week!
- Miscommunication is common
  - "Index starts at 0? I thought you said 1!"
- Who makes decisions?
  - Individual decisions are fast but trouble
  - Group decisions take time
  - Centralized decisions require a big picture view (someone who can be the "system architect")
- Often designating someone as the system architect can be a good thing
  - Better not be clueless
  - Better have good people skills
  - Better let other people do work
Coordination

- More people ⇒ no one can make all meetings!
  - They miss decisions and associated discussion
  - Example from earlier class: one person missed meetings and did something group had rejected
- People have different work styles
  - Some people work in the morning, some at night
  - How do you decide when to meet or work together?
- What about project slippage?
  - It will happen, guaranteed!
  - Example: phase 4 of one project, everyone busy but not talking. One person way behind. No one knew until very end – too late!
- Hard to add people to existing group
  - Members have already figured out how to work together

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

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?

  - User Requests
  - 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
  - SCAN: Implements an Elevator Algorithm: take the closest request in the direction of travel
    - No starvation, but retains flavor of SSTF
  - 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
**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?
  - \( U = \frac{10 \text{ ms}}{110 \text{ ms}} \approx 9\% \)
- What if there are two such processes?
  - \( U = \frac{(10 \text{ ms} + 10 \text{ ms})}{110 \text{ ms}} \approx 18\% \)
- 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

---

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

```c
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!
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**
  - 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 ⇒ 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 ⇒ 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

- **Directory Structure**
- **File Index Structure**

**File path**

- **Directory**
- **File Index**

**File number**

**Data blocks**

- **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 the kernel
  - Returns a "handle" (another int) to the user process

- **Read, Write, Seek, and Sync** operate on handle
  - Mapped to descriptor and to blocks

Components of a file system

- **file name** → **directory** → **file number** → **Storage block**

 directories

- Basically a hierarchical structure
- Each directory entry is a collection of:
  - Files
  - Directories
    - A link to another entry
- 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
- Low Level I/O

Syscall
- Streams
- Handles

File System
- Registers
- Descriptors

I/O Driver
- Commands and Data Transfers
- Disks, Flash, Controllers, DMA

Directory Structure

Data blocks

Summary
- Queuing Latency:
  - M/M/1 and M/G/1 queues: simplest to analyze
  - As utilization approaches 100%, latency \( \rightarrow \infty \)
  \[
  T_q = T_{ser} \times \frac{1}{2(1+C)} \times \frac{u}{(1 - u)}
  \]
- 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"
- Multilevel Indexed Scheme
  - Inode contains file info, direct pointers to blocks,
    indirect blocks, doubly indirect, etc..
- 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

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

Fileobjct (inode)

Position

Data blocks

File handle

...