CS 111: Final Review

Winter 2017, Prof. Reiher

Asynchronous Completion and Mutual Exclusion

Lecture 8: Mutual Exclusion and Asynchronous Completion

Condition Variables and the OS

- There is **always a queue associated**
- Generally condition variables are provided **by the OS**
    - OS checks and makes sure that it still definitely run
    - Or library code that implements thread
- **Blocks** a process or a thread when a condition variable is used
    - Move it **out of the ready queue**
    - **Observes** when the desired event occurs
    - When it does, **unblocks the process or thread** that was previously blocked
    - Places it back in the **ready queue**
    - May preempt the running process

Handling Multiple Waits

- Threads will wait on *several* different things
    - We can have condition variables for *all* the possible events we can wait for
    - It is pointless to wake everyone up on every event
    - Each OS allows easy selection of the right thread
    - Several threads can be waiting for the same thing
    - Do we wake some/several/all/one thread?

Waiting Lists

- Associated with **completion events**
- Suggests that each completion event **needs an associated waiting list**
    - When posting, we determine who is waiting for that event
    - The things we can do:
        - Wake everyone on the waiting list
            - All become eligible to run
        - Wake them up one-at-a-time in FIFO order
            - Event can only consumed by one
        - Wake them up one-at-a-time in a **priority order** → possible starvation
            - Change the order of the queue based on priority
            - **Priority = possibility of starvation**
            - If you have a lot of high priority threads → low priority threads are starved
    - Our choice depends on the event/application

Who to Wake Up?

- Who waits up when a condition variable is signaled?

        - pthread_cond_**wait** - at least **one blocked thread**
        - pthread_cond_**broadcast** - **all blocked threads**
        - The **broadcast** approach may be wasteful
        - If event can only be consumed once
        - Unbounded wait time potential
        - Competition based on **scheduling**
            - The rest of the threads have to **block and wait for the event again**
            - Each thread would have to be scheduled again and end up going to sleep
            - **Overhead in scheduling, context switches**, threads are brought in but they do **nothing**
        - **Waiting queue** would solve these problems
        - Each post wakes up **the first client** on the queue

Evaluating Waiting List Options

- Effectiveness/Correctness: should be goo
- Progress: trade-off involved with *cutting* in line
- Fairness: should be good
- Performance: should be very efficient, provided you aren't needlessly waking up processes
    - Depends on the frequency of spurious wakeups

Locking and Waiting Lists

- Spinning for a lock **usually is a bad thing** - try and use a **condition variable instead**
    - Locks should probably have waiting lists
    - A **waiting list** is a (shared) data structure
    - Implementation has **critical sections** that need to be updated properly
    - Protected by a lock
    - Seems to be a **circular dependency**
    - Locks have waiting lists
    - Which are protected by locks
    - What if we must wait for the waiting list lock?
        - Operation to lock is done very quickly → spin lock
        - Usually this is not a problem
        - The implementation makes sure that the waiting list **doesn't hold the lock for long**, so we can use a spin-loop

A Possible Problem

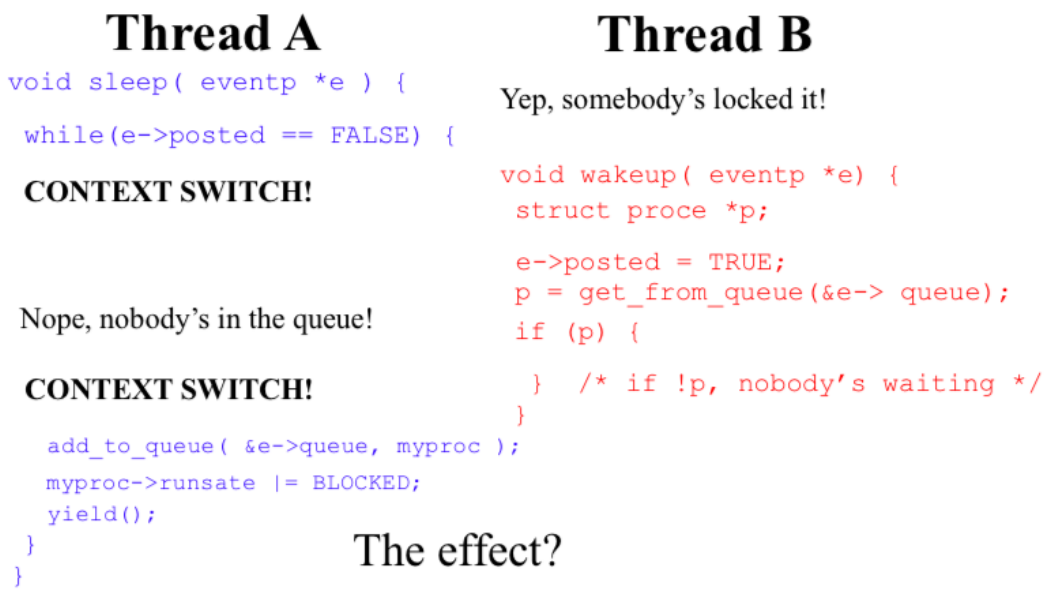
- The **sleep/wakeup race** condition

A Sleep/Wakeup Race

- Thread B has a locked resource and thread A needs to get that lock
- Thread A calls sleep() to wait
- Thread B finishes using the resource
    - So thread B calls wakeup() to release the lock

- No other threads

The Race At Work



- Thread A: wants something so it calls sleep
  - It checks if the event is posted
  - But a CONTEXT SWITCH occurs **before it can do anything else** - before it can get added to the queue
  - Thread B: finished with the resource, but since A is not yet on the queue, sees that the queue **is empty**
  - **B thinks that nobody is waiting**
  - Context returns to Thread A
  - Put on the queue and is blocked
  - NOW:
  - Thread B **no longer holds the resource**
    - **Never wakes up thread A again**
  - And Thread A ends up **sleeping forever**

Solving the Problem

- Our problem: context switch in the **middle of the critical section**
  - The thing we are trying to share **is the event**
- Critical section **in sleep()**
  - Starting before we test the posted flag
  - Ending after we put ourselves in the notify list
  - We want to prevent during this section:
  - Wakeups of the event
  - Other people waiting on the event
  - This is a **mutual-exclusion problem** that we already know how to solve using locks

Progress vs. Fairness

- Convoy can form

Semaphores and Locking

Lecture 9: Semaphores and Locks for Synchronization

Outline

- Locks
- Semaphores
- Mutexes and object locking
- Getting good performance with locking
    - Locking has bad performance implications

Our Synchronization Choices

- To reiterate:
    1. Don’t share resources
    2. Turn off interrupts to prevent concurrency
        - Only an option if in privileged mode
    3. Always access resources with atomic instructions
        - Usually not possible
    4. Use **locks to synchronize** access to resources
        - We need to use locks or higher level primitives
- If we use locks, we can
    1. Use **spin loops** when the resource is locked
    2. Use **primitives that block you (condition variables)** when your resource is locked and wake you later

Concentrating on Locking

- Locks are necessary for many synchronization problems
- To implement locks
    - Must always be correct
    - Else, someone may be locked forever or there will be improper synchronization
    - Ensure that locks are used in ways that **don’t kill performance**

Basic Locking Operations

- When possible concurrency problems occur, we
    1. Obtain a lock related to the shared resource
        - If we don’t get it, we **block or spin**
    2. Once you have the lock, **use the shared resource**
    3. **Release the lock** when finished
        - Need to do this in order to allow **other people to use the resource**
        - EX: unlock **before you exit the thread**
        - Usually when processes exit, OS will typically automatically release all locks it held

- Whoever implements the lock ensures that **no concurrency problems occur in the lock itself**
    - Using atomic instructions (test and set, compare and swap, etc.)
    - Or disabling interrupts
    - For **interrupt-based mutual exclusion**
    - Lock will have to be implemented **inside the OS**
    - Code itself **must be short** and have **no possibility of doing anything besides lock**

Semaphores

- Related to locks, but more complicated
- A theoretically sound way to implement locks
    - Extra functionality - critical for use in computer synchronization problems
    - Thoroughly studied and precisely specified
    - Not very usable
    - Gap between theory and implementation
    - **Correct in theory**
    - In computer, implementation of theory may not be perfect

The Semaphores We're Studying

- Introduced by Edsger Dijkstra
    - Cooperating sequential processes
    - The **classic synchronization mechanism**
    - Behavior is well specified and universally accepted
    - Foundation for many synchronization studies
    - Standard reference for all other mechanisms
    - **More powerful** than simple locks
    - Incorporate **a FIFO waiting queue** - can be used to build condition variables
    - Have a **counter** rather than a binary flag - more powerful

Semaphores - Operations

- Limited operations
- Semaphore has two parts:
    - An integer counter (initial value unspecified)
    - A FIFO waiting queue
    - P (proberen/test) - **"wait"**
    - Decrement counter, if count >= 0, return
        - You get what you were waiting for if you return
    - V (verhogen/raise) - **"post"** or **"signal"**
    - Increment counter
    - If the queue is not empty, **wake one of the waiting processes**
        - **Put it in the ready queue**
    - Semaphore only can do **two operations**

Using Semaphores for Mutual Exclusion

- Initialize semaphore count **to one**
    - We want mutually exclusive access
    - Count reflect **# of threads allowed to hold the lock**
    - Use P/wait operation to **take the lock**
    - The first will succeed
        - Decrement and checks if >= 0
        - Count decremented to 0
    - Subsequent attempts will block
        - Now counter will be < 0
    - Use V/post operation to release the lock
    - Restore the semaphore count to non-negative
    - If any threads are waiting, unblock the first in line
        - **Unblocked thread has not gotten it yet, the thread will repeat the P operation again**

Using Semaphores for Notifications

- Initialize semaphore count **to zero**
    - Completion events
    - Count reflects **# of completed events**
    - Use P/wait operation to **await completion**
    - If already posted, it will return immediately
    - Else, all callers will block until V/post is called
- USe V/post operation to **signal completion**
    - Increment the count
    - If any threads are waiting, unblock the first in line
    - **One signal per wait:** no broadcasts
    - Only wake up **the first in the queue**
    - As multiple occurrences happen of event, gradually people are pulled out of the queue

Counting Semaphores

- Initialize semaphore count to …
    - Count reflect **# of available resources**
    - Use P/wait operation to **consume a resource**
    - If available, it will return immediately
    - Else, all callers will block until V/post is called
    - Use the V/post to **produce a resource**
    - Increment the count = **free up a copy of the resource**
    - If any threads are waiting, unblock the first in line
    - One signal per wait: no broadcasts

Semaphores For Mutual Exclusion

```
struct account {
        struct semaphore s; //initialize count to 1, queue empty, lock 0
        int balance;
        ...
```

```
}
int write_check(struct account* a, int amount) {
	int ret;
	wait(&a->semaphore);	//get exclusive access to account
		if (a->balance >= amount){
			amount -= balance;
			ret = amount;
		} else
			ret = -1;
	post(&a->semaphore);	//release access to the account
	return(ret);
}
```

- Different from a lock
  - **Semaphores have queues!**
  - **Simple locks do not!**

Semaphores for Completion Events

```
struct semaphore pipe_semaphore = {0, 0, 0};
	//count = 0; pipe is empty
char buffer[BUFSIZE];
int read_ptr = 0, write_ptr = 0;
char pipe_read_char(){
	wait (&pipe_semaphore);	//wait for next input
	c = buffer[read_ptr++];	//get next input character
	if (read_ptr >= BUFSIZE)  //circular buffer wrap
		read_ptr -= BUFSIZE;
	return(c);
}
void pipe_write_string(char* buf, int count) {
	while (count-- > 0)  {
		buffer[write_ptr++] = *buff++;	//store next character
		if (write_ptr >= BUFSIZE) //circular buffer wrap
			write_ptr -= BUFIZE;
		post(&pipe_semaphore);		//signal that a char is available
	}
}
```

- Want a specific buffer order
  - Reader reads when buffer is full
  - Writer only writes when the buffer is empty

Implementing Semaphores

```
void sem_wait(sem_t* s) {
	pthread_mutex_lock(&s->lock);
	while (s->value<=0)
		pthread_cond_wait(&s->cond, &s->lock);
```

```
        s->value--;  //decrement only after you check, if you don't wait
        pthread_mutex_unlock(&s->lock);
}
void sem_post(sem_t* s) {
        pthread_mutex_lock(&s->lock);
        s->value++;
        pthread_cond_signal(&s->cond);
        pthread_mutex_unlock(&s->lock);
}
```

Limitations of Semaphores

- Very spartan and simple mechanism
  - Simple, have features
  - Designed for proofs, rather than synchronization
- They **lack many practical synchronization features**
  - Designed for theoretical usage
  - Limitations:
    - Easy to deadlock
    - Cannot check the lock without blocking
    - Do not support reader/writer shared access
    - No way to recover from a wedged V operation
    - No way to deal with priority inheritance
- Most OSes support them

Locking to Solve High Level Synchronization Problems

- Mutexes and object level locking
- Problems with locking
- Solving the problems (performance problems)

Mutexes

- Linux/Unix locking mechanism
- Intended to **lock sections of code**
  - Locks expected to be held briefly
  - Typically for multiple threads of the same process in pthread package
  - **Low overhead** and **very general**

Object Level Locking

- We don't want to lock code → we want to lock **objects/things**
- Mutexes **protect code** critical sections
  - Brief durations
  - Threads operating on the same data
  - All operating in a single address space
  - **Persistent objects are more difficult**
  - Don't do this with mutexes
  - Critical sections likely to be longer
  - Different programs can operate on them
  - May not even be running on the same computer

        - Solution: **lock objects** rather than code
        - Somewhat specific to object type

Linux File Descriptor Locking

- **int flock(*fd*, *operation*)**
- Thread locking between threads, doesn't work for multiple processes
- Supported *operation*s:
    - LOCK_SH - shared lock (multiple allowed)
    - LOCK_EX - exclusive lock (one at a time)
    - LOCK_UN - release a lock
    - Lock applies to **open instances of the same *fd***
    - Lock passes with the relevant fd
    - Distinct opens are not affected
    - Primarily used **in a single process with multiple threads**
    - **Multiple processes will not be aware**
    - Locking with flock() is **purely advisory**
    - Does not prevent reads, writes, unlinks

Advisory vs Enforced Locking

- **Enforced** locking - forcing lock, **done in implementation of object** or **the OS does it**
    - Done within the implementation of object methods
    - **Guaranteed to happen**, whether or not use wants it
    - May sometimes be too conservative
- **Advisory** locking - people/threads need to **follow the rules**
    - Convention that "good guys" are expected to follow
    - Users expected to lock object before calling methods
    - Give users **flexibility in what to lock, and when**
        - We know when locks will occur
    - Gives users **more freedom to do it wrong (or not at all)**
        - Bad synchronization
    - **Mutexes and flocks()** are advisory locks
    - No enforcement
        - If you aren't careful, benefits of synchronization can be lost
            - People aren't following the rules

Linux Ranged File Locking

- Use lockf for locking **between multiple processes**
- **int lockf(*fd*, *cmd*, *offset*, *len*)**
- Supported *cmd*s:
    - F_LOCK - get/wait for an exclusive lock
    - F_ULOCK - release a lock
    - F_TEST/F_TLOCK - test, or non-blocking request
    - *offset/len* specifies the portion of the file to be locked
    - Lock applies to file (not the open instance)
    - **Process specific**

        - Lock only given to that process
        - But other processes cannot access the file because it is locked
    - Closing any fd for the file **releases for all of a process' fds for that file**
        - If you have multiple file descriptors open for that file,
        - All are closed when one is closed
    - One party can hold the lock at a time
    - Locking may be enforced – **depends on the underlying file system**
    - May be advisory, may be enforced
    - Based on implementation of the file system
    - Nothing will check in OS whether it is mandatory
    - **Use as if it was an advisory lock**

Locking Problems

- Performance and overhead
- Contention
    - Convoy function
    - Priority function

Performance of Locking

- Locking often performed as an OS system call
    - Particularly for enforced locking
- Causes a **system call overhead**
    - Typical system call overheads for lock operations
    - If frequently used, **high overheads**
    - Even if it is not in the OS, **extra instructions are run to lock and unlock**
    - There is a cost paid for using locks

Locking Costs

- Locking called when you **need to protect critical sections to ensure correctness**
- Many critical sections **are very brief** – we don't want to hold the lock for very long
- **Overhead of the locking operation** may be **much higher** than the time spent in the critical section
    - Since the critical section is so short
    - Pay more to get and release lock than to actually run the program

What If You Don't Get Your Lock?

- Then you block
- Blocking is **much more expensive than getting a lock**
    - Microseconds to yield and context switch
    - Milliseconds if swapped-out or a queue forms
    - Performance depends on **conflict probability**

  $C^{expected} = (C^{block} * P^{conflict}) + (C^{get} * (1-P^{conflict}))$
- Depends on probability of not getting the lock when you try and acquire it

The Riddle of Parallelism

- Parallelism allows **better overall performance**
    - If one task is blocked, CPU runs another
    - So you must be able to run another
    - Concurrent use of shared resources is difficult
    - So we **protect critical sections** for those resources by locking
    - But **critical sections serialize tasks and thus, kill parallelism**
    - Meaning that other tasks are blocked
    - To prevent bad things from happening in parallelism, we kill parallelism and performance benefits
    - How do we balance performance and correctness?

What if Everyone Needs One Resource

- One process gets the resource
- Other processes get in line behind him (queue, some other data structure)
    - Forming a **convoy**
    - Processes in a convoy are **all blocked waiting** for the resource
    - Everyone is trying to get access to it
    - Parallelism is eliminated
    - Synchronize to access the resource
    - That resource **becomes a bottleneck**

Probability of Conflict

- More threads → probability of a conflict goes up

Convoy Formation

- Wait time increases as the line gets longer
    - Unless mean inter-arrival time goes down
- If the wait time reaches the **mean inter-arrival time** the line becomes permanent, parallelism ceases

Performance: Resource Convoys

- To avoid a convoy:
    - Avoid building resources everybody needs
    - Avoid having so many processes/threads scheduled

Priority Inversion

- **Priority inversion** can happen in priority scheduling systems that use locks
- Happens when you run **at a lower priority than you are supposed to be running**
    - A low priority process P1 has mutex M1 and is preempted
    - A high priority process P2 blocks for mutex M1
    - Process P2 is effectively reduced to priority of P1
        - P2 cannot run since P1 is running
        - Can’t run until P1 releases the lock
    - Depending on the situation, results could be anywhere from inconvenient to fatal

Handling Priority Inversion Problems

- In a priority inversion, **lower priority task runs because of a lock held elsewhere**
    - Preventing a higher priority task from running

Solving Priority Inversion

- When a low priority task is holding a lock, increase its priority
- Temporarily increase the priority and drop it back down when it is finished
- **Priority inheritance:** a general solution to this kind of problem

Solving Locking Problems

- Reduce overhead
    - Keep locking efficient and cheap
- Reduce contention
    - Reduce need for locking
    - Less locking, less serialization → better performance

Reducing Overhead of Locking

- Not much more to be done here
- Locking code in OS is usually **highly optimized**
- Typical users cannot do better

Reducing Contention

- **The only option**
- Do less locking
- Eliminate the critical section entirely
    - Eliminate shared resource, use atomic instructions
- Eliminate preemption during the critical section
- Reduce time spent in critical section
    - Make the critical section short
- Reduce the frequency of entering critical section
- Reduce exclusive use of the serialized resource
- Spread request out over more resources
    - Make copies → spread out the resources

Eliminating Critical Sections

- Eliminate shared resource
    - Give everyone their own copy
    - Find a way to work without it
- Use atomic instructions
    - Only possible for simple operations
- Great **when you can do it**
    - It is the best solution you can get
    - But often, **not possible**

Eliminate Preemption in Critical Section

- If your critical section cannot be preempted → no synchronization problems
    - Only works in a **single core environment**
    - May require **disabling interrupts**
    - Not always an option

Reducing Time in Critical Section

- Avoid staying in the critical section for a long time
- Eliminate **potentially blocking operations**
    - EX: allocating memory, I/O
    - Do these **before or after taking/releasing the lock**
    - **Minimize code** inside the critical section
    - Only put code that is subjective to race conditions
    - Move all other code outside
    - **Especially calls to other routines**
- Cost: this may **complicate the code**

Reduced Frequency of Entering Critical Section

- Call critical section less frequently to reduce contention
- Try to use critical section less often
    - Less use of high-contention resource/operations
    - **Batch operations**
        - Do more things at once in one single call
- Consider: **sloppy counters**
    - Every thread has its own **local counter**
        - When you update the counter in the thread, you only update the local counter
    - When the local counter **hits a certain point**, lock and update the central counter
        - Allows you to **unlock the global counter less frequently**
    - Move most updates to a private resource
    - Costs:
        - Global counter is not up-to-date always
        - Thread failure loses many updates
        - More complexity, loss of accuracy
    - Works well if you don't care about the value in the middle
    - Alternative:
        - Sum single-writer private counters when needed

Remove Requirement for Full Exclusivity

- We do not always have to lock shared data structures
- read/write locks - two types of locks
- Reads and writes are not equally common
- **Only writers require exclusive access**
- read/write locks
    - Allow many readers to **share a resource**
    - Only enforce **exclusivity when a writer is active**
    - Policy: when are writers allowed in?
        - Potential for starvation if writers must wait
    - Can get as many read locks as you need
    - Only need one read/write lock

        - But you can't read/write when someone is writing!
        - Potentially large performance win
        - Starvation issue for writers and there is **no great way to solve this**
        - Adds complexity

Spread Requests Over More Resources

- Change **lock granularity**
- **Coarse grained**: one lock for many objects
    - Simpler, and more idiot-proof
    - Greater resource contention (threads/resource)
- **Fine grained:** one lock per object (or sub-pool)
    - Spreading activity over many locks **reduces contention**
    - Dividing resources into pools **shortens searches**
    - A few operations may lock multiple objects/pools
    - time/space overhead associated with more locks, more gets/releases
    - Error-prone: harder to decide what to lock when

Lock-Granularity - Pools vs. Elements

- Most operations **lock only one buffer within the pool**
- But some operations require **locking the entire pool**
    - Adding to the cache
    - Looking for elements in the cache
    - The pool lock **could become a bottleneck** so instead
    - Minimize its use
    - reader/writer locking
    - Sub-pool use

The Snake in the Garden

- Locking is great for preventing improper concurrent operations
- With care, they can usually be made to perform well, but not well enough
- **Deadlock:** occurs when locking leads to the system freezing forever
    - Kills your program performance

Reading

Arpaci-Dusseau Ch. 29 - Lock-Based Concurrent Data Structures

- Locks in data structures to make them thread-safe

29.1 Concurrent Counters

- Simple data structure

→ Simple But Not Scalable

- Non-synchronized counter is trivial
- How do we make it thread-safe?
    - Add a single lock that is acquired when calling routine manipulates the data structure

        - Release the lock when returning from the call
    - Similar to data structure that is built with **monitors**
        - Locks are acquired and released automatically as you call and return
    - Data structure works → but there are performance issues
    - Two threads end up being **much slower** than a single thread
    - Perfect scaling would mean: threads compute just as quickly on multiple processors as the single thread does on one

→ Scalable Counting

- Need scalable counting
    - Without it, Linux systems would suffer scalability problems
- One possible approach, **sloppy counter**
    - Represents a single logical counter via numerous local counters (one/CPU core)
        - EX: four CPU cores → four local counters and four local locks for each counter, one for the global counter
- Basic Idea:
    - Thread on a core wishes to increment a counter
        - Increment the local counter → this is synchronized via the local lock corresponding to it
    - Each CPU has its own local counter
    - Threads can update local counters without contention
    - Counter updates are **scalable**
    - To keep global counters up to date
        - Local values are periodically transferred to the global counter
    - How?
        - Acquire the global lock
        - Increment by local counters value
            - Local counter is then set to 0
- How often does this transfer happen?
    - Determined by threshold, called **S (sloppiness)**
    - **Smaller S** → more the counter behaves like the non-scalable counter
        - Poor performance
    - **Larger S** → the counter is more scalable but the global value is further off the actual count
        - High performance but the global count lags
- Getting an exact value means **acquiring all local locks and the global lock** → not scalable

29.2 Concurrent Linked Lists

- Focus on a concurrent insert
- One (incorrect) approach:
    - Lock in insert routine upon entry
    - unlock/release when it exists
    - Must also release if malloc() fails → error prone

- Challenge: routine needs to **remain correct under insert** but also avoid an extra case **where failure path also requires an unlock**
- Instead, lock ONLY **the critical section** of the insert
    - Common exit path in the lookup code
- Malloc is thread safe
- We only need to lock **when updating the shared list**

→ Scaling Linked Lists

- They do not scale well
- **Hand-over-hand locking** or **lock coupling** should be used instead of one single lock for the entire list → have **one lock/node**
- When you traverse → the code graves the next node's lock and then releases the current node's lock
- High degree of concurrency but **difficult to make faster**
    - There is the overhead of **acquiring and releasing locks** for each node that is **high**
- Maybe use a **hybrid** (grab a new lock every so many nodes)

29.3 Concurrent Queues

- Standard method to make a concurrent data structure
    - Add a **huge lock**
- **Michael and Scott Concurrent Queue** has two locks
    - One lock at the head of the queue
    - One lock at the tail of the queue
        - Enables concurrency of enqueue and dequeue instructions
    - Add dummy node - separates the head/tail operations
- This queue - does not fully meet the requirements of multithreaded applications

29.4 Concurrent Hash Table

- Hash table that doesn't resize
- One lock/hash bucket
- Performs and scales well

29.5 Summary

- Lessons:
    - Be careful with acquisition and release of locks near control flow change
    - Performance problems should only be fixed if they appear
        - Avoid premature optimization
- Data structures without traditional locks: non-blocking data structures

Arpaci-Dusseau Ch. 30 - Condition Variables

30.2 The Producer/Consumer (Bounded Buffer) Problem

- producer/consumer problem AKA **bounded buffer problem** → synchronization problem → leads to semaphores

- Producer and consumer threads
    - **Producer**: generate data items and place them in the **buffer (the bounded buffer)**
    - **Consumer:** grabs those items from the buffer and consumes them
- Bounded buffers are a **shared resource** so we must synchronize it
    - EX: you can only get data from the buffer if it is full and only write to it when it is empty

→ A Broken Solution

- Putting locks around code doesn't work → we need a condition variable
- Producer wants to fill the buffer → must wait until it is empty
- Consumer wants to read from the buffer → waits for it to be full
    - These two are **similar conditions**
- With a single producer and consumer, we can have a single condition variable and lock work
    - Fails in the case of multiple consumers
    - If a producer wakes a thread up but the state of the buffer is changed before the thread gets to run → BAD
- **Mesa Semantics**: signaling threads only wakes them up → serves as a hint that the state as changed but **there is no guarantee** that when the thread runs, the state is as desired
- **Hoare Semantics**: provides a stronger guarantee that woken thread is run immediately after it awakens

→ Better, But Still Broken: While, Not If

- Change the if to a while
- When threads wake up, they check the state of the shared variable
- Simple rule with Mesa Semantics with condition variables
    - **Always use while loops**
- Another problem: having only one condition variable
- Consumers may wake other consumers when the buffer is empty → BAD
- Signaling should be more directed
- A consumer should only wake other producers, not other consumers (and vice versa)

→ The Single Producer/Consumer Solution

- Use 2 condition variables
    - Properly signal the type of thread that **should wake up** when a system changes state
- Producers **wait on empty and signal full**
- Consumers **wait on full and signal empty**

→ The Correct Producer/Consumer Solution

- Enable more concurrency and efficiency by adding **more buffer slots** so multiple values can be produced and consumed
    - In a single producer/consumer case: it will reduce context switches

        - In multiple producer/consumer case: allows concurrent producing or consuming
    - Changes:
        - Producers sleep when all buffers are full
        - Consumers sleep if all buffers are empty

30.3 Covering Conditions

- Another application of condition variables
- EX: memory allocation
    - When emory is free, which thread should be signaled
    - Some may release enough memory for multiple threads, or for one or two threads
- Instead of using pthread_cond_signal() → use **pthread_cond_broadcast()** which wakes up ALL sleeping threads
- Guarantees that all threads should be awakened once
    - **Can help prevent starvation**
- BUT, **negative impact on performance**
    - Some threads shouldn't be woken up yet
    - Those threads wake, and they check the condition, and immediately go back to sleep
- **Covering condition**: covers all cases where a thread needs to wake up

Arpaci-Dusseau Ch. 31 - Semaphores

- Semaphores used for both locks and condition variables
    - Because we need both to solve a broad range of concurrency problems

31.1 Semaphores: A Definition

- **Semaphore:** object with an integer value that we can manipulate with two routines:
    - POSIX: sem_wait() and sem_post()
- Need to initialize it to a value before calling other routines to interact with it
    - **The initial value determines behavior**
- After initialization, we can call either
    - **sem_wait():** decrement value of semaphore by 1 and waits to see if the value is negative
    - **sem_post():** increment value of a semaphore by one; if there are one or more threads waiting, awaken one of them
- Aspects of each:
    1. sem_wait(): can either return right away (because the value of the semaphore was previously 1 or higher) OR, caller suspends execution waiting for a post
        - Multiple threads may call sem_wait()
        - Waiting threads **are queued**

2. sem_post(): does not wait for a condition
    - Increments the value
    - Awakens waiting threads

- The value of the semaphore (when negative) = **the number of waiting threads**

31.2 Binary Semaphores (Locks)

- Using a semaphore as a lock
- Surround the critical section code with a sem_wait() & sem_post() pair
    - Initial value of the semaphore: **should be 1**
- EX: two threads
    - Thread 0 calls sem_wait()
        - Decrements value to 0
        - Waits if value is not greater than or equal to 0 → returns in this case
        - Thread 0 enters the critical section
    - Thread 0 is holding the lock (has called sem_wait() but not sem_post())
        - Thread 1 tries to enter by calling sem_wait()
            - Decrements the value of the semaphore to -1 and thus, has to wait
            - Thread 1 is in a sleeping state
        - When thread 0 posts, the value is now 0 and thread 1 can increment and run
- Semaphores used as locks are called **binary semaphores**

31.3 Semaphores for Ordering

- Semaphores - useful for **ordering events in a concurrent program**
    - Use as an **ordering primitive** (similar to condition variables)
- EX: parent and child, parent is waiting for the child
    - Have the parent call sem_wait() and child call sem_post()
    - The initial value should be **0**

    1. Parent creates child but child has not run yet
        - Parent calls sem_wait() before child calls sem_post() – we want the parent to wait
        - Parent will wait because the value is not greater than 0
        - Parent decrements then sleeps
        - When child runs and posts, the value will be 0
    2. Child completes before parent calls sem_wait()
        - Value is incremented to 1
        - Parent then decrements to 0 → returns immediately

31.4 The Producer/Consumer (Bounded Buffer) Problem

- Look at the producer/consumer problem AKA the bounded buffer problem

→ First Attempt

- Two semaphores: empty and full
    - Use to indicate when a buffer entry has been emptied or filled
- Scenario: two threads, producer and consumer

        - Consumer runs first and decrements full to -1
            - Now waits for someone to call sem_post on full
        - Producer runs and calls sem_wait on empty
            - (empty initialized to MAX(1)) → now decremented to 0 and producer puts the data in
        - Producer then calls sem_post → waking up the consumer
        - Two possibilities:
            - Producer continues and then is blocked
            - Consumer runs and consumes the buffer
    - Works but **with more buffers (MAX > 1)** → multiple consumers and producers can cause a **race condition**
    - Say, process A fills the buffer but doesn't get to increment before it is interrupted
    - When Process B begins to fill the buffer, Process A's data can be replaced
        - Data from the producer can be lost

→ A Solution: Adding Mutual Exclusion

- Filling of buffers and incrementing of index into buffer is actually **a critical section**
    - Use a binary semaphore to create locks
- But another issue is **deadlock**

→ Avoiding Deadlock

- Consumer holds the mutex and waits for someone to signal full
- Producer could signal full but is waiting for the mutex
- Both are **waiting for each other → DEADLOCK**

→ At Last, a Working Solution

- Need to reduce the **scope of the lock**
    - Move what the mutex acquires and releases to just around the critical section

31.5 Reader-Writer Locks

- Desire to create more flexible locking primitive that admits that different data structures need different kinds of locking
- **Reader-writer lock:** lock developed to support the following operation
    - Concurrent list operations: insert and lookup
        - Insert changes the state of the list
        - Lookup is reading the data structure
    - If we can guarantee that no read is happening, we can allow concurrency in lookups
- Thread wants to update data → call **rwlock_acquire_writeLock** to acquire a write lock and call **rwlock_release_writeLock** to release it
    - Use a writeLock semaphore to ensure only a single writer can acquire the lock
- For readers:

        - Reader acquires the lock and increments the number of readers → track the number of readers inside the data structure
        - rw_lock_acquire_readLock
            - Reader also acquires the **write lock** by calling sem_wait on the writeLock semaphore
            - Later release it
        - When a reader acquires a read lock, more readers can acquire it too
    - BUT, **any thread wanting the write lock has to wait** until **all readers** have finished
    - CONS: fairness → readers can starve the writer
    - Adds more overhead and may not necessarily speed up performance

31.6 The Dining Philosophers

- **The Dining Philosophers Problem**
    - Five "philosophers" sitting around a table with a fork between each pair
    - When philosophers think, they don't need the works
    - BUT, when the philosophers eat, a philosopher needs 2 fork, one of the left and another on the right
    - Contention for forks and synchronization problems ensue
- Loop for each philosopher:

```
while(1) {
        think();
        getforks();
        eat();
        putforks();
}
```

- getforks() and putforks() need to be written in such a way that there is no deadlock and no philosopher starves BUT, while concurrency is still high
- Need 5 semaphores, one for each fork

→ Broken Solution

- Initialize each semaphore to 1
- To acquire a lock
    - Grab a lock on each one
        - 1st: left
        - 2nd: right
    - Release when done eating
- PROBLEM: deadlock still possible
- If each philosopher grabs the fork on the left before any philosopher grabs the right one
    - All will be holding left forks and waiting for one other forever

→ A Solution: Breaking the Dependency

- Change how forks are acquired by **at least one** of the philosophers
    - One philosopher **acquires the forks in a different order**
- If that philosopher tries to grab right before left

    - No situation where each philosopher grabs one work and has to wait
    - Cycle of waiting is then broken

31.7 How to Implement Semaphores

- Use low-level synchronization primitives, locks, and condition variables to build zemophores
- Use: one lock and one condition variable and one state variable

Deadlocks

Lecture 10: Deadlocks – Problems and Solution

Outline

- The deadlock problem
  - Approaches to handling this problem
- Handing general synchronization bugs
- Simplifying synchronization

Deadlock

- Situation:
  - Process A has 1
  - Process B has 2
  - But Process A needs 2 and Process B needs 1
  - Can't get the resource because they **are waiting for each others' lock**
- **Deadlock**: a situation where two entities have each locked some resource
  - Each needs the other's locked resource to continue
- Neither will unlock they lock both resources
- Hence, **neither can ever make process**

The Dining Philosophers Problem

- Five philosophers, five plates of pasta, five forks
- Philosophers eat whenever they choose to
- Philosophers will not negotiate with each other
- A philosopher needs two forks to eat but **must pick them up one at a time**
- Want solutions that work **every time**

Dining Philosophers and Deadlock

- This problem is **the classical illustration** of deadlocking
- Created to illustrate deadlock problems
- Very artificial problems
  - Carefully designed to cause deadlocks
  - Changing the rules can eliminate deadlocks
    - But it wouldn't be able to illustrate deadlock
  - Actually, one point is to **see how changing the rules solves the problem**
- Helps us see **what really** causes deadlock
  - How to resolve it
  - What characteristics do we see in deadlock problems

Why Are Deadlocks Important?

- Because they happen
  - Especially if you lock multiple resources
- A **major peril** in cooperating parallel processes
  - Relatively common in complex applications

        - Result in catastrophic failures
    - Finding them through debugging is very difficult
        - Happen intermittently and are hard to diagnose
        - They are **easier to prevent at design time**
        - **Not deterministic**
            - Depends on how the system is running
    - Once you understand them, you can **avoid them**
        - Most deadlock results from careless/ignorant design
    - Design code to avoid deadlock

Deadlocks May Not Be Obvious

- Process resource needs are **ever-changing**, depend on
    - What data they are operating on
    - Where in computation they are
    - What errors have happened
- Modern software depends on **many services**
    - Most are ignorant of one another
    - Each requires numerous resources
- Services **encapsulate complexity**
    - We build our software off of other services
    - But we do not know what resources they require
    - Or how/when they are serialized

Deadlocks and Different Resource Types

- Deadlocks occur because we **lock resources**
- **Commodity Resources**
    - Clients need an amount of it (e.g., memory)
    - Deadlocks result from **over-commitment**
    - Avoidance can be done in resource manager
- **General resources**
    - Clients need a specific instance of something
        - Particular file or semaphore
        - Message or request completion
    - Deadlocks result from **specific dependency relationships**
    - Prevention can be done at **design time**

Four Basic Conditions For Deadlock

- For a deadlock to occur, these conditions must hold:
    1. **Mutual exclusion**
    2. **Incremental allocation**
    3. **No preemption**
    4. **Circular waiting**
- All four **must be true for any chance of deadlock**
- If any are not true, you **are guaranteed to have no deadlock**
- Preventing deadlock comes down to ensuring that **at least one** of these conditions is not true

Deadlock Condition 1: Mutual Exclusion

- Mutual exclusion provided using locks
- The resources in question **can each only be used by one entity at a time**
- If multiple entities can use a resource, just give it to all of them
- If only one can use it, once you've given it to one, nobody else gets it
    - Until resource holder releases it

Deadlock Condition 2: Incremental Allocation

- processes/threads are **allowed to ask for resources whenever they want**
    - As opposed to getting everything they need before they start
    - They can **ask for locks as they need them**
- Multiple resources that require mutual exclusion
    - As processes run, they need **mutually exclusive access to different parts**
        - They get the locks for different parts of section (incremental allocation)
- If they must pre-allocate all resources, either:
    - They get all they need and run to completion
    - They don't get all they need and abort
- In either case, no deadlock
- AKA Hold-and-wait
    - Threads hold locks while they ask for other ones
    - They block while waiting for a thread

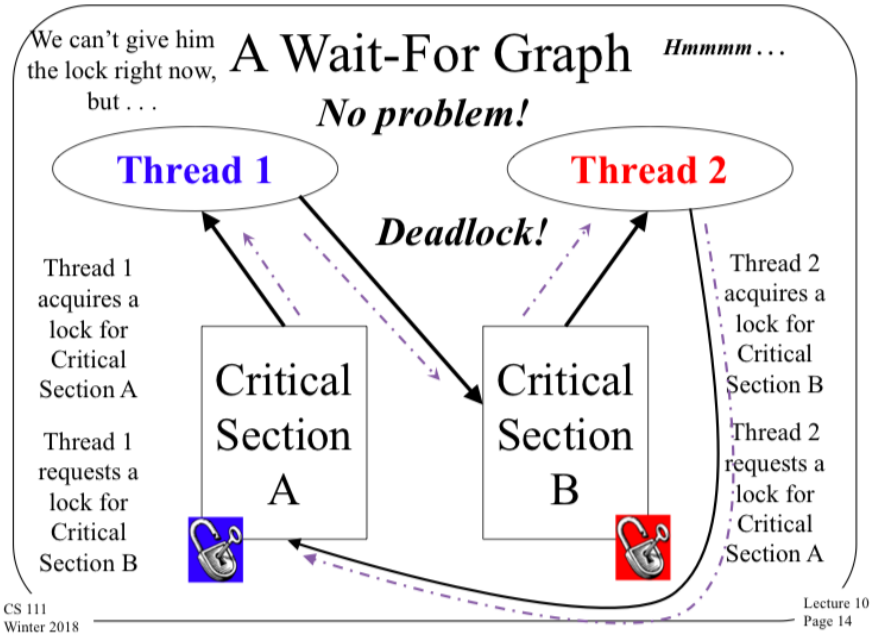
Deadlock Condition 3: No Preemption

- **Not the same as scheduling preemption**
- **Lock preemption:** giving a lock to another thread even if the first thread does not want to give it up
- When an entity has reserved a resource, you **can't take it away from him**
    - Not even temporarily
- If you can, deadlocks are simply resolved by **taking someone's resource away**
    - To give it to someone else
- But if you can't take anything away from anyone, you're stuck
- **Preemption of locks:** OS takes away locks from different processes
    - Sometimes BAD → could be in the middle of an operation that should be atomic

Deadlock Condition 4: Circular Waiting

- A waits on B which waits on A
- In graph terms, there's a **cycle in a graph of resource requests**
- Could involve a lot more than two entities
- But if there is no such cycle, **someone can complete without anyone releasing a resource**
    - Long chain of dependencies will eventually unwind
    - Maybe not very fast
- Someone will **eventually** get whatever they need → no deadlock

A Wait-For-Graph



Deadlock Avoidance

- Use methods that **guarantee that no deadlock can occur**, by their nature
- **Advance reservations** (for commodity resources)
  - The problems of under/over-booking
  - The Bankers' algorithm
- Practical commodity resource management
- Dealing with rejection
  - Requirement when using advance reservation policy
- Reserving critical resources

Avoiding Deadlock Using Reservations

- Advance reservations for **commodity resources**
  - Resource manager tracks **outstanding reservations**
  - **Only grants reservations** if resources are available at the beginning of program
- **Over-subscriptions** are detected early
  - Before processes ever get to the resources
- Client must be **prepared to deal with failures**
  - But these do not result in deadlocks
- Dilemma: **overbooking** vs **underutilization**
  - Under-utilization: people overestimate what they need
    - Reservations in this circumstance mean that people ask for a lot but don't use all of it
    - Wasteful
  - Overbooking: allow reservations of more

Overbooking vs. Under Utilization

- Processes generally cannot predict their resource needs perfectly
- To ensure they have enough, **tend to ask for more** than they will ever need
- Either the OS:
    - Grants the request until **everything's reserved - underutilization**
        - Most of it won't be used → wasted
    - Or grants requests **beyond the available amount - overbooking**
        - Someone won't get a resource he reserved

Handling Reservation Problems

- Clients usually don't need all resources all the time
    - Different resources are used during different phases
- All clients also don't need max allocation at the same time
- Question: can one **safely over-book resources?**
    - We want to minimize under utilization of resources
- What is a **"safe" resource allocation"?**
    - Everyone will be able to complete
    - Some people may have to wait for others to complete
    - Ensure no deadlocks

Commodity Management in Real Systems

- Advanced reservation mechanisms are common
    - Memory reservations
    - Disk quotas
    - Quality of Service contracts
        - Tells you about a limit
        - Overbooking
        - Ex: cloud computing
- Once granted, system must guarantee reservations
    - Allocation failures only happen at reservation time
    - Before new computation has begun hopefully
    - Failures will not happen at request time
    - System behavior will be more predictable and easier to handle
- Clients still need to deal with **reservation failures**

Dealing with Reservation Failures

- Resource reservation eliminates deadlock
- Applications need to deal with **reservation failures**
    - Design should handle failures gracefully
    - Should be able to report failure to requester
    - Must be able to continue running
        - Critical resources **must be reserved at start-up time**

Isn't Rejecting Application Requests Bad?

- Not great but **better than failing later**
- With advance notice, the application may be able to adjust the service to not need to resource
- If the application is in the middle of servicing a request

        - There is an expectation that the process runs to completion
        - We may have other resource allocated
        - The request may only be half-performed
        - If we fail then, all of this will have to be unwound
        - Can be **complex** or even **impossible**
    - We need to recover from the fact that only some part of it is done

System Services and Reservation

- System services **must never deadlock for memory**
- Potential deadlock: **swap manager**
    - Invoked to swap out process to **free up memory**
    - May need to allocate memory to build I/O request
    - If no memory is available, unable to swap out processes
    - If it can't free up memory, the system **wedges**
- Solution:
    - Make sure we **reserve (truly) some resources for critical operations**
    - Preallocate and hoard **a few request buffers**
        - **Reserve system resources for swap**
    - Keep reusing the same ones over and over again
    - A little bit of hoarded memory is a small price to pay to avoid deadlock
        - Slower because you have only a few buffers
        - But prevents deadlock

Deadlock Prevention

- Getting rid of conditions for deadlock
- Deadlock avoidance: tries to ensure that no lock ever causes deadlock
- Deadlock prevention: tries to ensure that a particular lock doesn't cause deadlock
- We do this by attacking ¼ necessary conditions for deadlock
- If any **one** doesn't hold → **no deadlock**

Four Basic Conditions for Deadlocks

- For a deadlock to occur, these conditions must hold:
    1. Mutual exclusion
    2. Incremental allocation
    3. No preemption
    4. Circular waiting

1. Mutual Exclusion

- This is a great solution if it works
- Deadlock requires mutual exclusion
    - P1 having the resources precludes P2 from getting it
- You can't deadlock over a **sharable resource**
    - Perhaps maintained with some atomic instructions
    - Even reader/writer locking can help
        - Readers can share, writers may be handled in other ways
- Can't deadlock on **your own private resources**

    - Giving each process its own private resource
    - Unless you can prevent **mutual exclusion for everything** → this simply makes deadlock less likely

2. Incremental Allocation

- Deadlock requires you to **block holding resources while you ask for others**
- Hold a lock while you request a lock
- You can either have one lock forever OR request all locks at once

1. **Allocate all of your resources in a single operation**
    - If you can't get everything, system returns failure and locks nothing
    - When you return **you have all or nothing**
2. **Non-blocking requests**
    - A request that can't be satisfied immediately will fail
    - When you request, if you don't get the lock, you **return or spin** but you **don't block**
        - Spinning forever is technically not deadlock but it still isn't good
3. **Disallow blocking while holding resources**
    - You must **release all held locks prior to blocking**
    - Reacquire them again after you return
    - Release all locks → people waiting will get your locks

Releasing Locks Before Blocking

- Could be blocking **for a reason not related to resource locking**
- Undo all work then release the lock
    - You will have to redo the work and remember it
- How can releasing locks before you block help?
- Won't deadlock just occur when you attempt to reacquire them
    - When you reacquire them → required to do so in a single **all-or-none transaction**
    - Transaction does not involve **hold-and-block** so there is no possibility of deadlock

3. No Preemption

- **Lease:** like a lock but **only valid for a certain amount of time**
- Deadlock can be broken by **resource confiscation**
    - Resource "**leases**" with time-outs and "**lock breaking**" - **preemption**
        - This needs to be done for **all locks** for this to work
    - Resource can be seized and reallocated to the new client
- Revocation **must be enforced** if trying to use leases
    - Need a **lock manager to enforce**
    - Invalidate the previous owner's resource handle
    - If revocation is not possible, then you **kill the previous owner**
- Some resources may be damaged by lock breaking
    - Resource needs to be in a **consistent state when the lease is taken away**
    - Previous owner may be in the **middle of a critical section**

- May need to include mechanisms to **audit/repair resource**
- Resources must be designed with **revocation in mind**
  - When you lose the lease, need to have a **fall back position**
    - May need to roll back or be in a bad state

When Can the OS "Seize" A Resource?

- This method is only useful for resources **that require OS help**
  - Otherwise, you can't enforce the policy
- When it can revoke access by invalidating a process' resource handle
  - If a process has to use a **system service to access the resource**, that service can no longer honor requests
- Only possible if the thread/process cannot use the resource without asking OS for permission
  - You cannot take away a resource form a process **if it doesn't need the OS to access the data** → instead, you have to **terminate the process**
- When is it **not possible** to revoke a process' access to a resource?
  - If the process has **direct access to the object (EX: data structures inside the process itself)**
    - Revoking requires destroying address space
    - Killing the process as a result

4. Circular Dependencies

- Preventing this ensures that there is no deadlock
- Use **total resource ordering**
  - All requester allocate **resource in the same order**
    - Ask for resources in a specific order
    - No possibility of a circular wait
  - First allocate R1 and then R2 afterwards
  - Someone else may need R2 but doesn't need R1
- Assumes that **we know how to order the resources**
  - Order by resource type (groups before members)
  - Order by relationship (parents before children)
  - May not be feasible and may be complex
- May require a **lock dance**
  - You have 5, want 3 and 5, need to release 5 and then get 3 and get 5 back again
  - Release R2, allocate R1, reacquire R2

Which Approach Should You Use?

- **No one universal solution**
  - Depends on characteristics of a system
  - Don't need one solution for all resources
  - Need one for each resource
- Solve each individual problem any way you can
  - Make resources shareable when possible
  - Use reservations for commodity resources

        - Ordered locking or no hold-and-block where possible
        - As a last resource, leases and lock breaking
    - OS must prevent **deadlocks in all system services** that the OS provides
        - But applications are responsible for their own behavior

One More Deadlock "Solution"

- Ignore the problem
- Deadlocks are improbable
- Avoiding them or preventing them may be expensive

Deadlock Detection and Recovery

- Allow deadlocks to occur
- **Detect them** once they have happened
    - As soon as possible preferably
- Do something to break the deadlock and allow someone to make progress
- Is this a good approach in general or when you don't want to avoid or prevent deadlocks?

Implementing Deadlock Detection

- To detect all deadlocks, need to **identify all resources that can be locked**
- Maintain a **wait-for-graph** or equivalent data structure
- When a lock is requested, the structure is **updated and checked for deadlock**
    - We analyze the graph and see if there is a cycle
    - Might it not just be better to **reject the lock request?**
    - And **not let the requester block?**
- To protect → maintain a structure that can prevent → might as well prevent it

Dealing with General Synchronization Bugs

- Deadlock detection seldom makes sense → take a performance hit
    - Extremely complex to implement
    - Only detects true deadlocks for a known resource
    - Not always clear cut what you should do if you defect one
- service/application **health monitoring** is better → good general choice
    - Monitor application progress
    - Submit test transactions
    - If the response takes too long → declare the service to be "hung"
- Health monitoring is **easy to implement**
    - Need to know what the system **should be doing**
- Can detect a **wide range of problems**
    - Deadlocks, live-locks, infinite loops & waits, crashes

Related Problems Health Monitoring Can Handle

- **Live-lock**
    - Process is running but won't free R1 until it gets message
    - Process that will send the message is blocked for R1
    - Technically not deadlock
    - Message – isn't a lockable resource → not deadlock
- Sleeping Beauty, waiting for "Prince Charming"

        - Process is blocked, **awaiting some completion that will never happen**
    - **Priority inversion** hangs
        - Talked about this before
    - **None of these are true deadlock**
        - Would not be prevented or detected by a deadlock detection program
        - All leave the system **just as hung as deadlock would**
    - **Health monitoring** would handle them

How to Monitor Process Health

- Look for obvious failures
    - Process exits or core dumps
- Passive observation to detect hangs
- External health monitoring
- Internal instrumentation

What To Do With "Unhealthy" Processes?

- **Kill and restart "all of the affected software"**
    - In complex systems, many processes are working together
- How many and which processes do we kill?
    - As many as necessary but **as few as possible**
    - Hung processes may not be the broken one
- How will kills and restarts affect current clients?
    - Depends on the service APIs and/or protocols
    - Apps should be designed for **cold/warm/partial restarts**
- Highly available systems define restart groups
    - Groups of processes to be started/killed as a group
    - Define inter-group dependencies (restart B after A, etc.)
- Should be able to **handle restarts**
    - Cold restart: start from beginning
    - Warm restart: restore saved state

Failure Recovery Methodology

- Retry if possible, not forever
- Roll-back failed operations and return an error
- Continue with reduced capacity or functionality
- Automatic restarts (cold, warm, partial)
- Escalation mechanisms for failed recoveries

Making Synchronization Easier

- Locks, semaphores, mutexes are hard to use correctly
    - May not be used when needed
    - May be used incorrectly
    - Can lead to deadlock, livelock, etc.
- Need to **make synchronization easier for programmers**

One Approach

- Identify **shared resources**
    - Objects whose methods **may require synchronization**

        - Identify problematic elements
        - Resources are what we are really synchronizing
    - Write code to **operate on those objects**
        - Just the code
        - Assume that all **critical sections will be serialized** once we specify that they need to be
    - **The compiler** generates the serialization
        - Automatically generated locks and releases
        - Using the appropriate mechanisms
        - Correct code in all required places

Monitors - Protected Classes

- Most common in **object oriented languages** where code is organized by objects
- Each **monitor object** has a **semaphore -** need to specify that it is a monitor
    - **Automatically acquired** on any method invocation
    - **Automatically released** on method return
- Good encapsulation
    - Developers don't need to identify sections
    - Clients don't need to worry about locking
    - Protection is **completely automatic**
- High confidence
- Does not prevent deadlock
    - Only ensures that the entire class is locked

Monitors: Simplicity vs. Performance

- Monitor locking is **very conservative**
    - Lock the entire object **on any method**
    - Locks for the entire duration of any invocation
- Can create **performance problems**
    - Because they eliminate conflicts by eliminating parallelism
    - If a thread blocks in a monitor, **a convoy can form**
- This is an example of **coarse-grained locking** which can create bottlenecks
    - Fine-grained locking is difficult and more error prone

Evaluating Monitors

- Correctness: **mutual exclusion** is assured, no synchronization issues
- Fairness: semaphore queue **prevents starvation**
- Progress: inter-class dependencies **can cause deadlocks**
- Performance: coarse-grained locking **is not scalable**
    - Complex methods will cause performance problems
    - With short methods, the performance is OK

Java Synchronized Methods

- Don't synchronize the entire class
- Each object has an **associated mutex** - not used for all methods
    - Only acquired for **specific methods**
        - Not all object methods need to be synchronized

        - Nested calls do not need to reacquire
            - If you have a mutex and make calls to other protected methods, **you don't need to reacquire the lock**
        - Automatically released upon final return
    - This is **not a semaphore** → there is no queue
    - Static synchronized methods lock class mutex
    - Advantages: finger lock granularity, reduced deadlock risk
    - Costs: developer needs to identify the serialized methods
    - **One lock/object**

Evaluating Java Synchronized Methods

- Correctness: correct if developer **chooses the right methods**
- Fairness: priority thread scheduling, potential for starvation
    - Not a queue, doesn't use semaphores
- Progress: multithreaded deadlocks are possible
- Performance: fine-grained (per object) locking
    - Selecting which methods to synchronize
        - Required for good performance
        - If you synchronize the **entire method,** you may need to separate methods into smaller methods

Reading

Arpaci-Dusseau Ch. 32 - Common Concurrency Problems

- Looking at concurrency bugs

32.1 What Types of Bugs Exist?

- Types of concurrency bugs that manifest in concurrent programs
- Study by Lu et. all → focused on four applications
    - mySQL
    - Apache
    - Mozilla
    - Open Office
- Different classes of bugs (non-deadlock, deadlock)

32.2 Non-Deadlock Bugs

- Majority of concurrency bugs
- Two major types: **atomicity violation** and **order violation**

→ Atomicity-Violation Bugs

```
THREAD 1: if (thd->proc_info) {
                ...
                fputs(thd->proc_info, …);
                }
THREAD 2: thd->proc_info = NULL;
```

- If thread 1 is interrupted and thread 2 runs, fputs may be trying to access a NULL pointer
- **Atomicity violation:** the desired serializability among multiple memory accesses is violated (code region is intended to be atomic but true atomicity is not enforced during execution)
- Code above has an **atomicity assumption**
- One solution: add locks

→ Order-Violation Bugs

```
THREAD 1: void init() {
              ...
              m_thread=PR_creatThread(mMain, …);
              }
THREAD 2: void mMain() {
              mSatate = mThread->state;
              }
```

- Thread 2 assumes that m_thread has already been initialized
- If thread 2 runs immediately but m_thread is not yet set
  - NULL pointer dereferencing
- **Order violation:** the desired ordered between two (groups of) memory accesses if slipped (A should execute before B, but the order is not enforced during execution)
- Solution: enforce ordering, using condition variables or semaphores

→ Non-Deadlock Bugs: Summary

- Easy fix
- Large fraction of non-deadlock bugs are either atomicity or order violation bugs

<u>32.3 Deadlock Bugs</u>

- Deadlock is a common problem
  - Occurs when thread holds a lock but is waiting for another one but the other thread holding that lock is waiting for the first one

```
EX:   thread 1:                    thread 2:
      pthread_mutex_lock(L1);      pthread_mutex_lock(L2);
      pthread_mutex_lock(L2);      pthread_mutex_lock(L1);
```

- Deadlock MAY occur – depending on the ordering

→ Why do Deadlocks Occur?

- Large code bases cause **dependencies**
- Due to nature of **encapsulation** - details are hidden from us
  - This doesn't work well with locking

→ Conditions for Deadlock

- Four conditions exist for deadlock to occur
  1. **Mutual exclusion**: threads claim exclusive control of the resources that they require

2. **Hold-and-Wait:** threads hold resources allocated to them while waiting for additional resources
3. **No preemption:** resources cannot be forcibly removed from threads holding them
4. **Circular wait:** circular chain of threads exist such that each thread holds one or more resources that are being requested by the next thread in the chain

- If any are not met → deadlock cannot occur

→ Prevention

- Circular Wait
  - Most practical method: write locking code that **will not induce a circular wait**
  - Provide a **total ordering** on lock acquisition
    - Strict ordering on when locks are acquired
  - Larger systems have more locks → use a **partial ordering**
    - Different groups of lock acquisition lock orderings
- Hold-and-Wait
  - Hold-and-wait requirement can be avoided by **acquiring all locks at once**
  - Problematic:
    - Encapsulation works against us
    - Requires us to know **exactly which locks must be held** and to acquire them ahead of time
    - Likely to decrease concurrency
- No preemption
  - Multiple lock acquisitions → when waiting for one, we hold another
  - Another flexible interface: **pthread_mutex_trylock()**
    - Grabs a lock if it is available, returns success
    - Returns an error code if the lock is being held
  - Use to build a deadlock-free, ordering-robust lock acquisition protocol
  - New problem: **livelock**
    - Two threads could repeatedly attempt and fail this sequence
    - Both systems running code sequence over and over but **no progress is being made**
  - Skirts around the hard parts of the trylock approach
    - Problems again due to encapsulation
    - Lock can be buried in some routine → make the jump back more complicated
  - Works well in limited circumstances
- Mutual Exclusion
  - Avoid mutual exclusion entirely
  - Design **lock-free** (and related **wait free**) data structures
    - Use powerful hardware instructions to build data structures that don't require explicit locking

        - Using compare_and_swap, we can build something that repeatedly tries to update the value to the new amount
            - No lock is required → no possibility of deadlock
            - BUT, livelock is still possible
        - EX: void insert (int v) {
                node_t* n = malloc(sizeof(node_t));
                n->value = v;
                do {
                    n->next = head;
                } while (compareAndSwap(&head, n->next, n) == 0);
            }
            - Continuously tries to swap the new node into the head position
            - Fails if some other thread swapped in a new head
                - The thread returns with the new head
- Deadlock Avoidance in Scheduling
    - Instead of preventing, we can try **deadlock avoidance**
    - Requires **global knowledge of the locks** that can be acquired
    - Schedule thread to guarantee no deadlocks
    - EX: T1 T2 T3 T4
      L1 Y Y N N
      L2 Y Y Y N
        - T1 and T2 can cause deadlocks since they both use L1 and L2
        - We know not to schedule them at the same time
    - This lengthens the length of the jobs
    - Fear of deadlocks **stops concurrency** and incurs a performance cost
    - Limited environment where a system has this kind of knowledge
    - Limit concurrence
    - Tactic is not widely used
- Detect and Recover
    - Allow deadlocks to **occasionally occur** → then take action once it has been detected
    - If deadlocks are rare, **this solution is pragmatic**
    - Database systems
        - Deadlock detector runs and checks for cycles (deadlocks)
            - If detected, it can restart or repair

Kampe - Deadlock Avoidance

Introduction

- Common situation where
    - Mutual exclusion is fundamental
    - Hold and block are inevitable
    - Preemption is unacceptable

        - The resource dependency networks are imponderable
    - Some cases, deadlock avoidance is easy and effective
    - There are situations where:
        - Process will free up resources when it completes
        - But process needs more resources to complete
    - **Deadlock avoidance:** making case-by-case decisions to avoid deadlock

Reservations

- Decline resource grants that **may lead to bad resource-depleted state** → will help prevent deadlock
- Failures of random allocation request mid operation are difficult to handle
- Common: ask processes to reserve resources **before** they need them
- sbrk(2) system call
    - Doesn't allocate more memory to the process
    - Requests OS to change the size of the data segment
    - Memory assignment occurs **after process gets new pages**
    - We can determine whether request will overtax the memory and return an error from sbrk
    - If we wait and there is a page referenced → we may have to kill the process
- Approach sbrk() takes is applicable in files, processes, and sockets
- There is a request → (we can return an error) before actual resource exhaustion → avoid resource exhaustion deadlock

Over-Booking

- Clients are unlikely to simultaneously all request the maximum resource reservation
- Relatively safe to grant **more reservations** than resources
- Reward for over-booking - get some more work done with the same resources
- Danger: demand we cannot handle

Dealing with Rejection

- What should a process do when a resource allocation request **fails?**
    - Simple program - will error message and exit
    - Stubborn program - retry
    - Robust - return error for request that can't be processed but continue serving new requests
    - Civic - attempt to reduce resource use

Kampe - Health Monitoring and Recovery

Introduction

- How do we determine if a system is deadlocked?
    - Identify:
        - All blocked processes
        - The resource on which the processes are blocked

        - Owner of the blocking resources (which may not be possible)
    - Determine:
        - Whether the implied dependency graph contains any loops
- How do we determine a system is wedged → to perform deadlock analysis
- Processes can simply be waiting for a message
- What do we do in the deadlock case?
- Formal deadlock detection in real systems is:
    - Difficult to perform
    - Inadequate to diagnose most hangs
    - Doesn't enable us to fix the problem
- **Health monitoring and managed recovery:** techniques to detect, diagnose, and repair a **wider** range of problems

Health Monitoring

- Detect whether a system is **making progress**
- Many ways to do so:
    - **Internal monitoring agent:** message traffic and transcript log
    - Clients submit **failure reports**
    - Server sends **heartbeat messages**
    - **External health monitoring service**
- Techniques have different strengths and weaknesses
- Many systems use a **combination** of these methods

Managed Recovery

- What do we do about failure?
- Services should be designed for **restart, recovery, and fail-over**:
    - Any process can be killed and restarted at any time
    - Multiple levels of restart
        - **Warm-start**: restore last saved state and resume
        - **Cold-start**: ignore the previous state
        - **Reset and reboot**: reboot the entire system, cold-start all applications
    - Progressively escalating scope of restarts
        - Restart a single process
        - Restart a group
        - Restart a single node
        - Restart a group of nodes, or the entire system

False Reports

- Declaring that a process has failed is an **expensive operation** that we do not want to do unless we are sure
- Trade-off:
    - If we don't confirm, **unnecessary service disruptions and fail-overs**
    - If we **mis-diagnose** the cause, make the problem worse
    - If we **wait too long**, prolong the outage
- "**Mark-out threshold**": requires good timing

Other Managed Restarts

- **Non-disruptive rolling upgrades:** take down nodes one at a time and upgrade and reintegrate
- Two tricks:
  - New software upwards compatible
  - Roll-back doesn't work → need a fall back option
- Prophylactic reboots: automatically restart every system at regular intervals

JAVA - Synchronized Methods

- Two basic synchronization idioms:
  - Synchronized methods
  - Synchronized statements
- To make methods synchronized, and "**synchronized**" keyword
- Two effects:
  - Not possible for two invocations to interleave
  - When method exits, establishes a happens before relationship with subsequent invocations
    - Guarantee that changes to state are visible to all objects

JAVA - Intrinsic Locks and Synchronization

- Build around an intrinsic/monitor lock
  - Enforces exclusive access
  - Establishes happens-before relationships
- All objects have **intrinsic locks** that threads acquire

Locks in Synchronized Methods

- Thread acquires when it evokes then releases when it returns

Synchronized Statements

- Specify object providing the intrinsic lock
- Provide concurrency

Reentrant Synchronization

- Threads can acquire the same lock more than once

Monitor (Synchronization)

- **Monitor:** synchronization construct that allows threads to have both mutual exclusion and the ability to wait for a certain condition
  - Can signal other threads
  - Contain a **mutex (lock)** object and **condition variables**
- **Condition variables:** container of threads waiting for a certain condition
- **Monitor:** thread-safe class, object, or module that uses wrapped mutual exclusion

Measuring System Performance

Metrics

- Determine **what you care about** and **what you want to measure**
- Choosing good metrics means they need to be **measurable**
- Need to be able to **capture results**

Complexity and the Role of Statistics in Measurement

- Different values of metrics each time you measure
- Need to analyze useful statistics using tools
    - **Mean:** capture entire set in one number
    - **Median:** middle point/where all data is centered
    - **Mode:** useful when one value appears more
- Indices of dispersion: range, standard deviation, confidence intervals

Comparing Alternatives

- Factors can alter performance
- Treat alternatives fairly

Source of Performance Problems

- Overloaded resource, poor scaling
- Problem affects how you look for it

Workloads

- Important aspect of performance experiment
- Think about different aspects of the workload
- Different types of workloads:
    1. Traces
    2. Live workloads
    3. Standard benchmarks
    4. Simulated workloads

Common Mistakes in Performance Measurement

1. Measuring latency without considering utilization
2. Not reporting variability of of measurements
3. Ignoring special cases
4. Ignoring costs of measurement program
5. Losing data
6. Valuing numbers over wisdom

Kampe: Load and Stress Testing

1. Introduction

- Regular suite of test cases
    - Establish conditions
    - Perform operations
    - Assert that assertions are satisfied
- The test cases should

    a. Adequately exercise program capabilities
    b. Capture and verify program behavior
    c. Be small enough to be practically implementable
- Load and stress testing are different
    - # of test cases is small but each is important
    - Ran in pseudo-random order for unspecified periods of time
    - No expectation that results are repeatable
    - No definitive pass indication
    - No complete set of assertions to determine correctness

2. Load Testing

- **Purpose:** measure the ability of a system to provide service at a specified load level
- **Harness:** issues requests to component at a specified rate and collects performance data
- **Load generators:** tools to generate the test traffic

→ 2.1 Load Generation

- **Load generator**: system that generates test traffic at a specified rate
- Characterized by:
    - Request rate
    - Request mix
    - Sequence fidelity

→ 2.2 Performance Measurement

- Ways to use load generator for assessment
    1. Measure response time
    2. Max throughput
    3. Use rate as a background for measuring other systems
    4. Use test load for performance bottleneck studies

→ 2.3 Accelerated Aging

- Can be used to simulate accelerated aging
- Simulate accumulation of problems

3. Stress Testing

- Simulate scenarios that are worse than what is expected to happen
- Random stress testing
    - Increase likelihood of encountering unlikely combinations of operations
    - Large numbers of conflicting requests
    - Wide range of generated errors
    - Swings in loads
- Provide confidence on robustness

Performances & Metrics

Lecture 11: Performance Measurement and Analysis

Outline

- Introduction to performance measurement
- Issues in performance measurement
- Performance measurement example

Performance Measurement

- Performance is almost always a **key issue in software**
- Especially in system software like the OS since **everything runs on top of the OS**
- Everyone wants the best performance possible
    - But this is not easy
    - Sometimes involves **trade off between other desirable quantities**
- How do we know what performance we've achieved?

Performance Analysis Goals

- **Quantify** the system performance (we want numbers)
    - For competitive positioning
    - To assess efficacy of previous work
    - To **identify future opportunities** for improvement
- **Understand** the system performance
    - What **factors are limiting** our current performance
    - What **choices** make us subject to these limitations
- Predict system performance

An Overarching Goal

- Applies to all performance analysis
- **We seek wisdom, not numbers**
- Point is not to produce a spreadsheet or a graph
- Point is to **understand critical performance issues**
- We want to know things like
    - What the system is limited by
    - How to improve the system

Why Are You Measuring Performance?

- To **understand system's behavior**
- To **compare to other systems**
    - See which ways some systems are better than others
- To **investigate alternatives**
    - In configuration or management of system
- To determine **how your sistem will (or won't) scale up**
- To **find the cause of performance problems**

Why Is It Hard?

- Components operate in a complex system
  - Many steps/components in processes
  - Ongoing competition for resources
  - Difficult to make clear/simple assertions
  - Systems may be too large to replicate in lab
  - Or have other **non-reproducible properties**, non-deterministic issues
- Lack of clear/rigorous requirements
  - Performance depends highly on specifics
    - what/how we measure
  - If we ask the wrong question → get the wrong answer
  - We need to know **which questions to ask**

Performance Analysis

- Can you characterize latency and throughput?
  - Of the system?
  - Of each **major component?**
- We need to look at latency and throughput for different components
  - **Which component used a large chunk of time**
- Can you account for all the **end-to-end time**?
  - Processing, transmission, queueing delays
- Can you explain how these things **vary with load?**
- Are there any significant **unexplained results?**
- Can you **predict the performance** of a system
  - As a function of its **configuration/parameters**
    - Can be used to test whether we can or what we can do to improve the system
  - Does it do what it is supposed to do?

Design for Performance Measurement

- Design the system for performance measurement in the future
- Successful systems will need to eventually have their performance measurements
- Becoming a successful system will generally require that **you improve its performance**
  - Which implies measuring in
  - **Build in a capability**
- Best to assume that your system will need to be measured
  - Adding in a capability **later is difficult**
- Put some forethought to make it easy

How to Design for Performance

- Establish **performance requirements** early
  - When you're initially designing the system
- Anticipate **bottlenecks**
  - Frequent operations
  - Limiting resources

        - Traffic concentration points
        - Points where you're doing a lot in the system
        - Ask these questions when you are at the **design stage**
    - Design to **minimize problems**
        - Eliminate, reduce use, add resources
    - Include performance measurement in design

Issues in Performance Measurement

- Performance measurement terminology
- Types of performance problems

Some Important Measurement Terminology

- Metrics
    - Indices of tendency and dispersion
- Factors and levels
- Workloads

Metrics

- A **metric** is a measurable quantity
    - Characteristic of a system represented by a number
    - **Measurable:** we can observe it in situations of interest
    - **Quantifiable:** time/rate, size/capacity, effectiveness/reliability
- A metric's value should describe **an important phenomenon in a system**
    - Needs to be **relevant** to the questions we are addressing
- Much of performance evaluation is about **properly evaluating metrics**

Common Type of System Metrics

- Duration/response time – how long
- Processing rate – how much work can we get done
- Resource consumption – do we effectively use our resources/do we need more
- Reliability – how often do messages get dropped, etc.

Non-Metrics

- Still good questions
- "Is it a good system"
- "Will it to the job I need done?"
- "Can I trust it?"
- And similar questions
- They are **not metrics**
    - No way to measure these things

Choosing Your Metrics

- Choosing irrelevant metrics → incurs **unnecessary costs**
- Pick metrics based on:
    - **Completeness**: will my metrics cover everything I need to know?
    - **(Non-)redundancy**: does each metric provide information not provided by others?
    - **Variability**: will this metric show any meaningful variation
        - Usually we are interested in measuring things that will vary

        - **Feasibility**: can I accurately measure this metric
            - May not be feasible to measure what we actually want to measure
            - EX: components that have been developed by someone else

Variability in Metrics

- Performance of a system is often **complex**
- In complex systems → there may be some elements of **non-deterministic behavior**
- Perhaps **not fully explainable**
- One result is variability in many metric readings
    - May get different results when you measure multiple times
        - How do you know if this number is characteristic of your system
    - Not a flaw in the metric but **something you must deal with**
        - Design system measurement approach to take this into account
- Good performance measurement **takes this into account**

Where Does Variation Come From?

- **Inconsistent test conditions**
    - Varying platforms, operations, injection rates
    - Background activity on test platform
    - Start-up, accumulation, cache effects
    - Different situations that can **affect system performance**
- **Flawed measurement choices/techniques**
    - Measurement **artefact**, sampling errors
        - **Artefact:** the technique you use to get the measurement affects the measurement itself
    - Measuring indirect/aggregate effects
- **Non-deterministic factor**
    - Queueing of processes, network, and disk IO
    - Where on disk files are located
    - Things that you don't have a lot of control over

Tendency and Dispersion

- Given variability in readings, how do we understand what they tell us?
- **Tendency**: what is **common or characteristic** about these readings?
    - Unifying element
- **Dispersion**: how much to the various measurements of the metric **vary?**
    - Where are the differences?
    - How do we characterize these differences
- Good performance experiments capture and report both

Indices of Tendency

- We want a **small set of numbers** that characterize the situation
- Examples:
    - Mean: average
    - Median: value of middle sample
    - Mode: most commonly occurring value
- Each of these tells us something different

- The one we want **depends on our goals**

Indices of Dispersion

- Compact description of **how much variation we observed** in our measurements
- Set of numbers reduced to a small set describing **how dispersed the data is**
    - Among the values of metrics supposedly under identical conditions
- Examples:
    - Range: high and low values
    - Standard deviation: statistical measure of common deviations from a mean
        - Characteristic set of things close to the mean
    - Coefficient of variance: ratio of standard deviation to mean
        - How far off things are from the mean
- Use index that is important for our goal

Capturing Variation

- Generally requires **repetition** of the same experiment
- Ideally **sufficient repetitions** to capture all likely outcomes
    - You don't know how many that is
- Design performance measurements with this in mind
- More experiments → more probability that you've learned about the system

Meaningful Measurements

- How do we get meaningful measurements
- Measure under **controlled conditions**
    - On a specific platform
    - Under controlled and calibrated load
    - **Removing as many extraneous external influences** as possible
- Measure the **right things**
    - Direct measurement of the key characteristics
- Ensure **quality of results**
    - Ensure we set up a **fair experiment**
    - Competing measurements we can **cross-compare**
    - **measure/correct** for artifacts
    - Quantify **repeatability/variability** of results

Factors and Levels

- Sometimes we only want to measure one thing
- More often, we are interested in **several alternatives**
    - We want to look at results for different options
    - EX: doubling memory, having things come in faster, using a different file system
- **Factors**: controlled variations for comparative purposes
    - Establishing the differences between different alternatives, things we are changing

Factors in Experiments

- **Choose factors related** to your experiment goals
    - What we are interesting in varying

- What differences to we want to observe

Levels

- Factors vary (by definition)
- **Levels:** describe which values you test for each factor
- Can be **numerical**
- Or **categorical**

Choosing Factors and Levels

- Experiment should look at **all vital factors**
- Each factor should be examined at **important levels**
  - Needs to be varied
  - Multiple levels are required
- But performance measurements also take **work**
- The **effort** involved in the experiment is related to (number of factors) * (number of levels)
- If you are not careful, this can cause your effort **to explode**
  - Especially if you repeat runs to capture variation
  - Optimization problems
- You need to choose **enough in order to get results** but not kill performance

Measurement Workloads

- Measurement programs require the use of a **workload**
- Some kind of work applied to the system you are testing
  - The work you put in affects the result directly
  - Preferably **similar to the work you care about** e.g., when you deploy the system
- Several different forms/ways to create workload to test with
  - Simulated workloads
  - Replaced trace
  - Live workload
  - Standard benchmarks

Simulated Workloads

- **Artificial load generation**: program puts out work that will be applied to the system
  - On-demand generation of a specified load
- Strengths
  - Controllable operation rates, parameters, mixes
  - Scalable to produce arbitrarily large loads
    - Usually care about scaling so this is good
  - Can collect excellent performance data
- Weaknesses
  - Random traffic is not a real usage scenario
    - Unrealistic, systems usually don’t get random traffic
  - Simulation may not create all realistic scenarios
  - Wrong parameter choices yield unrealistic loads

Replayed Workloads

- Captured operations from **real system**
    - Get a real workload trace
    - Analyze it by rerunning the trace
- Strengths:
    - Represent **real usage scenarios**
        - Represent real workloads
    - Can be **analyzed and replayed over and over**
- Weaknesses
    - Hard to obtain
        - Not a lot of good traces readily available
    - Not scalable necessarily
        - Can't scale the trace
            - Multiple instances are not equivalent to more users
    - Represent a limited set of possible behaviors
    - Limited ability to exercise little-used features
    - Kept around forever, and **become stale**
        - Expensive to obtain so data is kept for a long time
        - May not reflect changes
        - Solve this by periodically testing under a **live load**

Testing Under Live Loads

- Instrumented systems that are **actually serving clients**
- Instruments + external monitors
- Strengths:
    - **Real combinations** of real scenarios
    - Measured against **realistic background loads**
    - Enables collection of data on real usage
- Weaknesses
    - Must not get in the way of real work
    - Requires **good performance and reliability**
        - Performance measurements should not slow down or crash the system
    - Potentially **limited testing opportunities**
        - Cannot go back and change things
    - Load cannot be repeated or scaled on demand
        - Live meaning it won't happen again

Standard Benchmarks

- Carefully **crafted/reviewed simulators**
    - Similar to the first option but you don't have to build your own
    - Possibly derived from **real workloads**
- Strengths
    - Heavily reviewed by developers and customers
    - Believed to be representative of real usage
    - Standardized and widely available

        - Well-maintained
        - Allows comparison of competing products
    - Weaknesses
        - Inertia
            - Once a benchmark has been set, it isn't really changed
            - It is hard to make new benchmarks
        - Often used where they are **not applicable**
            - Not using the benchmark correctly

Types of Performance Problems

- Non-scalable solutions
    - Cost per operation becomes huge at scale
    - Worse-than-linear overheads and algorithms
    - Queueing delays associated with high utilization
- **Bottlenecks**
    - One component **limiting system throughput**
- **Accumulated costs**
    - Layers of calls, copies of data, message exchanges
    - Redundant or unnecessary work
    - Layering to get abstraction may be too much

Dealing with Performance Problems

- A lot like **finding and fixing a bug**
    - Formulate a hypothesis
    - Gather data to verify the hypothesis
    - Be sure you understand the underlying problem
    - Review proposed solutions
        - For effectiveness
        - And potential side effects
    - Make simple changes, one at a time
    - Re-measure to confirm effectiveness of each
- **Only harder**
    - You don't have incorrect behavior, it's just slow
    - May have to run entire performance measurement suite to see the changes

Common Measurement Mistakes

- Measuring time without **considering utilization**
    - Need to consider the workload
- Capturing **averages rather than distributions**
    - Outliers are usually interesting
- **Ignoring** startup, accumulation, cache effects
    - Need to measure steady-state performance
- Ignoring **instrumentation artefacts**
    - May greatly distort data
    - Getting the performance data incurs cost
    - Did running it get in the way of the system itself?

Handling Caching and Start-up Effects

- Cached results may **accelerate some runs**
  - This may not be what we want to see
  - Solutions:
    - Insert random requests
    - Overwhelm the cache
    - Disable or bypass cache entirely
- Startup costs may **distort total cost** of computation
  - Solutions:
    - Do all start-up operations prior to starting test
    - Long test runs amortize start-up effects
    - Measure and subtract start-up costs
- System performance may **degrade with age**
  - Solution:
    - Reestablish base condition for each test

Measurement Artifacts

- To measure performance, you write code
  - That typically runs on the same system
  - May lead to artifacts
- Cost of **instrumentation code**
  - Additional calls, instructions, cache misses
  - Additional memory consumption and paging
  - Take extra time
  - May affect the performance of the system
- Costs of **logging results**
  - Logging price may be much greater than implementation
  - May **dwarf** the costs of instrumentation
  - Increased disk load/latency may slow down **everything**
  - Writing to file is costly
- **Minimize frequency and costs of measuring**
  - Don’t measure things all the time
  - Use counters and accumulators instead of records
    - Instead of writing to disk, keep a counter in memory
    - Output it after the event has finished
  - In-memory circular buffer can reduce writing to files
    - Accumulate data in a large buffer
    - One large I/O cost is much less than many small I/Os
  - Probabilistic methods that don’t execute all the time
    - Don’t need to always record if you know the behavior is predictable
    - Maybe measure every other time, etc.

Measurement Tools

- Execution profiling

- Event logs
- End-to-end testing

Execution Profiling

- Usually **built into the compiler**
- Automated measurement tools
    - Compiler options for routine call counting
        - One per routine, incremented on entry
        - Simple, like turning on a flag
    - Statistical execution sampling
        - Timer interrupts execution at regular intervals
        - Increment a counter in **table based on PC value**
            - entry/subroutine → increment the entry
        - May have configurable time/space granularity
    - Tools to extract data and prepare reports
- Very useful in identifying **bottlenecks**

Time Stamped Event Logs

- Application instrumentation technique
- Event logs also relevant in **computer security**
    - Usually are **files**, sometimes necessary though **expensive**
- Create a **log buffer** and routine
    - Call the log routine for all interesting events
    - Routine stores the **time and event** in the buffer
        - Requires a cheap, very high resolution timer
    - Write into the log when an event occurs
- Extract the buffer, archive the data, mine the data
    - Time required for operations
    - Frequency of operations
    - Combinations of operations
    - Also useful for post-mortem analysis

End-to-End Testing

- How long does it take for the system to **complete a task**
- Measuring the performance from the client side
- Need instructions to identify **the start and end** of what we are interested in
- **Client-side** throughput/latency measurements
- Strengths
    - Easy test to run, easy data to analyze
        - High level
    - Results reflect **client experienced performance**
- Weaknesses
    - No information about **why** it took so long
    - No information about what resources were consumed
    - Only as relevant as your test clients are realistic
    - Doesn’t know anything about internal aspects of the code/system

Disks & I/O

Lecture 12: Devices, Device Drivers, and I/O

Outline

- Devices and device drivers
- I/O performance issues
- Device driver abstractions

World of Peripheral Devices

- Computers typically have **lots of devices attached to them**
- Each device needs to have **some code** associated with it
    - To perform whatever operations it needs to do
    - To integrate it with the rest of the system
- In modern commodity OSes, the code that handles these devices **dwarfs** the rest
    - Most OS code is **device drivers**

Peripheral Device Code and the OS

- Why does the OS take care of peripheral devices?
- Can they not be handled in user-level code?
- Sometimes they can but
- Some are **critical for system correctness**
    - The OS relies on these devices for correct behavior
    - This code needs to run **without bugs**
    - Needs to be done properly
    - EX: clock, disk, etc.
- Some of them **must be shared among multiple processes**
    - Which is complex
    - OS already knows about all processes
    - EX: the screen
- Some of them are **security sensitive**
    - EX: we don't want other processes to see data so we segregate it, the OS handles this

Where the Device Driver Fits In

- At one end, you have an **application**
- At the other end, you have a **very specific piece of hardware**
    - You need code that **understands** what hardware you have, **what signals** it uses, and **how to interact with it**
- In between is the OS
- When the application sends a packet, the OS needs to invoke **the proper device driver**
    - That **corresponds to that device**
    - Every particular piece of hardware built **must have a driver of its own**

        - Different drivers for different OSes
    - Which feeds detailed instructions to the hardware

Device Drivers

- Generally, the code is **pretty specific to the device**
- Basically code that **drives the device**, makes the device do things
    - Makes the device perform the operations it's designed for
- Typically, each system device is represented **by its own piece of code**, the **device driver**

Typical Properties of Device Drivers

- **Highly specific** to the particular device
    - System only needs drivers for the device it hosts
    - Only use for the devices you actually need
- **Highly modular**
- Usually interacts with the rest of the system in **limited, well defined ways**
    - Not involved with anything else
    - Only deals with that device
- Their correctness **is critical**
    - Device behavior correctness and overall correctness **depends on the device driver**
- Generally written by programmers who **understand the device well**
    - But have limited knowledge of the OS usually

Abstractions and Device Drivers

- We want to use simplifying abstractions for **higher level programs** to deal with
- OS defines **idealized device classes** to build **class abstractions**
    - Interface on how to interact with the device
    - Programs interact **with the abstraction**
- Classes define **expected interfaces/behavior**
    - All drivers in class support standard methods
    - OS provides this
- Device drivers **implement standard behavior**
    - Implements what the OS tells it to
    - Make diverse devices fit into a common mold
    - Protect applications from device eccentricities
- Abstractions **regularize and simplify** the chaos of the world of devices
    - Don't want an application to work with a specific device
    - We want applications to be able to **work with entire classes**

What Can Driver Abstractions Help With?

- **Encapsulate knowledge** of how to use the device
    - Map **standard operations** into operations for the device
    - Map **device states** into standard object behavior
    - **Hide irrelevant behavior** from users
    - Correctly **coordinate device** and **application behavior**
- Encapsulate **knowledge of optimization**

        - Efficiently perform standard operations on a device
        - We don't want the OS/applications to worry about these things
    - Encapsulate **fault handling**
        - Understand how to handle recoverable faults
        - Prevent device faults from becoming OS faults
        - Hardware faults should be handled in the driver

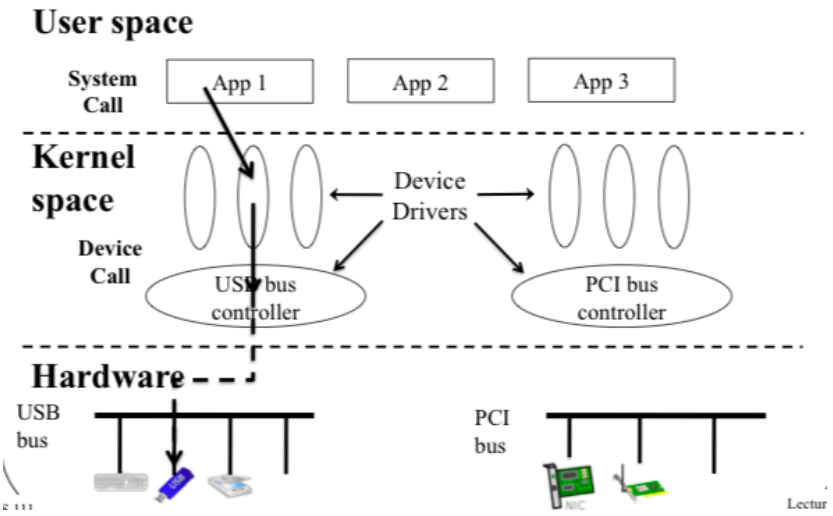
How do Device Drivers Fit into a Modern OS?

- A lot of them
- Each **pretty independent**
- May need to **add new ones later**
- Use a **pluggable model**
    - **Pluggable model:** when a device and a driver are found, plug in the device driver
    - Trusted code is plugged in
    - Need the OS to do this
- OS provides capabilities to **plug in particular drivers in well-defined ways**
    - **Detect a new device, find a driver, bring/plug it in**
        - The driver can be found from online, from a CD, or from the network, etc.
- Plug in the ones a given machine needs
- Easy to change or augment later

Layering Device Drivers

- Interactions with the **bus** down at the bottom are **pretty standard**
    - Each device is build to plug in in a particular way
    - How you address devices on the bus, coordination of signaling and data transfer, etc.
    - Does not depend much on device itself
- Interactions with the application at the top are **pretty standard also**
    - Using some file-oriented approach typically
- In between are **very device specific things**

A Pictorial View



- PCI bus controller and USB bus controller are **bus device drivers** since **buses are also devices**
- Devices usually connect onto a bus

Device Drivers vs. Core OS Code

- Device driver code **can be** in the Os but
- What belongs in **core OS vs. a device driver**
- **Common functionality** is in the OS
  - Caching
  - File systems code not tied to a specific device
  - Network protocols
- **Specialized functionality** belongs in the drivers
  - **Differ** in different pieces of hardware
  - **Only pertain** to the **particular piece** of hardware
  - Things that the driver does that you **don't want to expose** to the OS
- Putting things in hardware is **fast**
  - Building things into hardware is common
  - TODAY: hardware has more functionality

Devices and Interrupts

- Devices are **independent,** they work **at their own speed**
  - Sometimes they get **external signals**
- Primarily **interrupt-driven**
  - They are not schedulable processes
  - Interrupted when **interaction between CPU and driver** is required
- They work at **different speed than the CPU**
  - Slower usually
  - Work **asynchronously and in parallel** with the CPU
- They can **do their own work** while the CPU does something else
- They use **interrupts** to get the CPU's attention
  - Not processes so they cannot be scheduled

        - Use **pure interrupts**

Devices and Buses

- Devices are **not connected directly to the CPU**
- Both the CPU and devices are **connected to a bus**
- Sometimes to the same bus, sometimes to a different bus (from the CPU)
- Devices communicate with the CPU **across the bus**
    - Buses are used to **carry interrupts** and **transfer data**
- Bus used to **send/receive interrupts** and to **transfer data and commands**
    - Devices signal controller when they are done/ready
    - When the device finishes, controller puts the interrupt on the bus
    - Bus transfers the interrupt to the CPU
    - Leading to movement of data perhaps

CPUs and Interrupts

- Interrupts **look like traps**
    - Traps **come from the CPU**
    - Interrupts are **caused externally from the CPU**
        - Not generated by the CPU but by the device
    - Signal comes in and the CPU stops and deals with it
- Unlike traps, interrupts can be **enabled/disabled** by special CPU instructions
    - CPU may not want to be interrupted
    - Devices can be told when they may generate interrupts
    - Interrupt may be **held pending** until software is ready for it
        - You can **queue up the interrupts** until the CPU is ready to deal with it
- Need to be able to deal with situations where an interrupt may occur while dealing with another interrupt

Device Performance

- Importance of good device utilization
- How to achieve good utilization

Good Device Utilization

- Key system devices **limit system performance**
    - EX: file system I/O, swapping, network communication
- We need the **best possible performance** from these devices
- If the device is idle, **throughput drops**
    - If throughput is critical, we **don't want it to be idle**
    - May result in lower system throughput
    - Longer service queues, slower response times
    - We want the CPU doing computations **in parallel with disk drives**
- **Delays** can **disrupt real-time data flows**
    - Resulting in unacceptable performance
    - And possible **loss of irreplaceable data** or the data may come too late
- Important to **keep devices busy**
- How do we prevent idleness?

        - Have things queued up,ready for the device once the current job is complete

How to Do Better

- **Exploit parallelism**
- Devices **operate independently** of the CPU
    - We are able to parallelize because of this
- But devices need access to RAM sometimes
    - If both CPU and device need RAM, **RAM may become a bottleneck**

What's Really Happening on the CPU?

- Modern CPUs try to **avoid going to RAM**
    - Work with **registers** instead
    - **Caching** on the CPU chip itself
        - Hardware caches
    - Avoid going to slow RAM because we want to work **on the CPU speed**
- If things go well, CPU doesn't use the memory bus much
    - If not, it will be slow anyway
- So to parallelize, **let a device use the bus instead of the CPU**

Direct Memory Access (DMA)

- Devices **directly access the memory**
- Allows any **two devices** attached to the memory bus **to move data directly**
    - Without passing through CPU first
    - Allows **parallelism in I/O** related to memory and CPU performance
- Bypassing the CPU registers
- Bus can only be used for **one thing at a time**
- So if it is doing DMA, **not servicing CPU requests**
    - Bus controller takes care of who is using the RAM
- CPU usually doesn't need it anyway
- With DMA, bus moves from the device to memory at **bus/device/memory speed**
    - No instructions needed to move the data

Keeping Key Devices Busy

- Allow multiple requests **to be pending** at a time
    - Queue them
    - Requesters **block to await** eventual completion
    - Block things → need to signal and unblock them when they are ready
- Use **DMA to perform actual data transfers**
    - Data transferred at the device speed
    - Minimal overhead imposed on the CPU
- When the currently active request completes
    - Device controller generates a **completion interrupt**
    - OS accepts interrupt and calls the appropriate handler
        - May be a part of the device driver or the OS
    - Interrupt handler posts completion to requester
    - **Interrupt handler selects and initiates next transfer from the queue**

- This keeps the device **fairly busy** all the time and prevents slowdown

Bigger Transfers are Better

- On almost all devices
- Gets better throughput
- Disks have **high seek/rotation overheads**
    - Larger transfers **amortize down the cost/byte**
    - Less seeking, you delay only once
- All transfers have a **per-operation overhead**
    - Instructions to set up operation
    - Device time to start new operation
    - Time and cycles to service completion interrupt
    - Overhead is based on **instructions performed**
        - Usually is the same **regardless of size of transfer**
        - Smaller transfers have a larger overhead based on the size of the data
- Larger transfers have a **lower overhead/byte**

I/O and Buffering

- Most I/O requests cause data to come into the memory or to be copied to a device
- Data requires a **place in memory**
    - Commonly called a **buffer**
    - Special OS area that can't be messed with
- Use buffering
    - We need it regardless
    - Need to **move memory from device or vice versa**
    - Data needs to be placed in memory somewhere → a **buffer**
- Data in buffer is **ready to be sent to a device**
- An existing **empty buffer** is ready to **receive data from a device**
- OS needs to make sure that **buffers are available** when devices are ready to use them
    - Devote a portion of the RAM for I/O buffers
    - Used only for data going into/out of memory → devices
    - OS **protects the buffer**
- Size of buffer depends on transfer size of the device

OS Buffering Issues

- High overhead for small writes → want to **consolidate writes** by putting words into the buffer
    - When the buffer is full, we can write out to the disk
- fewer/larger transfers are more efficient
    - But may not be convenient
    - Natural record sizes tend to be relatively small
- Operating systems can **consolidate I/O requests**
    - Maintain a cache of **recently used disk blocks**

        - Accumulate small writes, flush out as blocks fill
        - **Read whole blocks** and deliver data as requested
            - OS takes more than 1 word than you ask for in a buffer
    - Enables **read-ahead**
        - OS reads/caches blocks **not yet requested**
        - Assumes that you'll **ask for more from the same data block**
        - Managed **directly out of RAM**
    - Caching done in buffer pool
        - Assumes that you'll read from it again

Deep Request Queues

- If you have a device that is **critical for system performance** try and always keep it busy
- Having **many I/O operations queued is good**
    - High device utilization
    - Reduces mean seek distance/rotational delay
        - Can minimize head movement as well by changing the order to **optimize**
    - Can combine adjacent requests
    - Can sometimes **avoid performing a write**
        - Can optimize if OS sees that two writes are to the same spot
        - No need to do both
    - Ways to achieve **deep queues**
        - Many processes making requests
            - Synchronization problems
        - Individual processes can make parallel requests
        - Read-ahead for expected data requests
        - Write-back cache flushing

Double-Buffered Output

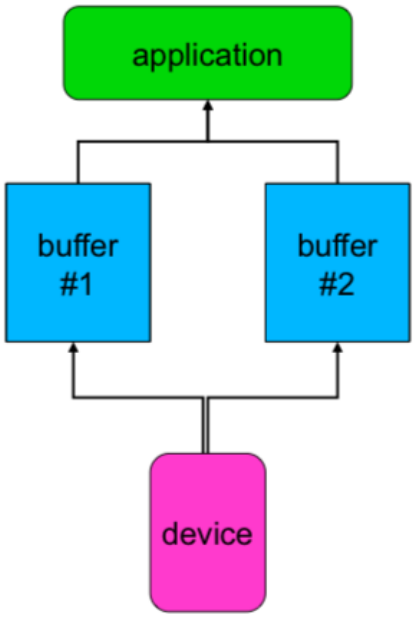
- Application writing to a device
- Create a buffer in the application itself
- Then ask for a write
- It will cause a block which may not be good
- Allocate **2 buffers instead of 1**
- When 1 buffer is full, begin sending to the device and filling the other buffer
- **Alternate** between filling out buffers and writing each to disk

Performing Double-buffered Output

- Have **multiple buffers queued up,** ready to write
    - Each write completion interrupt will start the next write to the other buffer
- Application and device I/O **proceed in parallel**
    - There are synchronization issues but you get good performance
    - Application will queue successive writes

        - Don't bother waiting for previous operation
    - Device picks up the next buffer as soon as it is ready
- If we are **CPU-bound** (more CPU than output/limited by the CPU)
    - Application will **speed up** because we don't wait for I/O
- If we are **I/O-bound** (more output than CPU/limited by I/O)
    - **Device is kept busy** which improves the throughput
        - Gives us a somewhat deep queue
    - Eventually may have to block the process
    - We use close-to-throughput of the entire device
- Can be done in the OS itself
    - But this may not be a good idea
    - Works well for writes but reads are complicated

Double-Buffered Input



- Application polling data off of the device
- Device fills buffer 1 and the application uses it
- Device then fills buffer 2 and the application uses it later
- **Alternate** between reading and filling buffers 1 and 2

Performing Double Buffered Input

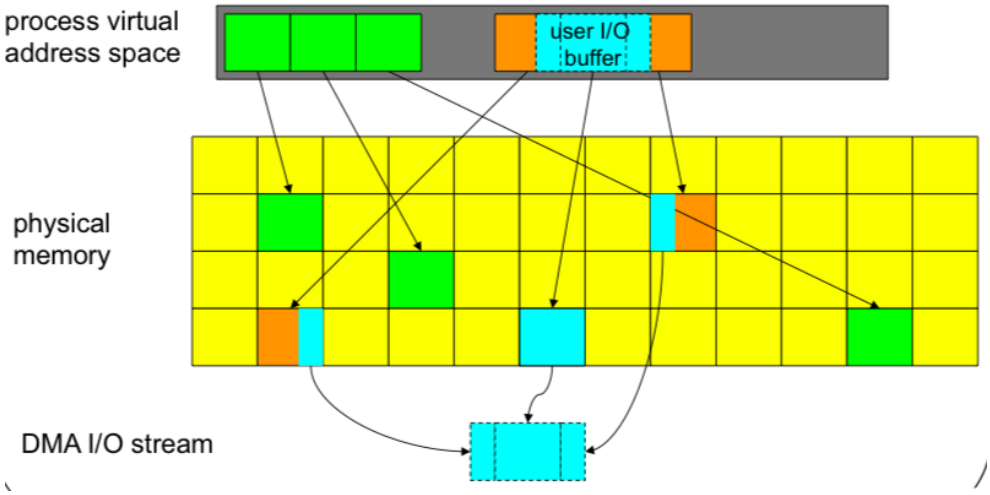
- Have to ask for multiple reads
    - Reads typically are blocking
    - Presumes that **next instruction depends on the last read**
    - Need to tell the process that we **don't need the data we read**
- Have **multiple reads queued up**, ready to go
    - Read completion interrupt starts read into the next buffer
- Filled buffers **wait until application asks** for them

    - Application doesn't have to wait for data to be read
- But **when can we do chain-scheduled reads?**
    - Apps will generally block until reads are completed
    - We can perhaps **queue reads from multiple processes**
    - Or we can do **predictive read-ahead**

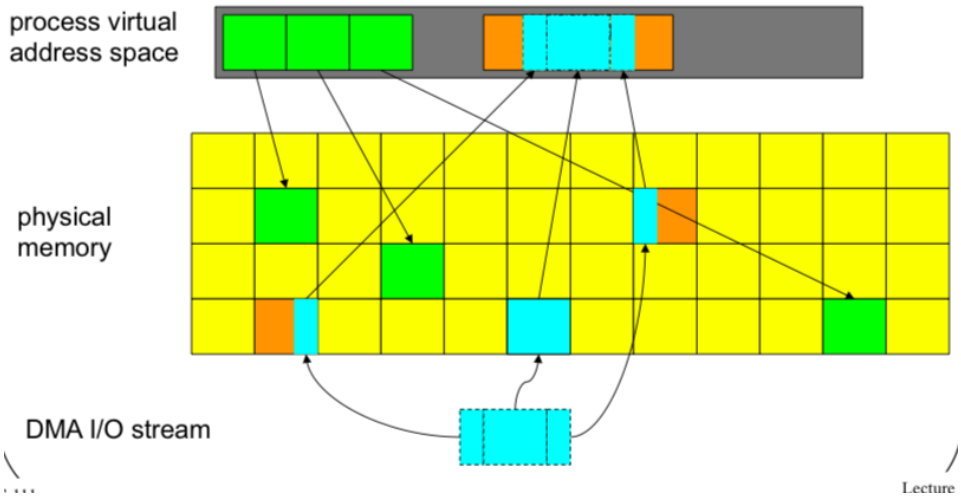
Scatter/Gather I/O

- Buffer is **scattered across multiple pages**
- But DMA reads **must be contiguous**
- This is inconsistent
- Device controllers support **DMA transfers**
    - But these transfers must be **contiguous in physical memory**
- User buffers are **paged in virtual memory**
    - The buffers themselves may be spread all over virtual memory
    - **Scatter:** read from device to multiple pages
        - Write out to different page frames
    - **Gather:** writing from multiple pages to device
        - Gather multiple page frames
- Three basic approaches
    1. **Copy all user data** into a contiguous physical buffer (one page frame)
    2. Split logical requests into **chain-scheduled page requests**
        - Overhead
        - Sometimes, may not need to copy with sophisticated hardware
    3. I/O MMU may **automatically handle** scatter/gather

"Gather" Writes From Paged Memory



"Scatter" Reads Into Paged Memory



Memory Mapped I/O

- DMA may not be the best way to do I/O
  - Designed for **large contiguous transfers**
  - Some devices may have many small sparse transfers
- Instead, treat registers/memory in device **as part of the regular memory space**
  - MMU needs to be aware of this
  - But this is **not actually RAM**
  - It is accessed by **reading/writing to those locations**
  - No need to call any open() or close() functions
  - read/write actual RAM locations, treated as addresses
- **Low overhead** per update and **no interrupts** to service
- **Relatively easy** to program
- Memory Mapped I/O presumes that **you know what you are doing**
  - Need to **understand the device** in order to do this
- There is **no verification**, the MMU tells you **what you can do**
- OS sets up an **address space** and the application **uses these addresses to access the data**
- OS still does **all the regular transfers** but they are done under the hood and it is not interrupted

Trade-Off: Memory Mapping cs. DMA

- DMA performs **large transfers efficiently**
  - Better use of both devices and CPU
    - Device doesn't have to wait for CPU to transfer
  - Considerable **per transfer overhead**
    - Expensive for small transfers
    - Setting up operation, processing completion interrupt
- Memory-mapped I/O has **no per-operation overhead**
  - Every byte is transferred **by a CPU instruction**
    - No waiting because the device accepts the data **at memory speed**
  - Belongs to **a single process/application** at a time

        - This helps **avoid a context switch**
- DMA is better for **occasional large transfers**
- Memory-mapped is better for **frequent small transfers**
- Memory-mapped devices are **more difficult to share**
    - Synchronization issues pop up in memory-mapped I/O
    - There is no file system locking, etc.

Generalizing Abstractions for Device Drivers

- Every device type is unique
    - Don't want to interact with these unique aspects
- Implies that each **requires its own unique device driver**
- But there are still many **commonalities**
    - We want to interact with general models and deal with details later
- Particularly among **classes of devices**
- We can simplify the OS by leveraging these commonalities and **defining simplifying abstractions**

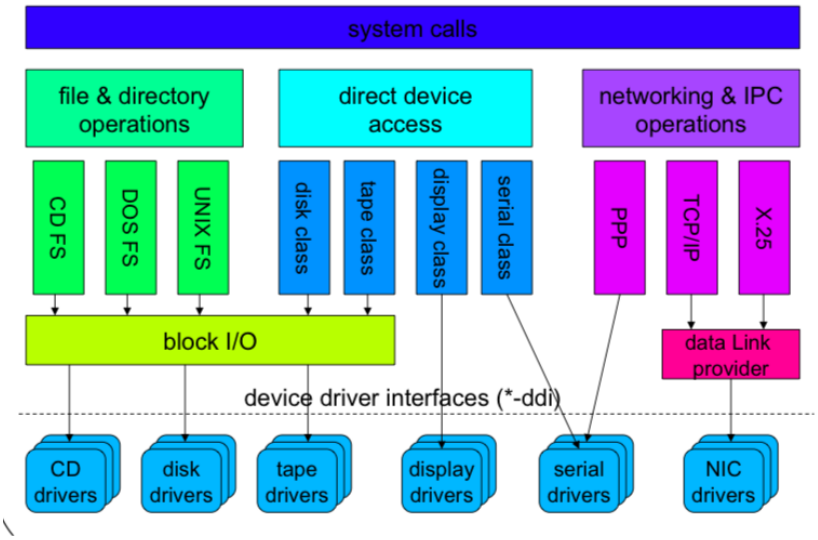
Providing the Abstractions

- The OS defines **idealized device classes**
- Classes define **expected interfaces/behavior** provided by the device driver
    - All device drivers in class support standard methods
- Device drivers **implement standard behavior**
    - Make diverse devices fit into a common mold
    - Protect applications from device eccentricities
- Interfaces are key to providing abstractions

Device Driver Interface (DDI)

- Standard (top-end) device driver **entry-points**
    - "Top-end" - from OS to driver
    - Basis for device-independent applications
    - Enables system to **exploit new devices**
    - A **critical interface contract**
- Some entry points **correspond directly to system calls**
- Some are associated with **OS frameworks**
    - Disk drivers and block I/O
    - Network drivers and protocols

Standard Driver Classes & Clients



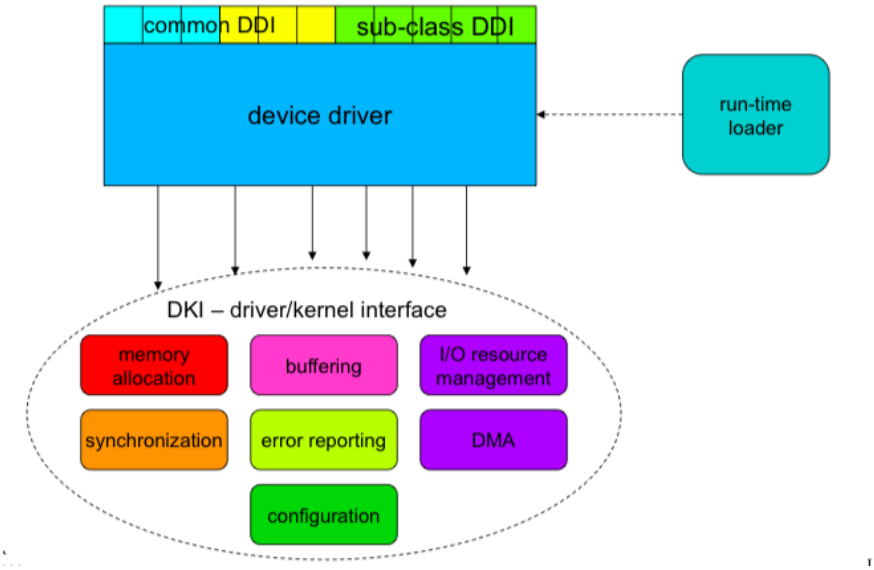
- Use file system interface to interact with devices and **pretend that devices are files**

Drivers – Simplifying Abstractions

- Encapsulate knowledge of **how to use a device**
  - Map standard operations to device-specific operations
  - Map device states into standard object behavior
  - Hide irrelevant behavior from users
  - Correctly coordinate device and application behavior
- Encapsulate knowledge of **optimization**
  - Device may optimize without our knowledge
  - Efficiently perform standard operations on a device
- Encapsulation of **fault handling**
  - Knowledge of how to handle faults
  - Prevent device faults from being OS faults

Kernel Services for Device Drivers

- Some devices may need help from the OS
- Ex: memory



Driver/Kernel Interface (DKI)

- What the kernel will do for the driver
- Specifies **bottom-end services** OS provides to drivers
  - Things drivers ask the kernel to do
  - Analogous to an ABI for device driver writers
- Must be **well-defined and stable**
- Each OS has its own DKI but **they are all similar** – things all drivers do that are not specific to the OS
  - Memory allocation, data transfer, buffering
  - I/O resource management, DMA
  - Synchronization, error reporting
  - Dynamic module support, configuration, plumbing

Criticality of Stable Interfaces

- Same reason as other interfaces
- Drivers are independent from the OS
- OS and drivers have interface dependencies
- Must be carefully managed

Linux Device Driver Abstractions

- Inherited from earlier Unix systems
- A **class-based** system
- Several superclasses
  - Block devices
  - Character devices
  - Networks maybe

Why Classes of Drivers?

- Provide **good organization** for abstraction
- Provide a **common framework** to reduce amount of code for each new device
- Ensure all device provide minimum functionality

- But a lot of driver functionality is specific to the device
    - Class abstractions don't cover everything

Character Device Superclass

- Devices that **read/write one byte at a time**
    - "Character" means byte
- May be either **stream** or **record structured**
- May be **sequential** or **random access**
- Support **direct, synchronous reads and writes**

Block Device Superclass

- Devices that **deal with a block of data at a time**
- Usually a **fixed size block**
- Common example: disk drive
- Reads or writes **a single size block** at a time
- Random access devices are accessible one block at a time
- Support queued, asynchronous reads and writes

Why a Separate Superclass for Block Devices

- Block devices span **all forms of block-addressable** random access storage
- Such devices require some **very elaborate services**
- Important system functionality is **implemented on top of block I/O**
- Designed to provide **very high performance** for **critical functions**

Network Device Superclass

- Devices that **send/receive data in packets**
- Originally treated as character devices
- But sufficiently different
- Used in context of **network protocols**

Identifying Device Drivers

- The **major device number** specifies **which device driver** type to use for it
- May have several distinct devices using the same drivers
- **Minor device number** distinguishes between the different specific drivers

Reading

Arpaci-Dusseau Ch. 33 - Event-Based Concurrency (Advanced)

- Event-based concurrency is used in GUI-based applications and in internet servers
- Addresses problems:
    1. **Managing concurrency in multithreaded applications is challenging**
    2. **In multi-threaded applications, the developer has no control over scheduling**
- We want to build a concurrent server without threads

33.1 The Basic Idea: An Event Loop

- **Event-based concurrency**: wait for something (an "event") to occur and when it does, you check what type of event it is and do the work it requires
- Event-based servers are based on an **event loop**:

```
EX:    while (1) {
              events=getEvents();
              for (e in events)
                    processEvent(e)          //an event handler
       }
```

- **Event handler:** code that processes each event, when processing an event it is the only activity taking place in the system
- Scheduling: deciding which events to happen next
- Allows explicit control over scheduling

33.2 An Important API: socket() (or poll())

- select() & poll() system calls are used to **receive events**
- Enable the program to check whether there is any incoming I/O that should be attended to
- select() - examines I/O descriptors of file descriptors passed in to see if they are ready for reading/writing or have an exceptional file descriptor
  - Lets you check whether descriptors can be read from or written to
  - Set a timeout argument
    - NULL - block indefinitely
    - 0 - return immediately
    - Can specify a timeout
- poll() - is similar
- poll() and select() give us ways to build non-blocking event loops (reads, and replies)

33.3 Using select()

- Look at how select() works
- Server enters the infinite loop
  - Initializes bits for descriptors of interest
  - Calls select to see if any file descriptors have data
    - Check for data using FD_ISSET()

33.4 Why Simpler? No Locks Needed

- Single CPU & event-based applications → **eliminate concurrency problems**
  - Only one event is handled at a time
- Cannot be interrupted

33.5 A Problem: Blocking System Calls

- Issue: event requires system call that might block
- Upon a block:
  - In a threaded environment, **different threads can run**
  - In an event-based application, **entire server is blocked** because there is **only one main thread**
    - Waste of resources

- Rule: in event-based systems, **no blocking calls**

33.6 A Solution: Asynchronous I/O

- OS's have introduced a new way to issue I/O requests to the disk using **asynchronous I/O**
- Interface allows application to issue an I/O request and **immediately return control to the caller** before the I/O is completed
  - Additional interface allows you to determine if it is complete
- Interface: **AIO Control Block** (struct aiocb)

```
struct aiocb {
      int aio_files;            //file descriptor
      off_t aio_offset;         //file offset
      volatile void* aio_buf;   //location
      size_t aio_nbytes;        //length
};
```

- To issue an asynchronous read:
  - Fill in the AIO control block with relevant information
- Issue using the **asynchronous read API**:

```
int aio_read(struct aiocb* aiocbp);
```

- Success - returns right away
- aio_error() - allows us to tell when an I/O is complete
  - Checks whether the event/request was completed
  - If yes, return EINPROGRESS
  - Use this to poll the system periodically
- This is cumbersome if you have a lot of I/O requests
- Interrupt-based approach: uses UNIX signals to notify asynchronous I/O completion

Kampe - Device Drivers

Introduction

- Device drivers represent
  - Generalizing abstractions: gather different devices together and synthesizes a few general classes and standard models, all implemented by devices
  - Simplifying abstractions: provide standard class interface and encapsulate details
- OS implemented in simple languages but device drivers have different realizations of class interfaces, derivations, and inheritances
- Number and diversity of devices is growing → there is a demand for object oriented code reuse
  - Want the system to behave similarly, regardless of underlying devices → higher level functionality is implemented in **common, higher level modules**

        - Minimize cost of developing drivers for new devices
        - Ensure that functionality and performance benefits are accrued in new device drivers and old device drivers
    - Satisfy these needs by implementing higher level functionality in **common code** and using **per-device implementations** of standard subclass drivers
    - This requires:
        - Deriving device-driver sub-classes for each major class
        - Defining subclass specific implementations to be implemented by the drivers of all devices in each class
        - Creating per-device implementations of standard subclass interfaces

Major Driver Classes

- Early Unix, there were two fundamental classes
    - **Block devices:** random-access devices
        - Addressed in fixed-size blocks
        - Drivers implement a request method
    - **Character devices**: sequential access or byte-addressable
- Division into two distinct classes is based on distinct class needs:
    - Block devices are used within OS by file systems to **access disk**
    - Character devices are used **directly by applications**
- Not mutual exclusive however
    - Both can initialize, cleanup, open, and release
    - Characters can read and write
    - Blocks can request and fsync

Driver Sub-Classes

- OS evolved and began to implement higher level services for other sub-classes of devices
    - New device driver methods defined to enable more communication between higher-level frameworks and lower-level device drivers in subclasses
- **Device Driver Interface (DDI)** - subclass specific interface
- PROS:
    - Subclass drivers are easier to implement
    - System has identical behavior over a range of devices
    - Enhancements are higher level code → should work on all devices in that subclass
- CONS:
    - Driver must correctly implement standard interface for subclasses
    - Driver's additional functionality is not defined in the standard interface for its subclass and thus is not exploited

Services for Device Drivers

- Device driver implementations require resources and services from the OS
- **Driver-Kernel Interface (DKI)** - collection of services exposed by the OS for use by device drivers
- Interface stability is important

- Need to maintain stable DKI entry points

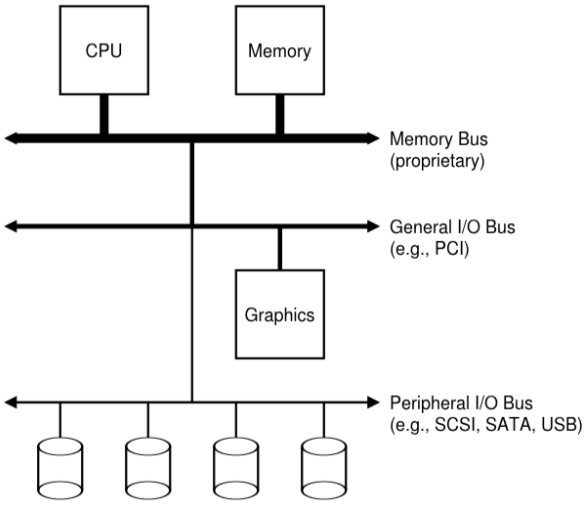
Conclusion

- Device drivers evolved from two basic superclasses to a hierarchy of derived sub-classes
- Each new subclass is a new implementation
    - Inherits pre-existing higher level frameworks

Arpaci-Dusseau Ch. 36 - I/O Devices

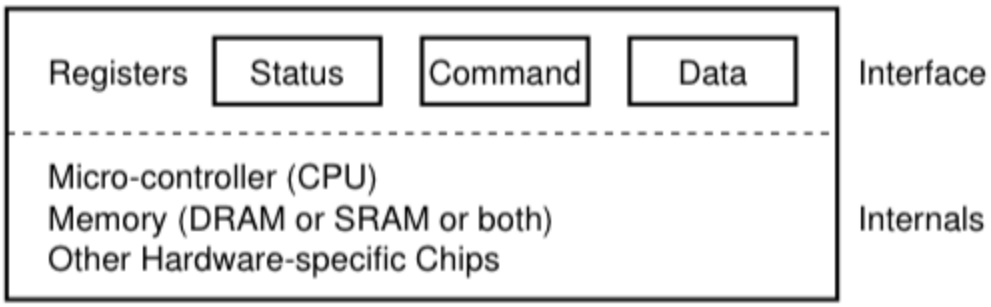
- Input-output device (I/O device)
    - OS interactions

36.1 System Architecture



- Hierarchical structure because of physics and cost
- **General I/O bus** - higher performance I/O devices
- **Peripheral I/O bus** - connect slowest devices of the system
- Faster bus → must be a shorter bus
    - Higher performance buses **cannot have many devices**
- Goes from higher to lower performance

36.2 A Canonical Device



- **(hardware) interface:** device presents to the rest of the system
    - Allows system software to control its operation
- **Internals:** internal structure
    - Implementation specific and responsible for implementing abstraction that the device provides

36.3 The Canonical Protocol

- The above device has three registers
    - **Status:** read to see the current status of the device
    - **Command:** tell the device to do a certain task
    - **Data**: pass data to the device
- The OS uses these registers to control the device
- Typical OS interaction

```
while (STATUS == BUSY)          //wait until the device is not busy
      ;
Write data to DATA reg
Write command to COMMAND reg    //starts the device and executes command
while (status == BUSY)          //wait until the device is done
      ;
```

- The protocol has four steps
    1. The OS waits until the device is ready to receive a command by checking the status register
        - Polling the device
    2. OS **sends data down to the data register**
        - **Programmed I/O (PIO)** - main CPU involved with data movement
    3. OS **writes the command to the command register**
        - Tells the device that data is present and the device should work on the command
    4. OS waits for the device to **finish by polling the status register again**
- Simple but inefficient and inconvenient due to **inefficient polling**

36.4 Lowering CPU Overhead with Interrupts

- Use interrupts instead of polling device repeatedly
    - OS can put calling process to sleep
    - And context switch to another task

- When the device is done, **raise a hardware interrupt**
    - CPU jumps into the OS at the **interrupt service routine (ISR)** or an **interrupt handler** that completes the request
- Interrupts enable **overlap of computation and I/O**
- Interrupts are not always the best solution
    - For devices that **perform very quickly**
        - Polling would be better
        - Interrupts would slow down the system
    - Perhaps use a **hybrid** that polls for a while and then sleep → called a **two-phased approach**
    - Networks – huge streams of packets, each generating an interrupt
        - OS can **livelock**, end up processing only the interrupts
- One optimization is **coalescing** multiple interrupts together

36.5 More Efficient Data Movement with DMA

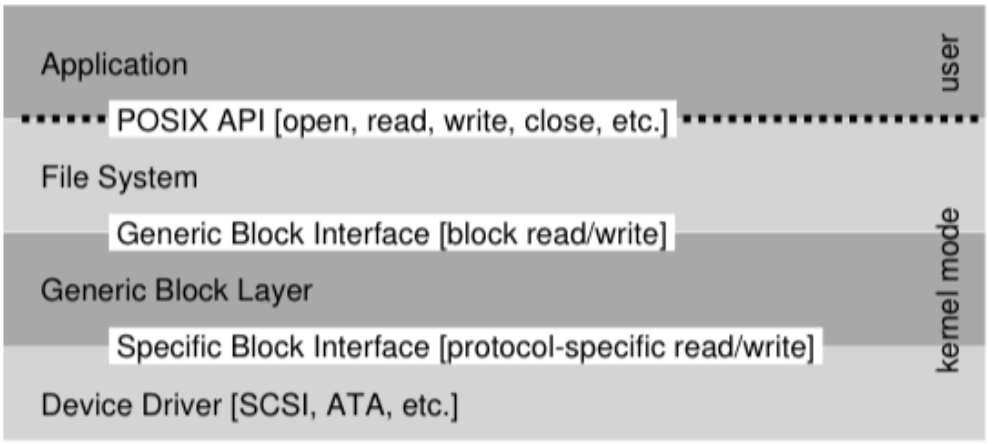
- When programmed I/O is used to transfer large chunks of data to a device
    - The CPU is overburdened with trivial tasks
- **Direct Memory Access (DMA)**: DMA engine is a device that can orchestrate transfers between devices and main memory **without CPU intervention**
- Using DMA to transfer data
    - OS provides DMA address and amount to copy and which device to send it to
    - Now, OS is done
    - When DMA is done, an interrupt is raised

36.6 Methods of Device Interaction

- How do we communicate with the device from the OS?
- Two methods primarily:
    - **Explicit I/O instructions**
        - Specify a way for OS to send data to device registers
        - **Privileged instructions**: only OS can directly communicate with drivers
    - **Memory-mapped I/O**
        - Hardware makes device registers available like memory locations
        - No new instructions are needed

36.7 Fitting Into the OS: The Device Drivers

- Fit devices (with very specific interfaces) into the OS (very general)
- We want to keep the OS **device neutral**
- Abstraction at the lowest level is a **device driver**, software in the Os that knows in detail, how the device works
    - Encapsulates specifics of device interaction



- File system: doesn't know which disk class it is using
- Hidden encapsulation and complexity
- Downsides of encapsulation:
    - Special capabilities may go **unused**
    - Becomes a **huge part of the kernel**
        - Device drivers make up over 70% of it

Arpaci-Dusseau Ch. 37 - Hard Disk Drives

- Particular I/O device is the **hard disk drive**

37.1 The Interface

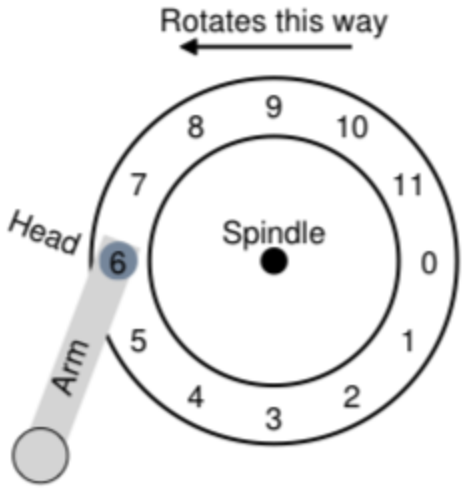
- **Drive:** consists of **sectors** (512 byte blocks) that can be read/written
    - Sectors are numbered 0 → N-1, the **address space of the drive**
    - Disk is an **array of sectors**
- Guarantee that 512 byte writes are **atomic**
    - Power loss may result in only a portion of a large write being completed, a **torn write**
- Other assumptions (not in the interface)
    - Accessing two blocks **closer together** if **faster** than accessing two blocks that are **far apart**

37.2 Basic Geometry

- Components of a modern disk drive
- **Platter**: circular hard surface on which data is stored persistently by inducing a magnetic charge
    - Disk may have one or more platters
- **Surface**: one of the two sides each platter has
- **Spindle:** where platters are bound together
    - Connected to a motor that spins the platter
    - Rate of rotation (rotations per minute, RPM)) normally in the 7200-15000 RPM range
- **Track:** one concentric circle of sectors where data is encoded

    - Single surface contains thousands of tracks
- **Disk head** and **disk arm**: mechanisms that allow us to either **sense (read)** the magnetic patterns on the disk or to **induce a charge (write)** them
    - Reading and writing are done by the **disk head**
        - One head/surface
    - Disk head is attached to the **disk arm** which moves across the surface to position the head

37.3 A Simple Disk Drive

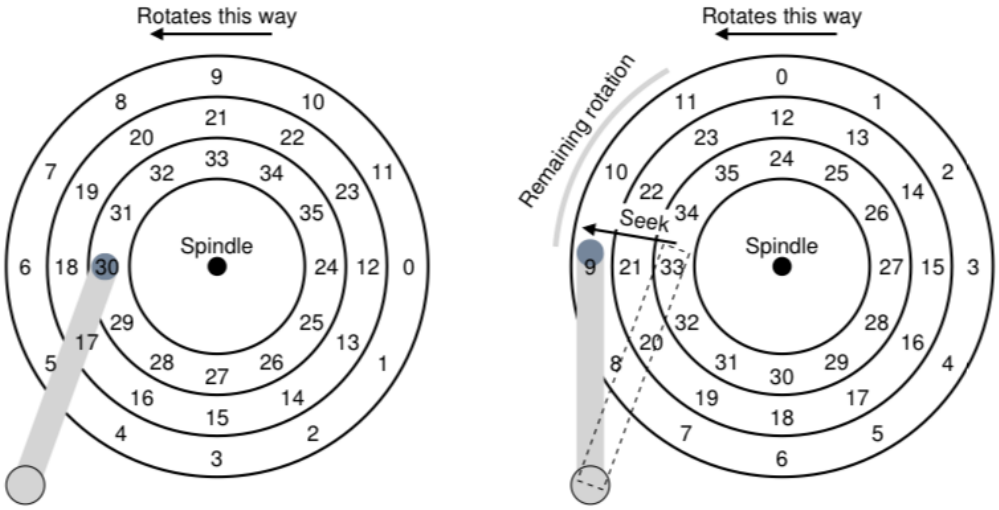


- 12 sectors, each 512 bytes in size
- Addressed 0-11
- Platter rotates around the spindle

→ Single Track Latency

- To service a specific block read, need to **wait until the sector rotates** under the head
- This wait is called the **rotational delay/rotation delay**

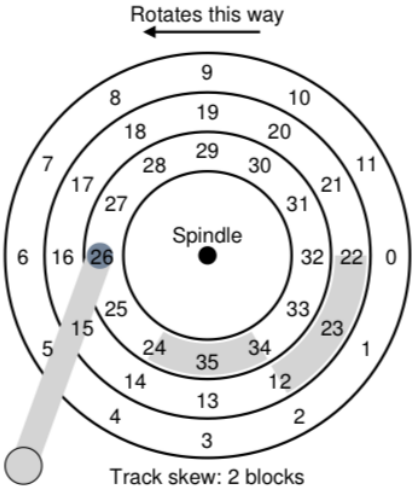
→ Multiple Tracks: Seek Time



- If we are currently at 30, to get to 11 (on another sector), drive needs to move the disk arm to the correct track, a **seek**
- **Seeking** - is costly, and has phases
    1. Acceleration: disk arm begins moving
    2. Coasting: arm moving at **full speed**
    3. Deceleration: arm slows down
    4. Settling: head positioned over the correct track
        - **Settling time** is quite significant
- When 11 passes under the head, the **transfer** can take place
    - Data is read from or written to the surface
- I/O time includes: **seek, rotational delay, transfer**

→ Some Other Details

- Drivers also have a **track skew**
    - Ensure sequential reads are correct **even when crossing track boundaries**



- Data stored is skewed to ensure that **when the head moves**, the next block **has not already passed the head**
- Outer tracks have more sectors than inner tracks - **multi-zoned disk drives**
  - Disk is organized into zones
  - **Zones:** consecutive set of tracks of a surface
  - Each zone has the same number of sectors/track
- **cache/track buffer**: small memory used to hold data read from or written to the disk
- On writes, disk has a choice:
  - Acknowledge that the write is complete when the data is in memory - **write-back caching** (immediate reporting)
    - May be faster, but is more dangerous
      - If an order is required for correctness, there are problems
  - Acknowledge after write is written to the disk - **write-through caching**

37.4 I/O Time: Doing the Math

- $T^{I/O} = T^{seek} + T^{rotation} + T^{transfer}$
- Total I/O time is sum of seek, rotation, and transfer time
- $R^{I/O}$ = (size of transfer) / ($T^{I/O}$)
- Examples show that:
  - There is a **gap in performance** between random and sequential workloads
  - **Large performance difference** between high-end performance drivers and low-end capacity drivers
- **High performance**: drives that go as fast as possible
- **Capacity**: drives are slower but pack many bits onto space

37.5 Disk Scheduling

- OS decides the order of I/Os issued to the disk
- **Disk scheduler:** given a set of I/O requests it examines them and decides which one to schedule next
- Length of "jobs" - can guess how long
- Estimate seek and rotational delay to know how long requests may take & pick the quickest one first
- Disk scheduler tries to follow to SJF policy (shortest job first)

→ SSTF: Shortest Seek Time First

- **Shortest-seek-time-first (shortest-seek-first(SSF)):** orders the queue of I/O requests by track
  - The nearest track is first
- Not a solution for all situations
- Has two problems
  1. OS doesn't know about disk geometry → only sees it as an **array of blocks**
     - Solution: implement nearest-block-first (NBF)
  2. **Starvation** is a fundamental problem

        - How do we implement SSTF-like scheduling while avoiding starvation

→ Elevator (aka SCAN or C-SCAN)

- **SCAN:** moves back and forth across the disk servicing requests **in order across tracks**
    - **Sweep:** a single pass across the disk (outer to inner/inner to outer)
    - Request comes for a block on a track that has already been served is **queued until the next sweep**
- Variants:
    - **F-SCAN:** freezes the queue during a sweep, places requests in another queue
        - Avoids starvation by delaying service
    - **C-SCAN (Circular Scan)**:
        - Algorithm sweeps from outer-to-inner only
        - Resets at the outer track
        - More fair to inner/outer track
            - SCAN favors the middle tracks
- Behaves like an elevator
- Ignores SJF principles → how do we combine them

→ SPTF: Shortest Positioning Time First

- Depends on whether seeking or rotation time is faster
- Difficult to implement in the OS
- Works when $T^{seek} = T^{rotate}$

→ Other Scheduling Issues

- Where should disk drive scheduling be performed
    - Inside disk if possible
    - Historically in the OS
    - OS picks the requests and sends them to the disk where they are refined
- I/O merging is performed by the disk scheduler
    - Merge nearby requests
- How long should the system wait before issuing I/O requests to disk?
    - **Work-conserving**: once disk has a single I/O → **immediately issue request**
    - **Non-Work-conserving**: anticipatory disk scheduling
        - **Wait for a better request**

Arpaci-Dusseau Ch. 38 - Redundant Arrays of Inexpensive Disks (RAIDs)

- **Redundant Array of Inexpensive Disks (RAID)**: technique using multiple disks together to build a faster, bigger, and more reliable disk system
- Externally: a disk, group of bocks
- Internally: complex, multiple disks, memory, and processors to manage the disk/system

- **Advantages:**
    - **Performance:** speed up I/O time
    - **Capacity:** large data sets need large disks
    - **Reliability:** spreading data out with some redundancy
        - Allows RAIDs to be tolerant of disk loss
- Advantages are provided transparently
    - Increases deployability of RAID
- Discuss: interface, fault model, evaluate

38.1 Interface and RAID Internals

- To a file system, RAID looks like a large, fast, reliable disk
    - Presents itself as a **linear array of blocks**
- Upon issuing a **logical I/O request** to RAID → RAID must calculate the disk(s) to access and then issue the **physical I/O request**
- Example: RAID has 2 copies of every block (each on a different disk)
    - **Mirrored RAID system**: RAID must perform 2 physical I/O requests for every 1 logical
- RAID: build as a hardware box with a standard connection to a host
- Internally complex
    - Microcontroller that runs firmware that directs operations of RAID
    - Volatile DRAM
    - Sometimes non-volatile memory
- Like a specialized computer that runs specialized software

38.2 Fault Model

- RAIDS are designed to **detect and recover** from **certain disk faults**
- Need to understand **fault models** to study RAID
- **Fail-stop fault model**
    - Disk has **2 states**: working or failed
    - **Working disk:** all blocks can be written to or read
    - **Failed disk:** assume that it is permanently lost
    - Assumes that disk failures are easily detected

38.3 How to Evaluate a RAID

- Different approaches to building a RAID
- RAID design is evaluated along three axes
    - **Capacity:** how much useful capacity is available to the clients of the RAID
        - N disks, B blocks
        - With no redundancy, N * B
        - With redundancy, say mirroring (2 copies of each block), N * B / 2
    - **Reliability**: how many disk faults can the design tolerate
    - **Performance**: difficult to evaluate, depends on the workload
- Three RAID designs
    - **RAID level 0** (striping)

- **RAID level 1** (mirroring)
- **RAID levels 4/5** (parity-based redundancy)

38.4 RAID Level 0: Striping

- No redundancy, it really isn't a RAID level
- Serves as a good **upper-bound** for performance and capacity
- Stripes the block across the disk

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

- Spread the blocks of the array in a RR-fashion
  - Designed to maximize parallelism
- Can also stripe based on block size

Disk 0	Disk 1	Disk 2	Disk 3	
0	2	4	6	chunk size:
1	3	5	7	2 blocks
8	10	12	14	
9	11	13	15	

→ Chunk Sizes

- Affects performance
- **Small chunk size**: files are striped across different disks
  - Increases parallelism
  - But also increases **positioning time** to access blocks for request
- **Big chunk size**
  - Reduces the intra-file parallelism
  - Relies on concurrent requests to achieve high throughput
  - But reduces positioning time

→ RAID-0 Analysis

- Perfect in terms of capacity
  - Given N disks of B blocks
  - Striping gives N*B blocks of useful capacity
- Reliability is also perfect in a BAD way
  - Disk failure leads to data loss
- Performance is good
  - All disks are used in parallel

→ Evaluating RAID Performance

- Two performance metrics

1. **Single-request latency:** understanding single I/O request → reveals parallelism in a single logical I/O operation
2. **Steady-state throughput:** total bandwidth of many concurrent requests
    - For RAIDs used in high performance environments, this is critical
    - Main focus of our analysis

- Workloads of interest
    - **Sequential:** requests come in large contiguous chunks
        - Common in environments
    - **Random:** small requests to different random locations in disk
    - **Real workloads** are often a MIX
- Different workloads give different results
    - In sequential workloads, the disk is **the most efficient**
        - Little time is spent seeking and waiting
        - More time is spent transferring
    - In random workloads, time is mostly spend **seeking and waiting**
        - Short transfer time
- Assume the system transfers sequential data at S MB/s and R MB/s
- In general S >> R

→ RAID-0 Analysis

- Latency of a single-block request similar to that of a single disk
- Steady-state throughput
    - Full bandwidth of N requests * S bandwidth
- Random throughput
    - Again N * R
- Simple, serves as an upper bound

38.5 RAID Level 1: Mirroring

- **Mirrored system:** make more than one copy of each block in the system
    - Each copy placed on a different disk
- Allows us to **tolerate disk failures**

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

- For each logical block, the system **keeps 2 copies of it**
- Different arrangements of copies:
    - **RAID-10 (RAID 1+0):** uses mirrored pairs (RAID 1) and stripes (RAID 0) on top of them
    - **RAID-01 (RAID 0+1):** two large striping arrays and then mirrors on top of them
- When reading: RAID has a choice between different copies to read

- When writing: NO CHOICE, RAID must update **both copies** of the data to ensure reliability
  - Can occur in parallel

→ RAID-1 Analysis

- Capacity-wise, RAID-1 is expensive
  - With mirroring level 2, capacity is (N*B)/2
- Performs well from reliability standpoint
  - Can tolerate failure of any one disk (or more)
  - Generally, up to 1 for sure, and up to N/2, depending on which disks fail
- Performance:
  - Single read - same as a single disk
  - Write requires **two physical writes** (in parallel)
    - Logical write waits for both physical writes to complete → worst-case seek and rotational delay
  - Steady-state throughput
    - Sequential workload: write to 2 blocks (N/2 * S)
      - ½ the peak bandwidth
    - Read performs just as poorly
    - Random reads is the best case
      - Distributes reads across all disks
      - Ful bandwidth N*R MB/s
    - Random writes: N/2 * R MB/s
      - 1 logical write → 2 physical writes

38.6 RAID Level 4: Saving Space with Parity

- Add redundancy using parity
- Attempt to use less capacity → overcome space penalty of heavily mirrored systems
- Cost: performance

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

- Disk 4 contains the **parity block**
  - For each stripe of data
  - Stores redundant information about the block
- **Computing parity:** need mathematical function used to withstand the loss of any one block from our stripe
  - XOR works

        - Returns 0 if there are an even number of 1s
        - Returns 1 if there are an odd number of 1s
    - The invariant that RAID must maintain in order for parity to be correct: **the number of 1s in any row must be an even number (not odd)**
    - To recover:
        - Say C2 is lost
            - To figure out which values in the column were lost, read all other values in the row and reconstruct the answer
        - Assume C2's first value is 1
            - Read the 0010 in the row
            - Know that there must be an even number of 1's
            - Lost bit must be a 1
    - To compute: XOR all data and parity bits together
    - Bitwise XOR across each data block and put the corresponding result **in the bit slot of the parity block**

→ RAID-4 Analysis

- Capacity-wise: RAID-4 uses one disk for storing parity info
    - Useful capacity: (N-1)*B
- Reliability: tolerates only 1 disk failure
    - More than 1 makes it impossible to reconstruct
- Performance
    - Steady-state throughput
        - Sequential read: uses all disks but parity
        - Sequential write:
            - Large chunk - **full-stripe write** optimization
                - Calculate P0 by XOR-ing across bits
                - Most efficient
        - Random reads: 1 block random reads → spread out but not on parity disk
        - Random writes: how do we update the parity?
- Methods to update parity:
    1. **Additive parity:**
        - Read in all other data blocks
        - XOR with the new block → the result is the new parity
        - Complete write by writing new data and the parity
        - Scales with the number of disks → PROBLEM
    2. **Subtractive parity:**
        - Read in old data and old parity
        - Compare old data and new data
            - If same, the parity bit remains
            - If different, flip the parity bit
            - $P^{new} = (C^{old} \text{ XOR } C^{new}) \text{ XOR } P^{old}$
        - Compute P over all bits in the block

- Performance analysis using subtractive parity
- Parity block is a **bottleneck** in cases of small writes to different stripes
    - **“Small-write” problem** for parity-based RAIDs
- Bandwidth of R/2 MB/s under small random writes
- **I/O latency:** 1 write → 2 reads → 2 writes
    - Latency: 2 times that of a single disk

38.7 RAID Level 5: Rotating Parity

- Address small-write problem
- **RAID-5** is identical to RAID-4 except it **rotates the parity block across drives**

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

- Parity block for each stripe is rotated across the disk
- Removes the parity-disk bottleneck of RAID-4

→ RAID-5 Analysis

- Analysis for effective capacity and failure tolerance is identical to that of RAID-4
- Sequential read/write - identical to RAID-4
- Latency of a single request - identical to RAID-4
- Random read performance is better, utilizing all disks
- Random write performance is noticeably better
    - Parallelism across requests
    - Able to keep all disks busy
    - Bandwidth of small writes: N/4 * R MB/s
- Identical to RAID-4 → replaced it in the market
- **Sometimes RAID-4 is used in cases of no small writes**
    - **Simpler to build**

38.8 RAID Comparison: A Summary

- Some details were omitted
    - Writing mirrored system
        - Average seek time > writing on a single disk
        - Performance on 2 disks for random < performance one a single disk
    - When updating the parity bit
        - First update → full rotation and seek
        - Second update → only a ortation
- Performance is priority so we use striping
- Want random I/O performance and reliability → mirroring is best

- Want capacity and reliability → use RAID-5
    - With a performance cost
- Want sequential I/O and capacity → use RAID-5

38.9 Other Issues

- Level 2 and 3 RAID designs, there is also a Level 6 RAID design
- Disk failure → use a **hot spare** to fill in
- More realistic fault models: latent sector errors, block corruption
- Software RAID - cheaper but has problems

Kampe - Dynamically Loadable Kernel Modules

- Choosing which device driver to add to the kernel once it is plugged in
    - How loading in a module works
- Systems provide common framework and problem-specific implementations are provided later
- This approach has some key elements:
    - Implementations provide functionality in accordance with an **interface**
    - Selection of implementation can be determined at runtime
    - Decoupling service from implementation
        - Allows the system to work with an open-ended set of algorithms and adaptors
- Most programs have implementation locked at build-time
- Systems can select and load implementations at runtime as **dynamically loadable modules** like device drivers
- Reasons for wanting device drivers to be dynamically loadable:
    - # of possible I/O devices is so large it is hard to build into the OS
    - OS must automatically identify and load
    - Devices are **hot-pluggable** meaning you don't know what drivers are needed until the device is plugged in
    - New devices are developed after the OS has shipped
    - Device drivers delivered independently from the OS

Choosing Which Module to Load

- **Probe method:** check which data and claim responsibility
    - Unreliable and dangerous
- **Self-identifying objects:** device has type, model, and serial number that are queries by walking the configuration space
- Use this information and device registry to select a driver for the device

Loading a New Module

- **Module:** self-contained or uses standard shared libraries
    - Loading a new module means allocating memory
- **Dynamically loaded modules** uses other functions
    - Has unresolved external references and requires a **run-time loader** (simple linkage editor)

- References are only from the dynamically loaded module into the program it is being loaded into (the OS)
    - Cannot be the other way around
    - Device driver may not always be there

Initialization and Registration

- Run-time loader returns a vector containing a pointer to an initialization point
- Device instance configuration and initialization → self-identifying devices have improved

Using a Dynamically Loaded Module

- OS provides means for processes to open device instances
- System maintains table of all registered device instances and associated device entry points

Unloading

- When a file descriptor into a device is closed and the drivers are no longer needed
- OS can call a **shut-down method** causing driver to:
    - Unregister itself
    - Shut downt the device it was managing
    - Return allocated memory and I/O resources

The Criticality of Stable Interfaces

- Everything depends on a stable interface
    - Entry-points and interfaces
    - Functions and interfaces

Hot-Pluggable Devices and Drivers

- **Hot-plug busses**: generate events when a device is added to or removed from the bus
- **Hot-plug manager:**
    - Subscribes to hot-plug events
    - When device is inserted, it walks the configuration space to find and load drivers
    - When the device is removed, it calls the removal of the associated driver
- Multiple power levels

Summary

- Stable and well defined interfaces are critical

File Systems

Lecture 13: File Systems

Outline

- File systems
  - Why do we need them
  - Why are they challenging
- Basic elements of file system design
- Designing file systems for disks
  - Basic issues
  - Free space, allocation, and deallocation

Introduction

- Most systems need to **store data persistently**
  - Data remains after reboot or power-down
- Typically **a core piece of functionality** for the system
  - Used all the time
- Even the OS itself needs to be stored this way
- So we must store **some data persistently**

Our Persistent Data Options

- Use **raw storage blocks** to store the data
  - Hard disk, flash driver, etc.
  - But these things make no sense to users
  - Not easy for OS developers to work with
- Use a **database** to store the data
  - Probably will provide **more structure** and maybe more overhead than we can afford
- Instead, use a **file system**
  - Organized way of **structuring persistent data**
  - Needs to make sense to users and programmers

File Systems

- Originally the computer equivalent of a physical filing cabinet
- Put related sets into **individual containers**
- Put them all into an **overall storage unit**
- **Organized** by some simple principle
- The goal is to provide:
  - Persistence
  - Ease of access
  - Good performance

The Basic File System Concept

- Organize data into **natural coherent units** usually specified by the user creating it
- Store each unit as its own self-contained entity
    - A **file**
    - Store these files in a way that allows **efficient access**
- Provide a simple, powerful **organizing principle** for the collection of files
    - Make it easy to find them
    - And to organize them

File Systems and Hardware

- File systems are usually **stored on hardware** that provide persistent memory
    - We want to use the **same abstraction**, regardless of what kind of hardware we use
- Expect that files put in one “place” will be there when we look again
- Performance considerations require us to **match implementation** to the hardware
- Ideally though, we should have the **same user-visible file system** that should work on **any reasonable hardware**

What Hardware Do We Use?

- File systems mostly designed for **disks**
- Required many **optimizations** based on disk characteristics
    - Minimize **seek overhead** and **rotational latency delay**
- The disk provided **cheap persistent storage** at the cost of **high latency**
    - File system had to hide as much latency as possible
    - Large quantity given at a low price

Data and Metadata

- File systems deal with two kinds of information
- **Data:** information that the file is actually supposed to store
- **Metadata:** information about the information the file stores
    - Ex: how many bytes, when it was created, etc.
    - Also called **attributes**
- Both data and metadata need to be stored persistently
    - And ideally on the same piece of hardware

A Further Wrinkle

- File system should be **agnostic to storage medium**
- Same program should access the file system **the same way** regardless of the hardware
- Should work the same for **disks of different types**

Desirable File System Properties

- Things we are looking for from our file system
    - **Persistence**
    - **Easy use model**
        - Expect it to be used by both programmers and human users
        - Easy to access files and to organize collections of files
    - **Flexibility**

        - No limit on number of files
        - Or on file size, type, contents
        - Limitations should be based on **hardware, not on the software**
    - **Portability** across hardware device types
        - No reliance on what type of hardware is underneath
    - Performance
    - Reliability
    - Suitable security

The Performance Issue

- Compensate for the fact that hardware is **much slower than RAM**
    - We want to operate as fast as the hardware can manage

The Reliability Issue

- Need an **error-free** file system
- Persistence implies **reliability**
- We want files to be there when we check, regardless of the situation

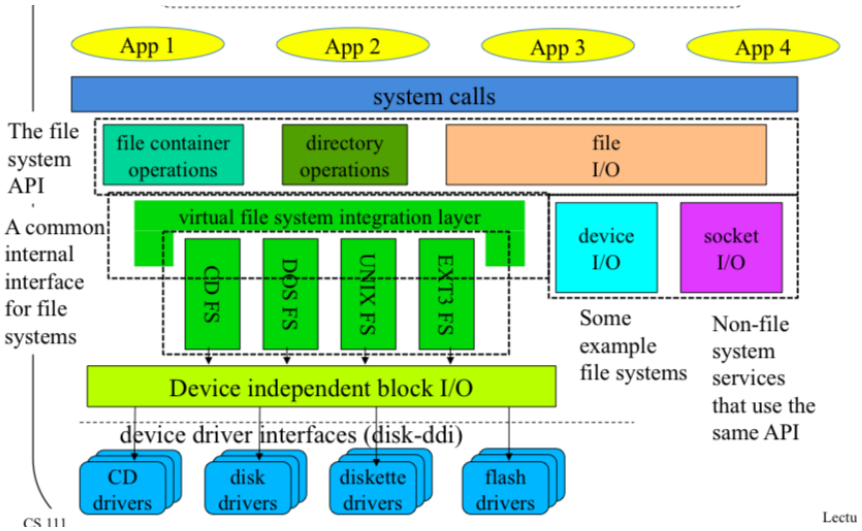
"Suitable" Security

- Files have "**owners**"
- Whoever owns the data should be able to control **who accesses it**
    - Using a **well-defined access control model** and mechanism
- Strong guarantees that the system **will enforce his desired controls**
    - Need to apply complete mediation
    - To the extent performance allows
        - To get better performance, may need to have some limitations in security

Basics of File System Design

- Where does the file system fit in the OS
- File control data structures

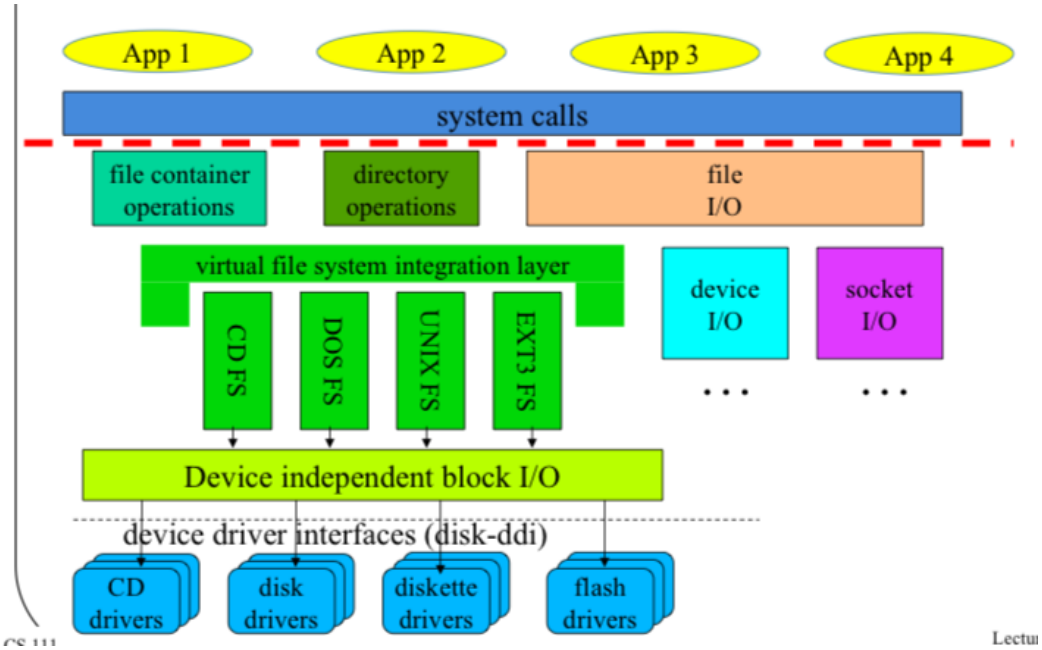
File Systems and the OS



File Systems and Layered Abstractions

- At the top, applications think that they are **accessing files**
- At the bottom, **various block devices are reading and writing blocks**
- Multiple layers of abstraction between

The File System API



- Provide a **single API** to programmers and users for all files
  - Provide access to files **regardless of what hardware** file system is using
- Regardless of how the file system underneath is actually implemented
- Required if one wants **program portability**
- Three categories of system calls here
  - **File container operations**
  - **Directory operations**
  - **File I/O operations**

File Container Operations

- Treat the file **as an object**, not caring about the individual bytes
- Standard file management system calls
  - Manipulate files **as objects**
  - Operations that **ignore** the contents of the file
  - Deal with the file **as a whole**
- Implemented with **standard file system methods**
  - get/set attributes, ownership, protection
  - create/destroy files and directories
  - create/destroy links
- Real work happens in the **file system implementation**
  - Changing metadata primarily

Directory Operations

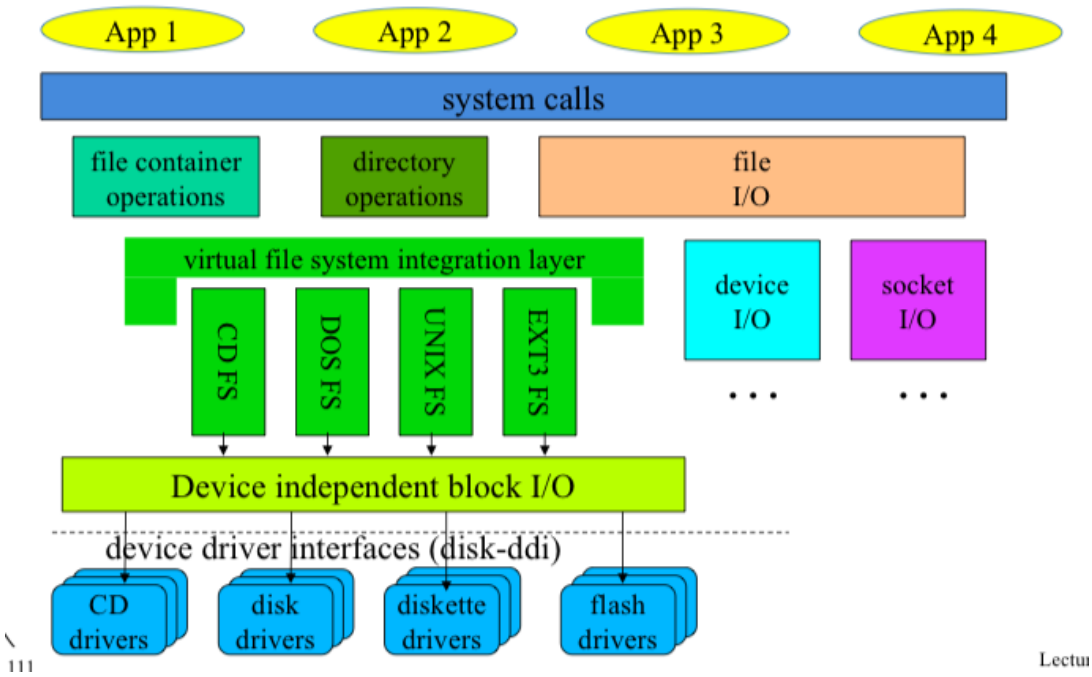
- Special files that contain **only references to other files**

        - Handled similarly to files
    - Directories provide the **organization** of a file system
        - Typically **hierarchical**
        - Sometimes have extra features
        - Higher level operations typically
    - At the core, directories **translate names** to a **lower-level file pointer**
        - Find a file by name
        - Create new name/file mapping
        - List a set of known names
    - Cannot call directory operations on regular files

File I/O Operations

- Accessing the information of the file
- **Open:** use a name to set up an open instance
- **Read** data from the file and write data to the file
    - Implement using **logical block fetches**
    - Copy the data between **user space and file buffer**
    - Request the file system to **write back block** when it is done
- **Seek**
    - Change the **logical offset** associated with an open instance
    - We don't want to just **throw away** first 1000 bytes if we want to read the 1001st byte
        - Allows us to just go in and read it
- **Map file** into address space
    - File block buffers are just **pages of physical memory**
    - Map into the **address space** and page it into and from the file system

The Virtual File System Layer



- Virtual file system integration layer
- There are multiple implementations of different file systems
- Allows you to ensure that **multiple file systems** work with other systems
- Translate opereations into virtual operations

The Virtual File System (VFS) Layer

- **Federation layer** to **generalize** file systems
    - Permits the rest of the OS to treat **all file systems the same**
    - Supports the dynamic addition of the new systems
- File systems **must meet the API of the VFS**
- Plug-in interface or file system implementations
    - All implement the same basic methods
- Implementation is **hidden from higher level clients**
    - Clients only see standard methods and properties

The File System Layer

- Different file systems
    - Different pieces of code
    - Independent from each other
- **Device independent block I/O**
    - Caching blocks of data for all file systems
    - All file systems work with **one block cache**
    - Also used by networking/IPC operations
    - Has pages of RAM of one size only
        - Each file system has to work with this size

The File Systems Layer

- Desirable to support **multiple different file systems**

        - Ex: file systems for flash drives, or hard disk drive, or DVD drive
    - All should be implemented **on top of block I/O**
        - Should be independent of underlying devices
    - All file systems **perform the same basic functions**
        - Map names to files
        - Map <file, offset> into <device, block>
        - Manage free space and allocate it to files
        - Create and destroy files
        - Get and set file attributes
        - Manipulate the file name space
    - Relies on the device independent block I/O cache

Why Multiple File Systems?

- Why not choose just one good one
- May be **multiple storage devices**
    - Ex: hard disk and flash drive
    - Might benefit from **very different file system**
- Different file systems provide **different services, despite the same interface**
    - Differing reliability guarantees
    - Differing performance
    - Read-only vs. read/write
- Different file systems are used for **different purposes**

File Systems and Block I/O Devices

- File systems typically sit on top of a **general block I/O layer** cache
- Generalizing abstraction
    - Makes **all disks look the same**
- Implements the **standard operations** on each block device
    - Asynchronous read
    - Asynchronous write
- **Maps logical block numbers** to device addresses
- **Encapsulates** all particulars of device support
    - I/O scheduling, initiation, completion, error handling
    - Size and alignment limitations

Why Device Independent Block I/O?

- A better abstraction than **generic disks**
- Allows **unified LRU buffer cache** for disk data
    - Caching for a lot of people
    - Better use if **everyone shares a single cache** then if everyone gets their own cache
    - Hold frequently used data until it is needed again
    - Hold pre-fetched read-ahead data until it is requested
- Provides buffers for **data re-blocking**
    - Adapting the file system block size to **device block size**
    - Adapting file system block size to **user request sizes**

- Handles **automatic buffer management**
    - Allocation, deallocation
    - Automatic write-back of changed buffers

Why Do We Need That Cache?

- File access exhibits **high degree of reference locality** at multiple levels
    - Users often read and write a single block in small operations, reusing that block
    - Users read and write the same files over and over
    - Users often open files from the same directory
    - OS regularly consults the same metadata blocks
- Having **common cache** eliminates many disk accesses

File Systems Control Structures

- **File: named collection of information**
- Metadata is used to keep track of how files are laid out
- Primary roles of the file system:
    - **Store and receive** data
    - **Manage the media/space** where data is stored
- When the file is deleted, we want to be able to **reuse blocks**
- "Blocks" are similar to paging/VM
    - Where we put these blocks matters in a hard disk drive

Finding Data on Disks

- Essentially a question of **how you managed the space on your disk**
- Space management on disk is **complex**
    - Millions of blocks, thousands of files
    - Files are created and destroyed continuously
    - Files can be extended after they have been written
    - **Data placement** has performance effects
    - Poor management leads to **poor performance**
- Must **track the space** assigned to each file

On-disk File Control Structures

- **On-disk descriptor** of important attributes of a file
    - Particularly where the data is located
- Tells where **all blocks** of data are in the file
- Keeps track of data about the file
- Virtually all file systems have such data structures
    - Different implementations, performance and abilities
    - Implementation can have **profound effects** on what the file system can do (well or at all)
- A **core design element** of a file system
- Paired with some kind of **in-memory representation** of the same information
    - On-disk: fine for persistence
    - Needed in RAM when working with the file for it to be fast

The Basic File Control Structure Problem

- A file typically consists of **multiple data blocks**
- The control structure **must be able to find them**
- preferably abe to find any of them **quickly**
    - Shouldn't need to read the entire file to find a block near the end
    - Should not use too many I/Os to find a data block
    - If possible, should be as few I/Os as possible
- Blocks **can be changed**
- New data **can be added** to the file
    - Or old data can be deleted
- Files can be **sparsely populated**
    - Should be able to write the 500th byte even if byte 0-499 are not yet written
    - In high performance computing, multiple machines/threads may be creating different parts of the file
        - Some parts may be created before others

The In-Memory Representation

- There is an **on-disk structure** pointing to disk blocks (and holding other information)
- When the file is opened, an **in-memory structure** is created
- Not an exact copy of the disk version **but is highly related**
    - The disk version points to disk blocks
    - The in-memory version points to **RAM pages**
        - Or indicates that the block isn't in memory
    - Also keeps track of **which blocks are dirty** and **which aren't**
- The copy in memory is updated but the copy in disk is not
- Need to **write out to disk eventually**

In-Memory Structures and Processes

- Multiple processes can have a given file open
- Should they **share one control structure** or have **one each**
- In-memory structures typically contain a **cursor pointer**
    - Indicating **how far into the file** data has been read/written
    - Multiple cursors for multiple processes
- Sounds like it should be a **per-process structure**

Per-Process Or Not?

- What if **cooperating processes** are working with the same file?
    - May want to **share a cursor**
    - Since we know they are already working together
- How can we know when **all processes are finished** with an open file
    - In order to reclaim space
    - And reuse it
- If the file is **closed** we can get rid of it in block I/O cache
- Implies a two-level solution
    1. Structure shared by all

2. Structure shared by cooperating processes

File System Structure

- How do I organize a disk into a file system?
  - Linked extents
    - DOS FAT file system
  - File index blocks
    - Unix System V file system

Basics of File System Structure

- Most file systems **live on disk**
- Disk volumes are **divided into fixed-size blocks**
- Most blocks will be used to **store user data**
- Some will be used to **store organized "meta-data"**
  - Description of the file system
  - File control block to describe individual files
  - Lists of free blocks
- All operating systems have data structures like this
  - But may have different implementations

The Boot Block

- The **0th block** of the disk is **usually reserved for the boot block**
  - Points to the OS
  - Gives you information about what to load
- Not usually under the control of a file system
  - Usually **ignores the boot block** entirely
- Not all disks are bootable
  - But the 0th block may be reserved **just in case**
  - Waste of block 0
- All file systems **start work at block 1**

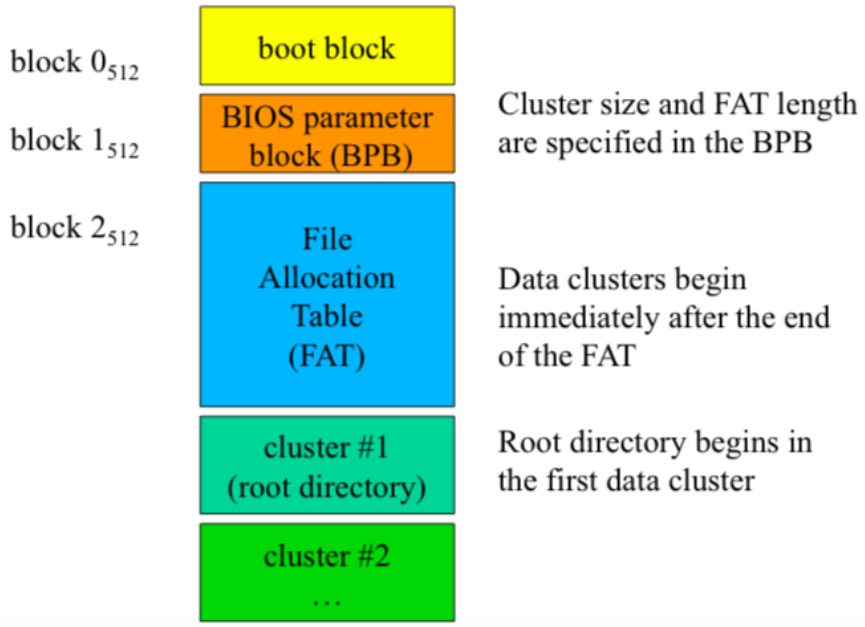
Managing Allocated Space

- A **core activity** for a file system, with various choices
- What if we give each file **the same amount of space – fixed partitions**
  - **Internal fragmentation**
- What if we allocate **just as much as the file needs – variable partitions**
  - **External fragmentation**, compaction
- Perhaps we should allocate **space in "pages"**
  - How many **chunks** can a file contain
  - Same concept as paging
  - **"Page"** is usually a multiple of how much you can write in a disk at a time
- The **file control data structure** will determine this
  - Only has room for **so many pointers**
- How do we want to **organize the space** in a file
  - How big can a file get
    - Depends on the data structure

Linked Extents

- A simple answer
- **File control block** contains exactly **one pointer**
  - To the **first chunk** of the file
  - Each chunk contains **a pointer to the next chunk**
  - Allows it to add **arbitrarily many chunks** to each file
- Pointers can be in chunks themselves
  - Takes away a little of every chunk
  - To find chunk N, you need to read the first N-1 chucks
    - Slow, would have to go back to the disk
- Pointers can be in an auxiliary **"chunk linkage" table**
  - Faster searches, especially if table is kept in memory
  - Contains only **chunk links → points to next chunk**

The DOS File System



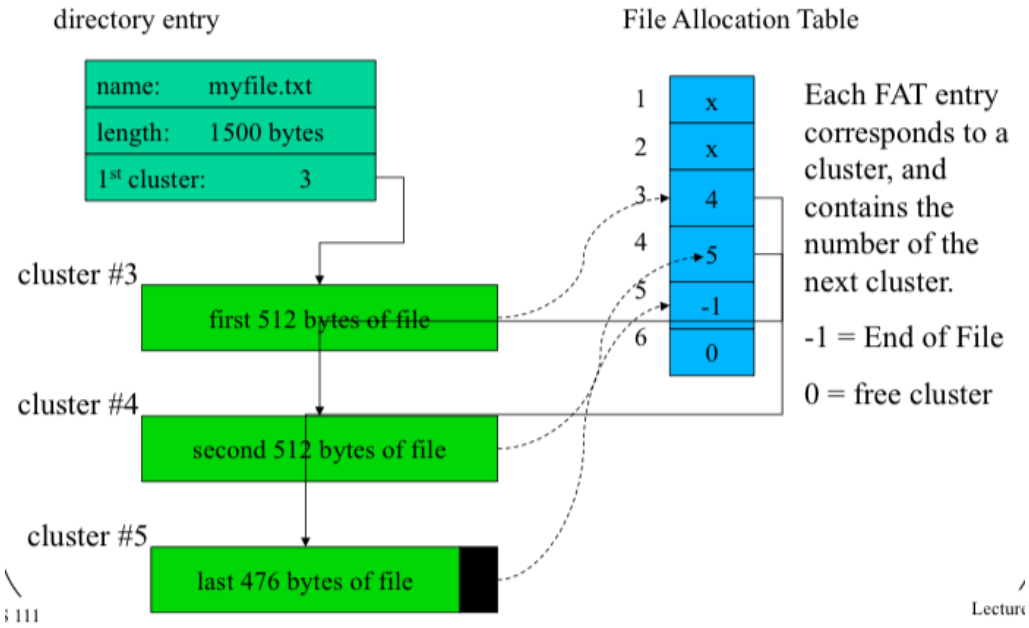
- When it came out, there were smaller disk drivers
- Parameter description of the file system
- Tells you where the first cluster is

DOS File System Overview

- DOS file systems provide space into "clusters
  - Cluster size fixed for each file system
  - Clusters are numbered 1 through N
- File control structure (**directory entry**) points to the **first cluster of a file**
- **File Allocation Table (FAT)**
  - One entry/cluster
  - Contains the **number of the next cluster** in the file
  - A 0 entry means **cluster is not allocated**

- A -1 entry means EOF
- File system is sometimes call “FAT”, after the name of this data structure

DOS FAT Clusters



- **Directory:** list of files
  - Each entry has metadata for that file
- The FAT is in memory at all times

DOS File System Characteristics

- To find a particular block of a file
  - Get the number of the **first cluster** from the directory entry
    - Don’t actually have to **read** this cluster
    - We can just go directly to the FAT table
  - Follow the **chain of pointers through the FAT**
- Entire FAT is **kept in memory**
  - No disk I/O is required to find a cluster
  - For very large files, **the search can still be long**
- **No support for “sparse” files**
  - Of a file that has a block **n, it must have all blocks <n**
- Width of FAT determines FAT determines max file system size
  - **Fundamental limitation**
  - Can have a file that uses a lot of entries but there would only be a few files

File Index Blocks

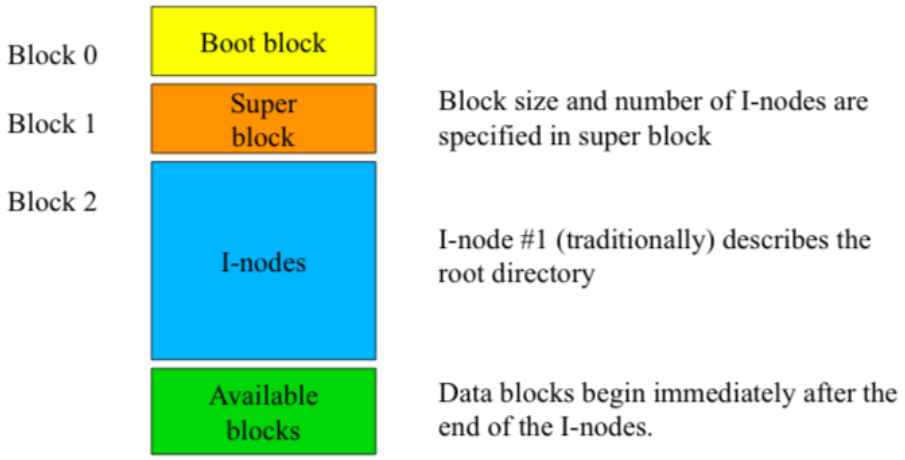
- A different way to keep track of **where a file’s data blocks are** on the disk
- A **file control block** points to all blocks in the file
  - Pointers to all the blocks
  - Very **fast access to any desired block**

    - How many pointeres can the file control block hold
- File control block could point at **extent descriptors**
    - Build files out of extents
    - Still gives us a fixed number of extents

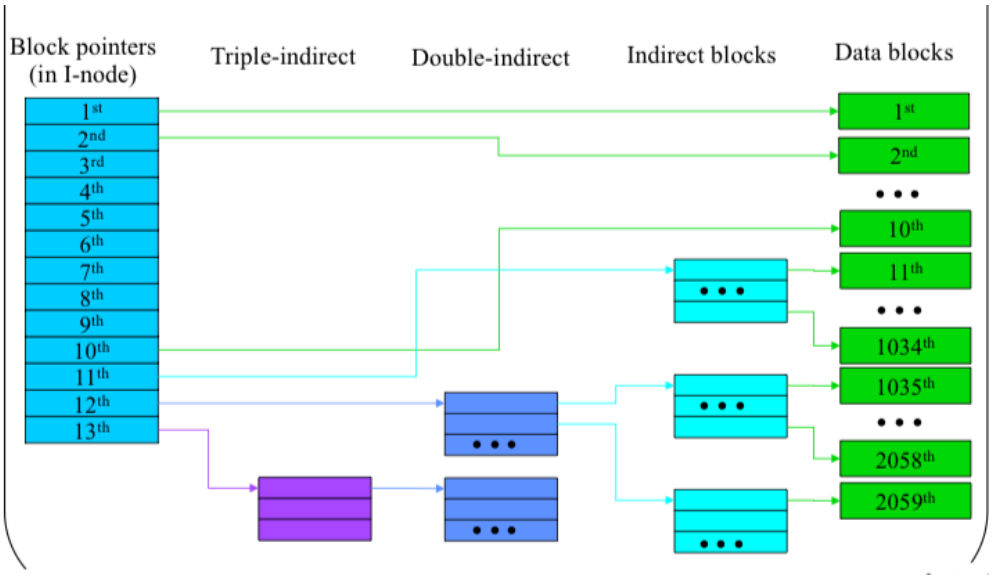
Hierarchically Structured File Index Blocks

- To solve the problem of **file size being limited** by **entries in the file index block**
    - Must be able to support large files
- The **basic file index block** points to blocks
- Some of these **contain pointers** which in turn **point to blocks**
- Can still point to many extents but still there is a limit
    - That limit **may be a very large number**
    - Potentially can adapt to a wide range of file sizes

Unix System V File System



Unix Inodes and Block Pointers



- **Indirect blocks:** block full of pointers
- **Double-indirect blocks:** points to indirect blocks
- **Triple-indirect blocks:** points to double-indirect blocks
    - Adds a level of indirection

Why Is This a Good Idea?

- The UNIX pointer structure seems ad hoc and complicated
- Why not something simpler?
    - E.g. block pointers are all triple indirect
- File sizes **are not random**
    - Majority of files are **small**
- UNIX approach allows us to access up to 40K bytes without extra I/Os
    - Remember that the **double and triple indirect blocks** must themselves **be fetched off of disk**
    - Need an I/O to get data block but no extra I/O to get information on how to FIND these data blocks
- It would be excessive to have triply indirect blocks to access small files

Unix Inode Performance Issues

- The inode is **kept in memory whenever the file is open**
- The first ten blocks **can be found with no extra I/O**
- After, we have to **read indirect blocks**
    - Real pointers are in indirect blocks
    - Sequential file processing will keep referencing it
    - **Block I/O will keep it in the buffer cache**
    - doubly/triply indirect blocks are treated in an LRU fashion
        - They aren't always in memory
        - May eventually by flushed/LRUed out
    - 1-3 extra I/O operations per thousand pages
        - Any block can be found in 3 or fewer reads
    - Index blocks **can support "sparse" files**
        - Not unlike page tables for sparse address spaces
        - Building a file piece by piece
            - Works well
            - Block pointer will be 0 whenever you haven't written
            - Will be nonzero where we've written to the file

Reading

Arpaci-Dusseau Ch. 39 - Interlude: Files and Directories

- Thus far, there have been two kep abstractions: processes and address space
- Third abstraction; persistent storage

- Persistent-storage devices (hard disk drivers, solid-state storage devices) store information permanently
    - Keeps data, even when power is loss

39.1 Files and Directories

- 2 key abstractions: files and directories
- **File:** linear array of bytes that can be read/written
    - Has a **low-level name** (usually a number), the **inode number**
    - Usually the OS doesn't know much about the file structure
        - The file system's responsibility: **persistence**
- **Directory:** abstraction that contains a list of **pairs**
    - Pair: User-readable and low-level names
    - Also has a low-level name
    - **Maps** user level name to low level name
    - Entries refer to either **files** or **other directories**
        - **Directory tree** or **directory hierarchy**
- Directory hierarchy:
    - Start at the root directory
        - Uses separators to name sub-directories

      EX: foo.txt in directory foo in root directory

      /foo/foo.txt → the **absolute pathname**
- Directory and file names **can be repeated** so long as they are in different locations/directories
- Names have **two parts**: foo.txt
    - "Foo" arbitrary
    - ".txt" type of file
    - Simply a convention → there is no real enforcement
- File system provides a way to name all files that we are interested in

Arpaci-Dusseau Ch. 40 - File System Implementation

- Simple file system implementation - **vsfs (Very Simple File System)**
    - Simplified version of UNIX file system
- File system is **pure software**, no hardware features

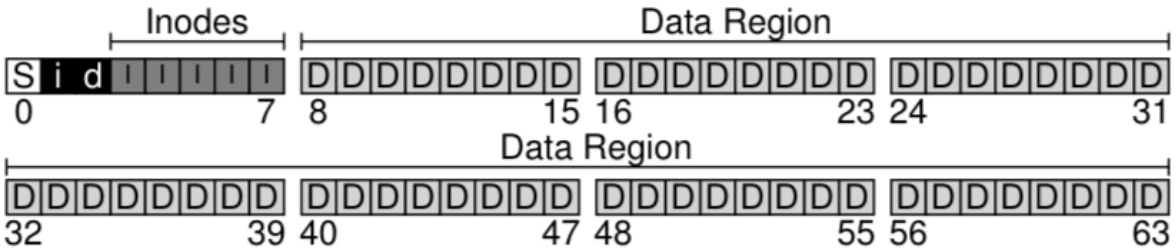
40.1 The Way to Think

- Two aspects: data structures and access methods
- **Data structures:** what structures are used to organize data
- **Access methods:** how do calls map to the data structure

40.2 Overall Organization

- Divide disk into blocks
- Simple file systems use **one block size**, here we use 4KB
- What do we store in the blocks?
    - **User data (D)** - stored in the **data region**
        - Reserve a fixed portion

- **Metadata (I)** to track data blocks containing files, the size, owner, access and modify rights and times
  - In a structure called an **inode**, stored in the **inode table**
  - Not usually very large
- **Inode and Data Bitmaps (i) (d)** way to track whether data or inode blocks are free or allocated in **allocation structures**
  - Possible methods: free list, bitmap, etc.
  - Bitmap for inode region and bitmap for data region
- **Superblock (S)**: contains information about the file system



- **Mounting** a file system means the OS **reads the superblock** first to initialize parameters
  - Then attaches the volume to the file system tree

40.3 File Organization: The Inode

- Index node AKA inode
  - Virtually all file systems have a similar structure
- Referred to by a number, **the i-number**, also the low-level name of a file
- Using the low level number, you can calculate the location on disk where the inode is located
  - EX: read inode 32
    - Calculate the offset into the inode region (32 * sizeof(inode))
    - Add to start of the inode table on disk
    - Arrive at the correct byte address/sector address
- In general:
  - Block = number * sizeof(inode_t))/block_size;
  - Sector = ((block * bock_size) + inode_start_addr)/sectorSize;
- Each inode can contains **metadata**:
  - Type
  - Size
  - The number of blocks allocated
  - Protection information
  - Time information
- Important decision in time of inode is **how it refers to where the data blocks are**
- Simple approach is to have **direct pointers inside the inode**

        - pointer refers to one disk block that belongs to the file
        - This is a **limited approach** since **large files cannot be represented**

→ The Multi-Level Index

- Need different structures in inodes to support larger files
- Idea: use a special pointer, the **indirect pointer**
    - Points to a block that contains other pointeres that **each point to user data**
- Inode may have a fixed
- number of direct pointers and a single indirect pointer
- When files grow large, an indirect block is allocated
- For even larger files, use **double indirect pointer**
    - Blocks of data containing pointers to indirect blocks
- Eventually, can even use **triple indirect blocks**
- Overall, this imbalanced tree is called a **multi-level index** approach to pointing to file blocks
- Why this system?
    - **Most files are small → we optimize for this case**
    - Use a small number of inodes
- In general
    - Files are small
    - Average file size is growing
    - File systems have a lot of files
    - File system is roughly ½ full
    - Most bytes are stored in large files
    - Directories are usually small

40.4 Directory Organization

- Directories have simple organization
    - List of entry name, inode number pairs
    - For each file/directory in a given directory, there is a **string and a number** in the data blocks
        - String may also have a length
    - . **directory** is the current directory
    - .. **directory** is the previous directory
- **Deleting a file** (calling unlink()) leaves space
    - There should be a way to mark this as being free
- **File systems** treat directories as **special files** marked as “directory” instead of “regular file”
    - Also has data blocks inode points to (and maybe indirect blocks)
    - Lives in data region

40.5 Free Space Management

- File system tracks which inodes and data blocks are free
- Vsfs uses **2 simple bitmaps**
    - Inode bitmap

        - Data bitmap
    - Files guaranteed continuity on some portion of the disk

40.6 Access Paths: Reading and Writing

- **Access path:** flow of operation during reading or writing

→ Reading a File From Disk

- Opening a file means:
    1. Finding the inode for the file
        - Traverse the path name to locate the node
        - Traversals being at the **root directory**
        - Need to find inode of the root directory
            - Usually the inode number of the root directory is known
            - “Well known” the file system knows when the system is mounted
            - It usually is number 2
        - Look through the data blocks of the root directory
    2. After finding it, traverse through the pathname until the inode is found
    3. Read the inode into memory
- Then we can read the file
    - 1st read: read in the first block using the inode
        - May update the inode
    - Successive reads: update the file table, updating the file offset for the next read
- Eventually the file is closed
- Open causes **many reads** to locate the inode
- After, each read to a block requires the file system to consult the inode in memory
- Amount of I/O generated is **directly proportional** to the length of the pathname

→ Writing to Disk

- Similar to opening/reading
- File must be opened → application issues write() to update the file → after, file is closed
- Writing may also **allocate a block**
- Writing to a new file:
    - Writes have to decide whether to allocate a block/which block to allocate and also write to disk
    - Need to update data structures
- **Each write to a file logically generates five I/Os**
    1. Read the data bitmap
        - Updated to indicate the newly allocated block
    2. Write to the bitmap
        - Reflect new state on disk
    3. Read the inode
    4. Write the inode

5. Write to the actual block
- Traffic worsens when you consider the creation of a new file
- To create the file:
  - System allocates an inode
  - System allocates space within the directory containing the new file
- **I/O traffic is high for creation**
  1. Read the inode bitmap to find a free inode
  2. Write to the inode bitmap
  3. Write to the new inode to initialize it
  4. Link the high-level name of the file to the inode number by writing to directory data
  5. Read and write to directory inode to update
- If the directory needs more space → more inode will be needed to allocate more space

40.7 Caching and Buffering
- Slow read/write
- Remedy: use system memory (DRAM) to cache
- Fixed-size cache: used by earlier system to hold popular blocks
  - LRU and other algorithms used to decide which blocks are kept in cache
  - Static partitioning can be wasteful
- Instead, use dynamic partitioning using a unified page cache
- Effects of caching
  - Reads: only first read generates I/O traffic but subsequent reads usually result in cache hits
  - Writes: cache doesn't help as much, usually still need to go to the disk in order to be persistent
    - Still has performance benefits, use **write buffering** to get performance benefits
    - Delay writes to batch updates into a smaller set of I/Os
    - Buffer writes in memory
      - Can schedule the I/Os to increase performance
      - Writes are avoided by delaying them
- Modern buffers **delay writes** to memory → may cause the data to be loss if something happens (e.g. system crashes)
  - Can force writes to disk (fsync()) using direct I/O interfaces that avoid the cache or using a raw disk interface and avoiding file system

Kampe: File Types & Attributes

1. Introduction
- Operating systems represent data sources and sinks as files
- **Files:** byte streams and 1-D arrays
  - Usually contains highly structured data

2. Ordinary Files
- Sequence of 0s and 1s

2.1 Data Types and Associated Applications
- Finding out how to interpret a file
- General approach:
    - User invokes the correct command
    - Registry that associates a program with a file type

2.2 File Structure and Operations
- Files can also be viewed as serialized representations of data

3. Other Types of Files
- Other types of files

3.1 Directories
- Represent name-spaces
- Association of names and data
- Namespaces from a directory includes files from different users

3.2 Inter-Process Communication Ports
- **Pipe:** where data is stored, where data is written

3.3 I/O Devices
- Some devices can use standard file access models

4. Attributes
- Files have data and metadata

4.1 System Attributes
- Standard set
    - Type
    - Ownership
    - Protection
    - Time information
    - Size
- All files have attributes
- OS depends on these attributes
- OS maintains these attributes

Wiki: Key-Value Database

- **Object storage** (object-based storage): data storage architecture that manages data as objects
    - Instead of file systems or block storage
- Objects include: data and metadata
- Allows retention of massive amounts of unstructured data

History

→ Origins
- 1995: Garth Gibson's research promoted the idea of splitting less common operations to optimize

- Object storage proposed in 1996
- Abstract writes and reads to data objects
- Fine grained access control

Architecture

→ Abstraction of Storage

- Abstract lower layers of storage from admin and applications
- Data is exposed and managed as objects instead of files or blocks
- **Objects:** have additional descriptive properties for better indexing or management
- No lower-level storage functions need to be performed
- Object storage – allows addressing and ID of objects by more than one file name/path
- Adds a unique identifier within a bucket or across an entire system
    - Larger name spaces
    - Eliminate name collisions

→ Inclusion of Rich Custom Metadata

- Separate metadata from data allows for additional capabilities
- Allows full fulciton, custom, object-level metadata
    - Capture info for better indexing
    - Data management policies
    - Centralize storage management across nodes/clusters
    - Optimize metadata storage and caching/indexing
- **Object-based storage devices (OSD)** manage metadata and data at the storage device level
    - Data organized into objects
    - Objects have data and metadata
    - Command interface allows creating and deleting, writing and reading, and getting and setting after

Kampe: An Intro to DOSFAT Volume and File Structure

1. Introduction

- Heavily used
- Reasonable performance with simple implementation
- Successful “linked list” space allocation

2. Structural Overview

- File system includes basic data structures
    - **Bootstrap:** code loaded and executed when computer powered on
    - **Volume descriptors:** information about the size, type, and layout and how to find other key metadata descriptors
    - **File descriptors:** information about a file, points to where the data is actually stored
    - **Free-space descriptors:** lists of unused space that can be allocated

        - **File name descriptors:** data structures mapping user chosen names to files
    - **DOS FAT file system:** divide volume into physical blocks grouped into larger logical divisions, all with a fixed size
    - First block of DOSFAT volume contains bootstrap with descriptor info
    - After this is the **File Allocation Table (FAT)**
        - Used as a free list
        - Used to keep track of which blocks have been allocated to which files
    - Remainder of the volume is data clusters
        - Can be allocated to files and directories

3. Boot Block BIOS Parameter Block and FDISK Table

- DOS combines first block and volume descriptor information
- Boot record:
    - Begins with branch instructions
    - Followed by a volume description
    - Real bootstrap code
    - Optional disk partitioning table
    - Signature

3.1 BIOS Parameter Block

- **BIOS Parameter block** describes the device geometry
    - # bytes per sector
    - Sectors per track
    - Tracks per cylinder
    - Total # of sectors on the volume
- Describes layout on the volume
    - sectors/cluster
    - # of reserved sectors
    - # of Alternate File Allocation Tables
    - Entries in root directory

3.2 FDISK Table

- Added to the end of the bootstrap block
- Allows **multiple file systems** to exist on a disk
- Four entries, each describing one disk partition
    - Partition type
    - ACTIVE indication (the one that the system is booted from)
    - Disk address where partition starts and ends
    - Number of sectors on that partition

4. File Descriptors (Directories)

- Combine file descriptors and file naming into a single file descriptor (directory entry)
- **DOS directory:** file that has fixed size directory entries (1 entry/file)
    - 11 byte name
    - Byte of attribute bits

        - file/directory
        - Has the file changed
        - System file
        - Hidden
        - Read-only
        - Volume label
    - time/date of creation
    - Date of last access
    - Pointer to logical block
    - Length
- First character of the name is NULL if the file is unused
- 0xE5 means that it is a deleted file

5. Links and Free Space (File Allocation Table)

- DOS design: one entry for each logical block in volume
    - If it is free, it is indicated by a FAT entry
    - If it is allocated, FAT entry gives logical block number of the next logical block in the file

5.1 Cluster Size & Performance

- clusters/bock - determined when the file system is created
- More space in larger chunks - improves I/O performance, reduces number of operations to read/write
    - High internal fragmentation

5.2 Next Block Pointers

- File's directory entry is a pointer to the first cluster of the file
- File allocation table entry: tells the cluster number for the next cluster
- Last cluster → the FAT entry is -1
- Need sequential access

5.3 Free Space

- Value 0 → file is free in FAT entry

5.4 Garbage Collection

- Garbage collection
- Files are not freed, simply crossed off
- Find valid entries and enumerate contents
- Clusters not found are marked as free
    - Able to recover their contents

6. Descendants of the DOS File System

- Evolved with time
- Wider FAT entries and other features

6.1 Long File Names

- Longer and mixed-case file names
- Put extended file names in additional auxiliary directory entries

File System Performance

Lecture 14: File Systems – Allocation, Naming, Performance, and Reliability

Outline

- Allocating and managing file system free space
- Other performance improvement strategies
- File naming and directories
- File system reliability issues

Free Space and Allocation Issues

- Keeping track of a file system's free space
- Allocating new disk blocks
    - Handling deallocation

The Allocation/Deallocation Problem

- File systems **usually are not static**
- Files are created and destroyed
- File contents are changed
- Changes **convert unused disk blocks to used disk blocks** or vice versa
- Need to keep track of allocated vs. free space
- Need to be able to take free space and add it to a file
- Need to do this **correctly and efficiently**
- Typically implies maintaining a **free list** of unused blocks
    - Maybe using a bitmap
    - Or a free list (linked list, tree, etc.)

Creating a New File

- You need space for both **data and metadata**
- Allocate a **free file control block**
    - In UNIX
        - Search the super-block free Inode list
        - Take the first free Inode
    - For DOS
        - Search the parent directory for an unused directory entry
- Initialize the file control block
- Give the file a **new name**
    - One that makes sense to application and to users
- DOS doesn't use inodes
    - Directory entries **hold all metadata**
    - If a directory is full, need to **allocate a new block for that directory** to name more entries

Extending a File

- Application **requests** new data be assigned to a file
    - May be an explicit **allocation/extension request**
    - May be **implicit** (writing to a non-existent block)
- **Find** a free chunk of space
    - May also need to get the indirect block as well
    - **Traverse** the free list to find an appropriate chunk
    - **Remove** the chosen chunk from the free list
- **Associate** it with the appropriate address in the file
    - Go to the appropriate page in the file or extent descriptor
    - Update it to point to the **newly allocated chunk**
- Need to update the **file control block**
    - Either inode or directory entry
- The free list, metadata structure, and file are all shared
    - Need atomicity
    - Need persistence
- Operations:
    - Change the free list
    - Change the metadata
    - Change the content of the block itself

Deleting a File

- **Release** all the space that is allocated to the file
    - UNIX: return blocks to the free block list
    - DOS does not free space
        - Use **garbage collection**
            - Because you allocate sequentially in the DOS FAT file system
            - Doesn't update the FAT table
            - Changes a bit to indicate that this chunk is empty
        - Search out the **deallocated blocks** and add them
- **Deallocate** the file control block
    - UNIX: zero the inode and return it to the free list
    - DOS: zero the first byte of the name in the parent directory
        - Indicate that the directory entry is no longer in use

Free Space Maintenance

- File system manager manages the free list
- We need to maintain the free list
- **getting/releasing** blocks should be fast operations
    - Extremely frequent
    - Avoid as much I/O as possible
        - Under the assumption of a hard disk drive underneath
- Unlike in memory, it **matters which block we chose**

        - Best to allocate new space **in the same cylinder** as the file's existing space
            - For quicker access later
            - Less head movement
        - User may ask for **contiguous storage**
    - **Free-list organization** must address both concerns
        - Speed of allocation and deallocation
        - Ability to allocate contiguous or near-by space
    - Same cylinder helps because all heads move in an out at once
        - Want heads to be on the same corresponding tracks on different surfaces

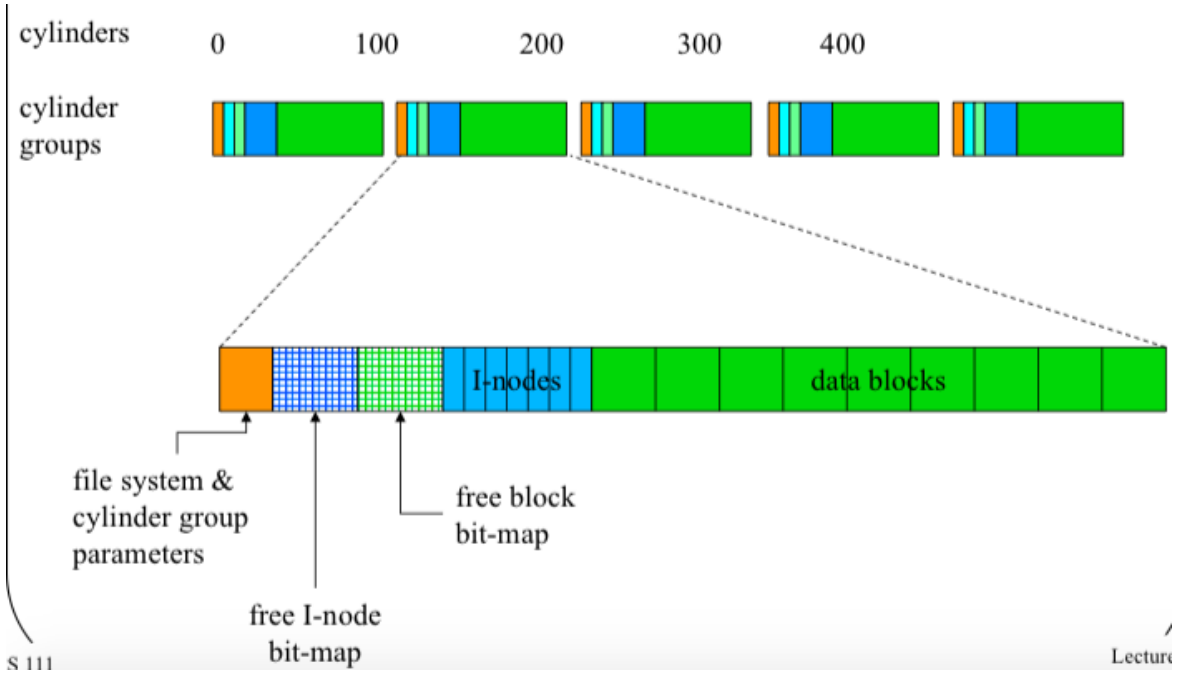
The BSD File System Free Space Management

- BSD is another version of UNIX
- Details of inodes similar to those of Unix System V
- Aspects are different
    - Free space management
    - More advanced typically
- Uses a **bitmap approach** to managing free space
    - **Keep cylinder issues in mind**

The BSD Approach

- Tries to keep related files in the same cylinder to **optimize head movements**
- Divides the file system into **cylinder groups**
    - Each cylinder group has its own control information
        - The **cylinder group summary**
        - Metadata for each cylinder
    - Active cylinder group summaries **are kept in memory** as cached information
        - Prevents extra I/O when switching between groups
    - Each cylinder group has its own **inodes and blocks**
    - Free block list is **a bitmap in cylinder group summary**
- Enacles **significant reductions** in head motion
    - Data blocks in file can be allocated in the **same cylinder**
    - Inode and data blocks are in **same cylinder group**
    - Directories and their files are in the same **cylinder group**

BSD Cylinder Groups and Free Space



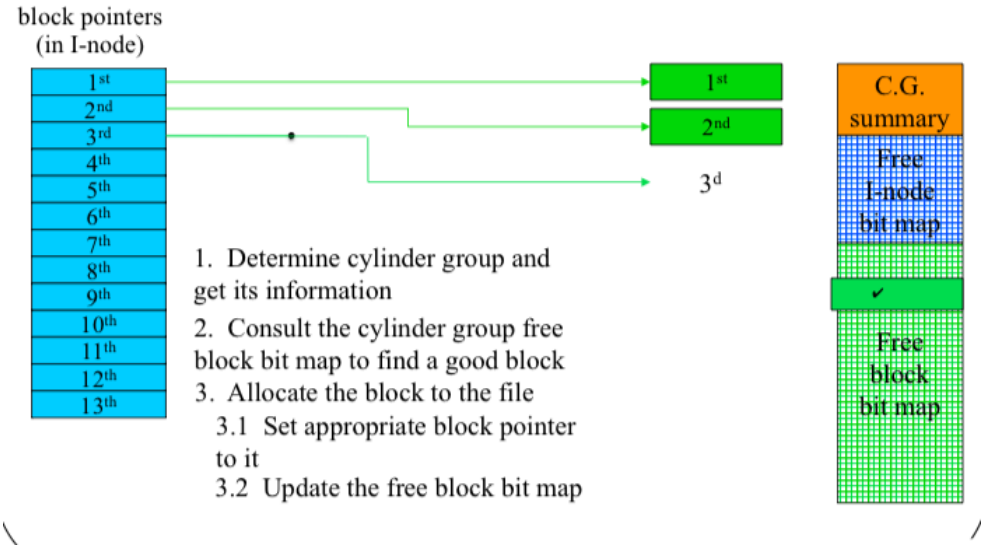
Bit Map Free Lists

- BSD Unix file systems use bit-maps to keep track of free blocks and free inodes in each cylinder group
- Quick to use
- Bit map is cached in RAM
    - Logical operations to access bits are fast
- 1 = free
- 0 = in use

Extending a BSD/Unix File

- Determine the **cylinder group** for the file's inode
    - Calculate from the **inode's identifying number**
- Find the cylinder for the previous block in the file
- Find a **free block** in the **desired cylinder**
    - Search the free-block bitmap for a free block in the right cylinder
    - Update the bitmap in memory to show it has been allocated
- Update the **inode** to point to the new block
    - Go to the appropriate block pointer in the inode/indirect block
    - If the new indirect block **is needed, allocate/assign it first**
    - **Update inode/indirect** to point to the new block
- So far, this has all been done **in memory**, the disk still has the old bitmap and inode

Unix File Extension



Other Performance Improvement Strategies

- Beyond disk layout issues
  - Only relevant for hard drives
  - All flash or SSD drives are equally slow
- Transfer size
- Caching

Allocation/Transfer Size

- **Per operation overheads** are high
  - DMA startup, seek, rotation, interrupt service
- Larger transfer units **are more efficient**
  - Move in big chunks when we can
  - **Amortize fixed per-operation costs** over more bytes/operation
- This requires **space allocation units**
  - Allocate space to files in **much larger chunks**
  - Large fixed size chunks lead to **internal fragmentation**
  - Therefore, we need **variable partition allocation**
    - Causes **external fragmentation** that we need to worry about

Efficient Disk Allocation

- Allocate space in **large, contiguous extents**
- Variable partition disk allocation is difficult
  - More complicated to find something that fits than to always use a single allocation size
  - Many files are **allocated for a long time**
- Fragmentation is more difficult to deal with if you **hardly deallocate** which is the case for files
- External fragmentation **eventually wins**
  - New files get smaller chunks, further apart
  - File system performance degrades with age

        - Do we care if the file is stored at this particular location?
        - We can do **relatively relocations since pointers are in the inodes**
        - We need to **compact the disk** but this is very slow
        - It goes through all of the pointers and figures out optimal placement
        - Then it will rewrite all of the data

Caching

- Caching for reads
- Caching for writes

Read Caching

- Disk I/O takes **a very long time**
    - **Deep queues, large transfers** will improve efficiency
    - They do not make it **significantly faster** though
- We need to **eliminate much of our disk I/O**
    - Maintain an **in-memory cache**, the block I/O cache
        - Block I/O cache should be organized to make checking possible and efficient
    - Depend on **locality**, and reuse of the same blocks
    - Check the **cache** before scheduling I/O

Read-Ahead

- Request blocks from the disk **before any process asked for them**
    - Put it in the block I/O cache before it is requested
- Reduces the **process wait time**
- When does this make sense?
    - When the client **specifically requests** sequential access
        - On files, signals that you will be doing operations that make sense for a read-ahead
    - When the client **seems** to be reading sequentially
- What are the risks?
    - May **waste disk access time** reading unwanted data
        - OS is asking for I/O that was not yet requested
    - May **waste buffer space** on unneeded blocks
    - If the block **wasn't really needed**, we will waste **time and space and other resources**

Write Caching

- Has both **performance and correctness risks**
- Usually writes occur in RAM
- Most disk writes go to a **write-back cache**, not persistent, in RAM
    - Flushed out later to disk
- Aggregates **small writes into large writes**
    - If the application does less than full block writes
- Eliminates **moot writes**
    - Application **rewriting the same data**

        - Application subsequently **deleting the file**
    - Accumulates **large batches of writes**
        - A **deeper queue** to enable better disk scheduling
        - **Batch** can be optimized
            - For minimal possible overhead movement

Common Types of Disk Caching

- General **block caching**
    - Popular files that are read frequently
    - Files that are written and then promptly re-read
    - Provides buffers for read-ahead and deferred write
- Special purpose caches
    - **Directory caches** that speed up the search of the same directory
    - **Inode caches** that speed up reuse of the same file
- Special purpose caches are **more complex**
    - But often work much better by **matching cache granularities** to actual needs
    - Don't cache at the level of a data block since things **are of different sizes**

Naming in File Systems

- Each file needs some kind of handle to allow us to refer to it
- **Low level names** like inode numbers aren't usable by people or even programs
- We need a **better way to name files**
    - User friendly
    - Easy organization of large numbers of files
    - Readily realizable in file systems
    - Requires a **translation step** that is easy and efficient

File Names and Binding

- File system knows files by **descriptor structures**
- Must provide more useful names for users
- The file system must handle **name-to-file mapping**
    - Associating names with **new files**
    - Finding the underlying **representation for a given name**, the appropriate mapping
    - **Changing names** associated with existing files
    - Allowing users to **organize files using names**
- **Name spaces:** the total collection of all names for files known by some naming mechanism
    - Sometimes all names that **could** be created by the mechanism
    - Total set of all possible names

Name Space Structure

- Many ways to **structure** a namespace
    - **Flat name spaces**
        - All names exist in a single level

        - Unique name for every file
        - Implies no directories
    - **Hierarchical name spaces**
        - A graph approach
        - Can be a **strict tree**
        - Or a more **general graph**, directed usually
- Are all files on the machine under the **same name structure?**
    - They don't have to be
    - More convenient for programmers if they are though
- Or are there **several independent name spaces?**
    - Every machine has its own name sace

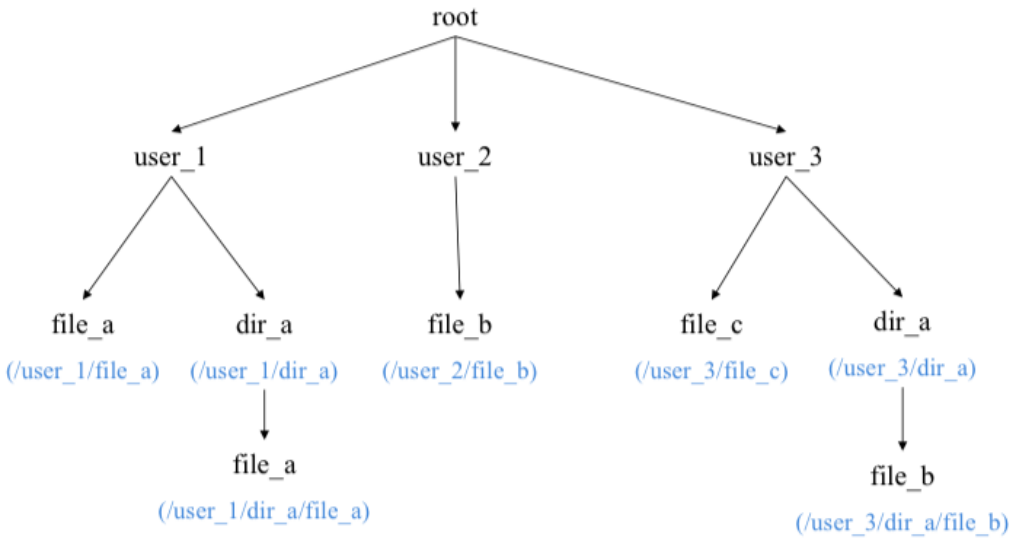
Some Issues in Name Space Structure

- How many files can have the same name?
    - **One per file system** - flat name spaces
    - **One per directory** - hierarchical name spaces
- How many **different names can one file have**
    - A single "true" name
    - One "true" name but aliases are allowed
    - Arbitrarily many
    - What's different about "true names?"
- Do **different names** have **different characteristics**
    - Does deleting one name make the others disappear too
    - Do all names **see the same access permissions**

Hierarchical Name Spaces

- Essentially a **graphical organization**
- Typically organized using **directories**
    - File containing references to other files
    - A non-lead node in the graph
    - Can be used as a naming context
        - Each process has a **current directory**
        - File names are interpreted **relative to that directory**
- Files: leaf nodes
- Directories: potential non-leaf nodes
- Nested directories can form a **tree**
    - A file name **describes a path through that tree**
    - The directory tree expands from a **"root" node**
        - Name beginning from the root is **"fully qualified"**
    - May actually form a **directed graph**
        - If files can have multiple names

A Rooted Directory Tree



- Some are the same name
- But they can be distinguished using their fully qualified names

Directories are Files

- Namespaces are **specified by directory**
- They are a **protected structure** defined by the file system that you cannot write
  - May mess up the file system
- Directories are **a special type of file**
  - Used by OS to **map file names** into the associated files
  - Protected by the OS
- Directory contains **multiple directory entries**
  - Each directory entry describes **one file and its name**
- User applications are **allowed to read directories**
  - To get information about the file
  - To find out what files exist
  - Cannot **write to them**
- Usually **only the OS is allowed to write them**
  - Users can cause writes with special system calls
  - File system depends on **integrity of directories**

Traversing the Directory Tree

- Some entries in directories point to child directories
  - Describing a lower level in the hierarchy
- To name a file at that level, **name the parent directory and the child directory, then the file**
  - With some kind of delimiter separating the filename components
- Moving up the hierarchy is **often useful**
  - Directories usually have some special entry **for parent**
  - The **..** **directory is the parent directory**

        - Implies you can only have **one parent**

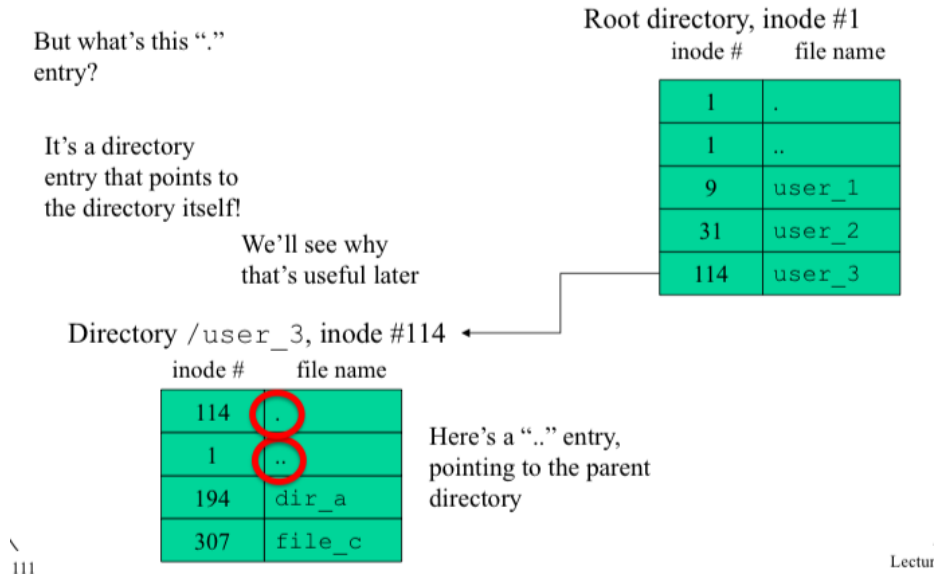
File Names vs Path Names

- In some name spaces, files had **“true names”**
    - One possible name for a fie
    - Kept somewhere in record
- EX: in DOS, a file is described by a directory entry
    - **Files have names associated**
    - Local name is specified in the directory entry
    - Fully qualified name is the path to that directory entry, the only path
- What if files had **no intrinsic names** of their own
    - All file names came from **directory paths**
    - In the Unix file system, the inode **doesn’t contain the name**
        - There is no true name
    - Metadata structure **has no name in it**

Example: Unix Directories

- A file system that **allows multiple file names**
    - No single “true” file name, unlike DOS
    - Every name is a **hard link**
- File names separated by slashes
- The **actual file descriptors** are the inodes
    - Directory entries **point to inodes**
    - Association of a name with an inode is a **hard link**
        - Multiple names that point to the same inode
        - Purpose of the name is to get to the inode
    - Multiple directory entries can point to the same inode
- Contents of a Unix directory entry
    - Name (relative to this directory)
    - Pointer to the inode of the associated file

Unix Directories



- Inode 114 doesn't know that it is a directory
- Only know that it is once we reach it and see that it is a directory

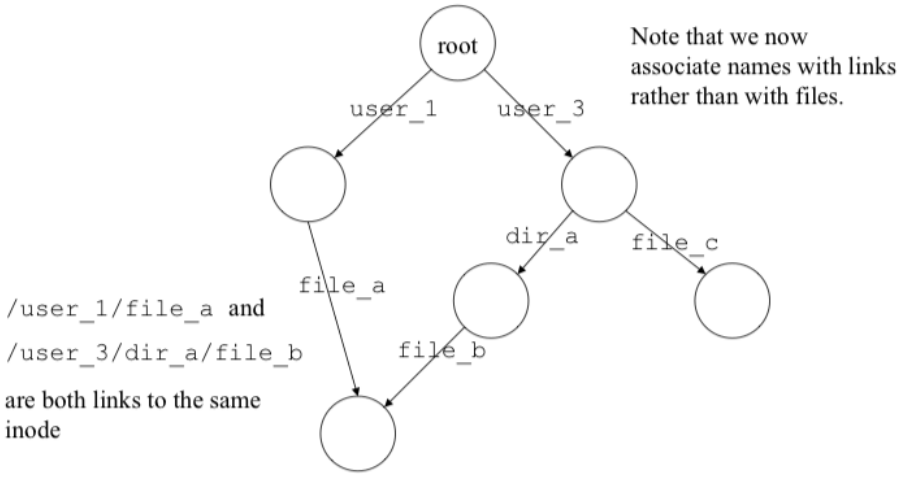
Multiple File Names in Unix

- How do **links relate to files**
  - They're only names
- All other **metadata is stored in the file inode**
  - Metadata is in the inode
  - Metadata contains no naming information
- All links **provide the same access** to the file
  - All names have the same access permissions
- All links **are equal**
  - Nothing special about the first link
  - No information about the name in the inode/metadata

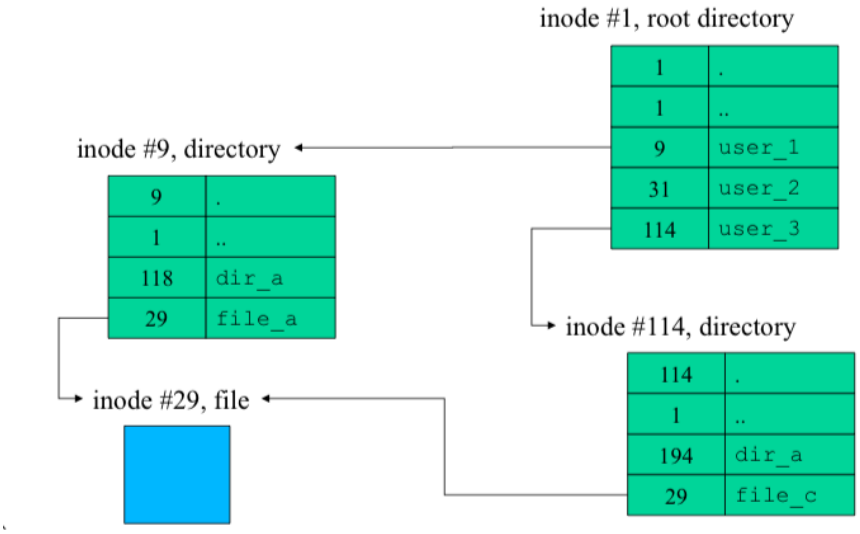
Links and Deallocation

- Files exist under **multiple names**
- What do we do if a name is removed?
  - Cannot deallocate the file
  - File still exists if **at least one name is left**
- Unix: file exists as long as **at least one name exists**
- Implies we need to **keep and maintain a reference count of links**
  - In the file inode, not in a directory

Unix Hard Link Example



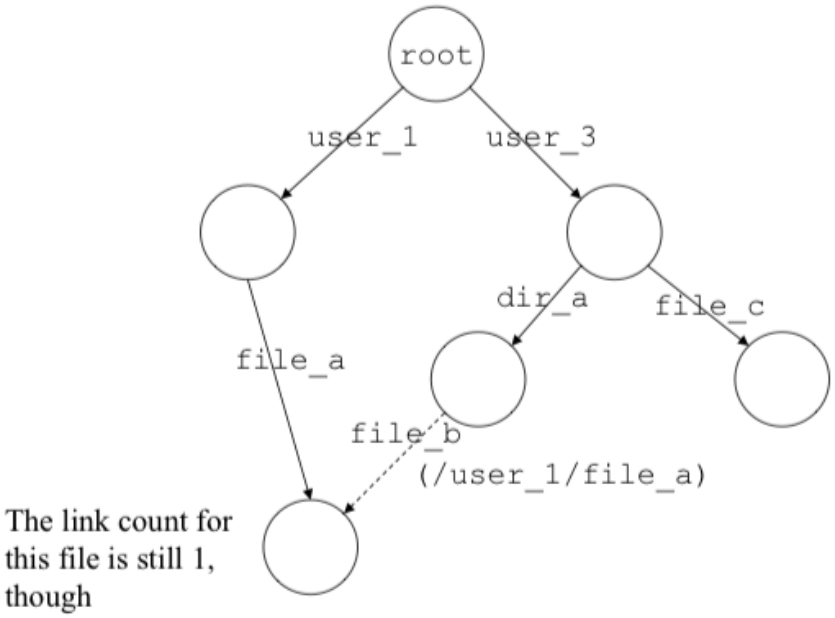
Hard Links, Directories, and Files



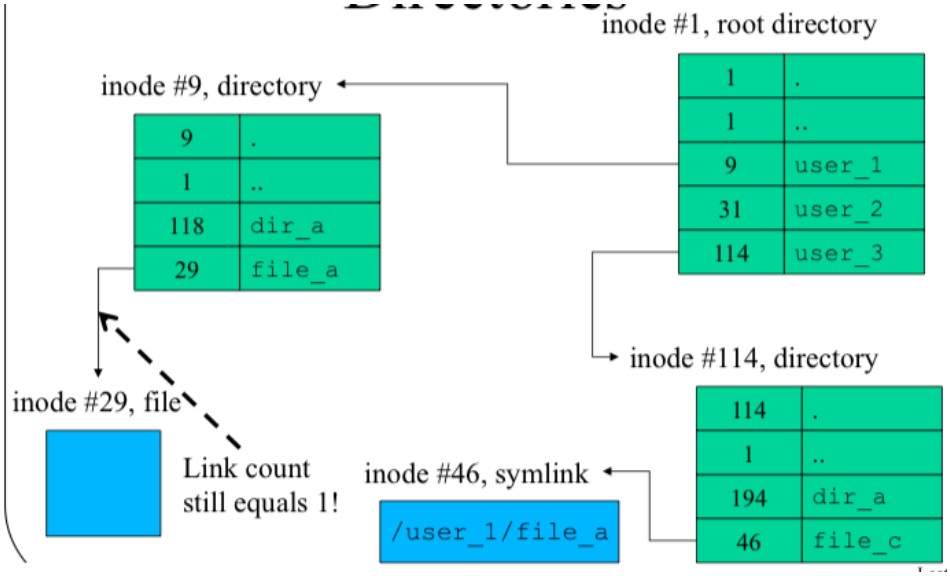
Symbolic Links

- A different way of giving files multiple names
- Symbolic links are **implemented as a special type of file**
  - Indirect reference to some other file
  - Contents is **a path name to another file**
- File system recognizes symbolic links
  - Automatically opens the associated file instead
  - Open will fail if the file is inaccessible or non-existent
  - **Opening a symbolic link – tries to follow the path**
- Symbolic links **are not a reference to the inode**
  - Symbolic links **do not prevent deletion**
  - It is just a **path name**
  - Does not guarantee **ability to follow specified path**
    - If the hard link is deleted, the file is deleted
    - The symbolic link won't work

Symbolic Link Example



Symbolic Links, Files, and Directories



File Systems Reliability

- What can go wrong in a file system?
- **Data loss**
  - File or data is no longer present
  - some/all of data cannot be read correctly back
- **File system corruption**
  - Lost free space
  - References to non-existent files
  - Corrupted free-list multiple allocates space
  - File contents over-written

        - Corrupted directories
        - Corrupted inodes

Storage Device Failures

- Unrecoverable **read errors**
    - Signal degrades beyond ECC ability to correct
    - Background **scrubbing** can greatly reduce this
- Misdirected or incomplete writes
    - Detectable with **independent checksums**
- Complete **mechanical/electronic failures**
- All are correctable with **redundant copies**
    - Mirroring, parity, or erasure coding
    - Individual block or whole volume recovery
    - At worst, a backup

File Systems - System Failures

- Caused by **system crashes or OS bugs**
- Queued writes that **don't get completed**
    - Client writes that will not be persisted
    - Client creates that will not be persisted
    - Partial multi-block file system updates
- Can also be caused by power failures
    - Solutions:
        - NVRAM (non volatile RAM) disk controllers
        - Uninterruptible power supply
        - Super-caps and fast flush

Deferred Writes - A Worst Case Scenario

- Process allocates a new block to file A
    - Get a new block x from free list
    - Write out the updated inode for file A
    - Defer the free-list write-back (happens often)
- The system crashes and after it reboots
    - New process wants a new block for file B
    - We get block x from the corrupted/stale free list
- Two different files now **contain the same block**
    - B is corrupted when A is written and vice versa

Robustness - Ordered Writes

- **Carefully ordered writes** can reduce potential damage
    - Data first → metadata second is **safer**
- Write out **data before writing pointers** to it
    - Unreferenced objects can be **garbage collected**
    - Pointers to incorrect info are more serious
    - Worse case: data block that isn't hooked up
    - But no corrupted data blocks, just an unused data block
- Write out **deallocations before allocations**

        - Disassociate resources from old files ASAP
        - Free list can be corrected with garbage collection
        - Shared data is more serious than missing data

Practicality of Ordered Writes

- Greatly **reduced I/O performance**
    - Eliminates head/disk motion scheduling
    - Eliminates accumulation of nearby operations
    - Eliminates consolidation of updates to the same block
- Bad for performance, prevents optimizations of head movements
- May not be possible
    - Disk drivers may **reorder the queued requests**
- Doesn't actually solve the problem
    - Doesn't eliminate **incomplete writes**
    - Chooses **minor problems** over major ones

Robustness – Audit and Repair

- Design a file system structure for **audit and repair**
    - **Redundant information** is stored in **multiple distinct locations**
        - Maintain **reference counts** in each object
        - Children have **pointers back to their parents**
        - Transaction logs of all updates
    - Resources can be **garbage collected**
        - Discover and recover unreferenced objects
- **Audit** file system for correctness (prior to mount)
    - All objects well formatted
    - References and free-lists and correct and consistent
- Use the redundant info to **enable automatic repair**

Practicality of Audit and Repair

- Integrity checking a file system **after a crash**
    - Verifying checksums, reference counts
    - Automatically correct inconsistency
    - Standard for many years
- **No longer practical**
- Need a more **efficient partial write solution**
    - File systems that are immune to them
    - And enable **fast recovery**

Journaling

- Create a **circular buffer journaling device**
    - Journal writes **are always sequential**
    - Can be **batched**
    - Relative small, may use NVRAM
    - Contiguous area on the hard disk drive that is **exclusively for the journal**
- Journal **all intended file system updates**

        - Inode updates, block write/alloc/free
    - Efficiently **schedule actual file system updates**
        - Allows us to optimize
        - Write-back cache, batching, motion-scheduling
    - Journal **completions** when real writes happen

Batched Journal Entries

- Operation is **safe after journal entry is persisted**
    - Caller must **wait** for this to happen
    - As long as the journal stas if you crash, you're safe
- Small writes are still inefficient
- Accumulate batch until **full or max wait time**
    - Block needs to be written out for operations to be safe
- The only benefit of journaling is helping us with failures

Journal Recovery

- Journal is a **circular buffer**
    - It can be recycled after all operations have been completed
    - Time-stamps distinguish new entries from old ones
- After the system restarts
    - Review the **entire (relatively small) journal**
        - Look at all the operations that were supposed to be performed
    - Note which operations are **known to have been completed**
    - Perform all writes **not known to have completed**
        - Data and destination both in journal
        - All of these writes are **idempotent**
            - These actions can occur as many times as you want and you'll get the same result
            - Simply writing over and over for the same data
    - Truncate journal and resume normal operation

Why Does Journaling Work?

- Journal writes are **must faster than data writes**
    - All journal writes are **sequential**
    - There is no **competing head motion**
- In normal operation, journal is **write-only**
    - File system never reads/processes the journal
    - We can recycle the parts that have already completed
- Scanning the journal on restart is **very fast**
    - It is very small, **compared to the file system**
    - Can read **sequentially with huge reads**
    - All recovery processing is done **in memory**

Meta-Data Only Journaling

- Why journal meta-data?
    - **Small and random** (I/O inefficient)
    - And **integrity-critical** (huge potential data loss)

        - If you get meta wrong, it's BAD
            - Metadata is the most important
            - But you can't journal everything
    - Why not journal data?
        - Often **large and sequential** (I/O efficient)
        - Would **consume most of journal capacity bandwidth**
        - It is **less order sensitive** (just precedes the meta-data)
    - Safe metadata journaling
        - Allocate the new space
        - Write the data
        - Then journal metadata updates
    - In general, if you **write the data before the metadata**
        - You're safer in crashes
    - Just journal the metadata once data is in memory
        - Journal **only metadata updates**
        - Solves the problem of large writes

Log Structured File Systems

- Only keep the journal, nothing else
- No sectors for data inodes
- This is **not as efficient for hard disk drives**
- The **journal is the file system**
    - Inodes and data updates are **written to the log**
    - Updates are **redirect-on-write**
        - Never overwrites old data locations
    - **In-memory index** caches inode locations
        - Now **metadata is scattered**
        - No data structure to keep track of where data is
    - When writing new data, just write **a new block and change the inode pointer** and create a new inode for it
- Becoming a dominant architecture
    - Flash file systems
    - key/value stores
- Issues
    - Recovery time (to reconstruct index/cache)
    - Log defragmentation and garbage collection
        - Garbage collection means that data needs to be copied somewhere else

Navigating a Logging File System

- Inodes **point at data segments** in the log
    - Sequential writes may be contiguous in log
    - Random updates can be spread all over log
    - Inodes are **spread out on disk**

        - Use an index in memory to be fast, but needs to still be persistent
    - Updated inodes are **added to the end of the log**
    - **Index** points to the latest version of each inode
        - Index is **periodically appended** to the log for persistency
    - Recovery
        - Find and recover the latest index
        - Replay all log updates since then

Redirect on Write

- Modern file systems do this
    - Blocks and inodes are **immutable** once written
    - Add new info to the log and update the index
- The old inodes and data remain in the log
    - If we have an old index, we can access old data
    - Clones and snapshots are almost free
- Price is **management and garbage collection**
    - Must inventory and manage old versions
    - Eventually need to recycle old log entries

Reading

Arpaci-Dusseau Ch. 41 - Locality and the Fast File System

- Old, simple file system
- Superblock: contains information about the file system
- After superblock is the inode block
- After the inode block is the data block
    - Most of the disk is taken by big data blocks

41.1 The Problem: Poor Performance

- Problem: performance was terrible
    - Only 2% of overall disk bandwidth eventually
- Problems:
    1. **Treated the disk like random-access memory**
        - Data was spread out with no regard to disk layout
        - Real and expensive positioning costs
    2. **The file system was fragmented**
        - Free space was not managed well
        - When accessing, you don’t get peak sequential performance
        - Defragmentation helps with this because the tools reorganize disk data to place files contiguously and make free space for contiguous regions
    3. **Original block size is small to minimize fragmentation**

        - Transferring data is inherently inefficient
            - Requires positioning overhead

41.2 FFS: Disk Awareness is the Solution

- **Fast file system (FFS):** design file system architectures and structures and allocation policies to be “disk aware”
    - Improved system performance
- Keep the same interface but **change the internal implementation**

41.3 Organizing Structure: The Cylinder Group

- First step: changes on disk structures
- FFS divides disk into **cylinder groups**
- **Cylinder**: set of tracks on different surfaces of a hard drive the same distance from the center of the drive
- FFS aggregates N consecutive cylinders into collections of **cylinder groups**
    - Modern systems organize the drive into **block groups**, consecutive potions of the address space, because the geometry is hidden by the disk
- To use groups for files and directories:
    - FFS needs to place files and directories into a group
    - And track necessary information
- To do this, FFS includes the structures necessary in the group
    - Superblock: reliability, needed to mount the file system
        - Multiple copies
        - If one is corrupted, you can still mount and access the file system
    - Inode and data bitmaps: tracks whether blocks are free or allocated
    - Data and inode data blocks: like the vsfs, takes up most of the disk

41.4 Policies: How to Allocate Files and Directories

- FFS needs to decide how to place files and directories and metadata on disk to improve performance
- The main idea: **keep related stuff together**, and unrelated stuff far apart
    - Needs to decide what is “related”
- Uses **placement heuristics**
- Placement of directories:
    - Find the cylinder group with a low number of allocated directories and high number of free inodes
    - Put directory data and inode in that group
    - Goal is to **balance directories** across groups
- Placement of files:
    - **General case:** allocate data blocks of files in the same place as the inode
        - Prevents long seeks
    - Place all files in the same directory in the cylinder group of that directory

- Files in a directory not arbitrarily spaced around the disk preserves namespace locality
- Policies are based on common sense
    - Files usually are accessed together
- Namespace locality will improve performance

41.5 Measuring File Locality

- Analyze the namespace locality
- Distance metric: measures how far up the directory tree you have to travel to find the common ancestor of two files
- Result: files usually are close together

41.6 The Large File Exception

- FFS exception is a **large file**
- Without a different rule, large files would **fill up the entire block group** which is not desirable
    - Prevents subsequent “related files” from being placed in the same block group
    - Hurts locality
- FFS does the following for large files
    - Some number are allocated in the first block group
    - The next large chunk is in another block group
    - The next chunk is in another
    - And so on,...
- Spreading blocks across disk hurts performance
    - We can address this by **choosing the block size**
    - Large chunk size means that the file system spends time transferring the data and **less time seeking** between chunks of the block
- **Amortization**: reduces overhead by doing more work per overhead paid
- FFS spreads large data files out based on the structure of the inode
- Transferring more data between seeks means that mechanical writes are more expensive

41.7 A Few Other Things About FFS

- Other run conditions
- Small files may cause internal fragmentation
    - Solution, use **sub-blocks**
- **Sub-blocks:** 512-byte smaller blocks that can be allocated to files
    - Continues until 4KB of data is reached
    - Then copy all of the data into 4KB → this is inefficient
- Avoid this by **buffering** then issuing 4KB chunks
- **Avoid sub-block specialization**
- Disk layout is optimized for performance
    - Solved the issue in sequential reads that may result in the disk having already rotated
    - Another layout is **skipping over blocks**

        - Figure out how many blocks should be skipped for a particular disk to avoid extra rotations, called **parametrization**
- Modern disks read tracks in and buffer on a disk cache called a **track buffer**
    - Subsequent reads return data from this cache
- Allowed long file names
- **Symbolic links** introduced
    - Allows the user to create an alias to another file or directory

Arpaci-Dusseau Ch. 42 - Crash Consistency: FSCK and Journaling

- File system data structures must persist → how do we update them despite a power loss or system crashes?
- Power losses and crashes make updating persistent data structures tricky, leads to the **crash-consistency problem**
    - Two data structures need to be updated to complete an operation
    - Disk serves one request at a time
        - One request reaches the disk first
    - If the system crashes or loses power after only one write is complete
    - The on-disk structure is in an **inconsistent state**
- The approach used by old systems is **fsck/file system checking**
- **Journaling** (write-ahead logging): is used for quicker recovery but incurs more overhead

42.1 A Detailed Example

- **Simple workload:** append a data block to an existing file
- Standard simple file system with inode, data bitmap, and inode and data blocks
- When adding/appending to a file, we add a **new data block**
    - Causes updates to **3 on-disk structures:**
        - Inode (points to new block and has larger size)
        - New data block D^b
        - New version of the data bitmap
    - Need to write three blocks to disk
- To achieve this transition, need to have **3 writes to disk**, one for the inode, bitmap, and data block
    - They sit in main memory in the page/buffer cache until the file decides to write them to disk
    - A crash may occur
        - If this happens after 1 or 2 of the writes, but not all three, the system state could be bad

→ Crash Scenarios

- Say one write succeeds → there are three possible outcomes
- Just the **data block is written**
    - The data is on disk but the inode doesn't point to it
    - There is no bitmap that indicates the block is allocated

        - Looks like **write never occured, no consistency issue**
    - Just the **updated inode is written**
        - The inode will point to where the data was about to be written
        - Which will yield **garbage data**
        - There is a new problem, **file-system inconsistency**
            - Bitmap says it is not allocated
            - Inode says it has been
    - Just the **updated bitmap is written**
        - Inconsistent state
        - Results in a **space leak** as block will never be used again
    - Say we attempt 3 and 2 succeed
    - The **inode and bitmap are written but data is not**
        - Metadata is consistent but the data block has garbage in it
    - **Inode and data blocks are written but bitmap is not**
        - Inconsistency between inode and bitmap
    - **Bitmap and data block are written but inode is not**
        - Inconsistency between inode and bitmap
        - Block is written and indicated so on the bitmap but **we don't know which file it belongs to** since there is no inode

→ The Crash Consistency Problem

- Issues occur because of crashes
- Ideally, we want to move from one consistent state to another atomically
- Difficult because disk only writes one at a time
- The problem: crash-consistency problem/consistent-update problem

42.2 Solution #1: The File System Checker

- Early file systems used a lazy approach
    - It let inconsistencies happen and fixed them later
- Example: fsck - tool for finding inconsistencies and repairing them
    - Doesn't fix all problems, like garbage data
    - Goal: make sure that **metadata is consistent**
- Run it before the file system is mounted and made available
- Once completed, file system should be consistent
- Summary of fsck:
    - Superblock: checks if it is reasonable
        - Sanity checks to see if file system size > number of blocks allocated
        - Goal: find a corrupt superblock
    - Free blocks: scans inodes, indirect blocks, double indirect blocks, etc. to see which are allocated
        - Produces correct allocation bitmap and checks for inconsistencies
        - Same check for inodes
    - Inode state: inodes check for corruption or other problems

        - If difficult to resolve problems exist, the inode is cleared and the bitmap is updated
    - Inode links: verifies link count of allocated inodes
        - **Link count:** number of different directories that contain a reference to this file
        - Scans through directory tree and builds link counts for every file and directory
        - If there is a mismatch
            - Fixes the count
        - Allocated inode with no directory is moved to the lost and found directory
    - Duplicates: check for duplicate pointers
        - clear/copy policy
    - Bad blocks: check for bad block pointers performed by scanning through all points
        - “Bad” meaning points to something outside the valid range
    - Directory checks: integrity checks on content of directories
        - Ensure that . and .. are entries 1 and 2
        - Inode referred to is allocated
        - No directory is linked more than once
- Requires knowledge of the file system
- **Fundamental problem: too slow**
    - Need to scan the entire disk
- Impractical and wasteful

42.3 Solution #2: Journaling (or Write-Ahead Logging)

- **Write-ahead logging/journaling:** when updating disk before overwriting structures, write note describing what you are about to do (on disk, in a well-known location)
    - Guarantees that if a crash takes place during an update, you can check the note and try again
    - Know exactly what to fix instead of scanning through the entire disk
- Adds work **during updates** but reduces it during recovery
- Linux ext3 incorporates journaling
    - New structure added called a journal
    - Occupies a space within partition or on another device

→ Data Journaling

- Here, 5 blocks are written

- TxB - tells you about the update
    - Information about the pending update
    - Transaction identifier (TIB)
- Middle blocks (I[v2], B[v2], Db) - content of blocks themselves
    - Physical logging: put exact physical contents into the journal
    - Logical logging: put compact logical representation into the journal
- TxE - marks the end of the transaction
- Once on disk, we can **overwrite the old structure** called, **checkpointing**
- To checkpoint the file system (update it with pending updates)
    - Issue writes for I[v2], B[v2], and Db to their disk locations
    - If they are successful, checkpoint is successful and we are done
- **Sequence of operations:**
    1. **Journal write:** write transaction, with begin block, all data and metadata updates, transaction end block, log, wait for writes
    2. **Checkpoint:** write pending metadata and data updates to their final locations
- Crash during writes to the journal → trying to write blocks in the transaction
- This is a simple method
    - They are done one at a time and this is slow
- Ideally we want to write **all 5 blocks at once**
    - Form 5 wires into a single sequential write
    - Fast but **unsafe**
        - Disk may internally schedule and split up large writes into any order
        - If disk loses power between these writes, there is a problem
- Avoid the problem by issuing in 2 steps
    - Write all but the TxE into the journal
    - When complete, write the TxE block
- Current protocol:
    1. **Journal write**: write events to log and wait
    2. **Journal commit:** write transaction commit block when complete, the transaction is complete
    3. **Checkpoint:** write contents of update to final disk location

→ Recovery

- **File system uses the journal contents to recover from a crash**
- A crash can occur at any time
- A crash before the transaction is committed means that the pending update is skipped
- Crash after the commit but before the checkpoint means that:
    - File system can recover from the update
    - When the system boots, file system recovery process scans the log and looks for committed transactions
    - Transactions are **replayed** in order

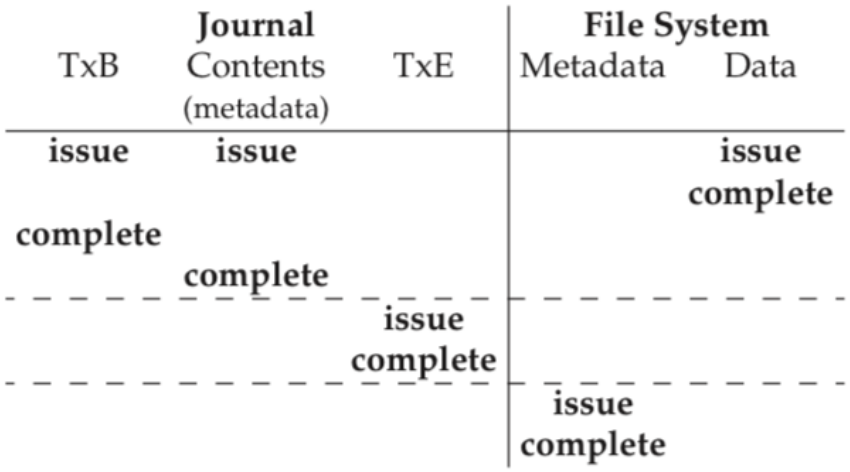
        - File system attempts to write out blocks
        - **Redo logging**
    - Crash during a checkpoint means that updates are performed again
    - Overall, crashes are rare
→ Batching Log Updates
    - Basic protocol results in heavy traffic
    - Creating one file and journaling means creating and writing a lot and commit all the new information to new logs
        - Two files in the same location can create a lot of redundant information
    - Remedy this issue by **not committing each update one at a time**
        - Instead, **buffer updates** into a global transaction
→ Making the Log File
    - Basic protocol: system buffers the updates in memory
        - Then writes out to journal
        - Then checkpoints to the final locations
    - Log is of a finite size
        - It may eventually get full and cause problems
            - Large log means longer recovery
            - Full log means that no transactions can be committed to disk, the file system becomes useless
    - To address these problems: use a **circular log** that we can reuse over and over
        - To do this, file system must take action after a checkpoint
        - Once the transaction is checkpointed, space it was using should be freed
            - Can do this by **marking oldest and newest non-checkpointed transactions** in a journal superblock, all of the other space should be free
            - **Journal superblock:** journaling system records information to know which transactions have not yet been checkpointed
                - Reduces recovery time
                - Enables re-use of the log
    - Basic protocol now:
        1. **Journal write**
        2. **Journal commit**
        3. **Checkpoint**
        4. **Free:** mark transaction as free in the journal superblock
    - Cost: writing data blocks to disk twice is **inefficient**
→ Metadata Journaling
    - Recovery now is fast but normal operation is slow
        - Each write to disk also means a write to the journal, doubling write traffic
        - Between writes to journal and to memory, there is a seek overhead
    - This incurs a high cost of writing data twice
    - Current approach: data journaling

        - Journals **all user data and metadata**
    - Another approach that is simpler and more common is **ordered journaling/metadata journaling**
        - User data is **not** written to the journal
    - Not writing the data twice reduces overhead
    - When we are done, write data to disk? The order becomes important
        - After the transaction, BAD, inode may point to garbage
        - Before transaction → used today
    - **Protocol:**
        1. **Data write**
        2. **Journal metadata write**
        3. **Journal commit**
        4. **Checkpoint metadata**
        5. **Free**
    - Force the data write first → guarantees that pointers won't point to garbage

→ Tricky Case: Block Reuse

- Corner cases that make journaling more tricky
- Reusing a block!
- EX: directory foo
    - User adds an entry → contents are written to log
    - User deletes everything
    - User creates file foobar, using the same block foo used before
    - Inode and data of foobar are committed to disk
    - Only the inode is committed to the journal, data is not included
    - CRASH
    - Recover replays, including overwrite of directory data
        - Overwrites user data of foobar with old data
    - This is incorrect
- Solutions:
    - Never reuse blocks until delete is checkpointed
    - Add new type of record, a **revoke record**
        - Deleting directory causes a revoke record to be created
        - Revoked data is not replayed after a crash

→ Wrapping Up Journaling: A Timeline



42.4 Solution #3: Other Solutions

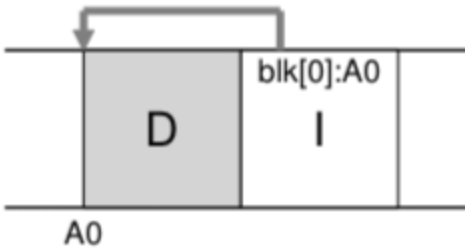
- Two approaches so far:
  - Lazy with fsck
  - Active journaling
- Another approach: **soft updates**
  - Orders all writes to ensure that no on-disk structures are in an inconsistent state
  - Challenge to implement: requires knowledge of data structures and adds complexity
- Another approach: **copy-on-write (COW)**
  - Never overwrite files or directories in place
    - Place new updates to previously unused locations on disk
  - After some updates, COW file systems flip root structure to include pointers to newly updated structures
  - Keeps file system consistency management fairly straightforward
- Another approach: **backpointer-based consistency (BBC)**
  - No ordering
  - To ensure consistency, a **back pointer** is added to every block
    - Each data has a reference to the inode it belongs to
  - When accessing a file, the file system can determine consistency by checking the forward pointer and whether it points to a block referring back to it
    - If yes, good
    - If no, inconsistent
  - Lazy crash consistency
- Reduce the number of times a journal protocol has to wait for disk writes to complete, called **optimistic crash consistency**
  - Issues as many writes to disk as possible
  - Uses the **transaction checkpoint** to detect inconsistencies

Arpaci-Dusseau Ch. 43 - Log Structured File Systems

- New file system: **log structured file system**
- Based on some observations
  - System memories are growing, the file system performance is determined by write performance
  - Large gap exists between random I/O performance and sequential I/O performance
  - Existing file systems perform poorly on many workloads
  - File systems are not RAID-aware
- Ideal file system should:
  - Focus on write performance
  - Make use of sequential bandwidth
  - Perform well on common workloads that write out to data and update on-disk metadata
  - Work well on both RAIDs and single disks
- **LFS (Log-Structured File System)** introduced as a result
  - Buffers all updates in an in memory segment when writing
  - When segment is full → it is written to disk in one **long, sequential transfer** to an unused part of disk
  - Never overwrites existing data, **always writes to free locations**
  - Segments are large so **disk or RAID is used efficiently**

43.1 Writing to Disk Sequentially

- Challenge: transforming all updates to file system to a series of writes to disk
- Writing data block D to a file
  - Data gets written to the disk along with the metadata that needs to be updated
  - Inode is written and points to data block D



- The basic idea: writing all updates to the disk

43.2 Writing Sequentially and Effectively

- Writing sequentially is not enough to ensure efficient writes
- Disk may rotate between writes → we need to **wait for disk to rotate before writing**
  - Need to issue contiguous writes to achieve good performance

- LFS uses **write buffering**
    - Before writing, LFS keeps track of updates in memory
    - At the sufficient number of updates, it writes **all to disk at once**
        - This large chunk of updates is a **segment**
- LFS buffers a segment in memory and then writes it to disk
- If segments are large enough, the writes are efficient

43.3 How Much To Buffer?

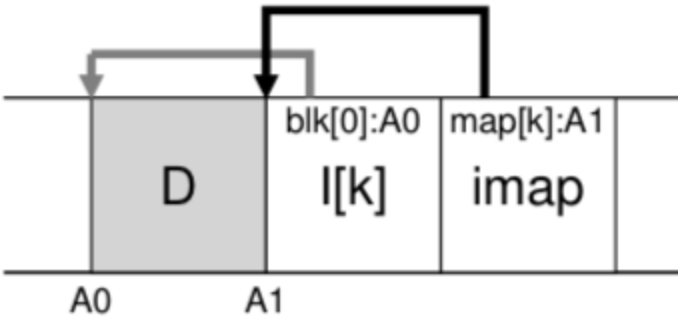
- Depends on the disk itself
- The positioning overhead vs. the transfer rate

43.4 Problem: Finding Inodes

- Review of finding an inode in a typical system:
    - Organized into arrays
    - Placed on disk at fixed locations (in inode block)
    - Need to know size and start of chunk into which inode tables are split
- LFS is more complicated
    - The inodes are **scattered all over the disk**
    - We never overwrite them, the **latest version keeps moving**

43.5 Solution Through Indirection: The Inode Map

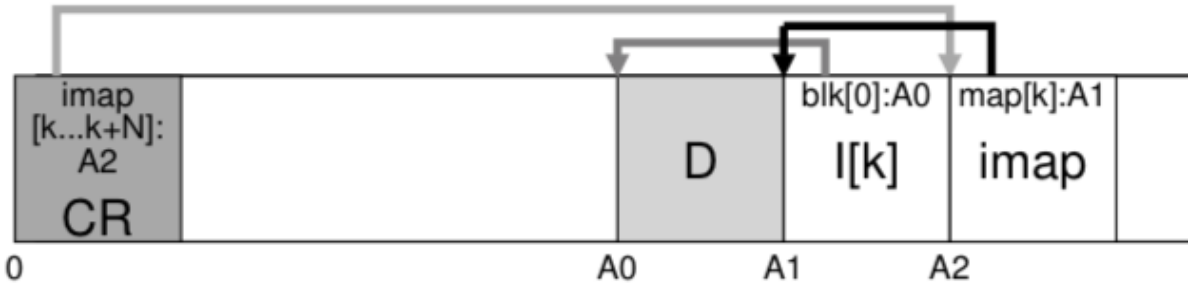
- LFS designers introduced a **level of indirection** between inode numbers and inodes using the **inode map (imap)**
- **Imap:** structure that takes an inode number as an input and produces a disk address of the most recent version of that inode
- Implemented as a simple array
- When inode is written to disk, the imap entry is updated
- But this needs to be persistent → where should it be on disk?
    - Could be on a fixed part
        - Requires updates to structures
        - Incurs writes to the map which is bad for performance
    - Place chunks of inode map next to where other new information is written
        - When appending a block to a file, LFS writes new data, inode, and a piece of the inode map



43.6 Completing the Solution: The Checkpoint Region

- How do we find the inode map?
    - File system has fixed and known location to **begin file lookup**

- **Checkpoint region (CR)**
    - Contains a pointer to the latest piece of the inode map
    - Inode map is found by reading the CR irst
    - Only updated periodically so performance is not affected
- On-disk layout has:
    - Checkpoint region
    - Inode map pieces
    - Inodes point to files



43.7 Reading A File From Disk: A Recap

- What happens when we read a file from disk
- Assume that there is nothing in memory
    1. **Read the checkpoint region**
        - LFS reads the inode map and caches it into memory
    2. **When inode number is given**
        - Look up the number in the imap and read the latest version
    3. **Read blocks from a file**
        - Same as Unix systems
        - Using direct/indirect pointers
        - In the common case, should be the same number of I/Os as typical file system
        - The imap is caches so we only need to look it up in the imap

43.8 What About Directories?

- Directory structure is basically identical to the classic Unix structure
- When creating a file, LFS writes new inode data, data, and directory data and inode
- Inode map contains information about location of new file and new directory
- To access the file, look up in the inode map to find inode of that directory
    - Then read the directory data
    - Gives you the name-to-inode mapping of a file
    - Consult the inode map again to read the file
- Serious problem in LFS is solved by the inode map, the **recursive update problem**
    - Arises in systems that don't update in place and instead, move them to new locations on disk

- When the inode changes/is updated, the location on disk changes
    - May have entailed an update to directory pointing to a file, which may have mandated an update to the entire directory tree, etc.
    - The problem is avoided with the inode map
- Inode location may change but the change **is not reflected in the directory**
    - The **imap structure** is updated while name to inode number mapping stays the same

43.9 A New Problem: Garbage Collection

- Problem: LFS repeatedly writes latest version of a file to a new location
    - Keeps the writes efficient but leaves old version on disk, accumulating **garbage**
- Two versions of the file: old and new
- Old version can be kept around to restore old versions, a **versioning file system** would keep track of these old versions
- LFS only keeps the latest version
    - Periodically, LFS needs to find dead versions and **clean them**, a process called **garbage collection**
    - Make the blocks free again for use in subsequent writes
- Segments are **integral to effective cleaning**
    - In LFS, cleared blocks would leave free blocks/holes
    - We want large contiguous regions
- Cleaner works on a **segment-by-segment basis**
    - Clean up large chunks
    - Periodically LFS cleaner reads in old segments
        - Determines the free blocks
        - Write out a new segment containing only live blocks
        - Free up the old blocks
- Expect the cleaner to read in M segments, compact them into N segments (N < M), then write the N segments to a new disk location
- Problems:
    - Mechanism: how to tell blocks are live/dead
    - Policy: how often to clean, which segments to pick

45.10 Determining Block Liveliness

- Mechanism
- Determine liveliness by adding extra information to each segment describing each block, including:
    - Inode number
    - Offset
- Recorded in a structure at the head of the segment called the **segment summary block**
- Use this information to determine whether a block is live or dead
- Given block D at address A

        - Look at the segment summary block to find the inode number N and offset T
        - Look in imap to find where N lives and read N from disk (or memory)
        - Use offset T to look at the inode and see where the inode thinks the Tth block of the file is on disk
        - If at disk address A, then block D is live
            - Otherwise, block D is listed, the version is no longer needed
    - pseudocode:

```
(N, T) = SegmentSummary[A];
inode  = Read(imap[N]);
if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```

    - Shortcuts using version number
        - When file is truncated or deleted, LFS increases the version number
        - Records new version number on imap and disk segment

43.11 Policy Question: Which Blocks to Clear and When?

- Determining when the clean is easy
    - Periodically, during idle time
    - When the disk is full
- Determining which blocks to clean is challenging
    - **Hot segment**: contents frequently overwritten
        - Best policy is to **wait** before cleaning it as more and more blocks are overwritten and thus freed
    - **Cold segment:** may have a few dead blocks but the rest of the contents are relatively stable
- Clean cold segments → this approach is not perfect

43.12 Crash Recovery and The Log

- Problem: system may crash while LFS is writing to disk
- Normal operation: LFS buffers writes in a segment
    - Then writes to disk
    - Writes organized in a log
        - The checkpoint region points to a head and tail segment
        - Each segment points to the next segment
    - Checkpoint region is periodically updated
    - Crashes during write to segment, write to CR
- **Crash during a write to the CR**
    - Ensure that CR write is atomic
        - LFS has two CRs, one at either end of the disk and writes to them alternately

        - LFS implements careful protocol when updating the CR with latest pointers to the inode map
            - First writes out to a header with the timestamp
            - Writes the body of the CR
            - And last block with a timestamp
        - Choose the most recent CR with consistent timestamps
            - Achieves consistent updates
    - **Crash during a write to a segment**
        - LFS writes to the CR at around 30 second intervals
        - Last consistent snapshot may be quite old
        - Reboot: LRS can recover by reading in the checkpoint region, imap pieces, and subsequent files and directories
            - But updates from last many seconds can be lost
        - To improve, LFS rebuilds segments through **roll forward technique**
        - Basic idea: start with last checkpointed region, find end of the log, and use that to read through next segments, checking the log for valid updates
        - If yes, LFS update the file system, recovering much of the data and metadata

Arpaci-Dusseau Ch. 44 - Flash-based SSDs

- Solid state devices (SSDs): no mechanical or moving parts, build out of transistors
- Unlike typical DRAM/random-access-memory, a **solid-state storage device (SSD)** retains information despite power loss
    - It is an ideal candidate for use in persistent data storage
- Technology it uses is **flash** (NAND-based flash)
- Unique property of flash:
    - To write a chunk of it (a **flash page**), you need to erase a bigger chunk (a **flash block**) which is quite expensive
    - Writing too often to a page will wear it out

44.1 Storing a Single Bit

- Flash chips: store 1 or more bits in a single transistor
    - Level of charge trapped within is mapped to a binary value
- **Single-level cell (SLC):** only a single bit is stored within a transistor (e.g. 1 or 0)
- **Multi-level cell (MLC):** two bits are encoded into different levels of charge (00-low, 01-somewhat low, 10-somewhat high, 11-high levels)
- **Triple-level cell (TLC):** three bits per cell
- SLC chips achieve higher performance and are more expensive

44.2 From Bits to Banks/Planes

- Flash chips are organized into **banks** or **planes**

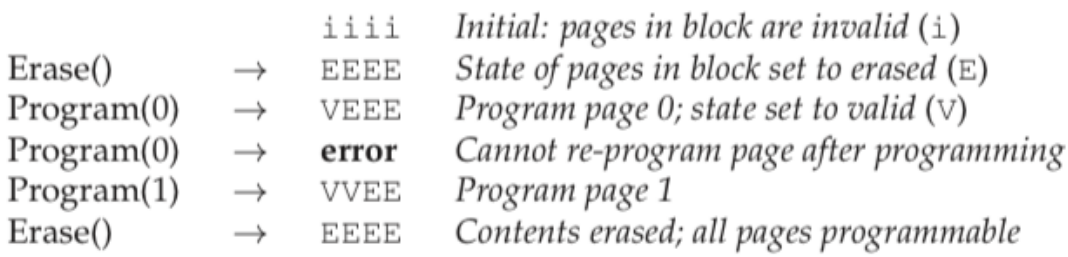
- Banks accessed in 2 different size units
    - **Blocks** (AKA erase blocks) – usually 128 KB or 256 KB
    - **Pages** – usually a few KB
- Each bank has a large number of blocks within
    - Within each block, there are a large number of pages

- This is a simple flash chips
- Distinguishing blocks and pages is critical for flash operations and the performance of the device
- To write a page within a block:
    - Have to **erase the entire block**
    - This detail makes building a flash-based SSD difficult

44.3 Basic Flash Operations

- Three low-level operations supported by the flash chip
    - **Read:** used to read a page from flash
    - **Erase and program:** used in tandem to write to a page
- Details:
    - **Read (a page)**: read by specifying read command and appropriate page number
        - Usually fast, regardless of location
        - The ability to access **any location** uniformly quickly makes this a **random access device**
    - **Erase (a block):** before writing a page to flash, need to **erase the entire block the page lies within**
        - Destroys the contents of the block by setting each bit value to 1
        - **We need to make sure that all data is copied elsewhere before executing the erase**
        - This is an expensive command
        - Once finished, the **block is reset and ready to be programmed**
    - **Program (a page):** after the block has been erased, the program command is used to change some values of 1s to 0s
        - Write the contents of a page
        - Less expensive than erasing
- Each page has a state
    - **Starts in an INVALID state**
    - **Erase the block that the page resides in will set the state to ERASED**
    - ERASED – resets the contents of each page in the block

        - Makes them programmable
    - **Programming a page sets its state to VALID**
        - Contents have been set and now can be read
- States are not affected by reads but you should read from a VALID page only



→ Summary

- Reading a page is easy, simply read
    - Flash performs well
    - Exceeds the random read performance of the disk drive
- Writing a page is tricky
    - The entire block must be erased
        - Need to move important data to another location
    - Desired page can then be programmed
    - This is expensive
    - Repetition can lead to reliability problem that flash chips face, **wear out**

44.4 Flash Performance and Reliability

- Read latency is good
- Program latency is higher and more variable
- Erases are expensive
    - Need to deal with this cost
- Reliability of the flash chip:
    - Mechanical disks fail easily, the flash is silicon so there are fewer mechanical reliability issues
    - Primary concern is **wear out**
        - When block is erased and programmed, it slowly accumulates extra charge
        - Extra charge buildup makes it difficult eventually to distinguish between a 0 and a 1
        - When it is impossible, the block can no longer be used
        - The lifetime is not currently well-known
    - Disturbance is another problem
        - When accessing a page, some bits may be flipped in neighboring pages
        - Known as **read disturbs** or **program disturbs**

44.5 From Raw Flash to Flash-Based SSDs

- Turning flash chips to storage devices

- **Standard storage interface:** block-based one where blocks can be read/written given a block address
- Flash-based SSDs need to provide the standard block interface on top of raw chips
- SSD internally: contains flash chips (for persistent storage), some volatile (non-persistent) memory (e.g. SRAM), and control logic
    - The volatile memory is used for caching and buffering of data or mapping tables
    - The control logic is used to orchestrate device operations
- **Control logic** satisfies client reads/writes by turning them into **internal flash operations**
    - Provided by the **flash translation layer (FTL)**
- **FTL:** takes read/write requests on logical blocks and turns them into low-level read/erase/program commands on physical blocks and pages
    - Should deliver high reliability and performance
- To deliver performance
    - Use multiple flash chips in parallel
    - Reduce **write amplification** – the total write traffic in bytes issues to flash chips by the FTL divided by the total write traffic (in bytes) issues by the client to the SSD
- High reliability concerns
    - Wear out – blocks being erased and reprogrammed too often become unusable
        - FTL should try and **spread out writes across blocks** as evenly as possible so all blocks will wear out roughly at the same time, a process called **wear leveling**
    - Disturbances
        - Minimize by programming pages in an erased block in order from LOW to HIGH
        - Sequential programming approach minimizes disturbance

44.6 FTL Organization: A Bad Approach

- Simplest organization of the FTL is **direct mapped**
    - Read to logical page N is directly mapped to physical page N
    - Write to logical page N is complicated
        - FTL reads in the entire block page N is in
        - Erases the block
        - Then FTL programs the old pages as well as the new one
- Problems of reliability and performance
- Performance of each write is problematic
    - Device has to read in the **entire block** (costly), erase it (more costly) and program it (still costly
        - Severe write amplification
        - Terrible write performance, **slower than typical disk drives**

- Poor reliability:
    - File system metadata and user file data is **repeatedly overwritten** which leads to quicker wear out
    - **Too much control** over the wear out to client workload
        - Workload needs to be spread out evenly across logical blocks

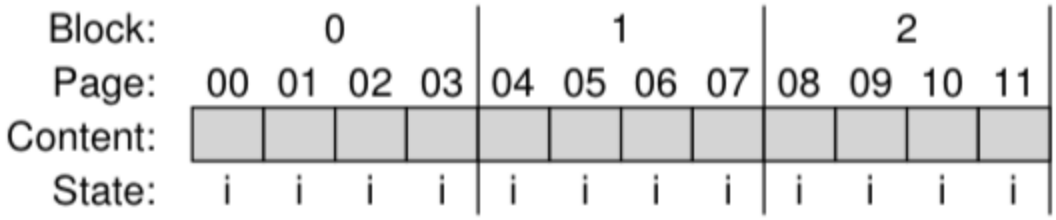
44.7 A Log-Structured FTL

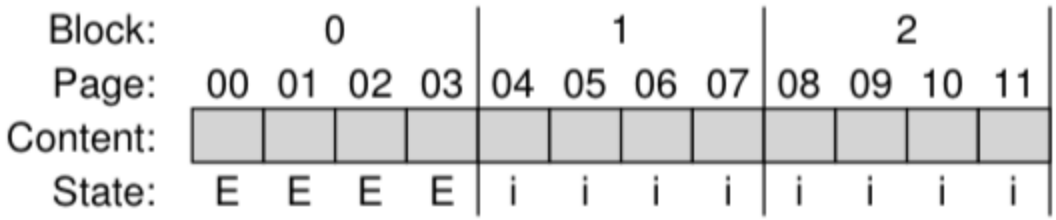
- Most FTLs are **log structured**: used in storage devices and in file systems
- Writes to logical block N means the device **appends the write to the next page spot in currently-being-written-to block**
    - This writing style is called **logging**
- Allow for reads later by keeping a **mapping table** in memory and on the device (persistently)
    - Stores the physical addresses of logical blocks in the system
- To the client, the device looks like a typical disk in which it can read/write 512-byte sections (or groups)
- Following sequence of operations:

    - Write(100) with contents a1
    - Write(101) with contents a2
    - Write(2000) with contents b1
    - Write(2001) with contents b2

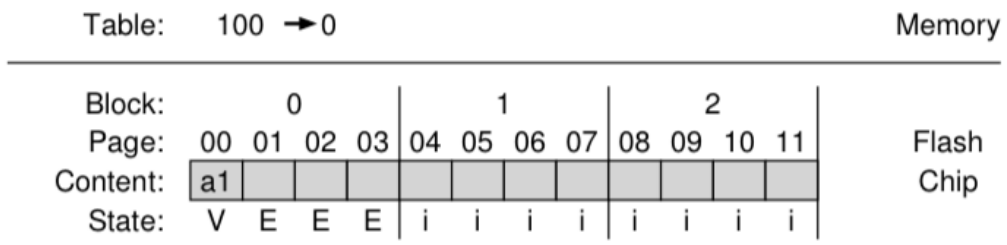
- Writes need to be transformed into erase and program instructions
- The addresses are **logical block addresses** used by clients to remember where the information is locate
- Need to record the **physical pages** of the SSD for each logical block address

1. Erase the blocks

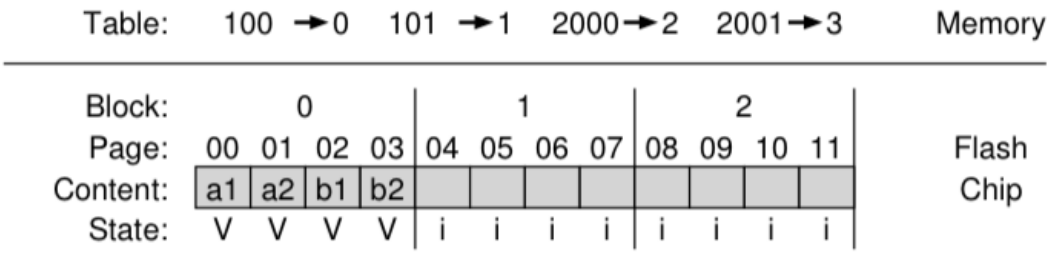




2. Program block 0
    - SSD writes pages in order from high to low to reduce **program disturbance**
    - Logical block 100 maps to physical page 00



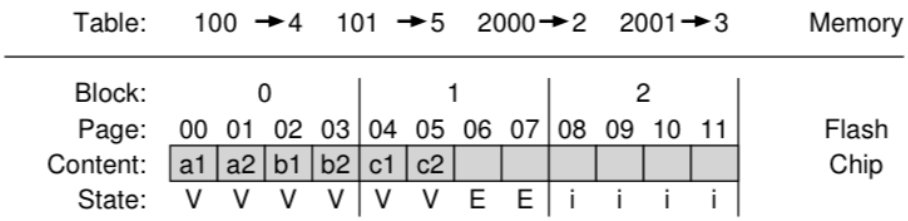
  - The in memory mapping table is used to accommodate the read
    - Read needs to map logical block 100 to physical page 0
- The client, when writing to the SSD:
  - SSD finds a location, usually the next free page
  - Programs the page with block contents
  - Records the logical-to-physical mapping in mapping table
  - Subsequent reads use the table to **translate** the logical block address to the physical page
- The final result:



- Approach improves performance
  - You only erase once in a while
  - **Read-write-modify is avoided**
  - Spreads the writes across pages → **wear leveling**
- Downsides:
  - Overwrites of logical blocks leads to **garbage**, the old versions of data that are left on the device will take up a page
  - Device needs to periodically perform **garbage collection (GC)** to find and free these blocks
    - Excessive GC drives up write amplification, lowering performance
  - Higher cost of in-memory mapping tables → larger device means more tables

44.8 Garbage Collection

- Garbage is created in a log-structured approach
  - Need to collect the garbage
  - E.g. dead-block reclamation
- Say we need to rewrite to blocks 100 and 101



- Now a1 and a2 are VALID but have garbage in them
- Log-structured nature causes overwrites to create garbage blocks
    - Device must reclaim these blocks
- **Garbage collection:** process of finding garbage blocks and reclaiming them for future use
- Basic process:
    - Find a block that contains one or more garbage pages
    - Read in the live pages from that block
    - Write out these live pages
    - And reclaim the entire block for use in writing
- Garbage collection can be **expensive**, it requires reading and writing of live pages
- Ideal candidates for reclamation are blocks containing **only dead pages**
    - Block can immediately be erased and used for new data
- Reduce GC cost by **overpromising** the device
    - Add extra flash capacity
    - Cleaning can be delayed until the device is less busy
    - More capacity will increase the internal bandwidth

44.9 Mapping Table Size

- Extremely large mapping table - one for each 4KB page in the device
- The **page-level FTL** scheme is **impractical**

→ Block Based Mapping

- Reduce mapping cost by keeping a **pointer per block** of the device instead of per-page
    - Reduces the amount of mapping information by a factor of S^{block}/S^{page}
- **Block-level FTL** is akin to sharing bigger page sizes in virtual memory
    - Fewer bits are used for the VPN and there is a larger offset
- Block-based mapping in a log-based FTl is not good for performance
- Big problem: the **“small write” problem** (writes < size of a physical block)
- NOW: logical block address has two portions: a **chunk number** and **an offset** (2 bits)
    - Logical address space is chopped into chunks the size of physical blocks
    - Chunks are mapped to physical blocks
- Reading is easy
    - FTL extracts the chunk number from the logical address taking topmost bits of the address

        - FTL looks up the chunk number to physical-page mapping in the table
        - FTL computes the address of the desired flash page by adding the offset from the logical address
    - Client writes to a page within the block
        - The entire block needs to be written out to a new location
        - Calls for updating the mapping table
    - An improvement needed because small writes will cause problems

→ Hybrid Mapping

- **Hybrid mapping** technique used to enable flexible writing and reduce mapping costs
    - FTL keeps a few blocks erased and directs all writes to them, they are **log blocks**
    - Keep a per-page mapping for log blocks
        - Allows the FTL to write any page to any location in the log block without copying
- Two types of mapping tables in memory
    - Small set of per-page mappings in the **log table**
    - Larger set of per-block mappings in the **data table**
- When looking for a logical block, the FTL consults the log table
    - If the logical block is not there, the FTL consults the data table to find and access the location of the block
- Key: keeping number of **log blocks small**
    - FTL needs to periodically examine log blocks and **switch them** into blocks that can be pointed to by a single block pointer
- Switch: 3 main techniques, based on the block contents
- **Switch merge:** all per-page pointers required replaced by a single block pointer
    - Client rewrites all pages in a block in the same order
    - Simply need to switch the data table pointer
- **Partial merge:** some pages are not overwritten, these pages are read in and appended to the log
    - Same result as switch merge but there is extra I/O
- **Full merge:** FTL must pull pages from many other blocks
    - Frequent full merges hurt performance

44.10 Wear Leveling

- **Wear leveling:** must be implemented in the background
- The basic idea: multiple erase/program may wear out the flash blocks
    - FTL should try and spread work across all blocks of the device evenly
    - All blocks wear out roughly at the same rate
- Log structures usually/initially does a good job
    - Eventually a block is filled with long-lived data that isn't overwritten
    - It is never reclaimed

- **To remedy this problem: FTL periodically reads all live data out of blocks and rewrites it elsewhere**
    - Increase amplification → this decreases performance

44.11 SSD Performance and Cost

- Examine SSD & HDD

→ Performance

- SSD - non-mechanical components
    - More similar to DRAM
- Biggest difference is in random reads/writes
    - Typical disk drive: only a few hundred random I/O per second
    - **SSD: performs much better**
- SSD random read is not so good as **random write**
    - Good because of the log-structured design that transforms random writes to sequential writes
- SSD sequential performance is similar
- Random I/O performance in general, the SSD is much better

→ Cost

- SSD: 60 cents/GB
- HDD: 5 cents/GB
- This difference dictates the design of large scale storage systems
- If performance is the main concern, use the SSD
- If you need to store massive amounts of data, use the HDD

Arpaci-Dusseau Ch. 45 - Data Integrity & Protection

- Ensuring that data is safe - data integrity or data protection
- Techniques to ensure that data is put into the system remains the same when you ask the system for it later

45.1 Disk Failure Modes

- Disks can fail
- Early RAID systems failure model: either entire disk is working or it fails completely → deletion is straightforward
    - **Fail-stop method** makes building RAID simple
- More failure modes → specifically, sometimes disks have issues accessing one or two blocks
- Two important single-block failures: **latent-sector errors (LSE)** and **block corruption**
- **Latent sector errors:** arise when a disk has been damaged in some way
    - EX: if a disk head touches the surface, a **head crash**, may damage the surface
    - EX:cosmic rays can flip bits → lead to incorrect contents
    - In-disk **error correcting codes (ECC)** are used by the drive to determined whether on-disk bits are good and how to fix them in some cases

    - If they are not good and the drive cannot fix it, the disk returns an **error on read**
- **Block corruption:** disk block becomes corrupt in a way that disk cannot detect
    - EX: buggy disk firmware may write to the incorrect location
        - ECC will indicate that this is fine but client may get the wrong block later
    - EX: block is corrupted when being transferred from the host to the disk by a faulty bus
        - Called a **silent fault**
            - Disk gives no indication of any problem when returning faulty data
- Modern view of disk drive failure – the **fail-partial** disk failure model
    - Disks can still fail in their entirety
    - Disks can also appear to be working and have 1 or more blocks be inaccessible (LSEs) or hold incorrect contents (corruption)
- When accessing a seemingly working disk:
    - **Non-silent partial fault:** error when reading/writing a block
    - **Silent partial fault:** return the wrong data
- These are both somewhat rare
- LSE findings:
    - Costly drives with more than 1 LSE are as likely as cheaper drives to have more error
    - Annual error rate increases in the second year
    - Number of LSEs increases as disk sizes increase
    - Most disks with LSEs have less than 50
    - Disks with LSEs are more likely to develop more LSEs
    - Significant amount of spatial and temporal locality
    - Disk scrubbing is useful for finding LSEs
- Corruption findings:
    - Chance of corruption varies across different models
    - The age affects are different across models
    - Workload and disk size have little effect/impact
    - Disks with corruption only have a few
    - Not independent within a disk or across disks in RAID
    - Spatial locality, some temporal locality
    - Weak correlation with LSEs
- Reliable systems need to **detect and recover** from LSEs and block corruption

45.2 Handing Latent Sector Errors

- Straightforward to handle because they are easily detected
- When a storage system tries to access a block and gets an error
    - It should use whatever redundancy mechanism it has to return the right data
- Precedence of LSEs has influenced RAID designs

- In RAID-4⁄5 systems, when both full-disk faults and LSEs occur in tandem
  - When an entire disk fails, RAID tries to reconstruct the disk onto a hot spare by reading through all disks in the parity group and reconstructing the values
  - If an LSE is encountered, the reconstruction cannot be completed
- Combat this with an extra degree of redundancy
  - Ex: NetApp's RAID-DP has 2 parity disks
    - Comes with cost of maintaining 2 parity disks
    - Log structured WAFL system mitigates the cost, with an extra cost being space

45.3 Detecting Corruption: The Checksum

- Detection of corruption is a key problem
  - Recovery is the same as before
- Focus on detection techniques
- Primary mechanism used is the **checksum**: the result of a function that takes a chunk of data as input and computes a function over said data, producing a small summary of the results
- **GOAL:** enable a system to detect if data has somehow been corrupted or altered by storing the checksum with the data and confirming upon a later access that the current checksum matches the original storage value

→ Common Checksum Functions

- Different functions that vary in strength and speed
  - Trade Off: more protection will cost more
- XOR (exclusive or) function
  - Checksum computed by XOR-ing each chunk of the data block being checksummed
  - Produces a single value representing the entire block
  - Limitations:
    - If two bits in the same position within each checksummed unit change, it is not detected
- Addition Function
  - Fast
  - Two's complement addition over each chunk of data, ignoring overflow
  - Detects changes but doesn't work if data is shifted
- **Fletcher Checksum** - more complex
  - Computation is simple, involves computation of two check bytes S^1 and S^2
  - Block D contains bytes d1 → dn
    - $S^1 = (S^1 + di) \bmod 255$
    - $S^2 = (S^2 + S^2) \bmod 255$
  - Strong, detects all single-bit, double-bit, and burst errors
- **Cyclic redundancy check (CRC)**
  - Checksum over D which is treated as a large binary number
  - Divide it by an agreed upon value K

        - Remainder of the division is the value of the CRC
        - This is efficient
    - There is no perfect checksum
    - Two data blocks with non-identical contents can have identical checksums, a situation called a **collision**

→ Checksum Layout

- Using a checksum
- Where should they be stored?
- The basic approach: store a checksum with each disk sector or block

- Checksums are usually small and disks can only write in sector-sized chunks
    - Format the disk with 520 byte sectors
        - Give extra 8 byte per sector for the checksum
- Disks without this functionality must pack the checksum into 512 byte blocks

    - N checksums are stored together followed by n blocks, this repeats
    - This works on all disks but is less efficient

45.4 Using Checksums

- When reading block D, client also reads the checksum from disk Cs(D), the **stored checksum**
- The client then **computes** the checksum over the retrieved block D, called the **computed checksum**, Cc(D)
- Client compares the two checksums, if Cs(D) == Cc(D), the data has likely not been corrupted
- Else, the data has changed since it was last stored and the checksum has detected a corruption
- What do we do about corruption?
    - If a system has a redundant copy, use it instead
    - If there is no copy, return an error

45.5 A New Problem: Misdirected Writes

- Unusual failure modes sometimes
- **Misdirected writes:** occurs in disk and RAID controllers that write to disk correctly but **in the wrong location**
    - In a single-disk, write to address y instead of x

        - In multi-disk situation, write to disk j instead of disk i
    - Handling misdirected writes
        - Add more information to the checksum like a **physical identifier (physical ID)**
    - C(D) can store the disk and sector number on the block
    - Result is **redundancy** which is the key to error detection
        - Each block has a repeated disk number
        - Offset is kept next to the block itself

45.6 One Last Problem: Lost Writes

- **Lost write:** when device informs the upper layer that a write has completed but in fact is never persisted
    - The old contents of the block remain
- Old checksumming strategies will not detect this
    - Old block likely has its matching checksum and physical ID
- Classic solution: **write verify** or **read-after-write**
    - Immediately read back the data written to ensure that the data has reached the disk
    - Slow, doubles the number of I/Os needed
- Some systems add a checksum elsewhere to detect lost writes
    - Checksum in each inode and indirect block for every block in the file
    - Even if the write to data block is lost, the checksum in the inode won't match the old data
    - Only if writes to inode and data are both lost will this scheme fail

45.7 Scrubbing

- When do checksums get checked
    - Sometimes when data is accessed
    - But some data is rarely accessed
- **Disk scrubbing** can be used
    - Periodically read through every block and checking the validity of the checksums
    - Reduces the chances of corruption
- Done on a nightly or weekly basis typically

48.5 Overheads of Checksumming

- Space and time overhead
- Space overhead:
    1. Disk-checksums **take up space on disk** that could have been used for user data
    2. **Memory of the system:** accessing data now requires room for checksums as well as data in memory
        - Not noticable if system checks once and discards it
        - Noticable if checksums are stored in memory
- **Time overhead: (more noticable)**

    - Minimally, the CPU must compute the checksum when data is stored or accessed
    - Reduce the overhead by combining data copying and checksumming into one streamlined activity
- I/O overhead
    - When the checksum is stored distinctly from data
        - Requires extra I/O to access
        - Reducible with system design
    - Scrubbing
        - Can be tuned to limit impact

OS Security

Lecture 15: Security and Privacy

Outline

- Introduction
- Authentication
- Access control
- Cryptography

Introduction

- OS provides the **lowest layer** of software visible to user
  - Everyone relies on the OS
- OS's are often **close to hardware**
  - Often have complete hardware access
- If the OS isn't protected, the machine is not protected
- Flaws in the OS can **compromise security** at higher levels
  - OS attacks hurt all other applications since the attacker has full control

Why is OS Security So Important?

- OS controls:
  - Access to application memory
  - Scheduling of processes
  - Ensures that users receive resources they ask for
- If OS isn't doing these things securely, **anything can go wrong**
- All other security systems must **assume a secure OS** at the bottom

Some Important Definitions

- Security
- Protection
- Vulnerabilities
- Exploits
- Trust
- Authentication and authorization

Security and Protection

- **Security** is a policy
  - Describes what you would like to happen
- **Protection** is a mechanism
  - Implements security policies
- Protection mechanisms implement security policies

Vulnerabilities and Exploits

- A **vulnerability** is a weakness that can allow an attacker to cause problems
  - Not all vulnerabilities can cause all problems

        - Most vulnerabilities are **never exploited**
    - An **exploit** is an actual incident of taking **advantage of a vulnerability**
        - Using the vulnerability to do something bad
        - Term also refers to the **code or methodology** used to take advantage of a vulnerability

Trust

- Extremely important security concern
- Do certain things for those you trust
- And don't do them for those you don't
- Isn't that simple
    - How to express trust
    - Why do you trust
    - How can you be sure who you're dealing with
    - What if the trust is situational
    - Trust may change
- We need to embed these qualities into code/binary algorithms

Trust and the Operating System

- Have to trust your operating system
- It controls everything
- Compromising your OS is a big deal

Authentication and Authorization

- We need to know **who wants to do something**
    - Trusted parties are allowed
    - Deny other parties
- When deciding whether or not we give access, it depends a lot on **WHO is asking**
- We need to know **who's asking**
    - Need an algorithm to determine identity
    - **Authentication**: figuring out who someone is
- Then need to check **if that party should be allowed to do it**
    - Determining this is **authorization**
    - Performed more often than authentication
    - Authorization usually requires authentication

Authentication

- Security policies allos **some parties** to do something but not others
- Implies we need to know **who's doing the asking**
- Determination will be made by a computer
- How?
    - Needs to be embedded in an algorithm
    - Based on evidence a computer can recognize and gather

Real World Authentication

- Recognition - who you are
- Credentials - what you have
- Knowledge - what you know

- Location - where you are
- All have cyber analogs

Authentication with a Computer

- Not as smart as a human
  - Steps must be **well-defined**
- Can't do certain things well
- But fast on computations and less prone to error
  - **Mathematical methods** used for authentication
- Often must authenticate **non-human entities**
  - Processes, other machines

Identities in Operating Systems

- Usually rely **primarily on a user ID**
  - Identifying based on a bit pattern
  - Uniquely identifies **some user**
  - Processes run **on his behalf** and thus, **inherit his ID**
  - Many processes have **human identities associated**
- Computers each have an independent set of user identity bit patterns
- Implies a model where **any process belonging to a user has all his privileges**
  - Has drawbacks
  - Every single process created has **identical set of privileges**
    - BAD IDEA with buffer overflow
    - Processes can be taken over by somebody else

Bootstrapping OS Authentication

- Processes **inherit user IDs**
- One ID/process
- Still need to create the **initial process and change its ID** from the kernel ID
  - Have a **special process** that does the login
  - Creates a process (login shell) for the user
  - Changes the ID of the processes
  - **Changing the ID of a process is a privileged instruction**
- Need to **create a process** belonging to a new user
  - Upon login usually
- How do we **tell who this newly arrived user is**

Passwords

- Authenticate the user by **what he knows**
  - Secret word he supplies to the system on login
- System must be able to **check that the password was correct**
  - **Storing it**
  - Or **storing a hash of it** (better idea)
- If correct, we can **tie the user ID to a new command shell** or a new window management process

Problems with Passwords

- Have to be **unguessable** yet easy to remember

- If you can guess them, it is **worthless**
- If networks connect **remote devices to computers,** they are susceptible to **password sniffers**
    - Password may be checked elsewhere
    - Must travel across the network which may not be secure
    - Programs which read data from the network and extract passwords when they see them are **password sniffers**
- Unless they are quite long, **brute force attacks often work**
    - If a hash file is obtained, the attacker can **hash all possible combinations and find the password**
    - May still be OK if you **salt your hashes**
- Widely regarded as **outdated technology**

Proper Use of Passwords

- Sufficiently long
- *Contain non-alphabetic characters*
- Unguessable
- *Changed often*
- Never be written down
- Never be shared
- Hard to achieve all of these things simultaneously

Challenge/Response Systems

- Authentication by what questions you can answer correctly
    - What you **know**
- System asks user to provide information
- If provided correctly, user is authenticated
- Safe if it is a different question each time
    - This is **not practical**
    - System needs to know a lot about you
- Also a bad idea
    - Things can be looked up
    - May not have many options to ask

Hardware-Based Challenge/Response

- Challenge is sent **to a hardware device** belonging to the appropriate user
    - Authentication based on **what you have**
- Sometimes **mere possession** of the device is enough
- Sometimes the device performs a **secret function** on the challenge

Problems with Challenge/Response

- If based on what you know, **usually too few unique and secret** challenge/response pairs
- If based on what you have, it **fails if you don’t have it**
    - Whoever does have it may try and pose as you
- Some forms are susceptible to **network sniffing**

Biometric Authentication

- Authentication based on **what you are**
- Measure some **physical attribute** of the user
    - Fingerprints, voice patterns, retinal patterns
- Convert measurements to a **binary representation**
- Check that representation **against a stored value** for that attribute
- Close match → authenticate user

Problems with Biometric Authentication

- Requires **very special hardware**
    - Some minor exceptions
    - Servers won't have it usually
    - Servers can't use biometric authentication
- Physical characteristics may **vary too much** for practical use
- Generally **not helpful** for authenticating programs or roles
    - Don't have biometrics
- Require **special care** when done across a network
    - People can still sniff

Errors in Biometric Authentication

- **False positives**
    - Biometric system was **too generous** in making matches
- **False negatives**
    - Biometric system was **too strict/picky** in making matches
- Need to have a **tradeoff**, system is usually not perfect

Biometrics and Remote Authentication

- Biometric reading is just a bit pattern
- If the attacker obtains a copy, can send the pattern over the network
    - Without actually performing a biometric reading
- Requires **high confidence in security of path** between biometric reader and the checking device
    - Usually OK when both are on the same machine
    - Problematic when the Internet is between them

Multi-Factor Authentication

- Rely on **two separate authentication methods**
- If well done, each method will **compensate for other's drawbacks**
- The **current preferred approach** in authentication

Access Control in Operating Systems

- OS can control **which processes access which resources**
    - OS can tie process to an identity
        - Only one with access
        - Can lookup the identity easily
- Giving it the chance to enforce security policies
    - Done with an **access control process**
- Mechanisms used to **enforce policies** on who can access are called **access control**

- Fundamental to OS security

Access Control Lists

- **ACLs - access control list**
- For each protected object, maintain a **single list**
  - **Managed by the OS**, to prevent improper alteration
  - Contains information about who can access and how they can access
- Each list entry specifies **who can access the object**
  - And the **allowable modes of access**
- When something requests access
  - **Check the access control list**

An Example Use of ACLs: the Unix File System

- ACL-based method for protecting files developed in the 1970s
- Still in **very wide use today**
  - Relatively few modifications
- Per-file ACLs (file are **objects**)
- Three subjects on list for each file
  - **Owner, group, other**
- And **three modes** for each entry
  - Read, write, execute
  - Have special meanings sometimes
    - Execute entry for directory means something else
- Part of the file's metadata
  - In the **inode** which is of a fixed size
  - Limited space for an ACL
  - In Unix, it is **9 bits**

Pros and Cons of ACLs

- Pros:
  - Easy to figure out who can access
  - Easy to revoke or change access
- Cons:
  - Hard to figure out what a subject can access
  - Changing access rights requires getting to the object

Capabilities

- A **data structure**
  - If you have this data structure, **you have permission to do something**
- Each entity keeps a **set of data items** that specify his allowable accesses
- To access an object, **present proper capabilities**
- Possession of the capability for an object **implies that access is allowed**
  - No check for ID, etc.

Properties of Capabilities

- Capabilities are just a **data structure,** a collection of bits
- Possessing the capability **grants access**
- How do we **ensure unforgeability** for a collection of bits

- One solution:
    - Don’t let the user/process have them
    - Store them in the OS
    - OS keeps track of capabilities
    - Good capabilities are ones under OS control

Pros and Cons of Capabilities

- PROS:
    - Easy to determine **what objects a subject can access**
        - Look at the capabilities for that object
    - Potentially can be **faster than ACLs** in some circumstances
        - ACL is stored on data so reading this can be slow
        - Capabilities are associated with the PCB so they are sitting in memory
    - Easy model for **transfer of privileges**
- CONS:
    - Hard to determine **who can access an object**
        - Need to look at all of the objects and check
        - Need to find **all capabilities in the system**
    - Requires an extra mechanism to **allow revocation**
        - Trivial in the ACL
        - Here, we have to find the capability and take it away
            - Simple in a single machine
            - Challenging in a distributed system
    - In network environments, need **cryptographic methods** to prevent forgeries
        - Capabilities are sent across the network
        - Network contains the capability bits

OS Use of Access Control

- OSes often use both ACLS and capabilities
    - For the same resource
- EX: Unix/Linux uses ACLS for file opens
    - File descriptor with a particular set of access writes
    - The **descriptor** is a capability
    - It is **hidden in the PCB**

Enforcing Access in an OS

- Protected resources must be **inaccessible**
    - Hardware protection must be used to ensure this
    - Only the OS can make them accessible
- Need to go through the OS for protected resources
    - Rely on hardware protection
    - Because only the OS can directly connect with the hardware
- To get access, issue request to OS
    - OS consults access control policy data
- Access may be **granted directly**

        - Resource manager maps resource into a process
        - Allows disk without going to OS like memory-mapped I/O
        - Avoids some waits
        - EX: game to illuminate pixels
    - Access may be granted **indirectly**
        - Resource manager returns a capability to a process
    - System call is needed to get access
        - Incurs overhead in system call
        - Performance cost for protection

Cryptography

- Computer security is about **keeping secrets**
- To do so, make it hard for others to read the secrets
- While making it simple for authorized parties to read them
- What **cryptography is about**
    - Transforming bit patterns in controlled ways to obtain security advantages
    - A **set of algorithms** that allow us to transform data in controlled ways

Cryptography Terminology

- Typically described in terms of **sending a message**
    - But used for many other purposes
- The sender is *S*
    - Encrypts the message
- The receiver is *R*
    - Decrypts the message
    - Should return the message in its original form
- **Encryption** is the process of making message unreadable/unalterable by anyone but R
- **Decryption** is the process of making the encrypted message readable by R
- A system performing these transformations is a **cryptosystem**
    - Rules for transformation are sometimes called a **cipher**

Plaintext and Cipher

- **Plaintext**: the original form of the message, often referred to as P
- **Ciphertext:** the encrypted form of the message, often referred to as C

Cryptographic Keys

- Cryptographic algorithms use a **key** to perform encryption and decryption
    - Referred to as K
- The key is a secret
- Without the key, **decryption is hard**
- With the key, **decryption is easy**
- Everyone knows the cipher but **there is a secret key**
- Everyone uses a different key but the same cipher
- Reduces the **secrecy problem** from your long message to the short key
    - Justifies cryptography

        - Sending a short key versus a long message
    - Some circularity: need to share the secret
        - Need to get the key to places that need the key without someone seeing the key
    - **Distribution of the key is where it falls apart**

More Terminology

- The **encryption algorithm** is referred to as E()
- C = E(K, P)
- The **decryption algorithm** is referred to as D()
- The decryption algorithm **also has a key**
- Combination of the two algorithms is a **cryptosystem**

Symmetric Cryptosystems

- C = E(K,P)
- P = D(K,C)
    - Use the same key, K
- P = D(K,E(K,P))
- E() and D() may not necessarily be the same operation however

Advantages of Symmetric Cryptosystems

- Encryption and authentication are performed in a **single operation**
    - Only party you are authenticating should know this key
    - Partners knows the message and who created the message if only 2 people know the key
    - The key authenticated you
    - How we authenticate across the internet
- Well-known and trusted ones **perform much faster than asymmetric key systems**
- No centralized authority required
    - Key servers may help a lot

Disadvantages of Symmetric Cryptosystems

- Encryption and authentication are performed in a single operation
    - Complicates some signature uses
    - We are using it for a bunch of things
- Non-repudiation is hard without servers
- Key distribution **can be a problem**
- **Scaling** is not good
    - Especially for use on the Internet

Some Popular Symmetric Ciphers

- The **data encryption standard (DES)**
    - Used, due to legacy
    - Weak
- The **advanced encryption standard (AES)**
    - Current US standard
    - Most widely used cipher
- **Blowfish**

- And many others

Symmetric Ciphers and Brute Force Attacks

- Attackers can try and attack using a brute force method
- **Brute force:** try every possible key until one works
- The cost of brute force depends on the **key length**

Asymmetric Cryptosystems

- Often called **public key cryptography (PK)**
- Encryption and decryption **use different keys**
  - K^E – used for encryption
  - K^D – used for decryption
  - $C = E(K^E, P)$
  - $P = D(K^D, C)$
  - $P = D(K^D, E(K^E, P))$
- Often works the other way as well
  - Good algorithms should also work in reverse
  - $C' = E(K^D, P)$
  - $P = D(K^E, C')$
  - $P = D(K^D, E(K^E, P))$

Using Public Key Cryptography

- Keys are **created in pairs**
  - Makes them difficult to create
  - Cannot do it randomly, need to ensure that is a **unique pair for encryption, decryption keys**
- One key is **kept secret by the owner**
- The other is **made public** to the world
  - Hence the name
- If you want to send someone an encrypted message, encrypt it with **his public key**
  - Only he will have the **private key to decrypt**

Authentication with Public Keys

- "Sign" a message by encrypting it with my private key
- Only I know the private key so **nobody else could have created the message**
- Everyone knows my public key so **everyone can check my identity claim directly**
  - Everyone knows that only you could have the private key needed to encrypt a message that can be decrypted by your public key
- Better than symmetric cryptography
  - Only the sender could have created a message
  - Versus where both parties could encrypt the message

Issues with PK Key Distribution

- Security of using public key cryptography depends on **using the right public key**
- Using the wrong one means that key's owner can read the message
- Need high assurance that **a given key belongs to a particular person**

        - How do you tell whose public key is whose?
        - Use a **key distribution infrastructure** or **certificates**
    - Both are problematic

The Nature of PK Algorithms

- Usually based on **some problem in mathematics**
    - Like factoring extremely large numbers
- Security is less dependent on brute force
- More dependent on the **complexity of the underlying problem**
- Implies that choosing key pairs is complex and expensive
    - Need a **large key and a complex algorithm** which is expensive

Example Public Key Ciphers

- RSA
    - Most popular public key algorithm
- Elliptic curve cryptography
    - Tends to have better performance

Security of PK Systems

- Based on **solving the underlying problem**
- In 2009, a 768 bit RSA key was successfully factored
- Keys up to 2048 bits may be insecure
- Size of keys will keep increasing
- The longer the key, the more expensive the encryption and decryption

Combined Use of Symmetric and Asymmetric Cryptography

- Very common to use **both** in a single session
- Symmetric cryptography: cheap but not as effective for authenticating
- Asymmetric cryptography: expensive but useful cryptography
- Asymmetric cryptography essentially can be used to **bootstrap symmetric cryptography**
- Use RSA to authenticate and establish a **session key**
- Use DES or AES with the session key for the rest of the transmission

Reading

Security for Operating Systems

Introduction

- It is particularly important for the OS to be secure
- Everything runs on top of the OS
    - If the software you are running on top of is insecure, then everything is insecure
    - Security flaws in the OS impact other software
- OS has control of underlying hardware
    - If the attacker can control this, it is BAD

- Large complex programs like the OS are harder to make secure
- OS's are designed to support multiple processes simultaneously
    - They are segregated by the OS
    - Need to be careful which processes are run
- OS also needs to provide abstractions, one of which is security behavior
    - Applications **build on security abstractions provided by the OS** to achieve their own security goals
- To secure our operating systems, there are important principles and tools

What Are We Protecting?

- We are protecting **everything** in the case of the OS
- It has complete control of the hardware
- Processes cannot protect themselves from the OS
    - Need to assure that the OS is not malicious
- Security flaws in the OS can hurt everything in the machine

Security Goals and Policies

- Specify the goals we have for security-relevant behavior
- Choose defense approaches to achieve these goals
- Three big security-related goals
    - **Confidentiality**: keep secrets
        - If the information is supposed to be hidden, keep it hidden
    - **Integrity**: don't allow adversaries to change the state of the component or information from what it is supposed to be
    - **Availability:** if the information or service is supposed to be available for use, make sure an attacker cannot prevent its use
- Extra dimension is **controlled sharing**
    - Ensure that only certain people can do certain things
- Also want **non-repudiation**: prevent people from denying something they previously said or did
    - Make it expensive for someone to repudiate their actions
- Need more detailed goals than these
    - Make these goals more specific in your system
- Need to be **very detailed** in specifying our goals for the system
- We also want flexibility in describing goals for our security mechanisms
- OS software enforces these policies so we need to encode them
- Need to turn current security goals to **specific security policies**
- In many caes, the OS already has the mechanisms necessary to implement a security policy but needs someone to tell it what the policy is
    - There are some exceptions
    - Usually the OS supplies general mechanisms that implement many policies
- Still need intelligent design of policies

Designing Secure Systems

- **Certain design principles**
    1. **Economy of mechanism:** keep system as small and simple as possible

2. **Fail-safe defaults:** default to security, not insecurity
3. **Complete mediation:** check if an action to be performed meets security policies every time it is taken
4. **Open design:** assume that adversary knows about your design
5. **Separation of privilege:** require separate parties or credentials to perform critical actions
6. **Least privilege:** give user/process the minimum privileges required to perform action you want to allow
7. **Least common mechanism:** for different users and processes, use separate data structures or mechanisms to handle them
8. **Acceptability:** don't ask too much of users

The Basics of Security

- Some goals are built by the OS and others are defined by users
- **Built-in goals**: extremely common, ensured in order to make more specific goals achievable
    - Usually relate to controlling hardware access
        - Hardware is shared
    - Services that the OS offers
        - Need to be controlled carefully
- System security is related to process handling
    - Need to maintain clean separation of processes
    - OS needs to be careful with allowing access to hardware
- System calls provide protection
    - OS can check that the requested service is allowed
    - Also for device drivers
- During a system call, the OS can determine the identity of the process
- Use **access control mechanism** to decide if the process is authorized

Authentication for Operating Systems

Introduction

- Context is vital in the OS's decision to perform or refuse a service depending on the security goals
    - Important element is **who is asking**
- **Principal:** party asking for something
    - Security-meaningful entities
    - Can request access to resources
- OS services requested by system calls made by user processes
    - Trap from user code into the OS
- OS takes control and performs some service in response
- Calling process has an OS-controlled data structure associated that **describes the process (like the Process Control Block)**
    - OS can check the data structure and identify the process

- Based on this, the OS can choose to perform or deny the request
- **Agent:** the entity performing the request on behalf of the principle is its agent
- **Object** of the access request: resource being requested
- OS determines if the agent process can access the object
    - If yes, OS will remember this decision and keeps track of it in the PCB
- **Credential:** any data created and managed by the OS that keeps track of access decisions for future reference
    - Needs information about the identity of the process to grant request and create credentials
    - EX: user identity, user group identity, process identity
- Need to attach identity to the process, this is a security issue
- Once a principal has been authenticated, the system relies on the authentication for the lifetime of the process

Attaching Identities to Processes

- **One way:the child inherits the parent's identity**
    - All processes will have the same identity
    - If this identity is the boot process identity, this is a problem
- Need to have different identities and use these differences to manage security policies
- Different identities of processes based on which human user created it is set for new processes
- Set identity in the process control block
- OS determines the user identity by having the user log in
- New requirement for the OS → query the identity from human users and verify then

How to Authenticate Users?

- If not an authorized user, they should be rejected
- If it is an authorized user, determine which one
- Authentication of human being has worked ⅓ ways:
    1. Based on what you know
    2. Based on what you have
    3. Based on what you are

Authentication By What You Know

- Performed by using passwords
- **Password:** secret known only to the party to be authenticated
- Party proves identity by divulging the secret
- Effectiveness depends on several factors
    - Other people may know it
    - Other people may be able to guess
- Other people may know the password
    - System knows the password sometimes
        - Password can leak out

        - System shouldn't store the password, instead, store a **hash of the password**
            - Can't reverse hashing algorithms
        - Hashed copies are not the passwords themselves
        - **Cryptographic hashes:** class of hashing algorithms that make it infeasible to figure out what the password is
            - They are hard to design
        - Modern systems use a cryptographic hashing algorithm that has been thoroughly studied and has no known flaws
    - Other people may be able to guess the password
        - The longer the password, the harder it is to guess
        - Expand possibilities from letters of the alphabet to special characters
        - **Dictionary attack:** guessing passwords by trying lists of names and words before random characters
        - Be smart about setting up the system → hackers should now be able to run a dictionary attack remotely
        - Attackers can steal password files, which are the **hashes of passwords**
            - If the hacker has passwords, the hashing algorithm, and a dictionary (specialized list of words, names, meaningful strings people tend to use), can crank away at the passwords by hashing the entire dictionary
            - Compare the file for results
        - Simple fix: before hashing a new password and storing it
            - Generate a 32-64 bit random number and concatenate it to your password
            - Store the random number
            - When the user logs in, take his input, concatenate the stored number, and hash it
            - This random number is a **salt**
                - Usually stored in the password file next to the hash password
            - Helps because there is now a translation needed for every possible salt

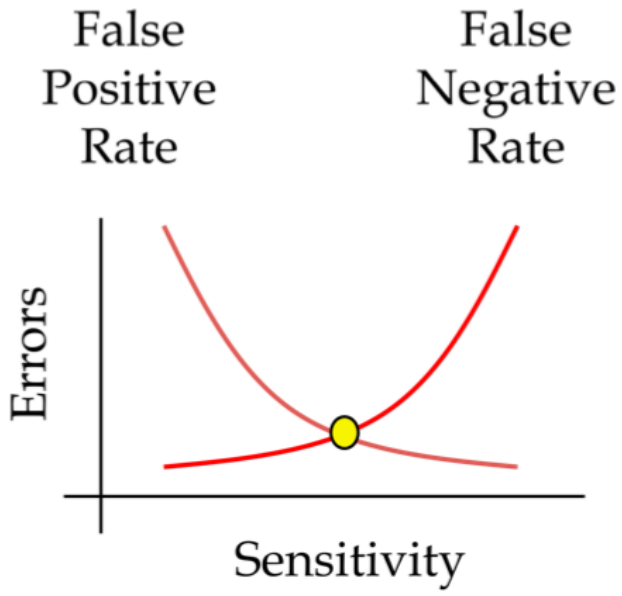
Authenticating by What You Have

- Something that plugs into a port
    - Can tell whether user trying to login has the proper device using security tokens called **dongles**
- Authenticating human user based on ability to transfer into from whatever it is they have to the system
- Devices rely on frequent changes of the information the device passes to the OS
    - Should be changed every few seconds
- Weak point: what if you don't have it

    - Problem: what if someone else has it

Authentication by What You Are

- Human characteristics that make you unique
- Computer can measure properties or characteristics and distinguish one person from another using those
- Difficult to measure
- Need to match sets of data
    - If “close enough”, we can authenticate
    - Some sloppiness is required
- **False negative:** incorrectly decided not to authenticate
    - Intolerant of differences
- **False positive:** incorrectly decided to authenticate
    - Too tolerant of differences
- Biometrics:
    - Any implementation will have false positive and false negatives rate
    - We want these rates to be low



- Intersection is the **crossover error rate**, a metric for describing accuracy of a biometric
    - Equal tradeoff between the two types of errors
- Another issue is that it **requires special hardware**
- Another issue is that there is a **physical gap between where it is measured and where it is checked**
    - Untrusted machine’s measurements can be dangerous

Authenticating Non-Humans

- Identities of different processes
- Make sure that only the real process has that identity

- Can have passwords for processes as well
    - System admin can log in as a web server and create a command shell to generate processes
    - Mechanisms whereby the privileged user can create processes that belong to someone else
    - Mechanisms that allow processes to change ownership
    - Allow a temporary change of identity
- Passwords are the most common authentication method
- Some trusted users are given the right to assign new identities to processes they create
    - In linux, sudo does this
- Identify groups of users who share common privileges
    - Associate group membership with processes
    - Or use the process' individual identity as an index into a list of groups

Access Control

Introduction

- Need to know what security policies we want to enforce
- Steps to figure out how to determine policy:
    1. Figure out if the request fits within our security policy
    2. If it does, perform the operation — if not, make sure it isn't done
- The first step is **access control**
    - Determining which system resources or services can be accessed by which parties in which ways under which circumstances
    - Boils down to another binary decision that fits well into computing paradigms: yes or no → how do we decide this?

Important Aspects of the Access Control Problem

- System will run some algorithm to make the decision
    - Takes certain inputs and produces binary output
    - Yes-or-no decision on granting access
- Access control, at a high level, is spoken in terms of **subjects, objects, and access**
- **Subject:** the entity that wants to perform the access
- **Object:** thing the subject wants to access
- **Access:** particular mode of dealing with the object
- Access control decision is about whether a particular subject is allowed to form a particular mode of access on an object
- **Authorization:** process of determining if a particular subject is allowed to perform a particular form of access on a particular object
- When will the decision be made?
    - The algorithm should be run every time the system needs to decide

- **Reference monitor:** code that implements authorization algorithm
    - Must be correct and efficient
- Need to balance cost (checking often) against security benefits
    - Try to find some special cases where we can achieve low cost without compromising security
- Give subjects objects that belong only to them
    - Object is inherently theirs
        - The system will let the subject access it freely
    - Virtualization allows us to do this
        - EX: virtual memory belongs to processes
    - Process is allowed to access its virtual memory freely
    - Can also do this with peripheral devices as well
    - In reality, these objects are shared
        - In the background, the OS is actually running keeping the illusion going
        - OS will still have to ensure the proper forms of access are allowed
        - Virtualization just pushes the problem down to protecting the virtualization function of the OS
- There are two basic approaches to dealing with the general problems that rely on different data structures and methods of deciding
- **Access control lists** and **capabilities**
    - **Access control list system:** only let certain verified/distinguished subjects in
    - **Capability:** key and lock, subjects with the key can unlock the resource
        - Can make new resources as needed
- Capabilities: if the process has capability specific to the file, it can access it
- Access control list: checks if the process is on the list
- Checks for both can be made at the moment that access is requested
- The check is made after trapping to the OS but before access is actually permitted
    - Early exit and error code if the access control check fails
- Need different data structures to support each

Using ACLs for Access Control

- Each file has **its own access control list**
    - Simpler, shorter lists
    - Quicker access control checks
- When a call traps to the OS, it consults the process's PCB to determine who owns the process
    - Indicates the owner of the process
    - Say the owner is X
- System then gets hold of the access control list for that object

    - Since the ACL is **metadata** it is stored with or near other metadata
- Look up whether owner X is on the list
    - If he is there, determine if the ACL entry allows the type of access he is requesting
    - If he is not there, then no access is granted
- The ACL is usually stored on the disk for persistence
    - Unless it is cached, it needs to be read in from disk
    - Should be close to something we've already been reading
        - At the file's directory entry, the inode, or the first data block
- How big is this list?
    - List can be of an arbitrary size if we use actual user IDs and access modes
        - May end up being extremely large
        - Files only belong to a few users, we don't want to reserve that much space in every ACL for every possible user to be listed
    - Some files should be available in some modes to all users
- Obvious implementation: big per-file list totally filled for some files and nearly empty for others
    - This is wasteful
    - Need to check or search the entire list to check access
        - Incurs overhead
- Use legacy feature from Bell Labs Unix System → three effective modes that they cared about: **read, write and execute**
    - Use three bits to handle most security policies
    - Partitioned the entries in the ACL into three groups:
        - **Owner of the file:** identity on the inode
        - **Members of a particular group/users:** group ID is also on the inode
        - **Everybody else:** people who aren't owners or members
            - No bits are really needed
            - Just complement the user and group
- Solution: solved the problem of the amount of storage potentially needed and the cost of accessing and checking
    - If the ACl is in the inode, no extra seeks or reads needed
    - Simple bit logic required
- PROS:
    - Easy to figure out who can access a resource
    - Changing permissions means only changing the ACL
    - ACL is kept with or near the file
        - If you can access the file, you can access the other information
- CONS:
    - Solution regarding size and where to store the ACL is relatively simple
        - Can limit functionality

- Need to check all ACLs in the system to see the entire set of resources a principal has access to
- In a distributed environment, need a common new way of identity across all machines

Using Capabilities for Access Control

- Running process has a set of capabilities that specify its access permissions
- A pure capability system will have no ACLs anywhere
- To perform things like open():
  - Application provides a capability permitting your process to open the file in question as a parameter or the OS finds the capability for you
  - OS checks if the capability allows you to open it or doesn't allow you to
- A capability is **a bunch of bits**
  - Likely to be long and be fairly complex
  - Specific to a resource
- No proprietary or reserved bit patterns that processes cannot create
  - Processes could trivially created more copies of the capability
  - Anyone can create any capability they want
  - Once someone has a working capability
    - Can make as many copies as they want
- This is not good from a security perspective
  - Have to make capabilities unforgeable
- The ability to copy a capability implies that we **can't take away access** because a process may have copies of it stored away
  - Process can also grant permissions to other processes by sending the capability over through IPC
- Deal with this problem by **never letting processes touch any capabilities**
  - The OS controls and maintains capabilities, storing them on protected memory space
- Process can perform operations on capabilities with the help of the OS
- OS will need to maintain its own protected capability list for each process
  - Can be done using the PCB, which is a protected per process data structure
  - Place a pointer to the capability list (which is in kernel memory) on the PCB
- When a call traps to the OS, the OS can consult the capability list of a process
- In general, the capability list containing all that a principal can access will incur high overheads
  - Usually if capabilities are used, system stores these persistently somewhere and then will import them as needed
  - The capability list attached to the process will not be very long in this case

    - Issue: which capability does a process need when it runs?
- Capabilities can be stored elsewhere (not on the OS) protected using cryptography
    - If they are protected and are long, they cannot be guessed in a practical way
- Make sense in a distributed system
- **Good and bad points about capabilities**
    - Easy to determine which system resources a principal can access
        - Just look through the capability list
    - Revoking access is the same as removing the capability from the list
        - Easy if the OS has exclusive access to capabilities
    - Can be cheap to check if it is already in memory
        - Capability can simply contain a pointer to data or software resource it protects
    - Determining set of principles that can access a resource is expensive
        - Need to check all principals' access lists
    - Methods for making capability lists short have not yet been developed
    - System must create, store, and retrieve capabilities in a way that overcomes the issue of forgery
- Other way to create processes with limited privileges
    - In ACL, process inherits the identity of parent and its privileges from the parent
        - It is difficult to give it just a subset
            - Need to create a new principal
            - Or some extension is needed for the ACL model
    - In capabilities, can just specify which capabilities to transfer specifically
- In practice, user-level access control mechanisms use ACLs
- Under the covers, the OS uses capabilities extensively
- EX: open() uses the ACL to check if open is possible
    - If the initial check is successful, while the file is open, the ACL is no longer examined
    - Instead, a capability is used, attached to the PCB
    - When a file is closed, this data structure is deleted
- A combination of ACL and capabilities allows the system to avoid problems associated with each mechanism
    - Cost of checking the ACL list is reduced
    - Cost of managing capabilities is avoided because the capability is only set up after the ACL is successful
    - If the object is never accessed, the ACL is never checked and no capability is required
    - Processes usually only open small fraction of total files so there is no scaling issue

Mandatory and Discretionary Access Control

- Owner of the resource decides what the access control on a computer resource should be
- **Users decide on user files**
- **System admin decides on system resources**
- Some systems and security policies need tighter security sometimes
- **Discretionary access control:** more common, access control is at the discretion of the owning user
- **Mandatory access control:** some elements of access control decisions are mandated by an authority with the ability to override
- Choice between discretionary or mandatory
    - Orthogonal to choosing ACLs or capabilities
    - Independent of other aspects o access control mechanisms
    - May include elements of each other

Practicality of Access Control Mechanisms

- Systems usually expose a simple or more powerful ACL mechanisms and use a discretionary control
    - Human user does not set all of the files' permissions
    - System allows user to have a default set of permissions
- Owner can alter the default ACL but users rarely do
- Software packages have taken care to set access control
- Practical issue: different people have different privileges
    - Organize access control based on roles
    - Users can switch between roles
- **Role-Based Access Control (RBAC):** switching permissions based on roles
    - Commonly used in large organizations
    - Similar to groups in ACL but more formal and systematic
    - Requires new authentication step when taking on new roles
        - Relinquishing old permissions
- **Type enforcement:** associates detailed rules to particular objects using a **security context**
    - Has performance, storage, and authentication implications
- Minimal RBAC can be built in Linux using ACls and groups using **privilege escalation**
    - Allows extension of privileges by allowing program to run with a set of privileges greater than those of the user that invoked them
    - Can use the sudo or setUserID command

Protecting Information with Cryptography

Introduction

- OS has no control over what happens on other machines or if the hardware it is controlling is being accessed by something else

- OS needs to protect something when it has no control over this
    - Do this by preparing the data structure
    - Assume we will lose the data
    - Or that an opponent will try and alter it improperly
- Key observation: if our opponent can't understand it, they can't alter or read it
- **Cryptography:** set of techniques to convert data from one form to another in controlled ways with expected outcomes
    - Goal: convert data from ordinary form to another using cryptography
- Opponent will not be able to determine the original data
    - However, must also be able to reverse the operation, but not too easily
- Cryptography is computationally expensive
- We need to be selective of when or where we use it
    - Well chosen, poorly performed → doesn't help security
    - Poorly chosen, well performed → may hurt

Cryptography

- Provides strong interface and well-defined behaviors
- Basic idea: take a piece of data and use an algorithm (a **cipher**), usually augmented with a second piece of information (a **key**) to convert the data to another form
- Want to be able to run another algorithm augmented with the same piece of data to get the original data
- The result: C = E (P, K) P = D (C, K)
    - C: the **ciphertext**
    - E: the encryption algorithm
    - K: the key
    - P: data we start with, the **plaintext**
    - D: the decryption algorithm
- P is usually the message we want to send and C is the protected version of that message
- Must be deterministic
- Should be hard to figure out P from C but not impossible
- Opponent should not know D() and K
- Usually we can't keep E() and D() a secret because it is hard to design good ciphers
    - Benefit of cryptography relies on **the key and its secrecy**
- Maintaining key secrecy is difficult
- Cryptography can be used to protect the integrity of data
    - Hash on received plaintext and compare to what it was before
- **Symmetric cryptography:** both sender and receiver know the key, same key is used to encrypt and decrypt the data

Public Key Cryptography

- To verify authenticity of encrypted data, need its key

        - Need to communicate this key to whoever might need to authenticate us
    - Use 2 different keys: one for encrypting and the other for decrypting
    - Encrypting and decrypting algorithms are now:
        $C = E(P, K^{encrypt})$ and $P = D(C, K^{decrypt})$
    - Notw the $K^{decrypt}$ can be public while the $K^{encrypt}$ can stay secret
    - Authenticated by encrypting it with the secret key
        - Other users can check authenticity by using the public key
        - Only you know the private key so it verifies that it is you
    - **Public key cryptography:** one of the two keys is known to the entire public while still achieving desired results
        - **Private key:** key only owner knows
        - **Public key:** key known to everyone
    - Solves an issue: distributing a key securely
        - Now, we can simply distribute the public key
        - This is still difficult
    - Public key cryptography can be done the other way around as well
        - Can use $K^{decrypt}$ to encrypt messages and $K^{encrypt}$ to decrypt these "reversed" messages
        - $K^{encrypt}$ is still secret so only the owner of $K^{encrypt}$ can decrypt these messages
    - PK allows authentication → and secret communication
        - If you want both, you need **2 pairs of keys**, this is expensive
    - PK cryptography is computationally expensive
        - We cannot use it for everything
    - Another issue: need to distribute keys securely
        - Need a key distribution infrastructure
    - Benefit relies on the privacy of the private key

<u>Cryptographic Hashes</u>

    - Can use hashing to ensure data integrity, but you can't use any hash function
    - **Cryptographic hash:** used to ensure integrity
        - **Special category of hash functions with several important properties**
            - Computationally infeasible to find 2 inputs with the same hash result
            - Changes in input result in unpredictable changes in the hash result
            - Computationally infeasible to infer properties of the input based only on the hash value
    - Data integrity can be ensured
        - Take the has of the data
        - Encrypt the data
        - Send the encrypted hash and data to partner
        - If someone messes with it in transit you can tell
        - Decrypt the hash and repeat hashing on the data

        - If you see a mismatch, you know it is corrupted
            - S=H(P)
    - S is usually shorter than P since crypto-hashes are a subset of hashes
        - May still be collisions but it will be difficult to use these collisions to decode
    - Cryptographic hashes are also used **for storing salted passwords**
    - Used to **determine if a stored file has been altered**
    - Used to force processes to perform a certain amount of work before submitting a request, called **"proof of work"**
        - Submitter required to submit a request that hashes a certain value
            - Requires lots of request format attempts
        - Takes time, requires a predictable amount of time
    - Standard algorithm may become obsolete eventually

Cracking Cryptography

- Using modern standards, there is no way to read data encrypted with algorithms without the key
- Can be done by exploiting software flaws to obtain ey or exploit a flaw in cryptography management system
- Software flaws in creating and managing keys is common
- **Improper cryptography management issues:**
    - Distributing secret keys
    - Transmitting plaintext versions of keys
    - Choosing from a limited set of keys
- Attackers can also guess the keys using a **brute force attack**
    - Reason why long keys should be used
    - Take longer to guess
- PK cryptography
    - Keys are longer but not random
    - Public and private keys must be somehow related
    - Mathematically difficult to derive the relationship

Cryptography and Operation Systems

- OS can decrypt all data because it has access to all private keys in the system
- OS can be compromised
    - May not be permanent but attacker may still have temporary access
        - If data is stored in an encrypted form, it is not as bad
        - IDEA: frequently encrypt, decrypt infrequently and dispose of plaintext quickly and carefully
            - Not very realistic and feasible
        - Don't keep plaintext key in system except when doing cryptographic operations
- Use cryptography to store passwords in an encrypted form
- Distributed environment

        - Encrypt data on one machine and then send it
        - Intermediaries won't be part of the machine and won't have access to the key
        - Data is protected in transit
        - The final machine will need the key
    - Data in hardware is encrypted, **at-rest data encryption**

At-Rest Data Encryption

- May want persistent data to be secured
    - Maybe store it in an encrypted form instead of in plaintext
- To perform computations on encrypted data, must first decrypt it
- Can be encrypted in different ways using different ciphers at different levels of granularities by different components
- Common use: **full-disk encryption**
    - Entire contents of storage device is encrypted
    - Usually provided in hardware or by system software
- At boot time, decryption key or information to get the key is requested from the user
    - If right information is provided, the keys necessary to perform decryption become available
        - Data sent to device is encrypted
        - Data from the device is decrypted
    - Data is decrypted as long as it remains in memory
    - New data sent is first encrypted
    - Process is transparent after the key request
- Computationally expensive
- Overhead incurred with full-disk encryption
- **Does not protect against**:
    - Users trying to access data they shouldn't see
    - Flaws in application that divulge data
    - Dishonest privileged users
    - Security flaws within the OS
- **Protects against:**
    - Storage device being moved onto another machine that shouldn't have access
        - If the hardware is stolen
- Other forms of at-rest encryption
    - Systems will address the same issues of how much, how to get the key, when to encrypt/decrypt the data
    - Requires software that ensures encrypted data is not left in the system/disk and key is not available
- Circumstances where this is useful:
    - Archiving data that needs to be copied and preserved
    - Storing sensitive data in cloud computing facilities

        - User-level encryption through an application
    - Special case is a **password vault/key ring**
        - Storing different passwords for each site on the machine
    - Password vaults and their specialties:
        - System needs to obtain the required key when data needs to be encrypted or decrypted
        - Needs to be stored somewhere safe
    - Keyring case: key to decrypt is obtained by asking for it and deriving it
        - Max security: destroyed key once it is decrypted
            - Implies user needs to renew the key whenever password is needed
        - Compromise between security and visibility
            - Remember the key in RAM
            - When the user logs out, the key is forgotten

Cryptographic Capabilities

- Need to create unforgeable capabilities
- Create long and securely encrypted data structure that indicates if a process has access
    - Given to a user
    - Present to owner to resource to gain access
- Can be created with public or private keys
- Symmetric key option: creator and system checking have the same key
    - Most feasible if they both are on the same system
    - Otherwise, need to move keys around
- Public key option: don’t need to be on the same computer
    - Resource manager can create credentials and encrypt them
    - Credentials can be validated
    - Secrecy
- Hold a lot of information
    - Expiration time
    - Identity
    - Etc.
- Reliable
- Can prevent arbitrary copying and sharing

Distributed Systems

Lecture 16: Distributed Systems

Outline

- Introduction
- Distributed system paradigms
- Remote procedure calls
- Distributed synchronization and consensus
- Distributed system security

Goals of Distributed Systems

- More than 1 computer, operating OS on different computers connected by a network
- **Scalability** and **performance**
    - Applications require **more resources** than one computer has
    - Grow the system capacity/bandwidth to **meet demand**
- Reason to build: not enough performance out of one computer
- Improved **reliability and availability**
    - 24/7 service despite disk/computer/software failures
    - More reliability with more computers
        - In the case of failure, other computers can pick up the slack
- **Ease of use** with reduced operating expenses
    - Centralized management of all services and systems
    - Buy better services rather than computer equipment
    - **Simple to maange** but need to manage multiple versions of software
- Enable **new collaboration** and business models

Transparency

- Ideally, a distributed system would just be like a **single machine system**
- Only works well if there is **a lot of transparency**
- Too complex to work with for regular users
    - Need to hide complexity
- Better
    - More resources
    - More reliable
    - Faster
- **Transparent** distributed systems look as much like single machine systems as possible

Deutsch's "Seven Fallacies of Network Computing"

1. The **network is reliable**
2. There is **no latency**
    - Messages are not instantaneous

3. The **available bandwidth is infinite**
4. The network **is secure**
5. The **topology of the network does not change**
6. There is **one administrator** for the whole network
7. The **cost of transporting additional data is zero**

- Bottom line: true transparency is not achievable

Heterogeneity in Distributed Systems

- Distributed systems **aren't uniform**
- **Heterogeneous clients**
    - Different ISAs
    - Different OS and OS versions
- **Heterogeneous servers**
    - Different implementations
    - Offered by competing service providers
- **Heterogeneous networks**
    - Public and private
    - Managed by different organizations in different countries
- Another problem for **achieving transparency**

Fundamental Building Blocks Change

- The old model:
    - Programs run in processes
    - Programs use APIs to access system resources
    - API services implemented by OS and libraries
- The new model:
    - Clients and servers run on **nodes**
    - Clients use APIs to access services
    - API services are exchanged via protocols
- Local is a **very important special case**
    - Higher performance and better security and control compared to going across the network

Changing Paradigms

- Network connectivity has become a **given**
    - Applications assume/exploit connectivity
    - Distributed programming paradigms emerge
    - Functionality depends on network services
- Applications demand new kinds of services
    - Location independent operations
    - Rendezvous between cooperating processes
    - WAN scale communication, synchronization
        - Need to synchronize across networks
        - Include communication delays
        - Locks will be held for longer

Distributed System Paradigms

- **Parallel processing**
    - Relying on tightly coupled special hardware
- Single **system images**
    - All nodes look like one big computer
    - Somewhere between **hard and impossible**
- **Loosely coupled systems**
    - Work with difficulties as best as you can
    - Typical modern approach
    - Make an effective system by **limiting expectations and providing what is possible**
- **Cloud computing**
    - Compute power in one location, not everywhere

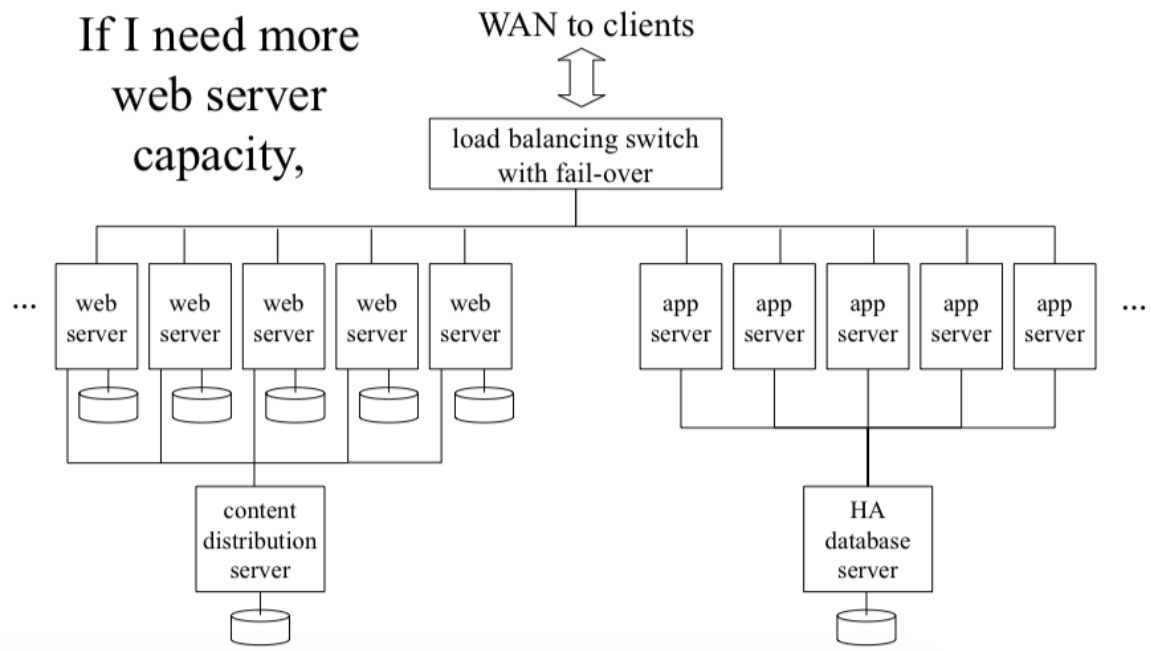
Loosely Coupled Systems

- Characterization:
    - Parallel group of independent computers
    - High speed LAN
    - Similar but independent requests
        - Given to single computers
        - Farm out requests to different computers
    - Minimal coordination and cooperating required
- Motivation:
    - Scalability and price performance
        - Add more computers to have more services/serve for requests
    - Availability
        - If the protocol permits stateless servers
        - If one crashes, still are able to handle other requests
    - Ease of management, reconfigure capacity

Horizontal Scalability

- Approach works based on **horizontal scalability**
- Each node is largely independent
- Add capacity by **adding a node**
- Scalability can be **limited by network** instead of **hardware or algorithms**
    - Or a load balancer
        - Special machine that puts load on computers
        - Also has a capacity
- Reliability is **high**
    - Failure of one of N nodes just reduces capacity

Horizontal Scalability Architecture



- Content distribution server
  - Distributes to your customer
  - Can be a bottleneck
  - Use aggressive caching

Elements of Loosely Coupled Architecture

- Farm of **independent servers**
  - Servers run the same software and **serve different requests**
  - Can share a common back-end database
- **Front-end switch** – load balancing
  - Use caching to make this not a bottleneck
  - Distributes incoming requests among available servers
  - Can do both load balancing and fail-over
- **Service protocol**
  - Stateless servers and **idempotent operations**
    - No state is preserved between different messages
      - Remembering state makes it much harder to manage distributed systems
    - Operations that can be performed N times with the same result
    - Don't have to remember if you did it already
  - Successive requests may be sent to different servers

Horizontally Scaled Performance

- Individual servers are **very inexpensive**
- Scalability is excellent, almost perfect scalability
- Service availability is **excellent**
  - Front-end automatically bypasses failed servers
  - Stateless servers and client retries fail-over easily

- The challenge is **managing thousands of servers**
    - Limited by management, not hardware or algorithms

Cloud Computing

- Set up a **large number of machines all identically configured**
    - 1000s, 10000s or computers
    - Need excellent power and AC
- Accept arbitrary job on one or more nodes
    - Do work on the computers for users/consumers
- Run each job on **one or more nodes**
    - Divide up the capacity among different nodes
- Entire facility probably running on a mix of **single machine and distributed jobs**
    - Some nodes run one job, others run 1/nth of a job, or run 12 jobs on one node

What Runs in a Cloud?

- Anything
- Distributed computing is hard
- Work is run using special tools
- Tools support particular kinds of parallel/distributed processing
- Either **embarrassingly parallel jobs**
- Or those using a method like **map-reduce**
- User does not need to be a distributed systems expert

Embarrassingly Parallel

- Really, really easy to parallelize them
- Data sets are easily divisible
- Exact same things are done on each piece
- Parcel out jobs among the nodes and let each go independently
- Everyone finishes at more or less the same time

MapReduce

- **Common cloud computing software tool/technique**
- Method of **dividing large problems into compartmentalized pieces**
- Can be performed on a separate node
- Eventual **combined set of results**

The Idea Behind MapReduce

- There is a **single function** you want to perform on **a lot of data**
    - Eventually get one summarized result
    - Searching it for a string
- Divide the data into **disjoint pieces**
    - Most sensible to divide them into equally sized pieces
- Perform the **function** on each piece on a separate node
    - **Map** each function onto pieces of data
- Combine the results to obtain
    - **Reduce** to combine the results

Reduce

- May have two or more nodes assigned to do the **reduce operation**
- Each will receive a share of data from the map node
- The reduce node performs a reduce operation to **“combine” the shares**
- Output its own result
- Match subsets of data to which node

But I Wanted A Combined List

- Run another MapReduce
- One reduce node that combines everything
- Problem: a node may fail
    - Need to recover from this

Synchronization in MapReduce

- Synchronize at the end and also at the map node output
- Each map node produces an output file for each reduce node
- Produced **atomically**
- Reduce node can’t work on data until the **whole file is written**
    - Don’t start reducing until **everyone has finished mapping**
    - This helps with error detection
    - If a node doesn’t return, it is assumed to have crashed
        - We know we have a problem
        - Since we know which node didn’t return, we know what has not yet been done
    - Give this node’s work to other nodes
- Forces a **synchronization point** between the map and reduce phases

Remote Procedure Calls

- RPC
- Program in one process makes a procedure call to another process
    - Code may not even be on my own machine
    - Procedure call across a network
- One way to build a distributed system
- Procedure calls are a **fundamental paradigm**
    - Easy for programmers to understand
- Natural **boundary between client and server**
    - Client makes a call and server returns the value
    - Turn procedure calls into message send/receives
- Some limitations
    - No implicit parameters/returns
    - No call-by-reference
    - No shared memory
    - Slower than procedure calls
- There is no hardware that makes procedure calls across a network
    - Reduced to **sending packets**

        - Need to change a procedure call into packets to be sent across the network
        - Need to use system calls to get messages to send packets across the network

Remote Procedure Call Concepts

- Interface specification
    - Methods, parameter types, return types
- **eXternal Data Representation**
    - Machine independent data-type representation
    - May have optimizations
        - Say if client architecture is the same as the server architecture
    - Use this if we don't know if specifications for things like word size, endianness, etc.
    - A common format, used to move data
- **Client stub**
    - Client-side proxy for a method in the API
    - Makes it look like the procedure call happened locally
        - Take the procedure call and use the client stub as a proxy
        - Take the parameters and information
        - Convert it to packets to be sent
- **Server stub** (or skeleton)
    - Server-side recipient for API invocations
    - Expects messages in a particular format

Key Features of RPC

- Client application **links against local procedures**
- All **RPC implementation** inside those procedures
- Client application does not know about RPC
    - All hidden by the client stub
- All of this is **generated automatically** by RPC tools
    - Capable of building custom-made stub for every procedure call you can call
- The **key to the tools** is the interface specification
    - Need to get this right

RPC is Not a Complete Solution

- This is a **mechanism, not a solution** to distributed system problems
- Expects a server to provide that function
- Need a live network of sufficient bandwidth
- Requires **client/server binding model**
    - Expects a live connection
- **Threading model** implementation
    - A **single thread service** requests one-at-a-time
    - Numerous **one-per-request worker threads**
- **Limited failure handling**

        - Client must arrange for timeout and recover
        - Not much, need to arange
    - Higher level abstractions can improve RPC

Distributed Synchronization and Consensus

- Hard to synchronize
- Tools used
- Consensus among cooperating nodes

What's Hard About Distributed Synchronization

- Before shared memory section was in a single machine
    - Now, there are different machines
    - Disagreement in orderings, etc.
    - We can't look directly at another machine's RAM
- **Spatial separation**
    - Different processes run on different systems
    - No shared memory for locks
    - Controlled by different operating systems
- **Temporal separation**
    - Impossible to "totally order" spatially separated events
    - Can't force things to happen
        - It would be expensive and difficult
- Independent modes of failure
    - One can die while others can continue
    - Before, if a system fails, the entire process would fail
    - Some machines can fail but others may not

Leases - More Robust Locks

- Obtained from a **resource manager**
    - Leases are given **by the machine owning the resource**
        - Need to go and ask for it
    - Gives client exclusive right
    - Lease **"cookie"** must be passed to the server on update
        - When trying to access resource, must present cookies
    - Lease can be released at the end of the critical section
- Only valid **for a limited period of time**
    - After that, the lease cookie expires
        - Updates with **stale cookies** are not permitted
    - After which, **new leases can be granted**
- **Handles a** wide range of failures
- But if a lease expires in the middle, **the resource may be left in an inconsistent state**
    - Need to worry about this

Lock Breaking and Recovery

- Revoking an expired lease is **fairly easy**
    - Lease cookie has a timeout

        - Operations involving a stale cookie will fail
    - Makes it **safe to issue a new lease**
        - Old lease holding will no longer be able to access the object
    - Object must be restored to **last "good" state**
        - Roll back to state prior to the aborted lease
        - Implement all-or-none transactions

Distributed Consensus

- Achieving simultaneously, **unanimous agreement**
    - Even in the presence of node and network failures
    - Required: **agreement, termination, validity, integrity**
    - Desired: bounded time
    - Provable **impossible in the general case**
        - Nodes can fail, messages can be lost
        - Hard to have everyone agree on the same thing
    - But can be done in **userful special cases** or if some requirements are relaxed
- Consensus algorithms **tend to be complex**
    - May take a long time to converge
- Tend to be used sparingly
    - Try to avoid having to reach consensus

Typical Consensus Algorithm

1. Each interested member broadcasts his nomination
2. All parties evaluate the received proposals according to a **fixed and well-known rule**
3. After allowing a reasonable time for proposals, each voter acknowledges the best proposal it has seen
4. If a proposal has a majority of the votes, the proposing member broadcasts a claim that the question has been resolved
5. Each party that agrees with the winner's claim acknowledges the announced resolution
6. Election is over when a quorum acknowledges the result

- The assumption is that **everybody is honest**, nobody is trying to prevent system from consensus
- Falls apart if there are enough bad guys
- Need to turn them into Byzantine set of protocols
    - Unpleasant

Security for Distributed Systems

- Security is hard in single machines
- Harder in distributed system

Why is Distributed Security Harder?

- OS cannot guarantee privacy and integrity
    - In the individual system, you trust the OS
    - Network transactions happen outside of the OS

        - Need to trust other OS's on other machines
    - Authentication is harder
        - Possible agents may not be in the local password file
    - Wire may not be secure
    - Hard to coordinate distributed security
    - Internet is an **open network**
        - May not be secure
    - As a rule, networks cannot be trusted

Goals of Network Security

- Secure conversations
    - **Privacy:** only you and your partner know what is said
    - **Integrity:** nobody can tamper with your messages
- **Positive identification of both parties**
    - Authentication of the identity of the message sender
    - Assurance that a message is not a replay or forgery
    - Non-repudiation
- Availability
    - This is difficult
    - Network and other nodes must be reachable when they need to be

Elements of Network Security

- **Cryptography**
    - **Symmetric cryptography** for protecting bulk transport of data
    - **Public key cryptography** primarily for authentication
    - **Cryptographic hashes** to detect message alterations
- **Digital signatures** and **public key certificates**
    - Tools to authenticate a message's sender
- **Filtering technologies**
    - Firewalls and the like
    - Try to identify bad things

Tamper Detection: Cryptographic Hashes

- **Check-sums** are often used to detect data corruption
    - Add up all bytes in a block and send the sum along with the data
    - Recipient adds up the bytes
    - If the checksum agrees, the data is probably OK
    - Checksum algorithms **are weak**
- **Cryptographic hashes are very strong check-sums**
    - **Unique:** two messages vanishingly unlikely to produce some hash
        - Particularly hard to find two messages with the same hash
    - **One way:** cannot infer original input from output
    - **Well distributed:** any change to input changes output
        - Slight change in the input should produce a very different output
    - Want to reduce a large quantity of data to small data
        - Like a strong checksum

Using Cryptographic Hashes

- Start with a message you want to protect
- Compute a cryptographic hash for that message
- Transmit the **hash securely**
    - This is a problem
- Recipient does the **same computation** on the received text
    - If both hash results agree, the message is intact
    - If not, the message has been corrupted/compromised

Secure Hash Transport

- Hash must be transmitted securely
    - Cryptographic hashes aren't keyed so anyone can produce them
    - Someone can change the message, provide a new hash, and send both
        - It looks like the message is intact because the hash matches
- How do we transmit it securely?
    - **Encrypt it**
    - Cheaper than encrypting the entire message
    - If you have a **secure channel**, you can also transmit it that way

A Principle of Key Use

- Both symmetric and PK cryptography rely on a **secret key** for their properties
- The more you use one key, **the less secure**
    - Key **stays around** in various places longer
    - More **opportunities for attacker** to get it
    - More **incentive for attacker** to get it
    - Brute force attacks **may eventually succeed**
- Therefore
    - Use a given key **as little as possible**
    - **Change** them often
    - Within the limits of practicality and required performance

Putting It Together: Secure Socket Layer (SSL)

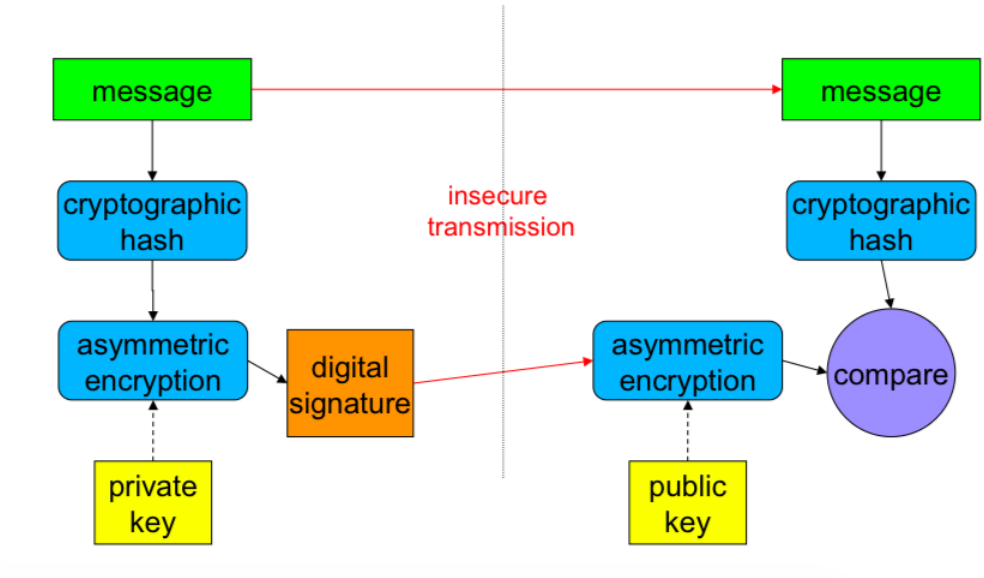
- General solution for **securing network communication**
- Built on top of **existing socket IPC**
- Establishes **secure link** between two parties
    - Privacy and integrity
- Certificate-based **authentication of server**
    - Typically but not necessary
    - Client knows what server he is talking to
- **Optional certificate-based authentication of client**
- PK used to **distribute symmetric session key**
- Rest of data transport switches to symmetric cryptography

Digital Signatures

- Encrypting a message with a private key **signs it**
    - Only you could have encrypted it, must be from you
    - Not been tampered with since you wrote it

- Encrypting **everything with your private key is a bad idea**
  - Asymmetric encryption is slow
- If you only care about integrity, **don't need to encrypt it all**
  - Compute a cryptographic hash of your message
  - Encrypt the cryptographic hash **with your private key**
  - Faster than encrypting whole message

Digital Signatures



- The public key is the **weak point**
  - Ensuring that the public key has not been tampered with is difficult

Signed Load Modules

- Certification authority
  - Verifies the reliability of the software
- They sign a module with their private key
- We can verify that signature with their public key
  - Proves module was certified by them
  - Proves module has not been tampered with

An Important Public Key Issue

- If you have a public key
  - Can authenticate received messages
  - Know they were sent by the owner of the key
- But need to verify the person who owns the ey
- Wants a **certificate of authenticity**

What is a PK Certificate?

- A **data structure**
- Used to **distribute public keys**
- Contains an identity and **a matching public key**
  - And some other information
- Contains a **digital signature of those items**

        - Computed over the identity and the public key
    - Signature is **signed by someone I trust**
        - And whose public key **I already have**

Using Public Key Certificates

- If you know the public key of the authority who signed it
    - Can **validate the signature**
    - Can tell that the certificate **has not been tampered with**
- If you trust the authority
    - Can trust they authenticated the certificate owner
- But first must trust and know signing authority

A Chicken and Egg Problem

- How do you get the public key of the signer?
- Use **out of band** method
    - Need to have a trusted outside mechanism
- Usually have the key **in a trusted program like a web browser**

Conclusion

- Distributed systems offer us more power than a single machine
    - Cost: complexity and security risk
- Handle complexity by using distributed systems in carefully defined ways
- Handle security risk by proper use of cryptography and other tools

Reading

Kampe – Distributed Systems: Goals and Challenges

- Why we build distributed systems
- Some fundamental problems

Goals: Why build Distributed Systems

- Collaboration and sharing
- Need to share work products

→ Client/Server usage Model

- Better functionality and same costs if the remote/centralized resources are used instead of having all resources be connected to the computer

→ Reliability and Availability

- Require higher reliability and availability from them
- Better reliability if we have multiple systems
- Distribute service over multiple independent servers
    - Independent
    - No **single point of failure:** single component whose failure would take out multiple systems
- Client and server instances distributed across multiple independent computers
    - Form of distributed system

→ Scalability
- Start projects on a small system then add more work
  - Need more storage capacity, network bandwidth, and computing power
- Inefficient to buy a new computer for more capacity
  - Eventually may need more computer power and capacity than a single computer can provide
- Design systems that can expand incrementally by adding more computers and storage as needed
- **Scale-out:** growth plan that builds out system using multiple independent systems/computers

→ Flexibility
- May want to run different parts on different computers or a single part on multiple computers
- Easy to change deployment model if components interact over a network

Challenges: Why are Distributed Systems Hard to Build?
- **Two reasons:**
  - Many solutions that work on a single system don't work on a distributed system
  - Distributed systems have new problems that were never encountered in single systems

→ New and More Modes of Failure
- In single systems, the entire system goes down if something bad happens
  - Bad for service
  - But don't need to worry about how some components can continue operating while others have failed
- Partial failures are common in distributed systems
  - One node can crash while others are still running
  - Network messages can occasionally be delayed or lost
  - Switch failures may interrupt communication between some nodes but not others
- Distributed systems introduce new problems never previously addressed in single systems
  - Distinct nodes operate independently of each other
    - Not possible to have a specific ordering
  - Independence of parallel events means different nodes may consider different resources to be in different states
    - Resources have multiple states
- Can have "mutex-like" effect to synchronize by sending messages to a **central coordinator** but this is very expensive
- Serialization approaches from before are too expensive now

→ Complexity of Management
- Different systems may be configured differently
  - Different user databases

        - Different service options
        - Different lists of capabilities
        - Switches configured differently with different rules
    - If we create distributed management service to push all updates to nodes
        - Some nodes may be down when updates are sent
        - Networking issues may create isolated island of nodes with different configurations

→ Much Higher Loads

- Build distributed systems to handle increasing loads
- Higher loads:
    - Uncover weaknesses that had never caused problems
    - If the load increases by a factor greater than a power of ten, new bottlenecks are discovered
- More nodes means more messages which means increased overhead and delays
    - Increased overhead means poor scaling, performance will drop
    - More delays means that race-conditions that were unlikely may now occur

→ Heterogeneity

- In a single computer system, applications:
    - Run on the same ISA
    - Run on the same version of the OS
    - Have the same versions of the same libraries
    - Interact with each other through the OS
- In a distributed system, applications may have:
    - Different ISAs
    - Different OS's
    - Different versions of software and protocols
- Components interact with each other through a variety of networks and file systems
- The number of combinations of possible component versions and interconnects is so high that testing exhaustively is impossible
- Interoperability issues and problematic interactions that never happen in a single system appear

→ Emergent Phenomena

- Complex systems exhibit **emergent behavior** not present in consistent components but arise from interactions at scale

→ Peter Deutsch's "Seven Fallacies" of Distributed Computing

- Research failed distributed computing projects

1. The network is reliable
    - Subroutine calls always happen
    - Messages and responses are not guaranteed to be delivered
2. Latency is 0
    - Subroutine time is negligible
    - Message exchange time can be slow

3. Bandwidth is infinite
    - In memory data copies so accesses are performed at very fast rates
    - In the network, throughput is limited
        - Large clients can saturate NICs, switches, and WAN links
4. The network is secure
    - OS is sufficiently well protected
    - Computers on a network are susceptible to attacks
        - Penetration, man-in-middle, denial-of-service attacks
5. Topology does not change
    - Route changes and new clients and servers appear and disappear continuously
    - Distributed applications need to deal with changing set of connections and partners
6. There is one administrator
    - No single database of all known clients
    - Different systems may have different privileges
    - Independent routers and firewalls may block messages
7. Transport cost is 0
    - Network infrastructure is not free
    - Capital and cost of equipment and channels
8. The network is homogenous
    - Nodes have different versions, different OS's, different ISAs, word lengths, byte orderings
    - Users may speak different languages, have different character sets, or different ways to represent information

Arpaci-Dusseau Ch. 48 - Distributed Systems

- Distributed systems have had a huge impact on the world
- Challenges arise when building distributed systems
    - **Failure**:
        - Machines and components can fail
        - But the entire system may still function
        - Try and build a distributed system that appears to rarely fail, despite the failures of its components
    - **System performance** is critical
        - Network connecting the distributed system together
        - Need to minimize messages sent and make communication efficient
    - **Security**
        - Need to ensure the remote party is who they say they are
        - Ensure a third party cannot monitor or alter ongoing communication
- Introduction to **communication** and how machines "talk" to each other

- Focus on how communication layers should handle failures

48.1 Communication Basics

- Central tenet of modern networking: communication is fundamentally unreliable
  - Packets are regularly lost, corrupted, or don't reach their destination
- Causes of packet corruption:
  - Bits are flipped during transmission
  - Elements in system are somehow damaged or not working
  - Network cables are accidentally severed
- Packet loss can also occur due to a lack of buffering within a network switch, router, or endpoint
  - This is a fundamental problem
  - Even if everything is working, this is still possible
- Packets arriving at a router must be placed in memory in the router to be processed
  - If there are too many packets, the memory within the router cannot accommodate
  - Only choice for a router is to drop one or more of the packets
- Packet loss because of overwhelming of the router or endpoint is possible
- **Packet loss is fundamental in networking**

48.2 Unreliable Communication Layers

- Don't deal with packet loss
- Some applications can deal with packet loss
  - Let them use a basic unreliable messaging layer
- **UDP/IP networking stack:** example of an unreliable networking layer
- To use UDP:
  - Process uses **sockets** API to create a **communication endpoint**
  - Processes on other machines (or the same machine) send UDP **datagrams** to original process
    - **Datagram:** fixed-size message up to some max size
- UDP/IP is unreliable
- Packets will get lost if you use it
  - Sender is never informed of the loss
- UDP still guards against failures using a **checksum** to detect some form of packet corruption
- Still need reliable communication on top of an unreliable network

48.3 Reliable Communication Layers

- Need new mechanisms and techniques to handle packet loss
- Use **acknowledgement** technique to let sender know receiver has received the message
  - Sender sends a message to the receiver
  - Receiver sends a short message back to acknowledge its receipt
- Sender may not receive the acknowledgement
  - Use a **timeout mechanism**

- When sender sends a message, the sender sets a timer to go off after some period of time
    - If no acknowledgement is received during this time, the message is **considered lost**
        - Sender can perform a **retry** of the send, sending the same message again
        - Sender must keep a copy of the message to do this
- Combination of timeout and retry is called the **timeout/retry** approach
- A receive may have received a message twice, generally this is **NOT OK**
- Want to **guarantee that each message is received exactly once by the receiver**
- Receiver needs to detect duplicate messages during transmissions
    - Sender needs to identify each message uniquely
    - Receiver needs to track whether the message has been seen before
- When receiver sees a duplicate transmission
    - Receiver should acknowledge the message
    - And then it **does not pass** the message along to the application receiving the data
    - Sender receives the acknowledgement but the message itself is received only once
- Myriad of ways to detect duplicate messages
    - Unique ID would be too costly, requires the receiver to track all possible IDs
- A simple approach: **sequence counter**
    - Sender and receiver agree on a start value for a counter each side maintains
    - When a message is sent, the current value of the counter is also sent, serving as an ID for a message
    - After the message is sent, the sender increments the counter by 1
    - Receiver uses the counter as an expected ID of the message
    - If ID (N) matches the receiver's counter
        - Acknowledge the message and passes it to the application
        - This is the first time the message has been received
        - Receiver increments
    - If the acknowledgement is lost, the sender retries the message, now with a higher counter value
        - Receiver sees this and knows that it already received the message
        - Acknowledges the message but doesn not pass it to the application
- **TCP/IP** is the most commonly used communication layer
    - **TCP** for short
    - More sophisticated than above paradigm

48.4 Communication Abstractions

- Abstraction of communication is used to build a distributed system
- One type: take OS abstraction and extent for distributed systems

- One example is the **distributed shared memory**
- **Distributed Shared Memory (DSM):** enables processes on different machines to share a large virtual address space
    - Turns distributed system into something like a multithreaded application
- DSM systems work by using virtual memory of the OS
- When a page is accessed on one machine, 2 possibilities:
    - Best case, the page is already local on the machine and data can be fetched quickly
    - Page can be on another machine
        - Page fault occurs
        - Page fault handler sends a message to another machine to fetch the page and install it in the page table of the requesting process
- Not widely used today
- Large problem is **handling failure**
    - If a machine fails, what happens to that machine's pages
    - Data structures could also be spread out across machines
- Performance problem
    - In DCM, memory accesses can be very expensive

48.5 Remote Procedure Call (RPC)

- Use programming language (PC) abstractions instead of OS ones
- Dominant abstraction is based on the idea of a **remote procedure call (RPC)**
- Goal of RPC packages: make process of executing code on a remote machine as simple and straightforward as calling a local function
    - To a client, a procedure call is made and later, a result is returned
    - Server only has to define the routines it wants to export
- RPC systems usually have two pieces:
    - A **stub generator** or a **protocol compiler**
    - The **run-time library**

→ Stub Generator

- Role: to remove the pain of packing a function's arguments and results into messages by automating it
- Benefits:
    - Avoids mistakes from writing such code by hand
    - Can optimize code and improve performance
- Input to the compiler is the set of calls a server wishes to export
- Stub generation takes input and generates a few pieces of code
- **Client stub** is generated for the client
    - Contains the specified functions from the interface
    - Client program would link with this client stub and call into it to make RPCs
    - Each function in the client stub do all work needed to perform the remote procedure call

        - To the client, it looks like a procedure call
    - Code in the client stub does the following:
        - Create a message buffer
        - Pack information into the message buffer
            - Includes identifier for the function to be called and the arguments
            - Process of putting information into the buffer is called **marshalling** of arguments or the **serialization** of the message
        - Send the message to the destination in RPC server
            - Communication between RPC server and its details are handled by the RPC run-time library
        - Wait for the reply
            - Function calls are usually synchronous
            - Call will wait for completion
        - Unpack return code and arguments
            - Stub goes through **unmarshalling** or **deserialization** process
        - Return to the caller
- Code also generated for the server
    - Steps the server takes:
        - Unpacks the message
            - unmarshalling/deserialization
            - Takes information out of the incoming message
            - Function identifier and arguments are extracted
        - Call into the actual function
            - The remote function is actually executed
        - Package the results
            - Marshaled into a single reply buffer
        - Send the reply
- Issues to consider
    - **Packing and sending complex data structures**
        - Need to use well-known types
        - Or annotate the data structure
    - **Organizations of server in terms of concurrency**
        - Inefficient if the server just waits for requests
        - Servers constructed in a concurrent fashion
            - A **thread pool**: finite set of threads created when the server starts
            - When a request comes, it is dispatched to one of the server threads
            - Main thread receives requests that are dispatched
            - Enables concurrent execution
            - Cost of this arise as well

        - Locks and other synchronization privileges

→ Run-Time Library
- Run-time library handles heavy lifting in the RPC system, including performance and reliability issues
- Comes with challenges
- Need to locate the remote service
    - Naming problem is common in distributed systems
    - Simple approaches build on existing naming systems
        - E.g. hostnames and port numbers provided by internet protocols
    - Client needs to know the host name and IP address of the machine running the desired IPC service and the port number
    - Protocol suite needs to provide service for addressing another machine on the system
- Need to know which transport level protocol RPC should be built on, reliable or unreliable communication layer
- Building on top of the reliable layer
    - Leads to inefficiency
    - “Extra” acknowledgement messages are sent
- Many RPC packages are built off unreliable communication layers
    - More efficient RPC system
    - Need to add responsibility to the system
- RPC layer adds responsibility to the system using timeout/retry and acknowledgements and sequence numbers

→ Other Issues
- Remote calls make take a long time
    - May appear to be a failure
    - Use explicit acknowledgement when reply isn’t immediately generated to acknowledge receipt
    - Client can periodically ask if the server is still working
- Handle procedure calls with large arguments
    - Larger than what can fit in a single packet
    - Some networks provide sender-side **fragmentation** (large → small) and receiver side **reassembly** (small → whole)
- **Byte-ordering** issue
    - Big endian: MSB → LSB
    - Little endian: LSB → MSB
    - Provide well-defined endianness within their message formats
    - XDR - eXternal Data Representation layer
        - If messages match the endianness of XDR, they are sent
        - Otherwise they are converted
- Whether to expose asynchronous nature of communication to clients to enable performance optimizations
    - Typical RPCs are synchronous, the client must wait after issuing

        - Some RPCs enable you to invoke and RPC asynchronously
            - RPC package sends and returns immediately
            - Client later needs to call back into the RPC to get results

WIKI - Representational State Transfer (REST)

- **Representational State Transfer (REST)** or **RESTful**: web services that provide interoperability between computer systems on the Internet
- REST-compliant web services allow requesting system to access and manipulate textual representations of web resources using a uniformed and predefined set of stateless operations
- **"Web resources":** documents or files identified by their URLs
    - NOW: every thing or entity that can be identified, named, addressed, or handled on the web
- **RESTful web service:** requests made to resources URL will elicit response in XML, HTML, JSON, or some other format
    - Response may:
        - Confirm some alteration has been made to the stored resource
        - Provide hypertext links to other related resources or collection of resources
- Aim for fast performance, reliability, and ability to grow by reusing components that can be managed and updated without affecting the system as a whole

Architectural Properties

- Constraints of REST architectural style affect the following properties:
    - Performance: component interaction factor in efficiency
    - Scalability: large number of components and interactions
    - Simplicity: uniform interface
    - Modifiability: component modification
    - Visibility: communication between components
    - Portability: moving program code
    - Reliability

Architectural Constraints

- 6 guiding constraints that define RESTful applications

➔ **Client-Server Architecture**

- Separation of concerns, user interface concerns and data storage concerns
- Improves portability and scalability
- Allows components to evolve independently

➔ **Statelessness**

- No client context is stored on the server between requests
- Each request contains all information necessary to service the request
- Session state is held in the client

  - State can be transferred to other services to maintain persistent state and allow authentication
  - Client sends requests when it is ready to transition
  - With more than 1 request, the client is in transition

➔ **Cacheability**

  - Clients and intermediaries can cache responses
  - Must define themselves as cacheable or not
    - Prevent clients from reusing stale or inappropriate data
  - Caching eliminates some client-server interactions

➔ **Layered System**

  - Client usually cannot tell whether it is directly connected to the server or an intermediary
  - Intermediaries can:
    - Improve performance by enabling load balancing or providing shared caches
    - Or enforce security policies

➔ **Code on Demand (optional)**

  - Servers can temporarily extend or customize functionality of client by transferring executable code

➔ **Uniform Interface**

  - Fundamental to REST service design
  - Simplifies and decouples the architecture
  - Four constraints
    - Resource identification requests
    - Resource manipulations through representations
    - Self-descriptive messages
    - Hypermedia as the engine of the application state

Kampe - Lease-Based Serialization

  - Leases are a better approach to serialization in distributed systems

Challenge of Distributed Systems

  - Fallacies of distributed computing included zero latencies, reliable delivery, stable topology, consistent management
  - Locking operations in distributed systems violate these fallacies
    - compare-and-swap/obtaining a lock take a long time
    - Mutex operation requests or responses can be lost
    - If a node holding a lock crashes without releasing it, other actors will hang indefinitely
    - If a process die, new meta-OS will observe the failure
  - Other issues:
    - No WAN-scale atomic instructions or interrupt disables
    - Operations may not be orderable

Addressing These Challenges

- Distributed consensus and multi-phase commits are complex
    - Too expensive for every locking operation
- Easier and more efficient to send all locking requests to a single server that implements them with local locks
- Replace locks with **leases** to deal with complex failure cases and greater deadlock risks
- Lock: grants owner exclusive access until he releases it
- **Lease:** grants owner exclusive access until he releases it or the lease expires
- For normal operations, a lease is just like a lock
- An important difference is that locks work **on the honor system**
    - Leases are often **enforced** and sent with the request
    - If the lease has expired, resource manager will refuse the request
- When a lease is no longer valid, the operation from the owner will no longer be accepted
    - Lease can be given to someone else
- **Two concerns with leases**
    1. **Expired leases stops the owner**
        - Owner may have been in the middle of multi-update transaction
            - Now left in an inconsistent state
        - Updates should be an all-or-none transaction
        - Should fall back to the last consistent state if it fails
    2. **Need to tune the choice of a lease period**
        - Too short → may need to renew too often
        - Too long → long recovery time after a failure
- Using leases for long-lived resources can be economical
    - Number of operations that can be performed under a single lease
    - Ratio of cost to obtain and the cost of operation

Evaluating Leases

- Mutual exclusion: at least as good as a lock, maybe better
- Fairness: depends on remote lock manager policies
- Performance: expensive by nature, but we don't lease all objects
    - Can be efficient if leases are rare
- Progress: immune to deadlock
- Robustness: more robust
- Leases still have issues
    - Recovery from lock-server failures is complex
    - Raises issue of checking time in distributed systems

Opportunistic Locks

- **Opportunistic locks:** requestor can ask for long term lease and handle it as a local lock until another node requests the resource
    - Lease is recovered and subsequent lock operations are dealt with by the centralized resource manager

- If contention is usually rare, use this approach

Summary

- Leases are better for complex situations

WIKI - Distributed Consensus

- Achieving system reliability in presence of faulty process is a fundamental problem of distributed computing
- Requires processes to agree on some data value used in computation

Problem Description

- Consensus problem requires agreement among a number of processes or agents for a single data value
- Processes may fail or be unreliable
    - Consensus protocols must be fault tolerant or resilient
- **Must put forth their candidates for a consensus value, communicate with one another, and agree on a single value**
- This is a fundamental problem
- One approach: generate consensus by having all processes agree on a majority value
    - Requires over one-half of the available votes
    - Faulty processes may skew resultant outcome so consensus may not be reached
- Protocols to deal with faulty processes
    - Deal with a limited number of faulty processes
- Protocols need to satisfy a number of requirements to be valid
    - Output of some protocol must be the input of another process
    - Process may decide upon and output a value only once and this decision is irrevocable
- Processes are correct in execution if they don't experience failure
- Consensus protocol tolerating halting failures satisfy the following properties:
    - **Termination:** every correct process decides some value
    - **Validity:** if all processes propose the same value v, then all correct processes decide v
    - **Integrity**: every correct process decides at most one value, if it decides some value v, then v must have been proposed by some process
    - **Agreement:** every correct process must agree on the same value
- A protocol that can guarantee consensus among n processes of which at most t fail is called **t-resistant**

Models of Computation

- Two types of failure:
    - **Crash failure:** process abruptly stops and does not resume

        - **Byzantine failure:** absolutely no conditions are imposed and there is failure
            - EX: an attacker
            - Far more disruptive
    - Consensus protocols that tolerate Byzantine failures must be more resilient
    - A consensus tolerating Byzantine failure:
        - Same termination
        - Same validity
        - **Integrity:** if a correct process decides on v, then v must have been proposed by some other correct process
        - Same agreement
    - Different models of computation may also define consensus problems
    - When input domain is much larger than the number of processes
        - In a synchronous message passing model, consensus may be impossible
    - In fully asynchronous message-passing distributed systems
        - If one process as a halting failure, consensus is impossible in worst-case scenarios
    - Asynchronous models
        - Failures can be handled by a synchronous consensus protocol
    - Synchronous model
        - Communication is assumed to be processed in rounds
        - One round: process sends all messages and receives messages
        - Message from one round → may influence any messages within the same round

Equivalency of Agreement Problems

- Three agreement problems

→ **Termination Reliable Broadcast (The General's Problem)**

- Collection of n processes numbered 0 to n-1, communicate by sending messages to each other
- Process 0 must transmit a value v to all processes such that
    1. If process 0 is correct, then very correct process receives v
    2. For any two correct processes, each process receives the same value

→ **Consensus**

- Formal agreements may include
    - **Agreement:** all correct processes must agree on the same value
    - **Weak validity:** if all correct processes receive the same value, then they all must output the same value
    - **Strong validity:** for each correct process, its output must be the input of some correct process
    - **Termination:** all processes must eventually decide on an output value

→ **Weak Interactive Consistency**

- For n processes in a partially synchronous system, each process chooses a private value

- The processes communicate with each other by rounds to determine a public value and generate a consensus vector with the following requirements
    1. If a correct process sends v, then all correct processes receive either v or nothing (integrity)
    2. All messages sent in a round by a correct process are received in the same round by all correct processes (consistency)

In Shared-Memory Systems

- Need to have concurrent object
    - A data structure that helps concurrent processes communicate to reach an agreement
- Two methods
    - Traditionally use a critical section, only one process can access the object at a time
        - May crash
    - A wait-free implementation
- Consensus hierarchy and number
    - Number is the max number of processes in the system that can reach consensus using the given object

Distributed System Security

Introduction

- Challenges in providing security in distributed system
- Operating system can only control its machine's resources
- **Two large problems:**
    1. Other machines may not properly implement policies we want or may be adversaries
        - Need to trust the validity of credentials and capabilities
    2. Machines communicate across a network none of them fully control
        - Generally the network cannot be trusted
        - Adversaries often have equal access to networks
- Use cryptography and other standard tools

The Role of Authentication

- Try and arrange to agree on policies and hope computers follow that agreement
- Usually difficult to get evidence of everyone agreeing
- Have to trust that some parties will behave well
- Can detect if they do or don't and adjust the trust or take some other action
- Need authentication of messages being sent across the network
- Distributed system authentication relies usually on one of two things:
    - Require remote machine to provide a password
    - Require it to provide evidence using a private key
- You need to know one of 2 things:
    - Password

        - Public key
    - **Passwords are useful when vast numbers of parties authenticate to one single party**
        - Provides evidence that someone knows the password
        - Party checking and party authenticating will know who it is
    - **Public key is useful if one party needs to authenticate to many parties**
        - Only one party can authenticate by knowing the private key
        - Many parties can know the public key
    - Both are used in different situations
        - Website authenticate itself to users → public key
        - User authenticate themself to the site → password
    - **The public key is needed for both methods**

Public Key Authentication for Distributed Systems

- Doesn't need to be a secret but we need to verify **who owns it**
- Try using a **third party** known to everyone
    - Have them authenticate that the party used their own public key
    - This signature verifies that the bits are the correct public key
- Signed bundle of bits is called a **certificate**
    - Contains information on:
        - Party that owns the public key
        - Public key itself
        - Other information
    - Ran through a cryptographic hash
    - Result is encrypted with the third party's private key - **signing the certificate**
- Obtaining a copy of the certificate = learning someone's public key
- Certificates are obtained from **certificate authorities**
- Certificate example:
    - Frobozz wants a certificate for its public key K^F
    - Acmesign runs a cryptographic algorithm on
        - Frobuzz's name
        - Public key K^F
        - And other information
    - This produces H^F
    - Acmesign encrypts H^F with its own private key P^A to produce a digital signature S^F
    - Combine all the information from H^F, Acme's own identity, and S^F into C^F
    - This is given back to Frobuzz
    - Frobuzz wants to authenticate itself over the internet
        - To get a public key, obtain C^F
    - Customer gets C^F and sees that it was run under SHA-3
    - Hashes the information provided (name, public key, etc.) through SHA to produce $H^{F'}$

        - Customer decrypts S^F using the public key of the certificate authority to get H^F
        - If H^F == H^F', everything is good
    - Properties of this approach:
        - Signing authority and company didn't participate in checking the certificate
        - Only needed one per customer
            - Just need to store the public key after
        - No need to trust company until after its identity has been proven by checking the certificate
    - Q: where do we get the public key of the certification authority?
        - Usually comes with some software you obtain and install like your browser
    - Anyone can create a certificate
        - But it must be that party being authenticated and party performing authentication trusts the person or company creating the certificate
    - Purpose of the certificate is to **distribute the public key**
    - **Replay attack:** an attacker copying the legitimate messages the site sent at one point and sending it again another time
    - Preventing these means that ensuring each encrypted message contains unique information not in any other message
        - Built into standard protocols
    - Public key cryptography is expensive

Password Authentication for Distributed Systems

- Works best in situations where only two parties need to deal with any particular password
- Works when an individual user is authenticating himself to a site that has many users
- Password is usually associated with a particular user ID
- Password is sent over a network
    - Need to add confidentiality over cross-network authentication
- Encrypt with the site's public key

SSL/TLS

- Add cryptographic features to sockets – the **SSL (Secure Socket Layer)**
- SSL is not fully correct
- A new version, **Transport Layer Security (TLS)**
- SSL is old an insecure
- SSL = SSL/TLS = TLS
- Concept behind SSL is moving **encrypted** data through ordinary sockets
    - Set up the socket
    - Encrypt on one side
    - Reverse on the other side
- Adding SSL is intricate

    - Need libraries and sequence of calls to set up an SSL connection
    - Allow a wide range of generality in options and usage
- Common requirement in setup is **securely distributing the cryptographic key you will use**
    - Brand new key to encrypt for each connection set up
    - Use symmetric cryptography to encrypt after authenticating
- To set up SSL
    - Bootstrap a connection based on symmetric cryptography
    - Set up negotiation to determine a set of cyphers and techniques that balance security and performance
- Use **Diffie-Hellman** key exchange to create the key
- Need to ensure who we are sharing the key with
    - Client obtains a certificate containing servers public key and uses the public key to verify authenticity
    - Transfer method **is not relevant to certificate security, the cryptography embedded is**
- Using certificate, we can start the Diffie-Hellman key exchange
    - Server signs Diffie-Hellman message with the private key
    - Client can determine that the partner is the right server
    - Client usually cannot sign the message
        - **Client knows who server is but not vice versa**
- Client usually uses password to authenticate itself
- Final word: this is a protocol, not a software package
    - Implemented by software packages

Other Authentication Approaches

- Other options to authenticate yourself
    - After entering the password once, the site assumes you are valid
- Site may make a security tradeoff
    - Verifies your identity in the past using your password
    - Relies on some other message in the future
- **Web cookies** is a common method
    - Pieces of data a website sends to the client
    - Client saves the data and sends it again later
- Usually built-in
- Some security problems:
    - An eavesdropper
    - Non-legitimate user may use machine
- challenges/response protocol
    - Remote machine sends you a challenge
    - To authenticate, perform some action on this challenge
    - May not need to be encrypted
    - Requires pre-arrangement
- Authentication server

        - Talk to a server
        - You trust it and the other party trusts it
        - Server vouches for your identity in a secure form and party checks it
    - Different types: online/offline

Some Higher Level Tools

- Work at a higher level: HTTPS/SSH

→ HTTPS

- Takes HTTP and adds SSL/TLS
- Basically taking HTTP and passing it through an SSL connection
- Relies heavily on authentication using certificates

→ SSH

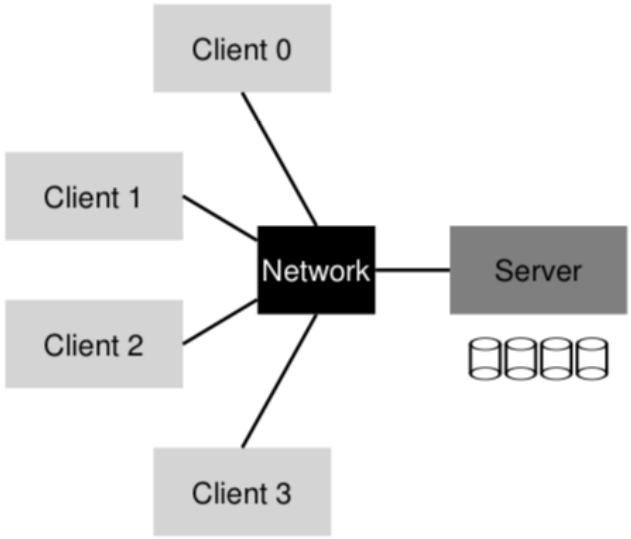
- “Secure Shell”
- Secure remote shell
- Allows secure interaction between computers

Remote Data Architectures

Reading

Arpaci-Dusseau Ch. 49 - Sun's Network File System (NFS)

- One of the first uses of **distributed client/server computing in distributed file systems**
- **Distributed file system**
  - **Client machine:** requests data through messages
  - **Server machine:** stores data on its disks



- Allows for easy sharing of data across clients
  - All have the **same view of the file system**
- Centralized administration
  - Backing up files done in servers instead of on the many clients
- Security benefits

49.1 A Basic Distributed File System

- client/server distributed file system has more components than a regular file system
- Client side application:
  - Client applications that access files and directories through the **client-side file system**
  - Client issues system calls to the client-side file system to access files stored on the server
  - File system doesn't appear any different to the client (except in performance)

- Client side file system
  - Role: execute the actions needed
  - Send message to the **server-side file system** or the **file server**
    - The **server-side file system** fetches/writes data to/from the disk and sends messages back to the client
  - Caching may occur

49.2 On to NFS

- Sun Network File Systems
  - Early and successful distributed system
- Build an **open protocol** instead of a proprietary and closed system
  - Specified the exact message formats clients and servers would use
- Different groups could develop their own WFS servers and compete

49.3 Focus: Simple and Fast Server Crash Recovery

- Classic NFS protocol, NFSv2
- Main goal: simple and fast server crash recovery
  - Makes sense in a multi-client, single server situation
  - If a server goes down, the entire system will go down

49.4 Key to Fast Crash Recovery: Statelessness

- Goal is realized by designing a **stateless** protocol
  - Server **doesn't keep track of anything** that is happening at each client
  - Designed to deliver **all the information needed** to complete a request in each protocol request
- EX: stateful protocol (not stateless)

```
int fd = open("foo", O_RDONLY);
read (fd, buffer, MAX);
...
read (fd, buffer, MAX);
close (fd);
```

- Client side opens the file using the open call
- The **descriptor** is sent back to the client **after being opened locally on the server**
  - This descriptor is used by the client later
- The descriptor is a **shared state** between the server and client
  - Will complicate crash recovery
  - File descriptor can be lost when the server crashes
  - May have to engage in some kind of recovery protocol

- Stateless approach: each client operation contains **all the information needed to complete the request**
    - May need to retry, but will not need recovery

49.5 The NFSv2 Protocol

- Problem: defining network protocol to enable stateless operation
- **File handle:** used to uniquely describe the file/directory a particular operation is going to operate on
    - Three components:
        - **Volume identifier:** which file system the request refers to
        - **Inode number:** which file in that partition
        - **Generation number:** reuse of inode numbers
            - Increment when reused
            - Client with an old file handle can't access newly-allocated files
    - Comprise a **unique identifier** for a file/directory a client wants to access
- Some protocol examples:
    - LOOKUP: used to obtain a file handle which is used to access data
    - READ: requires the file handle and an offset and a number of bytes
    - WRITE: similar to read
    - GETATTR: fetches file attributes for a given file

44.6 From Protocol to Distributed File System

- **Client-side file system:** tracks open files, translates application requests into protocol messages
- **Server:** responds to protocol messages
    - Each contains all the information needed to complete a request
- Client tracks relevant **state** for file accesses
    - Maps an integer file descriptor to an NFS handle
    - Current file pointer
    - Enables the client to turn read requests into protocol messages
- Subsequent reads from the same file handle use a different file offset
- Server interactions:
    - When the file is opened for the first time, use a LOOKUP message
    - For long path names, multiple LOOKUPs can be sent
- Server has all necessary information
    - Enables graceful recovery

49.7 Handling Server Failure with Idempotent Operations

- Many possible reasons for failure
    - Messages can be dropped
    - Servers can crash → eventually they are rebooted
    - What should clients do?
- Client handles server failures in a single way: **retry the request**
    - When making a request, set a time to go off after a certain period

        - If a reply is received before the timer goes off
            - Timer is cancelled
        - If the timer goes off before any reply is received
            - Client assumes that the request was not processed
            - Resends it
- Ability to retry a request depends on a property of NFS requests - **they are idempotent**
    - **Idempotent operation:** operations wherein the effect of performing the operation multiple times is equivalent to the effect of performing the operation a single time
- In general, reads are idempotent and data updates need to be examined more closely
- Idempotency in NFS of common operations
    - LOOKUP/READ are idempotent
    - WRITE is also idempotent
        - Contains the data, count, and also **the offset**
            - Because of the offset, writes can be repeated
- Some operations are not idempotent
    - MKDIR
    - Cannot make a directory twice

49.8 Improving Performance: Client-Side Caching

- Network is generally not fast
- Improve performance with **client-side caching**
- Client-side file system caches file data and metadata from server reads **into client memory**
- Cache also works as a temporary buffer for writes
    - Client writes buffer before writing out
    - **Write buffering** decouples application write() latency from actual performance
        - Looks like the write works immediately
        - Eventually data will get written to the server
- New problems introduced: **cache consistency problems**

49.9 The Cache Consistency Problem

- **Update visibility:** when do updates from one client become visible to others
- **Stale cache:** one client has finished its write but another client still has the old version
- Solve these two problems in 2 ways
- Two address update visibility: clients use **flush-on-close** or **close-to-open** consistency semantics
    - When a file is written and closed, **client automatically flushes all updates**
    - Ensures another open will be updated
- To address the stale-cache problem:

    - Clients **check if file has been changed before using the cached contents**
    - Use a GETATTR request to fetch attributes
        - Includes the last modified time
        - If the last modified time is more recent than the time that the file was fetched, **the client invalidates the file**, removing it from the cache
    - Introduces another problem: **the NFS server is flooded with GETATTR requests**
        - Attribute cache is added, including a timeout for the attribute cache entries

49.10 Assessing NFS Cache Consistency

- Flush-on-close introduced a performance problem
    - temporary/short-lived files would be flushed to server when deleted
    - Keep short-lived files in memory until they are deleted and remove the server interaction entirely
- Addition of the **attribute cache**
    - Made it hard to understand which version of a file you were accessing
    - Sometimes leads to odd behavior

49.11 Implications on Server-Side Write Buffering

- Write buffering
- NFS servers cannot return success on WRITE until write is in stable storage
    - Makes sure that clients see the correct data in the case that the server crashes
    - Write performance can cause major part of bottleneck

Arpaci-Dusseau Ch. 50 - The Andrew File System (AFS)

- Produced at CMU in the 1980s
- Goal of this project was **scale**
- Wanted to design a distributed file system such that a server can support as many clients as possible
- Numerous aspects of design and implementation affect scalability
- Design of protocol between client and server is important
    - In NFS: the protocol forces clients to check with the server periodically
    - Each check uses server resources
        - In turn, limits the number of clients a server can respond to
        - Hurts scalability
- AFS: differs because **reasonable user-level visibility is a first-class concern**
    - NFS: cache consistency is hard to describe
        - Depended on low-level implementation details
    - AFS: cache consistency is simple and readily understood
    - When a file is opened, a client receives the latest consistent copy

50.1 AFS Version 1

- Two versions of AFS
- AFS1 → AFS2
    - Re-design and final protocol is AFS2
- Basic tenet of all versions of AFS is **whole-file caching** on **the local disk of the client machine** that is accessing a file
- open() a file
    - The entire file is fetched and stored on your local disk
    - Subsequent read() and write() calls are redirected to the local file system
        - read() and write() require no network communication and are thus fast
- Upon a close(), if the file is modified, it is flushed back to the server
- Contrast with NFS which caches **blocks in client memory,** not local disk
- Upon calling open(), AFS client-server code (**Venus**) sends a fetch protocol message to the server
    - Passes the entire pathname of the desired file to the server (**Vice**)
    - Vice traverses the path name, finds the file, and sends the entire file to the client
- Client caches the entire file in local disk
- Subsequent read() and write() calls are **local in AFS** (no communication)
    - Blocks can be cached in client memory after read() and write() calls
    - Act like **calls to a local file system**
- When finished/close() is called, the AFS checks if the file has been modified and if so, flushes with a store protocol message
    - Send file and pathname to server for permanent storage
- Next access is more efficient
    - Client-side code contacts the server with a TestAuth message to determine if the file has changed
    - If no, the client uses the locally-cached copy

50.2 Problems with Version 1

- Two main problems with AFSv1:
    - **Path traversal costs are too high**
        - Entire pathname is passed to the server
        - Full pathname traversal must be performed by the server
        - Too much CPU time is spend just walking down directory paths
    - **Client issues too many TestAuth protocol messages**
        - Like NFS's overabundance of GETATTR messages
        - Much time is spent telling clients whether or not files have changed
- Two other problems:
    - Load not balanced across servers
    - Server used a single distinct processor/client

        - Induce context switching and other overheads
- Load imbalance solved by introducing **volumes**
    - **Moving data across servers**
- Context switched solved in AFSv2 by building server with threads instead

50.3 Improving the Protocol

- The two major problems limited scalability
- Bottleneck: server CPU

50.4 AFS Version 2

- Introduced a **callback** to reduce the number of client/server interactions
- **Callback:** promise from server to the client that the server will inform the client when a file that the client is caching has been modified
- By adding this state to the system
    - Client no longer needs to contact the server to ask if the file is still valid
- Client will assume that the file is valid until otherwise
    - Analogous to polling vs. interrupts
- Introduced the **file identifier (FID)** like the NFS **file handle** instead of using pathnames to specify files
    - Consists of a volume identifiers, a file identifier, and a “uniquifier”
        - If a file is replaced but the client doesn’t know, can check uniquifier
            - Otherwise,everything else may be the same
        - Enable reuse of the volume and file ID when a file is deleted
- Instead of sending entire pathnames, the client walks the pathname and caches the result to reduce server load
- EX: client accesses /home/remzi/notes.txt
    - Client fetches home, puts it on the local-disk cache
    - Set up call back on home
    - Then fetches remzi, set up callback on remzi, etc.
    - Finally fetches notes.txt
    - Caches it and sets up the callback
    - Returns a file descriptor
- Key difference: with each fetch of a directory or a file, AFS establishes a callback with the server
    - Ensures that the server will notify the client of a change in its state
- Benefit: first access to /home/remzi/notes.txt generates many client-server messages
    - Also establishes callbacks for all directories and the file
    - Subsequent accesses are **local** and require no server interaction
- In the common case: the file is cached at the client
    - AFS acts like a local disk-based file system

50.5 Cache Consistency

- In NFS, two aspects of cache consistency were considered: update visibility and cache staleness
- Cache consistency problem in AFS is easy to understand because of callbacks and whole-file caching
- Two important cases:
    - Consistency between processes on **different machines** and **the same machine**
- Different machines:
    - AFS makes update visible at the server and invalidates cached copies at the exact same time
        - When the updated file is closed
    - Client opens a file and writes to it
        - When it is closed, the new file is flushed to the server and is visible
        - Server “breaks” callbacks for any clients with cached copies
            - Accomplished by contacting each client and informing the callback they had for that file is no longer valid
        - Ensures that clients don’t read stale files
- Same machine:
    - Writes to a file are immediately visible
    - Behaves like you would expect (UNIX)
- Cross-machine case where processes on different machines are writing/modifying a file at the same time
    - AFS employs a **last writer wins** or **last closer wins** approach
    - Client that calls close() last updates the entire file on the server last
    - Result: file generated **entirely by one file or the other**
    - Different from NFS where **blocks are flushed out**
        - The result of which could be a mix of both clients

50.6 Crash Recovery

- More involved than in NFS
- If a client crashes, the client should first use TestAuth protocol message to check if cache is still valid
    - In case it missed any updates
- Server crashes are more complicated
    - Callbacks kept in memory are now gone
        - No idea which client machine had which files
    - Each client must realize server has crashed
        - Treat all cache contents as suspect
        - Establish validity
- Implement:
    - Server will send a message after crash
    - Clients send a heartbeat message

50.7 Scale and Performance of AFSv2

- More scalable than original
- Client-side performance is **close to local performance sometimes**
    - In the common case, files are local
    - Reads usually are in local disk cache or local memory
    - Creating new files/writing to a file → need to send message to the server
- Comparing AFS and NFS
    - Performance of each system usually roughly equivalent
    - Large-file sequential reread: AFS performs better
    - Sequential writes of new files have similar performance
    - AFS is worse on sequential overwrites
        - Fetch old files and rewrite them
    - Workloads that access a small subset of data within large files
        - Performs better on NFS

50.8 AFS: Other Improvements

- Provides true global namespace
- Takes security seriously
    - Incorporates authentication and privacy
- Facilitates flexible user-managed access control
    - User has control over who can access files
- Simpler management of servers for administration