UCB CS 162 - Operating Systems and Systems Programming

CS162 - Operating Systems and Systems Programming

Taught by Prof. John Kubiatowicz at UC Berkeley in Fall, 2020

Course Website
Fall 2020 Course Website
Lecture Videos (YouTube)

Lectures

1. Lecture 1 - What is an Operating System?
2. Lecture 2 - Four Fundamental OS Concepts
3. Lecture 3 - Abstractions 1: Threads and Processes
4. Lecture 4 - Abstractions 2: Files and I/O
5. Lecture 5 - Abstractions 3: IPC, Pipes, and Sockets
6. Lecture 6 - Synchronization 1: Concurrency, Mutual Exclusion
7. Lecture 7 - Synchronization 2: Semaphores, Lock Implementation
8. Lecture 8 - Synchronization 3: Atomic Instructions, Monitors, Readers/Writers
9. Lecture 10 - Scheduling 1: Concepts & Classic Policies
10. Lecture 11 - Scheduling 2: Case Studies, Real Time, Forward Progress
11. Lecture 12 - Scheduling 3: Deadlock
12. Lecture 13 - Memory 1: Address Translation and Virtual Memory
13. Lecture 14 - Memory 2: Virtual Memory, Caching and TLBs
14. Lecture 15 - Memory 3: Caching & TLBs, Demand Paging
15. Lecture 16 - Memory 4: Demand Paging Policies
16. Lecture 17 - Demand Paging, General I/O, Storage Devices
17. Lecture 18 - General I/O, Storage Devices, Performance
18. Lecture 19 - Filesystems 1: Performance, Queueing Theory, Filesystem Design
19. Lecture 20 - Filesystems 2: Filesystem Design, Case Studies
20. Lecture 21 - Filesystems 3: Case Studies, Buffering, Reliability, Transactions
21. Lecture 22 - Transactions, End-to-End Arguments, Distributed Decision Making
22. Lecture 23 - Distributed Decision Making, Networking, TCP/IP

Lecture 1: What is an Operating System?

An operating system is a piece of software that provides other software access to the system's hardware resources, and makes it seem much more convenient and less complex. It also gives applications isolated environments with useful abstractions of hardware resources.

It is like an illusionist. It provides clean, easy-to-use abstractions of physical resources and makes it seem to the user as if their machine has infinite memory, and makes it seem to the application that it has the entire machine to itself.

The operating system doesn't directly expose the processor and its registers to an application. Instead, it exposes "threads", which an application runs on. An application can run on one or more than one thread(s). It does not directly expose the memory to an application. Instead, it exposes "address spaces", "files", and "directories". It does not directly expose the network adapters and cards to the application. Instead, it exposes "sockets".

On top of threads, the OS packages each running application into its own "process". This makes it seem to the application that it has the entire machine to itself, because it isolates the application from every other application running on the machine. You can think of a process as and isolated execution environment with restricted rights provided by the operating system.

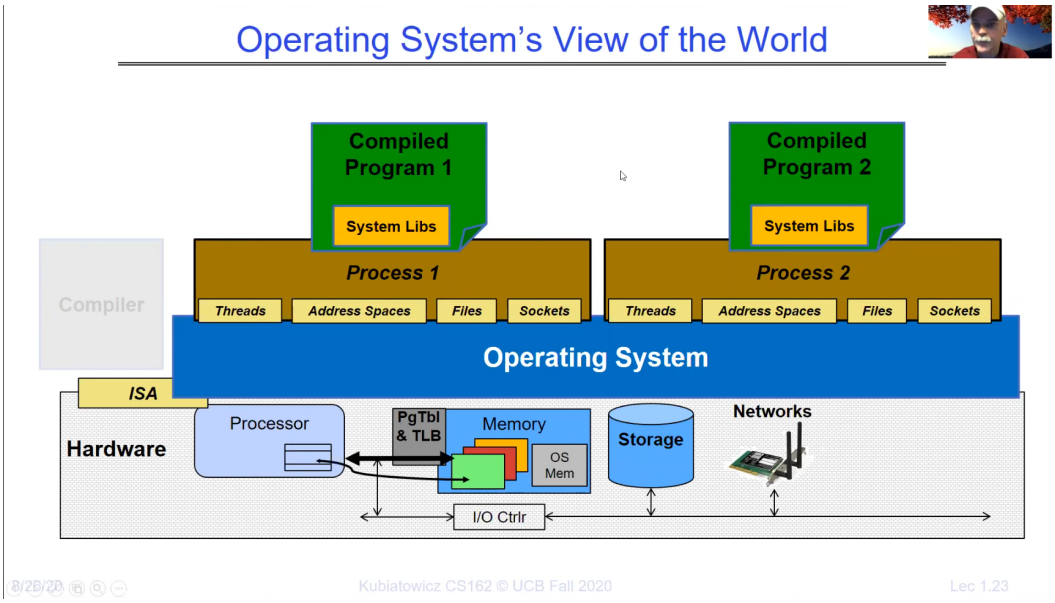


Figure 1: Image

Lecture 2: Four Fundamental OS Concepts

This lecture introduces four fundamental concepts of an operating system - 1. Threads: Execution Context 2. Address Spaces 3. Processes 4. Dual mode operation

Threads

A thread is a single unique execution context that has its own program counter, registers, execution flags, stack, and so on. This means that the block diagram you have studied for how a processor executes a program is really one single thread executing its own code.

A thread is said to be executing when it is "resident" in a core. Resident means that the PC of the core is pointing at the next executable instruction for the thread and the instructions of the thread are stored in memory. A resident thread's stack pointer holds the address of the top of the stack for that thread.

A thread is said to be suspended when its data (everything we have seen above) is not in memory.

The OS gives the user the illusion of multiple processors by multiplexing the execution of threads in time. All threads that want to run on the processor are stored in a chunk of memory called the Thread Control Block, and the data for each thread is swapped in and out of the hardware registers over time.

Swapping out the data of one thread for another takes some time, which is called context switching. We have to be careful not to swap between executing threads so often that majority of the time is spent in context switching, since the more context switching we perform, the more time is "wasted". However, if we don't context switch fast enough, we might get the feeling that our machine is unresponsive, since only one process would be running on the CPU for a long time, making all other processes hang.

Context switching can also happen because the thread voluntarily gave up control, or it asked for some i/o, and so on.

Address Spaces

An address space is a set of addresses that a thread or process can access. A 32-bit processor can access 2^{32} spaces, which is around 4 billion. A 64-bit processor can access 2^{64} address spaces, which is 10^{18}

When a thread or a process writes to an address space, one of many things can happen, depending on the type of memory location. A memory location can behave as -

1. A regular memory location, i.e. data written is stored in memory

2. Memory-mapped I/O, i.e. data written is mapped to I/O ports
3. Communication between two or more programs

An address space for a thread or process contains the parts shown above. The code segment contains the instructions that are being executed, the static data contains data that does not change throughout the life of the program (e.g. global variables), the heap is used to store dynamically allocated memory, and the stack is used to store local variables.

However, since this address space lives along with other address spaces around it, we want to make sure that no other thread or process can access this address space. We also want to make sure that this thread cannot access any other address space, especially address/memory where the OS lives.

This can be done using a simple check during each memory access. When the thread accesses a memory location, the OS first checks if the memory location is within the address space allocated to this thread. If it is not, then the thread or process can be killed. The address space therefore has a "base" (the start of the address space), and a "bound" (the end of an address space).

This simple model has lots of drawbacks, but the biggest one is fragmentation.

To fix that, we use a paged virtual address space. We divide the entire virtual address space into equal size chunks called pages. Each page has a base, and the bound is the global page size. All pages are of the same size, so it is easy to manipulate memory this way.

The hardware then translates addresses using a page table. A special hardware register stores a pointer to the page table. This lets us treat memory as a set of pages that are put into page frames. The thread or process operates on virtual addresses, and the virtual addresses are translated to hardware addresses by the page table.

Processes

A process is an isolated execution environment with restricted rights. It has an address space and contains one or more threads. You can think of it as an encapsulation of a running program. The process has an address space that is isolated from every other address space, but the threads inside the process share everything, including memory, hardware resources, network, i/o, etc.

This lets us do concurrency easily. Multiple threads in a single process can run on different cores (if available).

But one question that needs to be addressed is what stops a process from changing some data that it owns, that should not be changed (for example, the page table pointer for this process)? If a process could change its own page table pointer, the security offered by the concept of address spaces would mean nothing.

This is solved by the fourth concept: privilege levels and dual mode operation.

Dual Mode Operation

The hardware itself has at least two modes, kernel mode and user mode, which essentially give us privilege levels. Almost all processes installed in the machine can only run in user mode, which stops them from accessing/changing sensitive data. Certain operations are prohibited when a process is running in user mode, for example changing the page table pointer, disabling interrupts, interacting with the hardware directly, or accessing any memory outside its address space.

If the process needs to perform a "sensitive" operation, the OS provides a controlled transition between user mode and kernel mode.

The OS can itself switch between user mode and kernel mode if any of the following happen -

1. syscall
2. Interrupt
3. Exception

Lecture 3 - Abstractions 1: Threads and Processes

Multiprocessing vs. Multiprogramming

Multiprocessing vs. Multiprogramming is similar to the idea of concurrency vs. parallelism. Multiprocessing is when you have multiple physical cores, and you use them to perform operations at the same time completely independently. Multiprogramming is when you divide a program into multiple jobs/processes/threads.

Concurrency is when you are *handling* multiple things at once, not necessarily *doing* them. Parallelism is when you do multiple things *simultaneously*.

Threads

A thread can be in one of 3 states - running, ready, or blocked. A running thread is a thread whose instructions are actually being executed, a ready thread is a thread that is waiting for CPU time, and can be swapped out with the current running thread, and a blocked thread is a thread that is waiting for something to happen before it can run (typically i/o).

To write a multithreaded program in C, we usually use pthreads. The "P" in pthreads stands for POSIX. POSIX is an attempt at standardizing the programming interface that different operating systems provide to programs running in user mode.

Usually the structure of a multithreaded program is that we have a "main" thread that is created when the process is created. The main thread creates as many new threads as it needs. These threads run, do their own thing, and then "join" with the main thread. The main thread then continues to run.

All threads in a process share the process's address space, but they have their own independent stack. These stacks grow and shrink independently, and sometimes may run into each other. In this case, the OS decides what to do. The thread may be killed, or its stack may be reallocated.

The important part about multithreaded programming is correctness. The program you write must be able to handle any interleaving of thread execution that the OS scheduler gives you. One way to ensure correctness is mutual exclusion.

If more than one thread is going to access some data, the thread accessing that data acquires a "lock" over that data. This ensures that only that thread can read or write to that location. Once the thread has done whatever it needs to do with that data, it releases the lock. The part of code that exactly one thread can access at once is called the critical section.

Processes

All processes are created by other processes. If that is the case, how is the first process created? The first process is called the "init" process, and it is created by the kernel.

The OS also provides a process management API, that allows our programs to manage other processes. The common functions provided by this API are -

1. `exit` - Terminate a process
2. `fork` - Create a complete copy of the current process
3. `exec` - Change the program being run by the current process
4. `wait` - Wait for a process to finish
5. `kill` - Send an interrupt-like notification to another process
6. `sigaction` - Set handlers for signals

Lecture 4 - Abstractions 2: Files and I/O

Semaphores

Semaphore are a kind of generalized locking mechanism. A semaphore is just a number associated with critical section of code. It was first defined by Dijkstra in the 60s. A semaphore is a non-negative integer that supports the following operations -

1. `P()` or `down()` - An atomic operation that waits for the semaphore to become positive, then decrements it by 1.
2. `V()` or `up()` - An atomic operation that increments the semaphore by 1, and then wakes up a waiting `P`, if any.

This concept of semaphores can be used in two patterns. One is for mutual exclusion, which is easy to see. We set the initial value of the semaphore to be 1. Each thread first checks the value of the semaphore before entering the critical section. If the value is greater than 0, the thread enters the section and decrements the semaphore. If the value is 0, the thread is put to sleep and waits for the semaphore to increment.

The other pattern is for waiting for a thread to finish execution and join the main thread (or some other thread). We set the initial value of the semaphore to 0. When the main thread tries to decrement it, it is put to sleep. Then, another thread finishes its execution and increments the semaphore. This wakes up the main thread, and the main thread then takes over the execution. Essentially, we made the main thread wait for some other thread to join.

Processes & Signals

A common way of having one program launch or quit other programs is using `exec()` and signals from the process management API. `exec()` stops the execution of the current process, and gives up control to some other process that is specified as an argument to `exec()`. This is how the shell starts new programs. Look at the example code below -

```
pid = fork();
if (pid == 0) {
    // This is the child process launched by the parent
    exec(...);      // Start some other (child) process
}
else {
    wait(&stat);   // Wait for the child process to finish and exit
}
```

Files

A key idea in Unix/POSIX is that everything is a file. This means that (almost) all system calls have an identical interface. POSIX gives an identical interface for -

1. Files
2. Devices (terminals, printers, etc.)
3. Networking sockets
4. Interprocess communication (pipes, sockets)

and more. This standard interface is based around the system calls `open()`, `read()`, `write()`, `close()`. If you are writing a device driver that doesn't necessarily fit into this interface, you can use `ioctl()`, which stands for I/O control. A "folder" is just a special type of file that maps to a set of other files/folders, instead of having any arbitrary content.

The high level abstractions of files is streams. A stream is just an unformatted sequence of bytes. Kubi then explains the API C gives us for working with streams/-files, which I won't write about here, since it is fairly simple and can be looked up if you need it.

File Descriptors

A file descriptor is just an integer that corresponds to an entry in a system-wide file description table that is maintained by the kernel. The kernel maintains a list of open files, and a list of processes and which files they have access to.

On a successful call to `open()`, which is a low level function to open files (or anything that behaves like a file, which is everything), a file descriptor for the corresponding file is returned to the user, and an open file description is created in the kernel.

For each process, the kernel maintains a mapping from the file descriptor to the open file description. On performing system calls (e.g. `read()`), the kernel looks up the open file description using the file descriptor and services the system call.

Since everything in POSIX is a file, stdin, stdout, and stderr also have file descriptors associated with them. The file descriptors associated with these are -

- stdin -> 0
- stdout -> 1
- stderr -> 2

Not only that, but this means that other hardware resources like sockets also have file descriptors associated with them.

File descriptors is how pipes can be implemented. If you take the file descriptor for the stdout for a process, and map it to the file descriptor for stdin for another

process (using the pipe function), then the output of one process is fed to the input of another process.

All calls to high-level file handling APIs like `fread()`, `fwrite()`, etc. are buffered at the user level. This is how they are able to provide us functionality like "read until the next newline". Moreover, all reads and writes are also buffered at the kernel level by the OS. Another reason for having buffers at both the user and kernel level is because buffering in user level is faster than having to access the kernel buffer, for which the process would need to go into kernel mode (or make a syscall).

When you `fork()` a process, all of its file descriptors are also copied. This means that both the parent and the child process can read and write to and from the same file. This is a useful way to have the processes communicate with each other. This works not only for files, but also for hardware resources like network sockets.

Lecture 5: Abstractions 3: IPC, Sockets

IPC

IPC stands for inter-process communication. The process model we have seen so far is specifically designed so that one process is not able to affect another in any way. This is by design, but it does make inter-process communication more difficult.

Therefore, we have to have a specialized way for two processes to communicate with each other, if both processes agree. One initial idea may be to communicate through reading and writing to files that are shared by processes. We have seen that forking a process creates file descriptors that are shared by both the parent and the child, so maybe one process can write to a shared file descriptor, and another process could read from it. The drawback to this approach is, as you might imagine, efficiency. Communicating with disk is really slow, and this approach becomes completely impractical as the number of processes increases.

The approach we use is to have an in-memory queue maintained by the kernel, which provides the same POSIX interface for reading and writing as reading and writing to a file. This is called a "pipe". You can create a pipe using the `pipe(int filedes[2])` syscall.

Sockets

When we extend the principle of IPC to two processes that are not running on the same machine, we get communication across the network. This involves a "network connection", which is essentially a pair of sockets connected to each other (for a bidirectional connection).

Actually transmitting data over the network is a whole another task, and this course does not deal with that. We assume that the messages we send over the network are correctly transmitted and reach the other device across the network.

Aside from the fact that these two sockets are on different physical devices, they work in the exact same way that a local socket (or pipe) works.

The client has a socket that is connected to a socket on the server. The client opens the socket (which may establish a TCP connection using a 3-way handshake), and can then read from it or write to it. Data written to the socket is transmitted over the network, arrives at the server, which reads it, and sends a response. The server then closes the socket once the client's request has been served.

In order to serve multiple requests at once, the server may create either a new process or a new thread for each request. Creating a new process for each request gives us better isolation, but is more heavyweight. Creating a new thread is faster, but doesn't offer the same isolation.

If a server creates a different thread for each request, then there is a chance that a spike in traffic can create so many threads that the throughput actually decreases. To avoid that, we create a group of threads called the "thread pool". The thread pool has a certain number of threads waiting to serve requests. We maintain a queue of requests that are waiting to be served, and when a thread from the thread pool becomes free, it dequeues the next request from the request queue. This way, there is an upper limit on how many threads can be created, and requests are still served concurrently.

Lecture 6: Synchronization 1: Concurrency, Mutual Exclusion

Concurrency

One of the most important and central parts of the operating system is the scheduler. The scheduler maintains a queue of the active processes in the system and chooses which process gets to go next.

The scheduler maintains a data structure containing the process control blocks, and distributes CPU time to the different PCBs it contains according to some policy. It can also distribute access to other hardware resources like memory, IO, etc.

The lifecycle of a process is shown in the diagram below -

The pseudocode of the scheduler basically looks like this -

```
while (1) {
    RunThread();
    ChooseNextThread();
    SaveStateOfCPU(currPCB);
    LoadStateOfCPU(nextPCB);
 }
```

One can even argue that this is all that the operating system does.

Context Switching

When the scheduler decides to run a thread, it essentially gives up control of the CPU to the process it decided to run. It needs some way to get that control back from the process.

This can happen because of internal events, i.e. the thread yields control voluntarily, or because of external events like interrupts. The POSIX API provides a syscall `ptherad_yield()` or `sched_yield()`, that give up control of the CPU.

When a process yields, it goes into the kernel, tells the scheduler to run a new thread, and then performs a context switch. The context switch basically saves the state of all registers and the stack of the first thread/process into the TCB/PCB, and loads in the registers and the stack of the new thread/process.

Switching between threads is much faster than switching between processes (about 30x faster).

What happens if a process never yields or never does any operation (like waiting for I/O or disk read/writes) that causes it to voluntarily give up control of the CPU? In primitive operating systems like Windows 3.1, this would cause the OS to crash.

Since the OS has completely given up control of the CPU to the currently executing process, there is no way for the OS to regain control of the CPU.

In that case, we use external interrupts. Today's computers have hardware timers that are configured to interrupt the CPU at a fixed interval. A hardware interrupt is designed to stop the currently executing process and transfer control back to the OS.

Mutual Exclusion

We have seen mutual exclusion in a previous lecture. This lecture goes over the concept again and shows some examples. When we write multithreaded code, a big problem is correctness. Since threads share memory, if multiple threads access the same memory at the same time, and if at least one of those threads is doing a write, then we run into a race condition. The result of the program in that case is unpredictable.

The way we fix this is by using locks or semaphores. Before entering a critical section of code where multiple threads are going to access the same memory, we acquire a lock over that memory. This lock only lets one thread access that memory at a time. Once the thread is done using that part of memory, it releases that lock.

Semaphores

Semaphores are a higher level abstraction than locks. They are a little different, and can be used in a scenario where using locks is cumbersome. The lecture gives an example where using a semaphore is better than using a lock.

Consider a publish-subscribe application with producers and consumers connected with a buffer. We want the enqueue and dequeue operations on the buffer to be atomic, and we want a producer to go to sleep when it tries to publish an event when the buffer is full, and we want a consumer to go to sleep when it tries to read from an empty queue.

Using a lock to enforce this behaviour is not really ideal (you can try to write the pseudocode for it and see what problems arise). Instead, we use semaphores. The constraints we have on the resource (the buffer) are the following -

1. If the buffer is empty, the consumer must go to sleep (if it tries to read)
2. If the buffer is full, the producer must go to sleep (if it tries to write)
3. Only one thread can read or write at a time (standard mutual exclusion)

The general rule of thumb is that you have one semaphore for each constraint. So we would have to following semaphores -

1. `Semaphore fullBuffers`
2. `Semaphore emptyBuffers`

3. `Semaphore mutex`

Then the pseudocode for producers and consumers would look as follows -

```
Semaphore fullSlots = 0;            // No full slots initially
Semaphore emptySlots = buffSize;    // The max size of the buffer
Semaphore mutex = 1;                // Standard mutex

Producer(item) {
    semaphoreDown(emptySlots);      // Decrement empty slots or wait until we have space
    semaphoreDown(mutex);           // Mutex for actual enqueue/dequeue
    Enqueue(item);
    semaphoreUp(mutex);             // Release mutex after enqueue/dequeue
    semaphoreUp(fullSlots);         // Signal that an item has been enqueued. This wakes
}

Consumer() {
    semaphoreDown(fullSlots);       // Decrement full slots or wait until there is somet
    semaphoreDown(mutex);           // Mutex for actual enqueue/dequeue
    item = Dequeue();
    semaphoreUp(mutex);             // Release mutex after enqueue/dequeue
    semaphoreUp(emptySlots);        // Signal that an item has been dequeued. This wakes
}
```

Notice that the `semaphoreDown()` operation is used to keep track of the amount of available resources, in this case the space in the buffer, and the `semaphoreUp()` operation is used to signal that some change has taken place in the resource. This is why we have two semaphores. The `fullSlots` semaphore signals that at least one item has been enqueued, and wakes up any sleeping consumers. The `emptySlots` semaphore signals that the queue has some space left, and wakes up any sleeping producers.

Lecture 7 - Synchronization 2: Semaphores, Lock Implementation

This lecture gives an intuition about how locks are actually implemented in an operating system. We want an interface which lets us acquire a lock on a critical section of code, run the critical section, and then release the lock.

The simple implementation is described by the acquire and release pseudocode given below -

```
int value = FREE;   // Flag describing if the critical section is being executed or not
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread to sleep;
        go to sleep();      // careful!
    }
    else {
        value = BUSY;
    }
    enable interrupts;
}

Release() {
    disable interrupts;
    if (thread is sleeping on this lock) {
        wake up thread;
    }
    else {
        value = FREE;
    }
    enable interrupts;
}
```

A small problem with this implementation is in the `Acquire()` code, where if the `value` is `BUSY`, we put the thread (that is trying to enter the critical section) to sleep, and then we go to sleep ourselves, without ever re-enabling interrupts!

This problem is solved by following the convention that whichever thread wakes up after we go to sleep re-enables interrupts.

Another problem with this implementation is that we have to run this code at the kernel level, since it involves disabling and enabling interrupts. This means that the user level code has to go into the kernel by making a system call in order to acquire and release a lock.

Making a system call is around 25x costlier than a normal function call. So if we have hundreds of threads all acquiring and releasing locks, this can make the computer unusably slow. Therefore, we need a locking implementation that can acquire and release locks without having to go into the kernel.

Lecture 8 - Synchronization 3: Atomic Instructions, Monitors, Readers/Writers

The previous lock implementation we saw was impractical because it involved a system call every time we wanted to acquire or release a lock. A system call is at least 25x more expensive than a normal function call, so this approach limited our performance severely.

What we need is just to be able to read a value (of the lock), check if it is 0 or 1, and then write to the value atomically. If we removed the need to disable and re-enable interrupts for this operation, we would be able to implement locks at the user level.

Therefore, all architectures provide atomic instructions for this purpose. These instructions can read, modify, and write data atomically, such that they cannot be interleaved. Some of these instructions are -

```
// Return the value stored at address, and store a 1 there
test&set(&address) {
    result = M[address];
    M[address] = 1;
    return result;
}

// Swap "value" with value stored at address
swap(&address, value) {
    temp = M[address];
    M[address] = value;
    value = temp;
}

// If value at address == reg1, store reg2 there and return success, else failure
compare&swap(&address, reg1, reg2) {
    if (reg1 == M[address]) {
        M[address] = reg2;
        return success;
    }
    else return failure;
}
```

Implementing Locks With test&set

The lecture gives a few ways in which we can implement locks using `test&set()`, but I won't write about them here because they are very intuitive to come up with but result in busy waiting, so we don't use those implementations anyway.

Monitors

Monitors are a concurrent programming paradigm that are, in a way, cleaner and more intuitive than semaphores. A monitor is a lock and zero or more *condition variables* associated with it. A condition variable is just a variable that represents some constraint on the system.

We have the following functions to use with monitors - 1. `Wait(&lock)` - Automatically release the lock and go to sleep 2. `Signal()` - Wake up one waiter, if present 3. `Broadcast()` - Wake up all waiters, if present

For example, let us look at a database with multiple readers and writers. The constraints we have are that the readers can access the database only when there are no active writers or waiting writers, and the writers can only access the database when there are no active readers.

We maintain the following state variables - 1. `int activeReaders = 0` 2. `int waitingReaders = 0` 3. `int activeWriters = 0` 4. `int waitingWriters = 0` 5. `Condition okayToRead = NULL` 6. `Condition okayToWrite = NULL`

Then the code for a reader will look as follows -

```
Reader() {
    // First check self into system
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead,&lock);// Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
    // Perform actual read-only access
    AccessDatabase(ReadOnly);
    // Now, check out of system
    acquire(&lock);
    AR--; // No longer active
    if (AR == 0 && WW > 0) // No other active readers
        cond_signal(&okToWrite);// Wake up one writer
    release(&lock);
}
```

Lecture 10 - Scheduling 1: Concepts & Classic Policies

In a modern OS, all IO operations have the same interface, usually consisting of `read()`, `write()`, `open()`, `close()`, etc. This applies to file IO, network IO, peripheral device communication, etc. This is possible because of device drivers.

A device driver is software that runs on the kernel thread that corresponds to the user-level thread, and is responsible for providing the standard interface that the OS requires. This way, the device driver takes care of implementing each operation, and exposes it under the standard names like `read()`, `write()`, etc.

The life cycle of an IO request is shown below -

The flow chart is pretty self-explanatory in itself. What is important w.r.t. scheduling is that the thread requesting IO may be put to sleep either by the kernel in the second layer, when the kernel decides that the thread is requesting a large amount of data from some device, or by the top half of the device driver, when it receives the IO request, sends it to the device to process, and then puts the thread to sleep as it waits for the response to come back from the device.

Scheduling is the process of taking all the threads and processes that are waiting to run and choosing which thread or process should get the CPU next. Implementing a first-in-first-out policy may seem like the right thing to do, but that is not always the case. If one user has just one process running and another user has 5 processes running on a shared machine, the second user would get more CPU time just because they have more processes running.

Therefore, the scheduling algorithm that we decide to use has to be chosen carefully depending on the kind of workload we are trying to support. For example, consider the CPU burst model. Most programs use the CPU for a short burst of time, then perform some IO, then use the CPU, and so on. So each program has alternate "bursts" of CPU time and IO time. Now we can decide to schedule programs which are expected to have shorter CPU bursts first, or longer CPU bursts first. Each policy has its own advantages and disadvantages.

FIFO Scheduling

FIFO scheduling, also known as first-come-first-serve scheduling, is the most obvious scheduling policy. You run every process in the ready queue to completion, and then pick the first waiting process in the ready queue.

However, as you may expect, this is a bad strategy because a process that runs really long can take over the CPU. If a process runs infinitely long, it may take over the computer entirely. Also, if some processes come along which need very little CPU time to finish, they get stuck behind a long-running process and have to wait a long time.

This is a huge problem for responsiveness of the system. If the thread that handles user input via the keyboard has to wait for a few seconds before a long-running process finishes, then the user may not see what they typed on the screen for a few seconds after they type it.

This effect is called the convoy effect, where a short-running job is stuck behind a long-running job.

Round Robin Scheduling

An improvement over FIFO or FCFS is Round Robin. Round Robin puts an upper limit on the amount of time any process can run, and if it exceeds that time limit, it is preempted and swapped out with another process. This is good, because now the thread that handles user input via the keyboard can still get to run if a long-running process is trying to hog the CPU.

This is a simple but effective solution for the convoy effect, but for it to work well in the real world, we need to tune time quantum that is given to each process. If the time quantum is too large, we effectively revert back to FCFS, and if it is too small, we spend so much time context switching that even the short-running jobs are swapped out multiple times, which ultimately *reduces* the response time. In the lecture, Kubi shows the effect of different choices of time quantum on the average completion time and the average waiting time for each job.

Shortest Job First Scheduling

This is a hypothetical version of scheduling that puts the jobs with the shortest running time first. It is hypothetical because it requires us to know the future and know exactly how long each job is going to run. A preemptive variation of this scheduling policy is called Shortest Remaining Time First (SRTF), where if a new job comes along which has an even shorter time to completion that the currently running job, the currently running job is preempted and the new job starts running.

Although this policy is proven to be the best for improving response times, it has another problem (besides needing to know the future). If short jobs keep coming along, long jobs keep getting stuck behind short jobs and potentially never run. If the short jobs never stop coming, long jobs never run. This issue makes this scheduling policy extremely unfair.

In the real world, we use various heuristics to predict how long each job will take, along with trends observed from the previous runs of the job. So we can have an imperfect version of SRTF scheduling.

When writing a scheduler for a real world OS, you may have to come up with and test multiple scheduling policies to see which one works best for the type of workload

you have.

Lecture 11 - Scheduling 2: Case Studies, Real Time, Forward Progress

Multi-level Feedback Scheduling

This is a scheduling scheme in which we maintain multiple ready queues instead of just one. Each queue has a different time quantum for round-robin and represents different priority levels.

In the image above, the topmost queue has a time quantum of 8 and is for real-time, short-running jobs. The bottom most queue is for long-running jobs. Each new job gets added to the highest priority ready queue when it arrives for the first time. Then, every time its time slice expires, if it still hasn't finished, it gets dropped one level to the queue below with a longer time slice.

The CPU is shared among all the queues according to some percentage - each queue gets a certain amount of CPU time. That is, the topmost queue could get 70% of the CPU time, the one below it gets 20%, and the bottom most queue gets 10%. That means that 70% of the time, tasks in queue 1 are executing with a time quantum of 8, for 20% of the time, tasks in queue 2 are executing with a time quantum of 16, and for the remaining 10% of the time, tasks in queue 3 are executing FCFS.

Case Study: Linux O(1) Scheduler

The Linux scheduler for a while was a variant of the Multi-level feedback scheduler you saw above. Instead of 3 read queues, it had 140, of which the first 100 (of highest priority) were for real time tasks, and the remaining 40 were for user tasks.

It was a O(1) time scheduler, which means that it took constant time to run and figure out which job should be scheduled next. It achieved this by computing a large amount of heuristics every time it was called and maintaining a copy of all the 140 ready queues. Once a job was dequeued and started to run, it was put into the copy queue, and then the pointer to the active queue was swapped to point to the copy queue.

However, even though the scheduler worked well, it because impossible to maintain because the number of heuristics it used and their complexity grew too much, to the point that it was just discarded and completely replaced with a new policy.

Real-Time Scheduling

Real-time scheduling refers to the scheduling for a real-time operating system, which often needs to ensure that jobs that arrive are run before a certain "deadline". For example, the embedded system in a car that controls the brakes needs to ensure that once the driver presses the brake, the brakes are applied within a certain deadline.

These deadlines are critical to the safety of the application, and thus we need predictability of performance. A common scenario in real-time scheduling is scheduling periodic tasks that occur with a period P and have a burst time of C. The condition in this case is that a task should finish its computation of C before its period P is over. In this case, round robin scheduling does not work - we end up missing deadlines.

Earliest Deadline First (EDF)

This scheduling policy simply takes the task which has the earliest deadline and runs it. Every time it has to switch, it selects the job which has the earliest deadline.

EDF is proven to be optimal if you have a reasonable number of jobs to run. If the number of jobs keeps increasing, at one point EDF will not be able to guarantee deadlines. That point is given by -

$$\sum_{i=1}^{n} \frac{Ci}{Di} \leq 1$$

Where C_i is the burst time of task i and D_i is the deadline or period of task i. This inequality just says that the percentage of CPU used by each task should add up to be less than or equal to 100%. If it is greater than 100%, the scheduler obviously won't be able to meet all deadlines.

Ensuring Progress

We have already seen the problem of starvation - when a thread or a job fails to make progress for a long time because of the scheduling policy. This is not the same as deadlock, since starvation could possibly resolve under the right conditions.

Lecture 12 - Scheduling 3: Deadlock

Linux Completely Fair Scheduler

The scheduler used in Linux (as of the lecture in August 2021) is called the "Completely Fair Scheduler", and it approaches scheduling differently than a traditional round-robin or real-time scheduler. It tries to give every thread the same amount of time on the CPU, so if there are N threads, each thread gets 1/N of CPU time.

As each thread is running, the scheduler keeps track of how long the thread ran. Each thread has an allocated time slice, and if a thread gives up the CPU and goes to sleep before its time slice has expired, then it gets more priority to run when it wakes up, because the scheduler wants to make sure that each thread gets the same amount of time on the CPU.

The Linux CFS has a target latency, which is the period of time over which it wants to make sure that all threads are run at least once. For example, if the target latency is 20ms, and there are 5 threads to be scheduled, it will schedule each thread to run for 4ms, and keep track of the threads that run for a shorter (or longer) time.

However, if the target latency is 20ms, and there are 200 threads to be scheduled, then each thread gets a 0.1ms time slice, which may be too small, since we have to spend some time context-switching. Therefore, the CFS scheduler has a lower bound on the time slice, say 1ms. So in this case, all 200 processes will get a 1ms time slice, and we will not be able to meet our latency goal of 20ms.

So far this looks like a slightly modified version of round-robin, but the difference becomes clear when we see how CFS handles job priorities. It allows you to give each job (thread) a priority, and then instead of calculating the time slice for each thread equally, it calculates the time slice to give based on the weight of the job. The weight of the job is decided by the "nice" value of the job. A nice value typically ranges between -20 to 19, and a higher nice value means lower priority.

The weight of a job is calculated as

$$weight = \frac{1024}{1.25^{nice}}$$

The virtual time slice of a job is then calculated as

$$max(minslice, \frac{Wi}{\sum_p Wp} * targetlatency)$$

Where w_i is the weight of the ith job. CFS then tries to give every job this virtual time slice instead of real time.

Deadlock

A deadlock is a type of starvation that never resolves, and only occurs non-deterministically, when the schedule creates a cycle of accessing resources. A deadlock can be caused due to any resource - locks, terminals, memory, printers, sockets, pipes, etc.

There are 4 conditions that are necessary (but not sufficient) for deadlock to occur -

1. Mutual exclusion - There has to be a resource that only one thread can access at a time.
2. Hold and wait - A thread holding at least one resource is waiting to acquire more resources held by other threads.
3. No preemption - Resources are only released voluntarily by the thread, and the kernel cannot preempt it into giving up the resource
4. Circular wait - There exists a set of threads {T1, T2, . . . , Tn} such that
    - T1 is waiting for a resource held by T2
    - T2 is waiting for a resource held by T3
    - . . .
    - Tn-1 is waiting for a resource held by Tn
    - Tn is waiting for a resource held by T1

In order to detect deadlock, we can build a resource-allocation graph. A resource-allocation graph is a graph with 2 types of vertices - one representing threads and one representing resources. It has directed edges that signify the following -

- A directed edge from a thread vertex T to a resource vertex R means that the thread T is requesting resource R.
- A directed edge from a resource vertex R to a thread vertex T means that the thread T has ownership of resource R.

The slides show a few examples of resource allocation graphs. Below, a resource vertex has one or more dots which represent an instance of the resource.

Detecting Deadlock

Just because there is a cycle in this graph does not mean that there will be deadlock. To detect deadlock, we have a simple algorithm. Instead of describing it here, I'll just attach a screenshot of the lecture slide.

Dealing with Deadlock

There are 4 common ways of dealing with deadlock -

- Deadlock prevention - Write your code in such a way that deadlock won't occur. This means that you trust the apps running on your machine to be written this way as well.

- Deadlock recovery - Let deadlock happen, and then figure out how to recover from it, maybe by rolling back a transaction (if transactions and rollbacks are supported).
- Deadlock avoidance - Dynamically delay resource requests so that deadlock doesn't happen. Requires you to do extra work while writing and maintaining the kernel.
- Deadlock denial - Don't deal with deadlock

The Linux operating system makes sure that the kernel is not involved in any deadlock, but ignores any deadlock that happens in applications.

There is an algorithm that can detect if allocating a resource to a particular thread would lead to deadlock called the banker's algorithm (described in the lecture and the slides).

Lecture 13 - Memory 1: Address Translation and Virtual Memory

Virtual Address Spaces

An address space is a set of protected memory addresses that are accessible to a program. These are virtual addresses distinct from actual physical memory. Not all processors have virtual address spaces. In order to have virtual address spaces, you need support for address translation from hardware.

The mapping between physical addresses and virtual addresses is done by a unit called the Memory Management Unit (MMU). This mapping lets every process assume that it has the entire RAM to itself, and is able to access any address. Whenever the processes accesses a particular (virtual) address, the MMU maps it to the specific physical address assigned to it, and gives the process data from the actual physical address. However, the process never knows exactly which physical address the data it accessed came from.

Memory Translation

The most important function of the MMU is to be able to allocate non-contiguous chunks of memory to a single process. Without the MMU, in very early operating systems, when a process was loaded into memory, it was assigned a big contiguous chunk of memory. However, allocating contiguous chunks of memory quickly leads to fragmentation of the memory.

To get around this problem, the MMU only allocates contiguous chunks of memory to different "segments" of a process. For example, each process needs memory for its code, static data, heap, and stack. The MMU only allocates contiguous memory for a process's code, static data, heap, and stack, but all of these four parts may be stored in different chunks of physical memory as well as virtual memory.

This approach has another advantage. If we allocate a contiguous chunk of memory for a process' heap, we can map that same chunk of memory to another process' heap and get inter-process communication.

The information about where each segment has been stored is contained in the segment map. For each virtual address, we take the first few bits of the virtual address. These bits are used to index into the segment map and find the base of the chunk of physical memory allocated to this segment. These first few bits are usually the first 2 or 3 bits. Then, these first few bits are replaced by 0s, and the resulting virtual address is considered to be the offset from the base that we got from the segment map. So this offset is added to the base, and we get a physical address.

In the lecture Kubi steps through an assembly program and goes through all the

steps I described above in order to get a physical address from a virtual address, around the 1:07:00 timestamp.

This segment table is unique to a process, and when a process is swapped out during a context-switch, this segment table also needs to be swapped out.

Now, if we have so many processes that we run out of room for storing each segment for each process, we can actually use disk to store these segments for a process. This is called swapping, and this lets us use the huge size of the disk as DRAM. However, accessing disk has a huge cost in terms of time, and each disk access takes roughly on the order of 1M cycles. Therefore, this is not really a practical strategy, but only used when there is no other option.

So far, this strategy helped us a little bit with fragmentation. Instead of allocating a huge chunk of contiguous memory to a process, we were able to divide the memory a process requires into different segments, and only allocate those contiguously. However, when the size of the segments keeps growing, especially with the process stack and heap, fragmentation starts coming up again.

Fragmentation can be of two types - external and internal. External fragmentation is the type we have been talking about so far. But internal fragmentation is when we allocate a small(er) chunk of memory to a segment, but the process doesn't need that entire chunk. This leads to internal fragmentation.

In order to better deal with fragmentation (both external and internal), we need a smaller quanta than a segment of a process. We can come up with a smaller quanta, and call it a "page".

Paging

Now each process gets a page table which maps a virtual page number into a physical page number. If you have a virtual address, the last few bits of the address are used to represent an offset into the page. So if a page is of 1024 bytes, we will need to use the last 10 bits of the virtual address to specify the offset into the page. The remaining bits of the address are used to represent the virtual page number.

Now that you have this virtual address, we use the virtual page number as an index into the page table. The page table gives you the physical page number. You then access that physical page and go to the offset you have, and you get to the address you wanted.

However, this is still not quite what we want. Next lecture extends this idea to better deal with fragmentation.

Lecture 14 - Memory 2: Virtual Memory, Caching and TLBs

So far, the implementation of paging that we have seen is not very efficient. If we assume that a page is 4KB, then we need 12 bits to represent the offset into each page. If we have a 32 bit machine, then the remaining 20 bits will be used as an index into the page table, which means that the page table has to have 2^20 entries, each of 4 bytes, which means we use around 4MB to store the page table for a process. That may not be too much for today's machines to handle, but if we have a 64 bit machine, then using a 12 bit offset, we use 52 bits as an index into the page table, which means our page table takes around 36 exabytes.

How do we fix this? We can split the page table into multiple levels. Instead of just having one "level" of virtual address translation, we can have two. Considering the example of a 32 bit machine, we would have 12 bits for the offset into the page, and then instead of using the remaining 20 bits as an index into the page table, we will use only the top 10 bits. These 10 bits will be used to index into a table which will point to the next "level" of the page table. Then the next 10 bits will give us the actual physical page.

As processor cores get bigger and bigger and we start getting to 64-bit architectures, even multi-level page tables start becoming infeasible. In that case, we can switch to using a hash table. A hash table has the advantage that its size only depends on the number of pages used by the process, and not the size of the virtual address space. However, this means that we don't have cache locality, and managing this hash table is more complex than a simple page table.

TLBs

There is one issue with the concept of paging that we must address. The issue is that we have effectively made all caching useless because in order to check if an address is in the cache, we have to translate it to a physical address, for which we have to go to the page table, which is stored in DRAM!

A simple solution to this is to add another cache for storing virtual addresses and their mapping to physical addresses. This is called the TLB - Translation Look-aside Buffer.

Lecture 15 - Memory 3: Caching & TLBs, Demand Paging

Demand paging is a simple but powerful idea. Most modern programs require a lot of memory, but they don't use all of their memory at once. On average, they follow the 90-10 rule - 90% of the time the program executes 10% of its code, and 90% of the time, the program only needs 10% of its memory.

We can use this trend to our advantage in order to keep only the most used data in DRAM and cache, and the rest out on disk, instead of trying to load the entire program and the data it needs into the DRAM.

In demand paging, we don't fully populate the page table when we load a program into memory. As the program runs, it tries to access a page that is not mapped by the page table. In this case, we get a page fault, which is equivalent to a cache miss. We then go out to disk and load the page into memory, update the page table, and run the instruction again.

Lecture 16 - Memory 4: Demand Paging Policies

Demand paging aims to improve the average memory access time. The average memory access time (AMAT) can be calculated as -

```
AMAT = Hit Rate x Hit Time + Miss Rate x Miss Time
```

Which can be simplified to

```
AMAT = Hit Time + Miss Rate x Miss Penalty
```

You can think of demand paging as being a cache managed in software. The pages live on the disk, and only the pages that are needed are brought from the disk into the DRAM. The mechanism of demand paging is as follows -

1. Processor tries to access an address
2. It checks the page table entry for the address
    1. If the entry is valid, we go ahead and access that physical address
    2. If it is invalid, we get a page fault, and we need to get that page from the disk into DRAM

In case we get a page fault, the OS gets the page from the disk into DRAM -

1. Choose an old page to replace (assuming we don't have free space in DRAM)
2. If the old page is modified, write it back to disk
3. Change the page table entry and TLB entry to be invalid
4. Load new page from disk into DRAM
5. Update the page table entry and invalidate the TLB entry to force the MMU to fetch the new address
6. Retry the memory access that faulted before

Memory Access Behaviors

There are two common types of memory access behaviors that a process follows -

1. A process has a specific set of pages that it accesses, and it doesn't access any other pages. This working set can change over time. This is called the working set model.
2. A process does not have a specific set of pages it accesses, but it accesses some pages more frequently than others. In this case we can say that some pages are popular for a process, and give each page it accesses a rank depending on the number of times it is accessed. This is called the Zipf model.

Cache hit rate under the working set model grows in steps as the cache size increases.

Cache hit rate under the Zipf model grows logarithmically as the cache size increases.

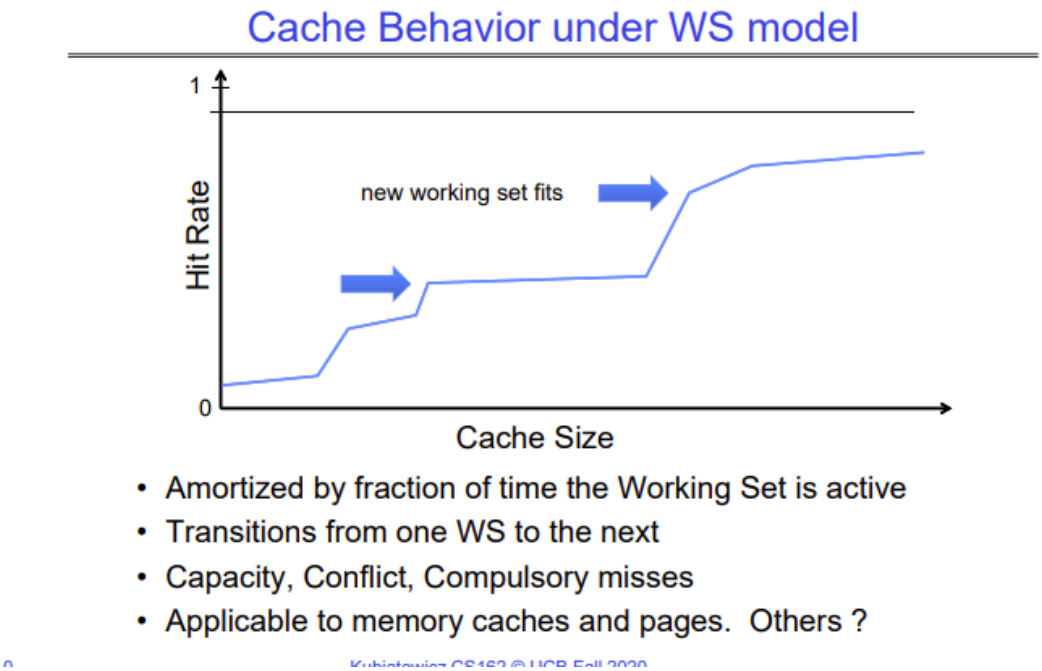


Figure 2: Cache behavior under the working set model

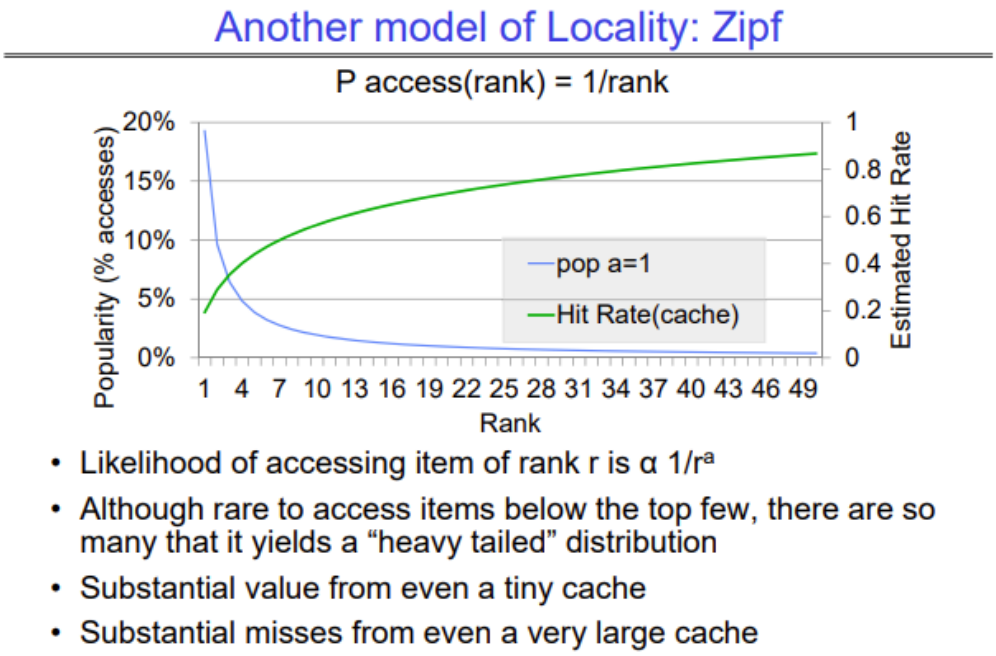


Figure 3: Cache behavior under the Zipf model

Demand Paging Replacement Policies

When we implement demand paging, we have to take a special care about the replacement policy we use to choose which page to evict from the DRAM to make space for the new page we will bring in from the disk. This is because the cost of a miss/page fault is huge, since going out to the disk takes around 1M cycles. This means that if you start page faulting, your performance comes to a grinding halt very fast.

In the lecture, Kubi shows a rough calculation which shows that even if you have a miss rate of 1 in 1000, you slow down by a factor of 40! If you want to keep your effective memory access time to be only around 10% more than accessing a page from DRAM, you need a miss rate of less than 0.00025% or 1 in 400,000.

The lecture covers a few replacement policies -

1. FIFO - Replace the page that came in first. This is a bad policy, since it is likely that the page that came in first could be a page that is accessed very frequently.
2. Random - This works well for caches since the miss penalty of a cache means going to the DRAM, which is not as bad. However, when the miss penalty is going out to the disk, we cannot risk randomly choosing a very frequently accessed page.
3. Min (Optimal) - Replace the page that will be used farthest in the future. This strategy is provably optimal, but we obviously can't implement it because we can't know the future.
4. LRU - LRU is a good approximation for Min, but it is not practical to implement since it involves operations that have high time complexity.

Approximating LRU - The Clock Algorithm

Since LRU is impractical to implement, we approximate LRU using something called the Clock algorithm.

We first link all the pages in memory in a circular linked list. We then keep a pointer to a node in this list, and call it the clock hand. On every page fault, we check if this node has been set as "used" or accessed by the processor. If it has, we reset its used bit to zero, and move the clock hand to the next node. We eventually reach a node which is not marked as used. This means this page is an old page which has not been used in a while, so we can replace it.

This is an approximation to LRU because we evict an old page, but not necessarily the oldest page. In the worst case, we could loop around the circular linked list and not find a single unused page. In this case, we eventually come back to the start node, whose used bit we had cleared, and evict that page. Therefore, in the worst case, the clock algorithm becomes FIFO.

Instead of keeping track of a single used bit, we could keep a counter, and give each page N chances instead of just 1 before we choose to evict it. This is a variation of the clock algorithm.

Lecture 17 - Demand Paging, General I/O, Storage Devices

An important design decision is how many page frames we should give to each process/thread. Too few page frames (memory) allocated to a process will lead to thrashing, where the process just spends all or most of its time swapping pages in and out of memory. Too many frames allocated to a process will waste memory that could have gone to a process that needs the extra memory.

We can use a static policy to decide the number of page frames, but it is better to have a dynamic policy that changes the amount of memory allocated to a process based on its behaviour. We can detect that a process needs more memory and that it is thrashing by monitoring the number of page faults it is encountering. A large page fault rate means a process is likely thrashing. We want to keep the page fault rate between an upper limit and a lower limit. We don't want the page fault rate to get too low, because that means we could be allocating too much memory to a process that doesn't need it.

Meltdown

In 2017, a bug was discovered that allowed a user level program to read kernel code, because of the way memory was mapped by operating systems. The lower addresses of virtual memory were mapped to user addresses, and the upper addresses of virtual memory were mapped to kernel addresses. The image below shows the organization for 32 bit and 64 bit machines. The memory map for 64 bit machines has a big hole (called the Canonical Hole), simply because the address space that can be mapped by 64 bits is huge, and no machine has that much RAM yet.

A user program accessing a kernel address would cause a page fault, and ultimately a core dump.

However, processors have something called speculative execution, which is when an instruction is executed even though it performs an illegal operation (like accessing kernel level addresses), or when a branch prediction is needed, etc, and when the illegal operation (or error) is discovered, all the registers relating to that instruction are cleared so that no data is leaked. This technique is usually beneficial and improves the speed of the processor.

Meltdown exploited this technique in a very clever way -

```
uchar array[256 * 4096];
flush(array);       // Make sure the array is not in cache

try {
    uchar result = *(uchar *) kernel_address;       // Try to access kernel memory
```

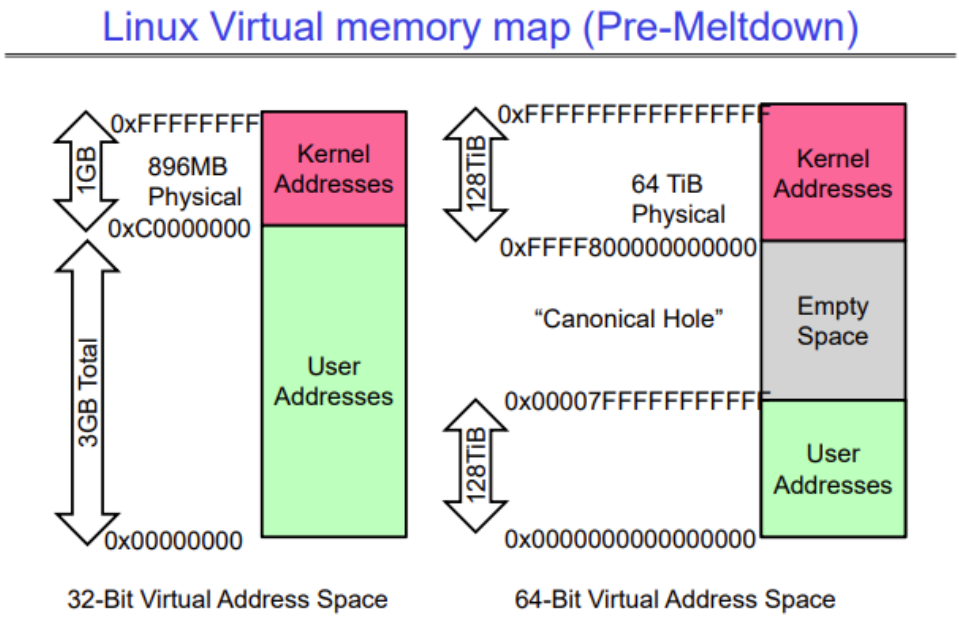


Figure 4: Pre-meltdown Linux Memory Map

```
    uchar dummy = array[result * 4096];             // Force the processor to store (res
} catch () {;}
```

We first try to access some kernel memory, which the processor does successfully because of speculative execution. We then immediately force the processor to cache the value we got from the kernel address in the next instruction. When the processor realizes that we performed an illegal operation, it clears the value of the registers and throws an error, which we catch. However, the value is still in cache. We then try to access each item in the `array` array, and measure the time it takes to access each item. When we get to the index that was in cache, our access time is very fast, so we can tell what value we got back from the kernel address we just accessed.

Lecture 18 - General I/O, Storage Devices, Performance

In previous lectures we have seen that the OS has a standardized interface for communicating with the external world. This means that even though there are hundreds of different types of devices, each of which has a different internal structure, the operating system can work with all of them because the devices support this standardized interface.

This interface is provided to the OS by the device driver. The device driver is a process that knows how to work with the specific hardware of the device, and can work with the device in order to provide the standard API calls to the OS.

There are three main types of devices -

1. Block oriented devices - Devices which access blocks of data, e.g. disk drives. When you read from or write to these devices you can only do so in "blocks" (typically 4KB), and not individual bytes.
2. Character oriented devices - Devices which communicate individual bytes or character, e.g. mouse, keyboard, etc.
3. Network oriented devices - Devices which deal with packets and are slightly different from block or character oriented devices, e.g. ethernet, wireless, bluetooth, etc.

To each of these devices, there are three main types of interfaces -

1. Blocking interface - When a process requests some data from this device, the process is blocked and possibly put to sleep.
2. Non-blocking interface - When a process requests some data from this device, the device returns whatever it can return immediately, and the process continues executing. If the process needs more data, then it must keep asking the device for more data.
3. Asynchronous interface - When a process request some data from this device, it continues executing and the device starts fetching the data. Once the data is ready, the device signals to the process saying that the data it had requested is ready.

Hard Disk Drives

Although SSDs are becoming more common now, HDDs are still around. HDDs work by storing information on magnetic disks. A track positioned on top of the disk can read or write to the disk.

The disk latency (time taken for a read/write request to be completed) is the sum of the queueing time, disk controller's execution time, seek time, rotational delay, and transfer time.

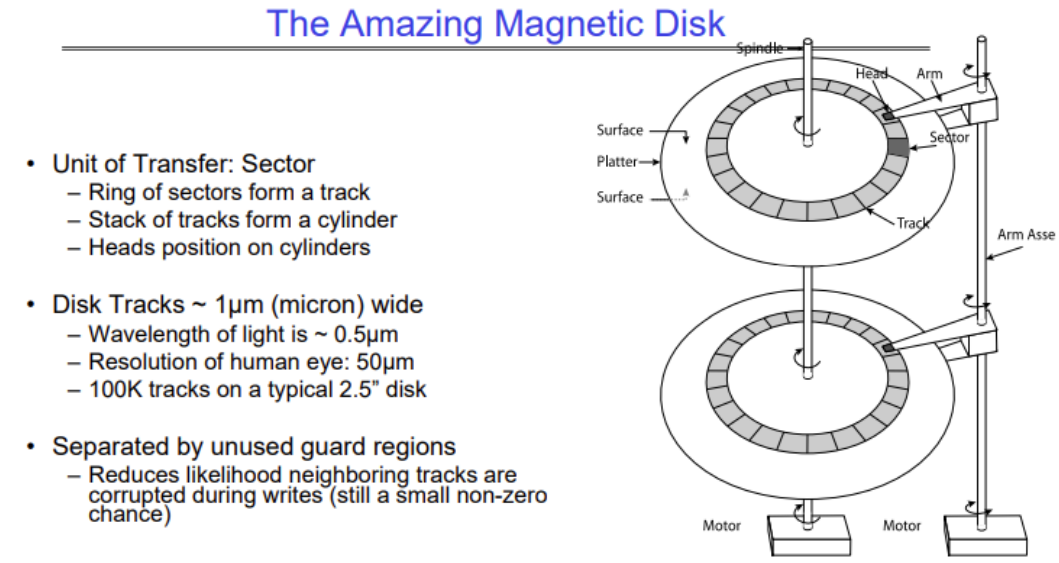


Figure 5: Magnetic Disks

All of these factors means that HDDs are painfully slow. Their data transfer rate is usually 50 to 250 MB/s.

Lecture 19 - Filesystems 1: Performance, Queueing Theory, Filesystem Design

When we measure the performance of an I/O system, we have to consider queueing time, because the data can pass through multiple queues before our I/O request is satisfied. These queues are present in the OS itself (the ready queue, wait queue, etc. that the process has to go through), and the I/O device itself.

Each queue has a latency L and a bandwidth B. In a simple system, the (maximum) bandwidth is the inverse of the latency.

$$B = \frac{1}{L}$$

If we consider a pipelined system, if we have a pipeline of K stages, our bandwidth increases K times, as expected.

$$B = \frac{K}{L}$$

Another important parameter could be the number of things/operations inside the system at any time. If the latency is L, and the current bandwidth is B, then there are $ B*L $ things/operations inside the system at this time (roughly, assuming the bandwidth isn't constantly changing). This is known as Little's Law.

Lecture 20 - Filesystems 2: Filesystem Design, Case Studies

The filesystem is responsible for bridging the gap between the API exposed by the operating system to the user (which is byte-oriented), and the API provided by the device driver for the disk to the OS (which is block oriented). The disk only lets you read and write data in blocks which can range from 512 bytes upto 4KB. However, the user is given the illusion that they can access any byte on the disk as they need.

The filesystem provides two entities - files and directories. Files are just a collection of bytes, and a directory is an index that maps a name to a set of files. Each sector of the disk is given an integer address, and the filesystem remembers which files are stored at which addresses.

The filesystem also needs to track additional information such as -

- Free block addresses, in case it has to create new files.
- Which blocks belong to which file, in order to read and write existing files.
- Which directories contain which files.

Since we want all of this data to be persistent after the machine is shut down, we will need to store this data somewhere on the disk. This may become a recursive problem! We can solve this using a small set of fixed addresses for this metadata, or some standard positions for the root of the filesystem.

Components of a File System

A file system has two components - a directory structure map, and a file header structure. The directory structure map is a mapping in a directory that maps a file name to a file number (called an inumber). This inumber is used to find the inode of a file (also called the file header). The inode of a file contains a list of all the blocks that belong to the file.

When a process opens a file, a name resolution is performed in the directory structure to first find the inumber of the file. This inumber is used to look up the inode of the file. This inode describes all the blocks that belong to the file, and now we can read/write these blocks as we want.

A directory in a file system is actually a file that contains a list of mappings of the format `<file_name: file_number>` for all the files the directory contains. Each of these `<file_name: file_number>` mapping is called a directory entry, and these are stored in a format that the OS decides. The OS doesn't let a user process read the raw bytes of a directory (the directory entries), because if these entries are corrupted by an incorrect user process, all the data in the directory could be lost.

When we perform actions on these directories like creating them, moving them, or

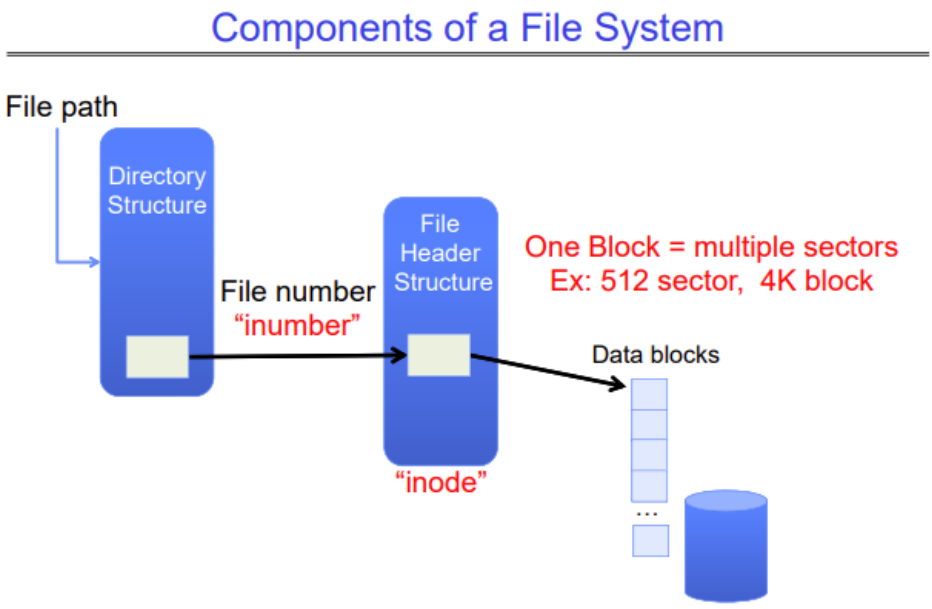


Figure 6: Components of a file system

deleting them, we are actually just modifying this structure. So syscalls like `open`, `create`, `readdir` traverse this structure, and `mkdir`, `rmdir` add/remove entries from this structure.

For example, say you wanted to open a file `/my/book/count`. The OS follows the following steps -

1. Read the file header (inode) for the root `/`.
2. Using the inode, read the first data block for the root.
3. In this data block, find the directory entry for `my`.
4. If we find it in this block, we proceed, otherwise we read in the next block and look for it there.
5. Once we find the directory entry for `my`, we read the first block for `my`.
6. We look for the directory entry for `book`.
7. We read the first block for `book`.
8. We look for the directory entry for `count`.
9. We get the inode for `count`, and now we can read the blocks for `/my/book/count` as we need.

This is all stored on the disk, however, when we try to open a file, these structures are also created in memory -

Each process has a table that contains the file descriptors of the files the process

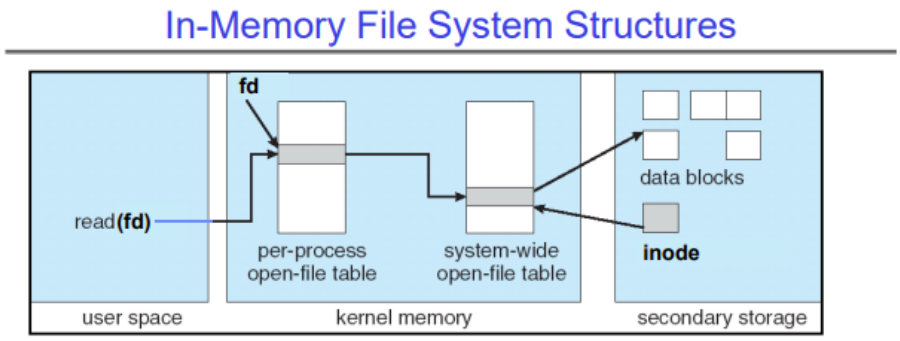


Figure 7: In-memory file system structures

has opened. When the `open` syscall is executed, a file descriptor is created for the file and stored in this table. This file descriptor points to an entry in a system-wide open file table, which contains the inode of this file, fetched from the disk and put into RAM.

Case Study: File Allocation Table (FAT)

FAT is a very simple file system. It basically just maintains a table that maps from a file number to a disk block number. Assuming we have a way to turn a file path into a file number, we just look up the file number in the FAT, and it gives us the disk block for the start of that file. If a file spans across multiple blocks, each FAT entry stores a pointer to the FAT entry for the next block of the file.

FAT stores directories in the same way, since directories are just files that store file name to file number mappings. The root directory is stored in FAT entry 2.

The FAT filesystem is simple by design, and is therefore not the most efficient. It is prone to fragmentation, random access is slow, and sequential access can also become slow over time because of fragmentation.

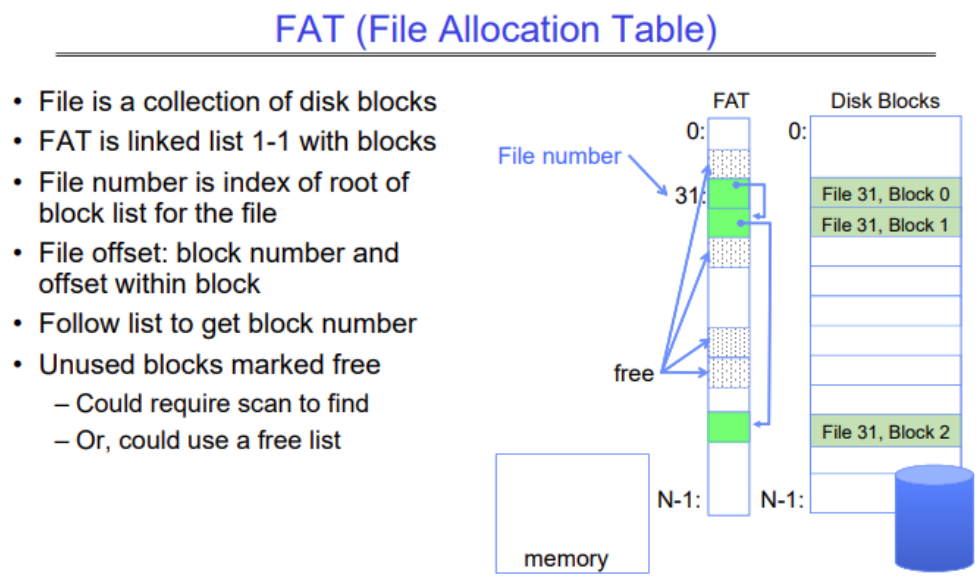


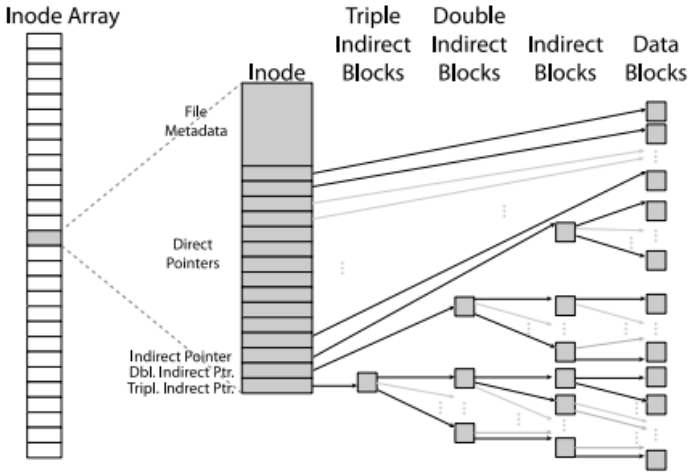
Figure 8: FAT Filesystem

Case Study: Berkley Fast File System

Instead of a file allocation table, the Berkeley Fast File System uses inodes. Inodes first appeared in BSD 4.1 (Berkeley Standard Distribution). An inode is a structure that holds information about a file. Every file has an inumber, which is an index into an array of inodes. The inodes store some metadata about the file, and then they have a few pointers -

1. Direct pointers - These pointers point directly to the first few blocks of the file. Each pointer points to one block. There are around 10 to 12 direct pointers, which point to the first 10 to 12 blocks of the file. These pointers are maintained because most files are small, so those small files can be accessed directly.
2. Indirect pointers - Indirect pointers point to indirect blocks, which are one level above the data blocks. These indirect blocks mean that you have to follow a tree-like structure to get to the data blocks. This allows the file system to store very large files efficiently.

Inode Structure



Filesystems 3: Case Studies, Buffering, Reliability, Transactions

In previous lectures, we saw that a directory is just a file mapping file names to file numbers. There are two such types of mappings -

1. Hard links - Maps a file name to a file number. The fist hard link is created when the file is created. There can be multiple hard links, and hard links can be created and removed using syscalls like `link()` and `unlink()`. When all hard links to a file are removed, the file is deleted.
2. Soft links - These are also called symbolic links (symlinks). They map a file name to a different file name.

Memory Mapped Files

Traditional I/O involves reading and writing to regions of memory on disk, usually with buffers and caches in between. Memory mapping of files is a technique in which we "map" the file into RAM by giving it a virtual address, and then we can use the RAM as a cache without needing buffers, and the RAM is backed by the file in memory. This technique is often used to set up inter process communication.

Buffer Cache

The buffer cache is just a region in memory that is used to cache (hash table) disk blocks, inodes, directory entries, etc. It is a write back cache with LRU replacement, and is completely implemented in software by the kernel. It also supports prefetching for sequential disk access. It also writes blocks back to the disk periodically, in case of a crash.

It improves the performance a lot, but it also means that we have to worry about reliability. In the lecture, Kubi describes 3 important properties of a system -

1. Availability - The probability that the system is responsive, but not necessarily working correctly.
2. Durability - The ability of the system to recover despite faults.
3. Reliability - The ability of a system to perform correctly under the stated conditions.

Transactions, End-to-End Arguments, Distributed Decision Making

Transactions in a file system are a mechanism to improve the reliability of the system. They extend the concept of atomic updates from memory to persistent storage. A transaction is an atomic sequence of writes that takes the file system from one state to another. If the system crashes before the transaction is complete, none of the writes in the transaction make it through and the file system is restored to the state it was in before the transaction.

Transactions are generally implemented using a log-based approach. There is a log which supports only one atomic operation - appending to it. When we write to a file as part of a transaction, all the individual steps of the write are written as instructions in the log. Once all the instructions are written, we write a "commit" instruction to the log, which says that the transaction is complete and the changes can be applied to the file system.

The OS then goes through the instructions in the log and applies them one by one. If we crash at any time before reaching the commit instruction, we just start from the first instruction and perform the writes again. This is okay because we're just overwriting the same data. If we crash before the commit message is written to the log, all the instructions for this transaction are just thrown out from the log and we say we rolled back.

We use NVRAM for the log to make sure unapplied commits are not lost in case of power failures.

A file system that uses a log in this way to support transactions is called a transactional file system. A transactional file system can be log structured or journaled. In a log structured file system, the data stays in log form. In a journaled file system, the log is only used for recovery and all data is written directly to the disk.

Log Structured File System (LFS)

If we take the concept of a log further, we can make the log itself the file system. It is one continuous sequence of blocks that data is just appended, not modified. As actions are taken (files or directories are created, read, deleted), those actions are also appended to the log. The present state of the file system is then inferred by replaying the actions stored in the log.

As you might expect, this type of filesystem is efficient for writes but not so efficient for reads. However, a good block cache with enough space can make this file system feasible, especially when used with flash memory instead of a hard disk drive. An implementation of a log structured file system is F2FS, a flash file system used in many mobile devices, including the Pixel 3.

Distributed Decision Making, Networking, TCP/IP

General's Paradox

The general's paradox is a problem in which two generals have to decide on a particular time to coordinate their attack. They can only exchange physical messages through messengers, who can be captured by the enemy. If they successfully decide on a time to attack, they will succeed. If they cannot decide on a time and attack at different times, they will fail.

The general's paradox doesn't have a solution. Since the messengers can be captured, you can never be sure that the last message you sent was received. So even if you send an ACK, you can never be sure that the other person got the ACK.

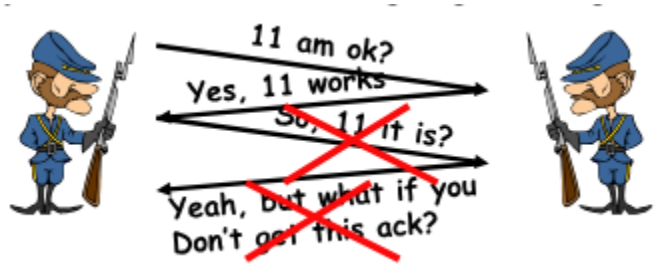


Figure 9: General's Paradox

Two-Phase Commit

The two-phase commit protocol was developed by Jim Gray, a Turing award winner and Berkeley alumni. The two-phase commit protocol solves a problem related to the General's paradox, since we cannot solve the general's paradox itself.

The 2PC algorithm has one coordinator node and n workers who are trying to decide whether to commit or abort a transaction. The transaction can be a database transaction, a distributed file system transaction, or any other action that can be committed or aborted.

The high-level overview of the algorithm is -

1. The coordinator asks all workers if they want to commit or abort
2. All workers decide individually if they are ready to commit or if they want to abort
3. All workers reply with their desired action
4. If the coordinator receives "commit" from all workers, it sends a "global-commit" message to each worker, which causes each worker to commit their transaction
5. If the coordinator receives at least one "abort", it sends a "global-abort" message to each worker, which causes each worker to abort their transaction

All workers and the coordinator have an internal log in which they record their decision so that they will remember their decision even after a crash.

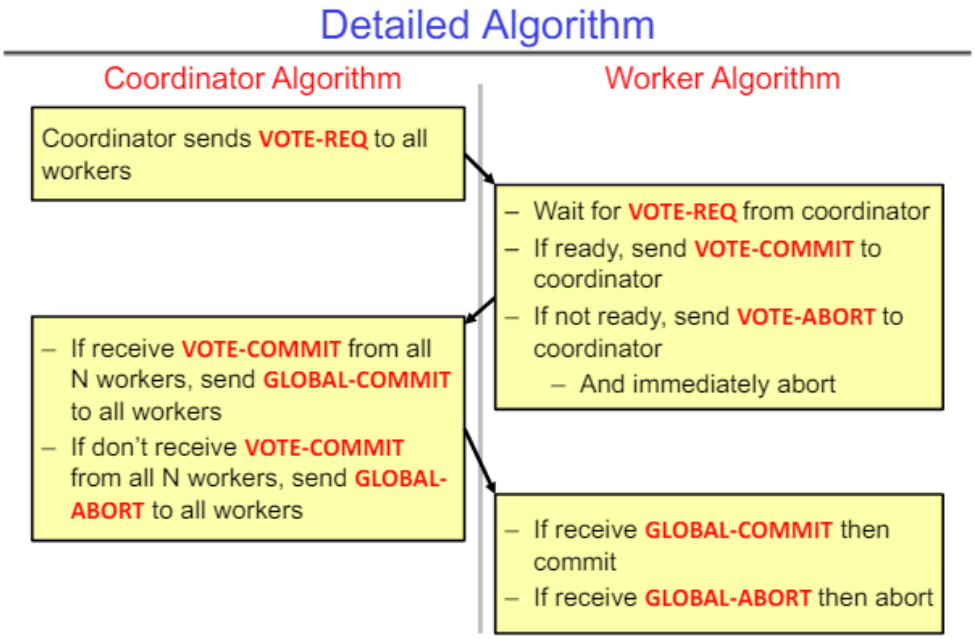


Figure 10: 2PC algorithm

A key drawback of the 2PC algorithm is that it is blocking. If a node fails or keeps failing, the other nodes are blocked and cannot make progress. A blocked node can also hold resources like locks, which make the blocking problem worse. There are some alternatives to the 2PC algorithm, such as the 3 phase commit, which has 3 phases instead of 2, PAXOS, a complicated distributed decision making algorithm used by Google in its internal systems, etc.

Another limitation of the 2PC algorithm (and also the alternatives to it) is that it assumes all nodes are working correctly, and no node is malicious. If a distributed system can have malicious nodes, you need a different class of algorithms. This situation is demonstrated by the Byzantine general's problem.

Byzantine General's Problem

The Byzantine general's problem has n players (nodes), out of which one or more can be malicious. One of the nodes is the General (leader), and has to send a boolean message to all nodes. The outcome must be that all correctly working nodes perform the same action, regardless of how the malicious nodes try to throw them off.

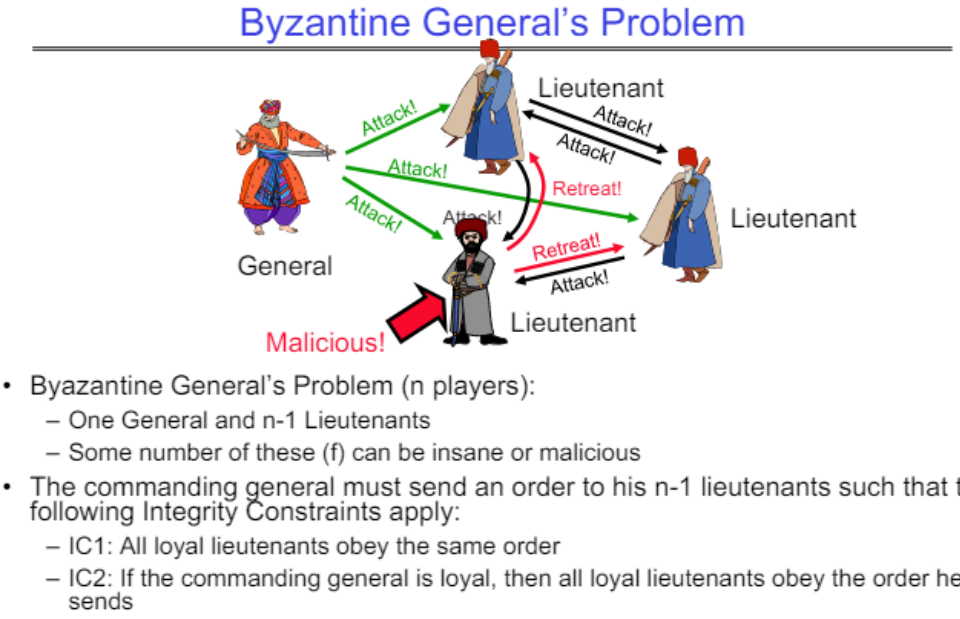


Figure 11: Byzantine General's Problem

The leader itself could be compromised. In this case, all the correctly functioning nodes still perform the same action, but that action may not be what the leader communicates to each node.

The lecture doesn't actually describe the algorithms that solve this problem, but today's algorithms are $ O(nˆ2) $, and block chain based algorithms can even be $ O(n) $ in the number of messages that need to be exchanged between the nodes.