162 Notes

Enzo Massyle

September 4, 2024

1 Low Level File IO

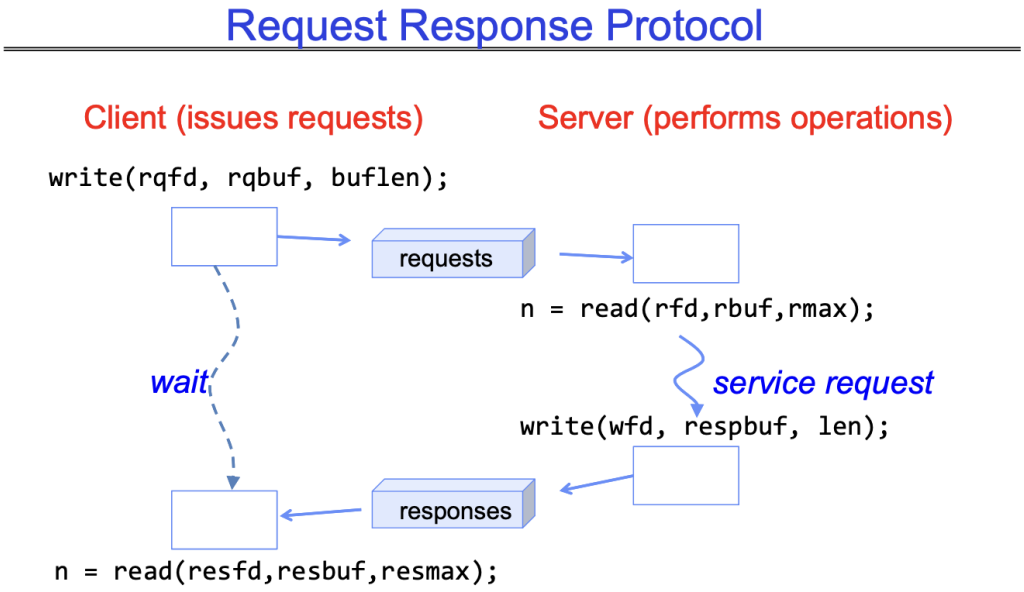
While we usually use high level file IO when writing programs, it is still important to know how low level File IO works to understand what is really going on under the hood. Every time we call a low level file IO function, we will make a system call. Here are some low level file IO functions:

- int open → open a file given a file name, will return a file descriptor
- int create → create a file
- int close → close a file given a file descriptor
- ssize t read → Read from the file using the file descriptor
- ssize t write → Write to the file using the file descriptor
- off t lseek → Reposition where we are pointing to in the file.

Note these functions will mostly deal with file descriptors rather than file descriptions. This is because if we were able to obtain the file description, we could potentially modify things that we shouldn't. So it is mainly a security purpose. The description is kept under the hood.

2 Using File IO in communication across networks

With the POSIX principle, it can even be extending to communication between processes or communication across networks! we will first look at the following diagram:

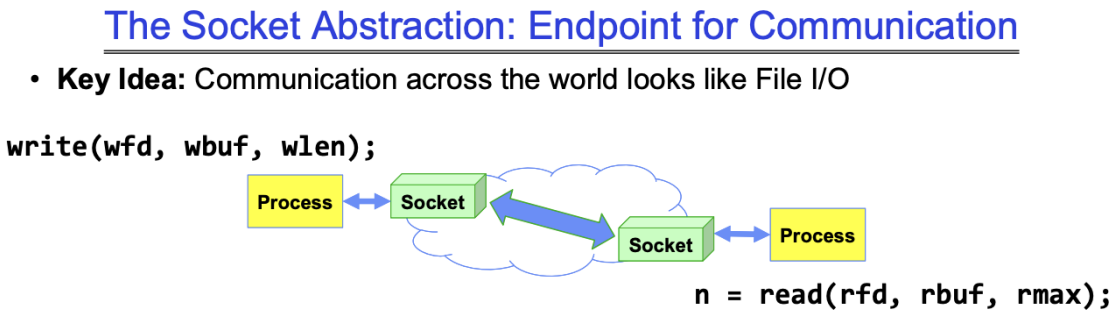


This diagram represents the protocol on how processes can communicate with each other using these file operations. If a client process wants to make a request, it will issue a write to the Server process that will read in this write and write its output, then send it back to the client. The client will then read this response and use it for whatever purpose they have.

This idea can be extended to network communications. Here we will introduce an abstraction that allows us to communicate across the world.

Socket: An abstraction for an endpoint of a network connection

This image will show how a socket is used for network communication.



The key idea here is that we can use our file IO to communicate across a network, and we can use sockets to be the established points where a process can recieve input/ouput from other processes. The question you might be asking right know, given now that we can communicate is things such as, how do we know where the request is coming from? How does the client request know where to connect? We will go over the namespace communication for sockets.

Namespace

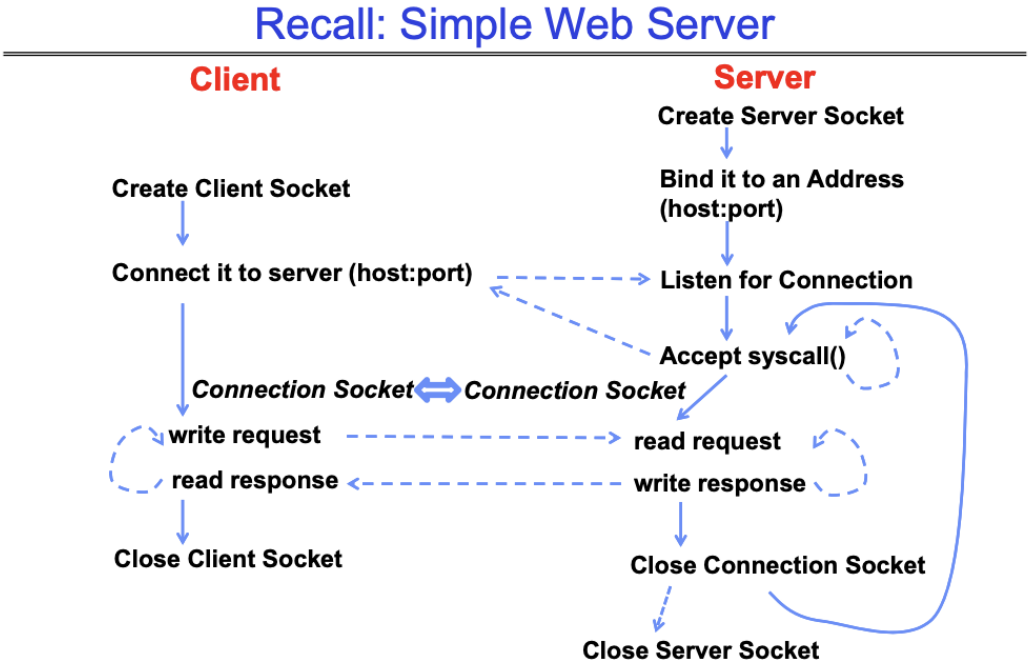
- IP address: a unique identifier that each machine has that is connected to the internet
- Port number: Essientially where on the Server the client will connect to to complete their request.

So when a connection is made, a **5-tuple** is sent with each connection. This will contain:

- Source IP Address
- Destination IP Address
- Source Port Number
- Destination Port Number
- Protocol (Usually TCP)

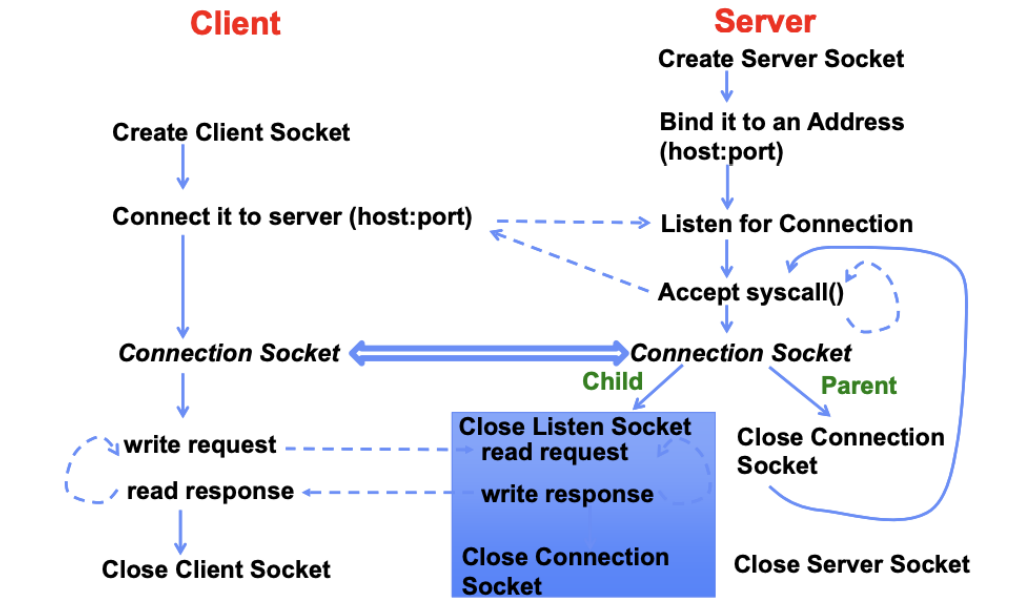
Lastly, there are some things that we will assume when working with connections. First is reliability. That is when we write to a file or across a network and read it from the other side. No data is lost. Second is the property of things being in order. if I send byte x then byte y. The receiver should get byte x then byte y in that order.

2.1 A Simple Web Server Connection



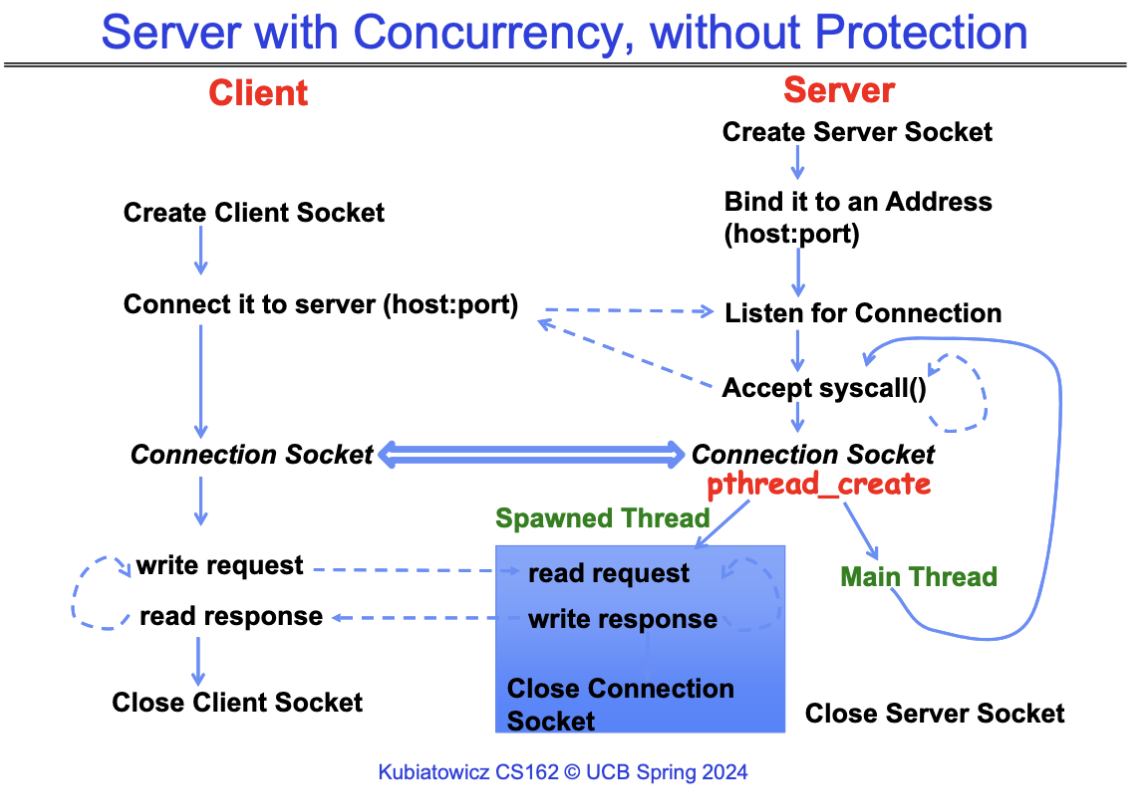
Above is a diagram of how a simple we server works. The client requests to connect to the server and the server will wait and listen until a client request. After the request has been recognized. It will accept the connection with the accept() syscall. We will then read what the client's request is and then write the response. Remember this is using the POSIX principle that everything is a file.

The main thing that is wrong with this setup is the server can only serve one client at a time. Because it needs to finish the client's request before it can close the socket and accept the next connection. What we can do instead is every time we get a new connection, we fork our process and have the child process create the connection socket and handle the request. That way the server socket can go on to accept the next client request.



We see in this setup, once the child process is created. The parent immediately closes the connection socket and looks for the next connection. We also see that the child will close the listen socket because there is no need for it since the connection has already been made. This is to make good use of our resources and not have something handing around if it doesn't need to.

While this method is better than the simple web server. We can do better performace wise. We know that creating new processes and switching between them is expensive. They have more overhead than threads. So why don't we just use threads instead?



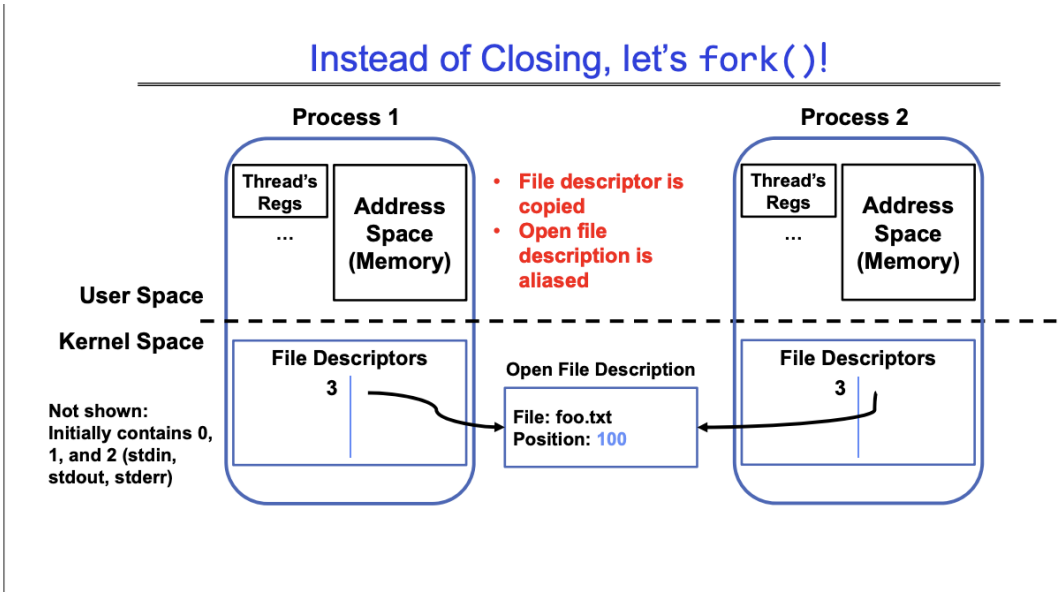
We see this is very simliar to the process creation version of this server, but the main thread contains the server socket and can be seen as the "manager". The spawned threads can be thought of as the "workers"

2.2 Looking at File Descriptors inside of Processes

We already know what our definition of a process is: The instance of a running program. When a process is created. It has its own address space and one or more threads. We have mentioned the process control block, which holds vital information about the process such as the register values, state of process, process id (pid), its memory space and more. What it also holds is a file descriptor table which is held inside the kernel. This file descriptor table will show all of the files that are currently present in a given process. Every process will have files with file descriptors 0,1, and 2 These correspond to the following:

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

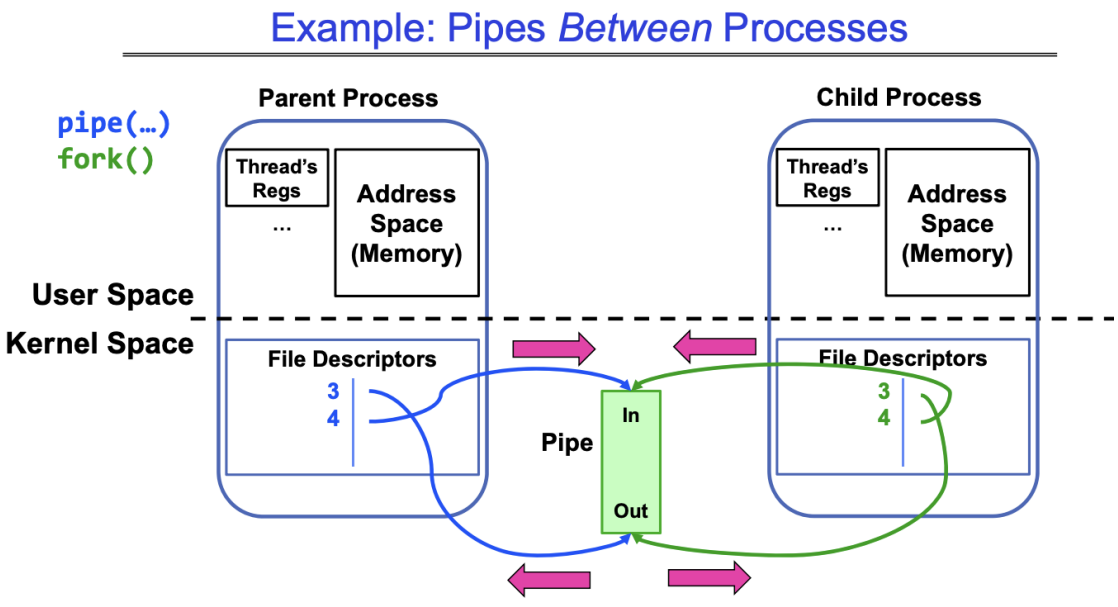
  So in a process, it we call something like read() on a file descriptor, it will move the file pointer (where we are pointing to in the file) for that specific file description. This will make more sense when we see an example using fork.



So we see we fork on process 1 to get process 2. Everything gets copied over. So both file descriptors will actually point to the same file description. Meaning we can perform file IO from either process and it will affect both processes. With things such as the where the file pointer currently is and what contents are in the file. So we can close one of the file descriptors and still have the other file description pointing to that file description, so that file description is not lost! This is important for how pipes work.

2.3 Pipes

A pipe is a communication method that allows one process to send data to another process. We do this via file IO once again.



We essentially have two file descriptors in each process, one pointing to the in portion of the pipe and one pointing to the out portion of the pipe. The in portion of the pipe and the out portion are both file descriptions.

3 More on Threads

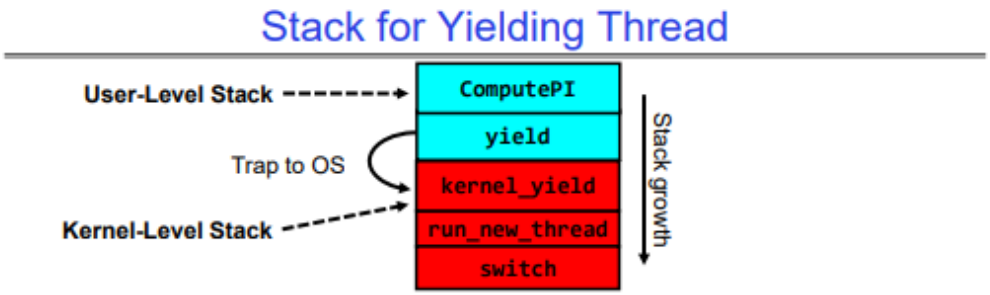
So far in this class, we know what threads are, they are an execution context in a give process. And in a thread, it gets its own stack to work with, but also shares sections of memory such as the heap, code and static. Now we will go more over how we actually run a thread. Here is the process:

- First load its state which are things like the registers, the PC, stack pointer into the CPU
- Load the environment (virtual memory space)
- Once everything is set up, Jump to the PC

After all of these things, happen, control is given to the user code and the thread begins to run. Since for now, we will work with a single-core CPU, once the thread is executing, nothing else can be running at that time, not even the OS.
So how does the OS, gain back control. There are two general ways this can happen.

3.1 Internal Events

Internal events are things that happen inside the thread while it is running examples of this are things such as IO requests, waiting on a signal from another thread, or explicitly calling yield() in a thread. All of these ways will cause the thread to yield to the CPU, letting another thread execute.
When we switch to a new thread, we will first save vital information about the thread such as the PC, regs, stack pointer. Here is how we run a new thread.



We see from this picture, that when we want to begin running a new thread, we go into the kernel, to begin running kernel code. We then select which thread to run next, and then switch the execution context from the current thread to the new thread.

One important note is that when we switch threads, the kernel stack remains until we continue execution of the thread we yielded, It might be a little hard to explain in words so I would suggest watching the recorded lecture on this part. But essentially, once we return back to executing this thread, the kernel stack will disappear and resume execution like normal.

One of the most important things about switching the thread that is executing is the switch function itself, this is really really really vital in ensuring a deterministic program, and if it screwed up in the slightest way, we will get what is called a subtle bug, which are the hardest bugs to find. Really ensure that your switch functions are implemented correctly.

3.1.1 The overhead of context switching

Switching threads in the same process are much cheaper than say switching between process. This is simply because we have less things we need to change. As we know threads share the same address space, we just need to switch out the program counter, the registers and the stack pointer.

3.2 External Events

So what happens in the case that no internal events happend, that is there is no IO, no waiting for other threads, and no yielding? This is then external events come in.

External events usually come in the form of interrupts, Which signals the hardware or software to stop running the code currently running and jump to the kernel. These interrupts usually happen from a timer, which is a hardware elements what will go off every so many milliseconds so send an interrupt to switch the context thread. When an interrupt executes, the same process with internal events take place, we save all of the important information in the thread control block and load in the next thread from its respective thread control block.

3.3 Starting a Completely New Thread

As we know the scheduler in the CPU will decide which thread to run when it is time to decide. But when we switch to a new thread, we want to immediately start executing the next instruction in the thread, we do not want to deal with any setting up of the thread. This problem arises when a new thread is created and we must combat it.

So when we make a new thread. We need to do some setup before we can start executing. These things will include

- Setup with the TCB to point at the new user stack and ThreadRoot Code
- Set up the stack with putting the instruction pointer at the start of the function and putting arguments on the stack.

Hopefully this provides some context on how to properly make new threads and switch the context to begin executing another thread.

4 Concurrency Vs Parallelism

We have discussed a little bit about what concurrency and parallelism is, we will review some definitions we have covered already:

- **Multi-threading:** Having multiple threads exist in a process
- **Multi-plexing:** Sharing a single resources (like a CPU) among multiple threads.
- **Concurrency:** Being able to interleave multiple items arbitrarily.
- **Parallelism:** Running multiple process or threads at the same time.

4.1 Issues With Single-Threaded Programs

Usually the issue with single-threaded programs is that they are slow when waiting for IO operations or we could be taking advatnage of all of the processing cores on our CPU. We will look at some solutions to speeding up our single-threaded programs.

4.1.1 Event-Driven Programming

In an event-driven program, the program is designed to react when events happen. Some advantages is has over single-threaded programs is that they employ non-blocking IO. That is if an IO operation takes places, the application can initiate the IO operation then continue to execute other code, compared to a single-threaded application where it would have to wait for the IO operation to finish. It is usually compromised of an event loop, which is a control structure that will wait to see if new events have come. When a new event comes, we need to handle it with what are called event handlers.

The issue with event programming is that some events might be dependent on other events, which would block some events from happening. While event driven programming is useful in certain contexts like graphical programming, we will instead look for a different approach in this class:

4.1.2 Multi-threading

As we defined before, multi-threading utilizes more than one thread in a given process. This allows us to divide up work in a given program to speed it up, to utilize all the cores on our machine and not be blocked by IO operations.

The most apparent problem with multi-threading, is that they can produce a non-deterministic output. This really happens because the schedule can interleave the execution of our threads in any way it pleases. And we should always assume the scheduler will schedule the threads to run in the worst way possible. This is where we will need some new terms to combat this issue:

- **Atomic operation**: An operation that when started, must finish in its entirety before switching to another thread. It always runs til completion.
- **Synchronization**: using atomic operations to ensure the cooperation between threads.
- **Mutual Exclusion**: Ensuring that at a critical section of code, only one thread does a particular thing at a time.

You may think that Mutual exclusion kind of defeats the purpose of multi-threading. I mean hell now we cannot parallelize. However, this only happens for certain sections of code, mainly sections where we are manipulating shared resources amongst the threads. Because if none of the threads share any resources, then theoretically no data races will occur. So we only create critical sections on shared resources amongst the threads. We can do this with the following operations

- **Lock():** Prevents another thread from doing something, We would do this before entering a critical section
- **Unlock():** Allows other threads to do things, do this when leaving a critical section.

4.2 Synchronization

So know that we know what multi-threading is and some of the possible dangers that come with that, we will aim to go more into depth on synchronization issues and how to make our programs both fast and safe to use.

4.2.1 Example: Too much milk

This example was taken from lecture. Imagine we live in a house in which the number one rule is that there is always milk in the fridge. Whenever someone comes home, they open the fridge, if there is no milk in the fridge, they leave to get some milk. Now imagine this situation.

- Student A comes home at 3:00. They see there is no milk in the fridge so they leave to the store. They get to the store at 3:10 and buys milk at 3:15
- Student B comes home at 3:05 and sees there is no milk in the fridge so they also leave to the store. They get to the store at 3:15 and buys milk at 3:20

Do you see the problem here. There was no communication between student A and student B, which made them both go to the store and buy milk. Now some of it will probably go bad.

This is a very similar issue to synchronization in programs, We need to establish some sort of way so when something happens involving pieces of data shared amongst threads. We must have some sort of communication between threads so no one goes stepping on other threads toes. First lets define some things both new and what we already know.

- **Acquire:** Acquiring a lock will make it so no other thread will be able to execute instruction that are located after the acquisation of the lock
- **Release:** Releasing a lock will end of a critical section, meaning other another thread can execute instructions within this section.
- **Atomic operation:** An instruction or sequence of instructions that if started by a thread, will always complete in its entirety (i.e a context switch would never happen during the middle of one of these operations)

So now the question becomes, how do we implement these locks? one could consider this naive implementation

- **Acquire() :** Disable interrupts
- **Release() :** Enable interrupts

Do you see the problem with this implementation. One thing that could happen is a user acquires a lock, then goes into an infinite loop, this mean that this process would just spin forever, since no other thread can execute their instructions since external interrupts were disabled by Acquire(). We need to think of a better way to **Acquire** and **Release** locks. Consider the following:

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

```
Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
}
```

Lets break **Acquire** down:

- Acquire: First we disable interrupts so that the **Acquire** is an atomic operation
- Next, we want to check if the critical section is BUSY, if it is BUSY, that means that another is thread is currently running in this context. If it is busy, we will put the current thread on the wait queue to execute this critical section, and then put it to sleep
- If the critical section is not BUSY, then we will simply set the critical section to BUSY and re-enable interrupts, this thread will then execute the critical section.

Next, we will break **Release** down:

- Since we are releasing the lock on a given thread, we will check if any thread is on the wait queue. This means that another thread is waiting to execute this critical section. If there is a thread on the wait queue, we will remove the thread from the wait queue and place it on the ready queue, meaning it is ready for execution.

- If there is no thread on the wait queue, we will set the critical section as FREE meaning, no one is currently in the critical section and another thread can begin executing when it gets there.
- Lastly re-enable interrupts.

This is definitely an improvement on the naive implementation we had before for **Acquire** and **Release**. But this implementation we came up with still requires us to go into kernel mode, which has a lot of overhead. We want to improve this further.

We will try to use an atomic operation that will allow us to stay in user mode when executing **Acquire** and **Release**

- Simple lock that doesn't require entry into the kernel:

```
// (Free) Can access this memory location from user space!
int mylock = 0; // Interface: acquire(&mylock);
                //            release(&mylock);
acquire(int *thelock) {
   while (test&set(thelock)); // Atomic operation!
}
release(int *thelock) {
   *thelock = 0;              // Atomic operation!
}
```

To understand the way this works, lets look at the following cases;

- if the lock is FREE (i.e lock is 0), TestAndSet will return 0 and set the lock to 1. Here, the 1 indicates the lock is now BUSY
- if the lock is BUSY(i.e lock is 1), TestAndSet will return 1 and set the lock to 1. So really nothing has changed, which makes sense since the the lock is still BUSY
- When the lock is released, we simply just set the lock to 0, indicating that it is FREE and another thread can acquire the lock.

The advantage with this implementation is that we do not need to enter kernel mode, which saves a lot of overhead. A disadvantage with this implementation, is that if a thread is trying to execute a critical section that is busy, it will spin (kinda just keep executing until the lock becomes free) instead of going to sleep. This ends up using valuable resources too such as the cache and network bandwidth.

4.3 Semaphores

Semaphores are another synchronization tool we can use for programs. The ways they work is they use non-negative integer to coordinate threads. Semaphores have two atomic operations

- Up: Increment count
- Down: Decrement count

How a Semaphore works is it starts at some value, and every time we have a thread entering something, we decrement the count. Then every time a thread leaves something, we increment the count. A thread cannot enter a given section protected by a semaphore if the semaphore count is 0. This is the protection part of the semaphore that controls how many threads are doing something. In fact if we initialize the lock to 1, it behaves just like a lock.

4.4 Monitors

Before we get into what monitors are, lets look at an example that motivates the use of monitors.

4.4.1 The Coke Machine

So we have a coke machine and at has consumers and producers. Whenever a consumer wants to go to the coke machine, they will first check if the machine is not empty (has at least one can of coke). And then take a coke.

Then there is the producers perspective. They will come to fill the coke machine. First they will check if the machine is not full (there is at least one empty slot) and fill the machine.

```
lock buf_lock;                  // Initially unlocked
condition buf_CV;               // Initially empty
queue queue;                    // Actual queue!

Producer(item) {
    acquire(&buf_lock);         // Get Lock
    enqueue(&queue,item);       // Add item
    cond_signal(&buf_CV);       // Signal any waiters
    release(&buf_lock);         // Release Lock
}

Consumer() {
    acquire(&buf_lock);         // Get Lock
    while (isEmpty(&queue)) {
        cond_wait(&buf_CV, &buf_lock); // If empty, sleep
    }
    item = dequeue(&queue);     // Get next item
    release(&buf_lock);         // Release Lock
    return(item);
}
```

Kubiatowicz CS162 © UCB Spring 2024

This is how we can do this. We need a way to schedule our constraints on for this problem, because when a consumer comes and has to wait for coke, they should wait until a producer comes and fills the machine with coke. The problem with what we have learned so far is that there is that locks are used for mutual exclusion and semaphore are dual purpose, meaning they can be used for locks or mutual exclusion. The problem with them being dual, is that in some situations of mutual exclusion, they can be very complicated to implement and the order of things really matter. The code above uses what is called monitors to accomplish our synchronization. Monitors are a scheduling tool to help with synchronization and they have 3 main functions:

- wait: puts a thread on the wait queue
- Signal: wake up one thread (if any) on the wait queue to run
- Broadcast: wake up all threads on the wait queue to run.

Monitors use what are called condition variables to synchronize threads. What a condition variable is a queue of threads that are waiting to enter a critical section. Note that we don't really need to specify the actual condition inside the condition variable, meaning the condition variable's value does not really represent the condition. It is more used as a marker for what threads are waiting to be signals so they can run a specific critical section. There are two main types of Monitors that are used today: Mesa and Hoare monitors

4.4.2 MESA

The main thing how MESA scheduling works is that when a thread is put on the wait queue, there is no priority for these threads. Meaning that we can put thread x on the wait queue, then thread

y, and when we signal, thread y could be the thread that comes off the queue, even though we put x first. This is easier to implement so for most OS, this is the method that is used.

4.4.3 Hoare

How Hoare works is pretty much the exact opposite of how MESA works. When threads are put on the wait queue, the queue is in the order of how they were put. So if I put thread x , then thread y on the queue. When they get signaled, thread x will always wake up first.

4.4.4 Readers and Writers Example

We will go over one more synchronization example that will go over monitors once again. In this example, we will have a data base where there are readers and writers. Here are some correctness constraints for this data base

- Only one writer can be in the database at a time
- If there is a writer in the database, a reader must wait until the writer is finished.
- If there are no writers in the database, many readers and be in the database
- Priority is given to the writers

```
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);
}
```

Here is the code for the reader. Notice we first acquire the lock. This is because we must make sure that the reader has the opportunity to check if the database is available atomically. We first check if there are any active readers or waiting readers currently in the database. Note we also check if there are any waiting writers because in this database we give priority to the writers. So if there are any writers that are currently waiting in the database, we must give them priority to go first. If there are any writers that are active or waiting, we will use a monitor to have the current reader wait. After we get out of the while loop we are now active so we can release the lock, letting other readers attempt to come in the database. Lastly we need to leave the database. So once again we will acquire the lock which must be done synchronously. The main thing to do here is the check the number of active readers and waiting writers. We want to signal writers that is good to write if there are no active readers and there exists at least one waiting writer. This will fit our correctness constraints.

```
Writer() {
  // First check self into system
  acquire(&lock);
  while ((AW + AR) > 0) { // Is it safe to write?
    WW++;                 // No. Active users exist
    cond_wait(&okToWrite,&lock);  // Sleep on cond var
    WW--;                 // No longer waiting
  }
  AW++;                   // Now we are active!
  release(&lock);
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  acquire(&lock);
  AW--;                   // No longer active
  if (WW > 0){            // Give priority to writers
    cond_signal(&okToWrite);// Wake up one writer
  } else if (WR > 0) {    // Otherwise, wake reader
    cond_broadcast(&okToRead); // Wake all readers
  }
  release(&lock);
}
```

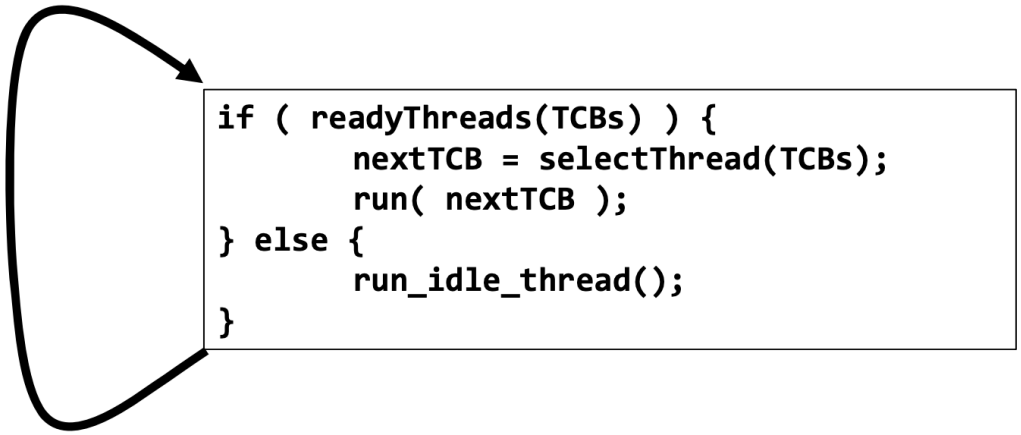
Here is the writer code. Notice the structure of it is very similar to the reader code. There are a few differences. In the first while loop, we are checking if there are any active readers or active writers. Because in our correctness constraints, we cannot allow a writer in if either of those are true. In the exit, we simple check if there any waiting writers and if there are we signal them to come through to write. If there are no waiting writers, we also will check if there are any waiting readers, if there are we can do a broadcast, which is signaling all of them to come through. Notice.

Notice how this scheme gives priority to the writers, since writers will always get the green light to go before the readers. The reason this is done is policy. The policy that was decided for this scheme was to give the writers priority so the readers can have the most up to date information from the database.

The important thing to take away from this example. Is one, another use of monitors, and also showing that this is a form of scheduling, were we have the power to synchronize in a way that fits some policy that we have decided is the best. We will go over scheduling more in the next section.

5 Scheduling

We just briefly touched on scheduling from the last section, and now we will talk about CPU scheduling. That is how does the CPU decide which process to schedule next on one if its cores? In this section we will go over different policies that are used.



This is the simplest scheduler you can have, we just get the next available thread and run it. This is similar to the first scheme we will go over which is First Come First Serve (FCFS).

5.1 Calculating Throughput, Wait Time, Completion Time

Here are the ways we calculate each:

- Throughput: Given certain amount of time and the number of processes completed in that time, we simply take the ratio to get through throughput
- Average Wait time: For each process we calculate the total amount of time it waited to be ran since it was last scheduled. Take all of these times, sum them up and divide by the amount of processes
- Average Completion Time: For each process, take note of the time in which the process finished, then again take all the processes, sum them up and divide by the amount of processes.

5.2 Scheduling Policies

In this section we will go over different schemes such as FCFS, Round Robin (RR). Shorest Job First (SJF) and shorrest remaining time first (SRTF)

5.2.1 First Come First Serve

The name for this policy is exactly what it sounds like. Whoever is at the front of the queue will get the CPU and will run the whole process til completion. It would look something like the following:

- First-Come, First-Served (FCFS)
  - Also "First In, First Out" (FIFO) or "Run until done"
    - » In early systems, FCFS meant one program scheduled until done (including I/O)
    - » Now, means keep CPU until thread blocks
- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

  - Suppose processes arrive in the order: P_1 , P_2 , P_3 The Gantt Chart for the schedule is:

  - Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
  - Average waiting time: (0 + 24 + 27)/3 = 17
  - Average Completion time: (24 + 27 + 30)/3 = 27
- *Convoy effect:* short process stuck behind long process

In this figure, P1 came first into the queue, then P2, then P3. So this will be the order that they are ran. Notice how this is not very effiction when it comes to the average completion time and average waiting time. This is because the shorter process are stuck behind the longer processes. Causing them to be completed later. However in FCFS we have the lowest amount of overhead because there is no context switching involved due to external sources. We simply run the whole process until it is completed and then switch.

5.2.2 Shortest Job First

The next scheme we will talk about is similar to FCFS, SJF is similar to FCFS in that the whole process will run before switching to another process. The main difference is the order the process are scheduled are sorted in increasing order in terms of time to complete. Hence the shortest job is ran first. This will improve our average completion time and the average waiting time. However there is some issues that this policy does not address. First, this policy can lead to starvation. That means that, some process in the back of the queue might not ever get ran because process that have a shorter time will get scheduled before it. It also does not account for if we are running a given process, and then another process comes into the queue that has a shorter time to finish than the current process. This is where the next policy solves that.

5.2.3 Shortest Remaning Time First

SRTF is a policy that is preemptive. Preemptive simply means here that the cpu can interrupt the current process that is running to switch it out for another process. The pre-emption that is done is based the on the current remaining time to complete a process. So if we get a process that has a shorter time remaining to finish than the process currently being ran (the previous shortest process) we will interrupt the current program being run and perform a context switch to run the shorter one first. The main issue, similar to shortest job first is this leads to starvation.

5.2.4 Round Robin

Round Robin is the policy that is most commonly used in actual scheduling in the real work because it provides the most fairness and overall performance. This is how it works: Each process has a given amount of time to run, we call this amount of time the quantum q after the time running the process has reached q, we will perform a context switch to start running the next process. The important thing with RR is how we pick our quantum. Here are some characteristics the quantum:

- $q \to 0$: lots of context switching, not very efficient because there is lots of overhead
- $q \to \text{inf}$: FCFS, no interrupts would occur because a process will not take the quantum.

We ideally whan to choose the quantum so it is about 100x bigger than the time it takes to do a context switch

5.2.5 Multi-lvl Feedback Scheduling

In multilevel feedback scheduling, we use the concept of round robin, but the difference is that we will have multiple queues each with a different quantum. There are different priorities to the queues, the queues with the higher prioity are the ones with the smaller quantums. Here is the general algorithm of how it works:

- Every task is first assigned to the highest priority queue, which is the one with the smallest quantum.
- If a task that is currently running takes more time to finish than the quantum, it gets demoted to the next highest priority queue with a larger quantum, the quantum is usually 2x longer.
- If a task takes shorter than the quantum allowed, promote it to the higher prioirty queue, or stay at the top if it already highest priority.

In this way of scheduling, there are different policies we can implement. One policy could be to always serve the queue with the highest priority queue first. However, this way could lead to starvation, that is the lower priority tasks will never run. Another way of doing it is dividing up CPU time to the different queues. One scheme could be give 70 percent to the highest queue 20 percent to the next queue and so on.

5.2.6 The Linux O(1) scheduler

One of the first linux schedulers was the O(1) scheduler. It is called the O(1) scheduler because every single operation to do with the scheduler is O(1) time. It had 140 prioirites, 40 of them for user tasks and 100 of them for kernel tasks. More abou this scheduler is covered in lecture, but the problem with this scheduler was its complexity. It had so many features that it became really hard to manage when someone wanted to add a new thing to the OS or change something about it. It might make one person satisfied but it could make previous users dissatisfied with how it is now.

5.3 Multi-Core Scheduling

Most concepts about scheduling stay the same when we move to a multi-core system. However, there are a couple things to note about it.

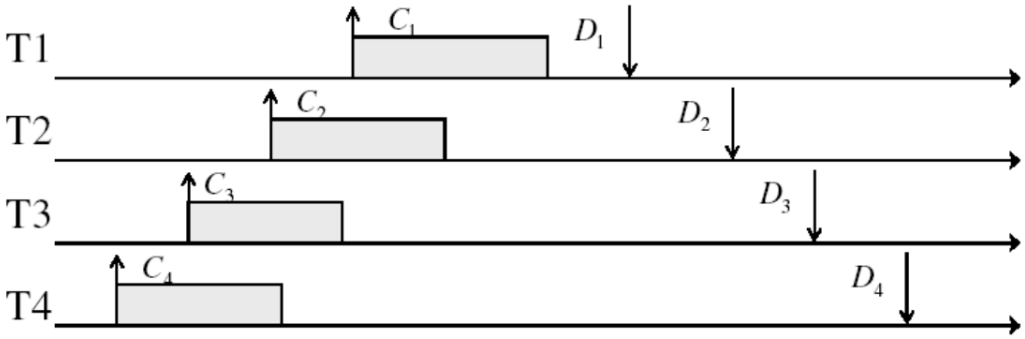
One thing is that if a thread is scheduled on a certain core, when that thread is scheduled again, the CPU will attempt to put it on the same core. This is because of better performance from the Cache and branch prediction, these things might still have information from when that thread was previously ran.

5.3.1 Gang scheduling

The idea of gang scheduling is that if we are doing some sort of parallel opertaions (like SIMD from 61C), we like to schedule the threads together (in a gang). This is also a way that spin locks might be better than putting threads to sleep while waiting. It could be faster to have a thread be spinning while the other threads complete their operations and perhaps release the lock, instead of putting the thread to sleep and waking it up, because as we mentioned, there is a lot of overhead in doing those things.

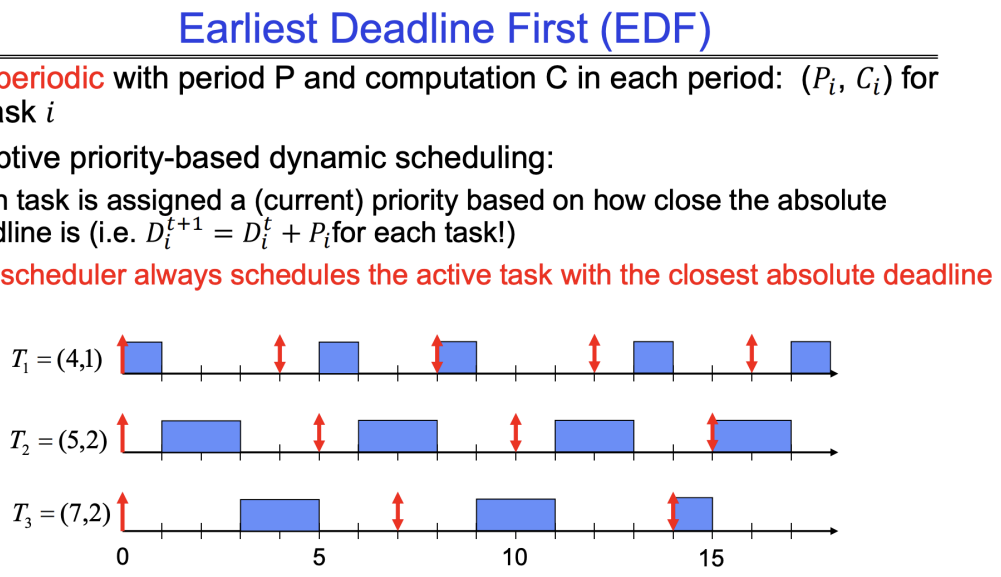
5.4 Real-time Scheduling

The goal with real-time scheduling is that we want to be able to predict the worst-case possible runtime for a given task, that way we can better schedule the tasks. Take a look at this picture for example.



we have 4 tasks, labele $T1...T4$ each with a computation time C and a dealing D we need to finish each task before its deadline. We see the way this is scheduled does not make sense because at almost any time, there are 2 tasks that are scheduled at the same time, we need to come up with scheduling schemes that will meet the requirements we deined

5.4.1 Earliest Deadline First



It idea with EDF is that we will always schedule the tasks that has the earliest deadline. In EDF, each task has a period P and a computation time C. We see the dealine is calculated from $D_i^{t+1} = D_i^t + P_i$

From looking at the picture, we can think of it like this. The read line is the deadline we need to meet so whatever thread has the earliest deadline, we will run that task first. This is the best thing we can do to ensure that all tasks will be finished by their deadline.

Even with EDF being the best we can do, it is still not possible to accomplish every task before their deadline if there are too many tasks, we will simply use too much of the CPU (more than 100 percent). There is a way to check if using EDF will accomplish all tasks by their deadline.

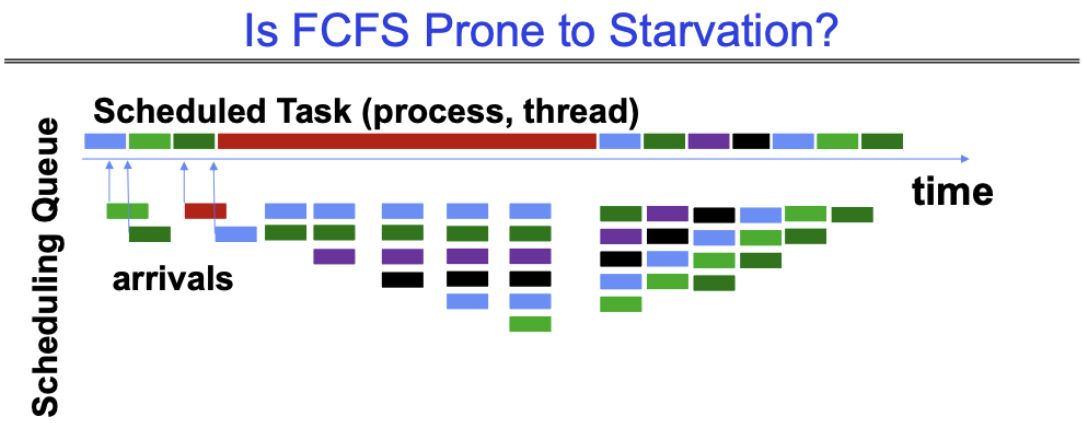
$$\sum_{i=1}^{n} \left(\frac{C_i}{D_i} \right) \leq 1$$

This inequality says that if we have n tasks each with a completion time C and deadline D if the sum of their ratios is greater than 1, it is not possible to complete all tasks before their deadline.

5.5 Starvation

5.5.1 FCFS Starvation

Recall in FCFS, threads will run in the order they are queued, like a line in the grocery store. So you would think that FCFS is not prone to starvation, since all the threads will be ran in the order they are queued. But imaging you are in line in the grocery store and someone is trying to use all their damn coupons and they take fucking 30 minutes to checkout. The same idea can be applied here.



We see there that the red tasks is taking up all the CPU time, this can either be a malicious task or a buggy tasks, either way imaging if red never finished, then the thread queued after will be starved.

5.5.2 Priority inversion

The main thing that we have already talked about with priority scheduling is that it leads to starvation, meaning that tasks with lower priority will not run (starved) since the higher priority tasks will always be scheduled. Recall this. The next priority issue that relates to this has to do with locks and is called a priority inversion. Basically, imaging a thread with low priority is ran and acquires a lock, then another thread of high priority is ran and wants to acquire that same lock. The issue not is that higher priority task has to wait and there could be other high priority tasks that are run after, meaning that the low priority task is never ran and the high priority task can never proceed because it is forever waiting on the lock. To counter this, we do what is called a priority donation. That is if a higher priority H thread wants to acqure a lock from a lower priority thread L , we will donate H's priority to L, so now L has the same priority as H and will be scheduled.

So now the question is how can we still give priority to tasks that need to be completed fast, while not starving out other tasks? That is where Proportional-Share Scheduling comes in

5.6 Proportional-Share Scheduling

The idea with with scheduling scheme is that we want to give each job a share of the CPU according to its priority. So higher priority jobs will get ore of the CPU than low prioirty jobs. So how are we gonna do this?

5.6.1 Lottery Scheduling

For each job we will give it a certain number of lottery tickets. Higher priority jobs will get more lottery tickets than low priority jobs. To avoid starvation, each job will get at least 1 lottery ticket. When it is time to schedule the next job, we will pick a random lottery number and whatever job has that number will be scheduled. The problem with this is the randomness of the scheduler, if we had two jobs both with the same number of tickets, it is not guaranteed that the two jobs will be given the same CPU time. This leads to unfairness.

5.6.2 Stride Scheduling

To try and counter the problem that we had in lottery scheduling, we have stride scheduling. This is the outline of stride scheduling:

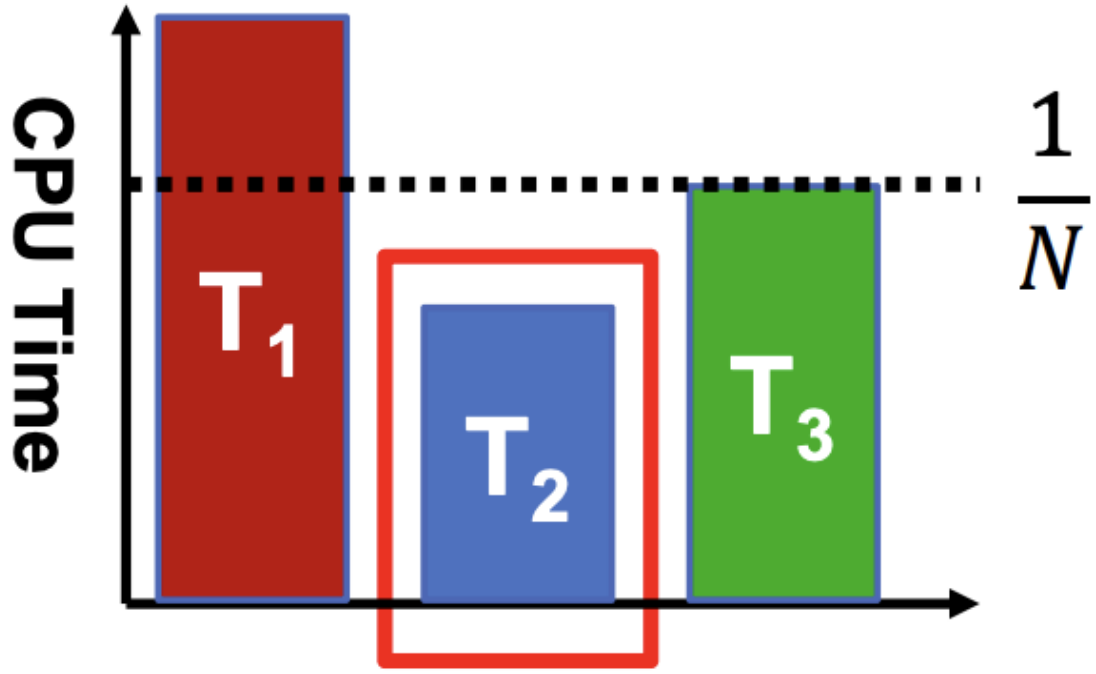
- The formula for calculating a stride is $\frac{W}{N_i}$ where W is the total number of tickets alloted and N_i is the number of tickets task i has.

- Each task then has a pass counter
- schedule the task with the lowest pass counter
- Before the scheduler switches tasks, increase the current job's pass counter by the stride

This method automatically prioritize jobs with a higher priority (more tickets).

5.6.3 Completely fair scheduling

Another method to promote fairness is to give each task in the CPU equal time. If we had N resources on the CPU, each task would get $\frac{1}{N}$ resources. Note that we can just give out $\frac{1}{N}$ resources to each task, in hardware this just really isn't possible. So we would do something like the following:



We are essentially keeping track of how much CPU time we have given to each task. so the task that has been ran the least will be scheduled next. There are some goals that we want with this scheduler so it is usable:

- **Low response time**: we want the OS to be able to support inter activeness like typing on a keyboard or something like that. To do this, we will need the notion of a target latency, that is what is the maximum amount of time we want any program to run? Remember the quantum is the amount of time we will give to a given task to run. We can assign the quantum as $\frac{TL}{N}$ where N is the number of tasks and TL is the target latency. Notice if the number of processes become very large, our quantum becomes very small, which recall is bad because that means we will have lots of overhead from context switching.
- **Throughput** To counteract the problem we were having with the quantum being too small, we will introduce the concept of **minimum granularity**, that is, the quantum will be at least the minimum granularity. So if the minimum granularity was 1ms and the quantum was calculated to be 0.1ms, it does not matter and the quantum will be assigned to be 1ms.

From previous scheduling schemes, we know that we will want to prioritize some tasks over others, since some are higher priority. So how do we translate this idea to the completely fair scheduler?

We can have different quantums for each task. We can calculate this by using weights for each task and calculating the quantum that way. It would be like the following:

$$Q_i = \frac{w_i}{\sum_p w_p}$$

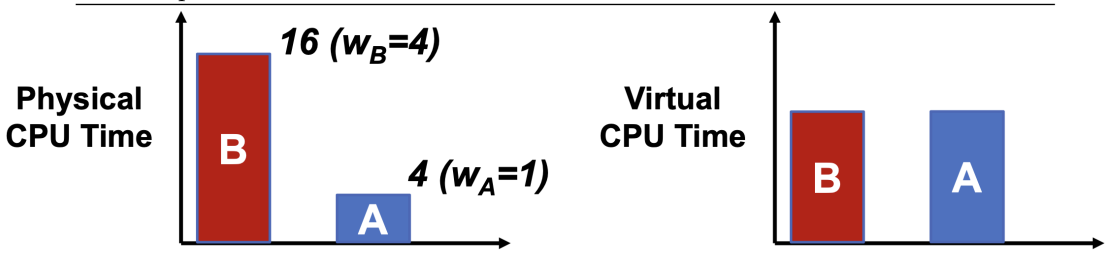
where p is the number of processes.

So we still need a way to calculate the weight of a given process. We will introduce the **nice** syscall in C which is a way to raise a given process' priority. Typically when we call nice, we are lowering the process' priority, since we are being 'nice' to other processes by letting them get more time. Note that when we lower the priority, we raise the 'nice' value. This is because we calculate the weight at such:

$$W = \frac{1024}{(1.25)^{nice}}$$

Note that the 1024 and 1.25 here are arbitrary. The main idea here though is that when we raise the **nice** value, it's weight will decrease.

So now that we are assigning weights to each task, it kind of is no longer completely fair scheduling. The way we will bring it back to this is running the CFS scheme according to the virtual CPU time. Here is a picture that describes this:



For example, we can look at task B. since it has 4 times more weight than A, after task B has finished running, we will divide the total time it ran by 4 and then add that to the counter that is keeping track of how much each task has ran. This way, in the way it looks to the scheduler, each task has the same weight, allowing us to run CFS.

5.7 Choosing the right scheduler

Here is a picture on the pros of each scheduler:

I Care About:	Then Choose:
CPU Throughput	FCFS
Avg. Response Time	SRTF Approximation
I/O Throughput	SRTF Approximation
Fairness (CPU Time)	Linux CFS
Fairness – Wait Time to Get CPU	Round Robin
Meeting Deadlines	EDF
Favoring Important Tasks	Priority

6 Deadlocks

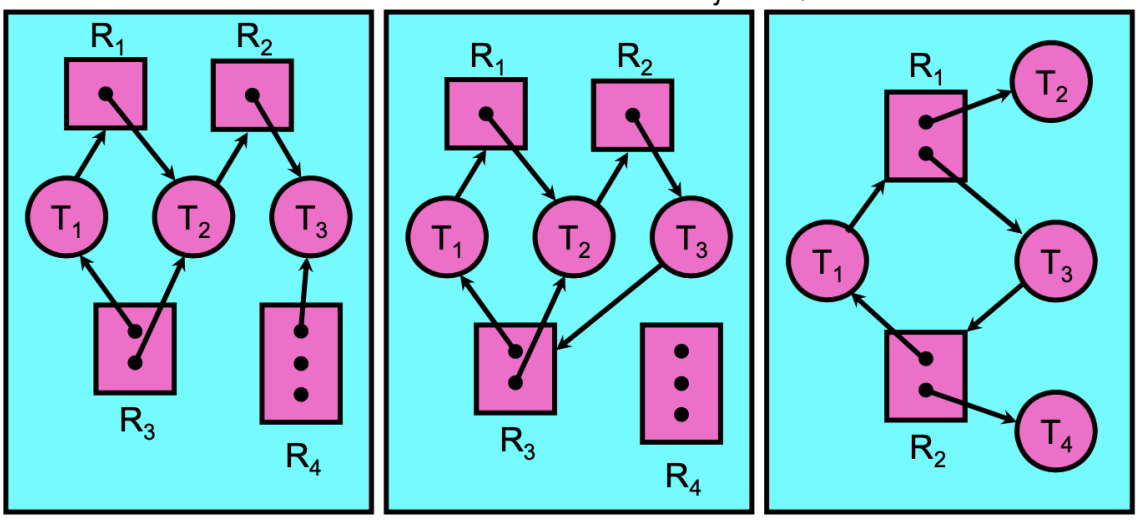
Deadlocks are a special type of starvation, we have already touched on the concepts of deadlocks with mutual exclusion, since they are very prone to come up if we are not careful. The reason deadlocks get their own category when talking about starvation, is because, usually in starvation, we would eventually be able to be fed (run our task). However in a deadlock, we will never be fed. Here are the 4 requirements to get a deadlock.

- Mutual exclusion: only one thread at a time can use a resrouce
- Hold and wait: All threads is holding a resource and is waiting to acquire another resource
- The threads themselves are the only ones that can release resources, in other words, there is no preemption
- A circular wait exists. That means the following:
  - There exists a set of threads $T_1, ...T_n$
  - T_1 holds a resources and is waiting for T_2's resource
  - T_2 holds a resources and is waiting for T_3's resource
  - ...
  - T_n holds a resources and is waiting for T_1's resource

The one nice thing about deadlocks is that all of these requirements need to be true for a deadlock to happen. That means if we can prove that one of these requirements is not present, we do not have a deadlock.

It would be nice to have a way to model resource allocations, so we can see if a deadlock exists. Luckily we can use graph theory for this.

- Model:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph

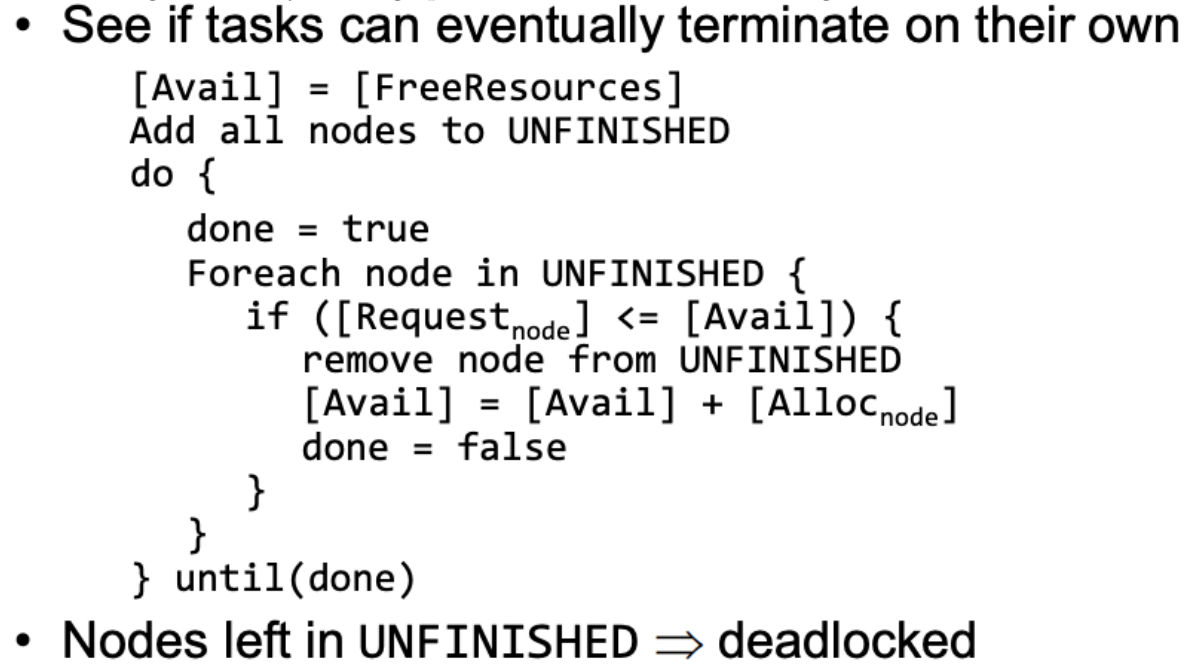
Allocation Graph With Deadlock

Allocation Graph With Cycle, but No Deadlock

One thing is that if no cycle exists, then it is not possible to have a deadlock. This is because there is **no circular wait**. This would be the case in the graph on the left. However, it is important to note that just because a circular wait exists, it does not mean that we have a deadlock. Because we have to remember that we can prove that one of the other deadlock requirements is not present. For example if we look at the far right graph, a cycle exists, but we see that T_2 is not waiting for another resource, since we fail that all threads must do this, we do not have a deadlock.

6.0.1 Detecting a Deadlock

Instead of having to manually look at graphs to determine a deadlock, An algorithms exists to do this for



us.

The for loop of this algorithm does is the following, it iterates through all the unfinished nodes:

- Have done initially set to True
- If what resources the current unfinished node is requesting is available, then that node can finish. So we will add that node's resoruces to the **Avail** vector and set done to **false**. This is because now there might be other nodes that can now finish since we freed the current node's resources

6.0.2 Dealing with a deadlock

There are four main ways we can deal with a deadlock

- prevention: don't write code that has deadlocks
- detection and recovery: let a deadlock happen and recover from it using some preemption
- avoidance: delay programs that are going to deadlock (requesting resources that would cause a deadlock)
- ignore: ignore the deadlock

6.0.3 Avoiding a deadlock: Bankers algorithm

The Banker's algorithm is a way to avoid a deadlock. Pretty much if granting a resource will put the thread in an unsafe state (can potentially have a deadlock), we will not grant the resource to it until it is safe, here is the general structure:

Banker's Algorithm for Avoiding Deadlock

```
[Avail] = [FreeResources]
     Add all nodes to UNFINISHED
     do {
        done = true
        Foreach node in UNFINISHED {
           if ([Max_node]-[Alloc_node] <= [Avail]) {
              remove node from UNFINISHED
              [Avail] = [Avail] + [Alloc_node]
              done = false
           }
        }
     } until(done)
```

- » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
- » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:
  ([Max_node]-[Alloc_node] <= [Avail]) for ([Request_node] <= [Avail])
  Grant request if result is deadlock free (conservative!)

What this algorithm is doing is the following, whenver a thread wants a resource, we will check if granting that resource will bring the program into an unsafe state. To do this, we will pretend that we just granted this resource and then run our deadlock detection algorithm that we desribed earlier, with a slight tweack, instead of checking if the number of resources is available, we will check that the maximum number of resources minus the allocated resrouces of that node is less than what is available. This pretty much guarantees that the node will finish.

7 Memory:

7.1 Intro

7.1.1 OS Roles

Recall some of the roles of the OS, it is an illusionist and a Referee. When we are dealing with memory, we want to make it seem like every process has all of the memory available to them and also ensure that no process has conflicting memory spaces when it is not supposed to. There are ways to ensure this functionality

7.1.2 Address Spaces

The definition of an address space is a set of accessible addresses and the state associated with them. Some basic things that you should know is if you have k bits, you have 2^k unique addresses,

7.1.3 Virtual Memory VS Physical Memory

There are two types of memory that both play key roles. We will see how these two types of memory are used to provide the functionality we see above.

Virtual Memory is the memory that processes deal with most of the time. It is not the real addresses of the system but doing this is able to provide the illusion that the process does not need to worry about what memory other processes are using. It makes it look like we have access to all the memory.

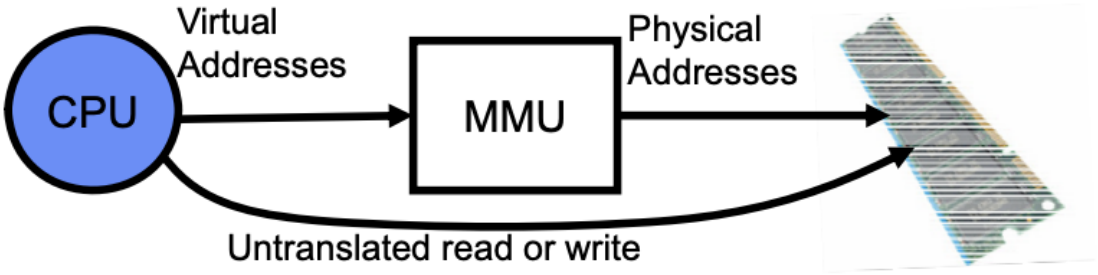
Physical Memory is the memory that is actually in the DRAM and Disk. These are the locations where data is actually stored. We will usually access the physical memory on load and store operations in a process.

So the memory where the data we care about is the physical memory, but we work with virtual addresses in each process so we are able to provide the illusion of infinite memory. So there is a need to have a way to translate between virtual and physical memory

7.2 Translation

7.2.1 The Memory Management Unit (MMU)

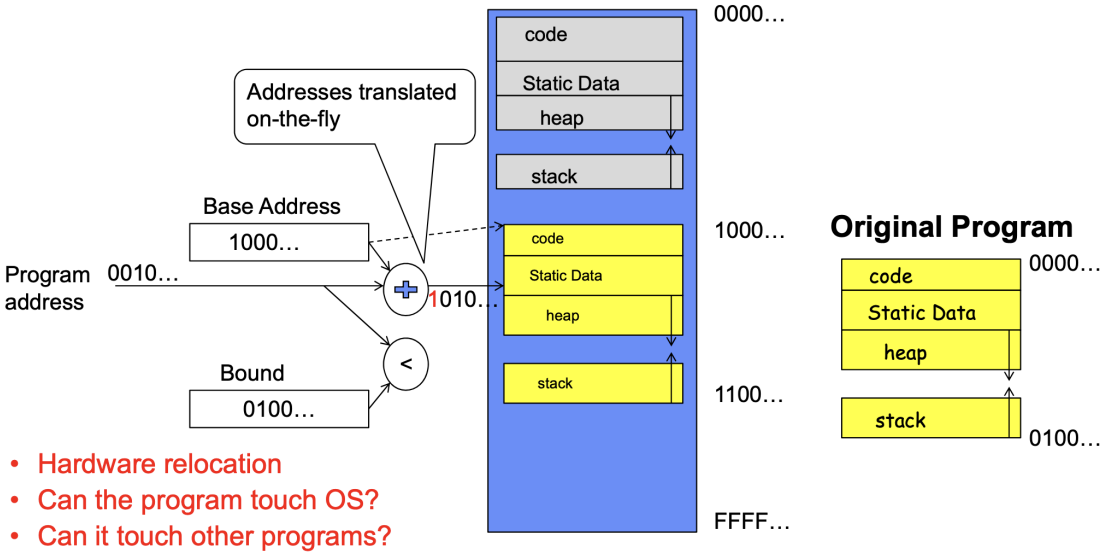
The memory management unit is the device that will take care of translating virtual addresses to physical addresses. You can think of the memory management unit like a seperate CPU for memory address



translation.
There are many ways of translating from virtual to physical addresses, we will go over them now.

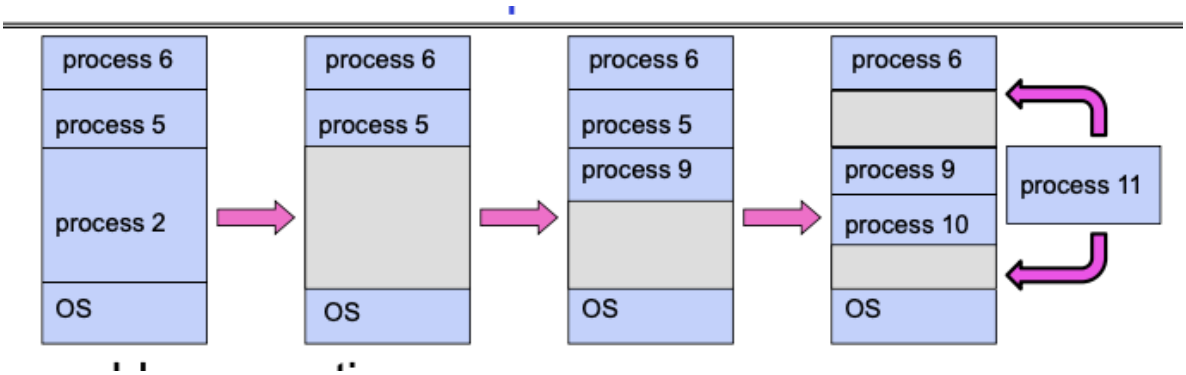
7.2.2 Simple Base and Bound

We have talked about simple base and bound before. Here is a picture as a reminder:



So for each process, it has a base address and then some bound of how much space the process has in its address space, So the total address space of the program is going to be from the **base to the base + bound**.

The issue with the simple base and bound scheme, is it leads to external fragmentation. This is because since each process will take up a different amount of physical memory, it will lead to gaps in physical memory that are fragmented, meaning we will only have space to fit in new processes that are small enough to fit into these gaps, but a process might not fit, hence we would be wasting space.



You can see from this image, that we have process 11 but we do not have room to put it even through we have two gaps.

7.2.3 Segmentation

In segmentation, we pretty much can split up the memory of the process into different sections. One example could be we make sections for the Code, Static, Stack, and Heap. So when we have a virtual address we need some way of knowing which segment it belongs to so we can get the correct physical address.

We see this address is split up into two different parts, first is the segmenttation number, which indicates which segment we are trying to access. The rest of the address is the offset, so where in the segment we are trying to access. The number of segment bits really depends on how many segments we need. From what we know about address spaces, if we have the split up we previously mentioned, we would need 2 segmentation bits, because we need to represent $2^2 = 4$ sections.

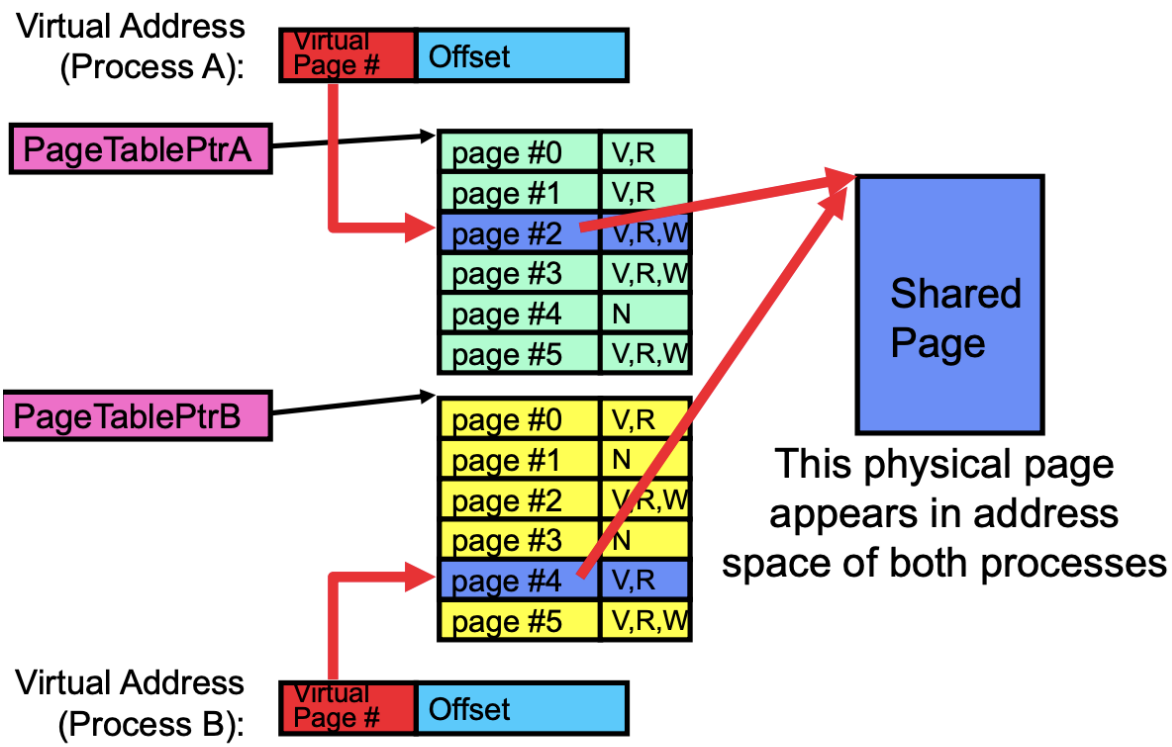
Notice that this still runs into the same issue as with simple base and bound, since the length of each segment is still variable in length, so external fragmentation still occurs. However, it does give us the ability to not be forced to keep the whole address space for a process all in the same place, we can split it up which can provide some benefits.

The solution to external fragmentation is called paging, which allocates memory in fixed size chunks.

7.2.4 Simple Paging

Simple paging involves having a page table, in why each entry in the table contains a translation from a virtual page to a physical page. We access this page table from what is called the page table pointer, which resides in the physical memory. Similar to segmentation, we will split up our virtual address into a virtual page number and the offset, we will use the virtual page number to access the correct entry in the page table to get the physical page number and then attach that offset to get the physical address we are looking for.

What is cool about this is that it allows multiple processes to share the same point in memory, since two page tables can map to the same physical page. Here is a picture to describe this:



7.2.5 Improvements on the Simple Paging Scheme

Some good things about the simple paging scheme we came up with is that it is very easy to implement, and sharing resources is pretty easy as well. However some cons is that if our processes are very big, our page table also needs to be very big, so our page table might be very sparse, since we might not need the whole address space. So one thing we can try to do to improve upon this scheme,

7.2.6 Multi-Level Page Table

TODO: Finish

7.3 Demand Paging

Demand paging is what happens when you try to get a page from the page table and it results in a page fault. This usually means we tried to access a page that is not mapped in our page table. What will happen is that the MMU will catch this and call the page fault handler, in which we will try and fetch from disk the data we want, once we get it, we can put it in memory. Here are the general steps that happen:

- We need to choose an old page to replace
- If the old page chosen was dirty, we should update its disk contents with the most up to date data
- Next we swap out the old page table entry for a new one with the page we want to pull in from disk.
- Lastly, we will continue the thread from its last starting location.

We can draw similarities from demand paging to caching. They both have similar policies when we miss on the piece of data we want to get, we need to swap out something old for the new thing we

want to bring in. So just like with caches, we can compute the effective access time which if you recall is

$$EAT = R_{HIT} * T_{HIT} + R_{MISS} * T_{MISS}$$

$$EAT = T_{HIT} + R_{MISS} + P_{MISS}$$

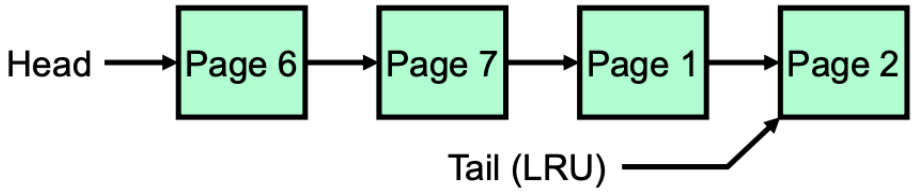
The effective access time will rise very quickly, even if we have a page fault every 1000 times. So it is very important to minimize the number of page faults we get.

7.3.1 Page replacement policies

Since it is very important that we have very few misses, the replacement policy we use will be extremely crucial in the speef of our system. Here are a few policies we have seen so far and how they would fair in demand paging:

- **FIFO:** Throw out the oldest page in the cache. This is the most fair policy since each page will live in the cache for the same amount of time. However, this performs poorly since heavily used pages will have the same lifespan in the cache as hardly used pages.
- **Random:** We mentioned how in caching, Random was almost as good as LRU, hence was a valid replacement strategy, However here, even though it is almost as good as LRU, it is not good enough, since any page faults will severely slow down our system.
- **Min:** Replace the page that will not be used for the longest time. This would be great if we can tell the future and see what pages we are done with for a while, but predicting the future is hard :(
- **LRU:** The page that will be thrown out is the page that was last used the longest time ago. This will prove to be the best policy we can do since it prioritizes keeping heavily used pages in the cache while throwing out pages that are not being used anymore.

From the replacement policies we have looked at, we can see that LRU seems to be the best way to go when choosing a replacement policy. The only gripe with this is that implementing true LRU is very inefficient. We would need to use a linked list that keeps all the pages, and then we can keep the LRU page on the tail. Here is a picture of what I mean.



However when a page is used, we will need to reorder this list, making it pretty inefficient if our cache is decently big. Also note that LRU is not full proof, meaning it will perform well in any paging scenario, remember we talked earlier about having N spots in our cache but sequentially accessing $N + 1$ things, which would cause a page fault every access. This is just to show that no matter the paging policy you use, there is always a ordering of pages such that your scheme performs bad, LRU is still generally conceived as the best we can do in termes of demand pagin policies. So we want LRU, but realisticly, it is too inefficient, so we need a way to approximate LRU.

7.3.2 Examples with different policies

First lets look at the MIN replacement policy:

- Suppose we have the same reference stream:
  - A B C A B D A D B C B
- Consider MIN Page replacement:

Ref: Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

Lets break down how this works. The first three misses are compulsory misses, since the cache started out cold. So currently it has the contents A, B, C. When we try to get page D we will get a page fault, C will be chosen to be kicked out the cache since A and B will be used sooner than C. Similary when we want to bring C back into the cache we will kick out A because at that point A is no longer used farther down the reference stream.

Lets take a look at LRU:

Consider the following: A B C D A B C D A B C D

LRU Performs as follows (same as FIFO here):

Ref: Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

- Every reference is a page fault!

On to this example, again the first three misses are compulsory misses since the cache started out cold. When page D is accessed, it is a conflict miss and A is kicked out since it was the LRU, then when A is access, B is kicked out since it was the LRU, you can see this pattern will continue, kicking out the LRU page, which will be accessed next in the reference stream.

7.3.3 Adding more memory

In general adding more memory to our page table will reduce the number of page faults we see. This is true for policies such as LRU and MIN policies. However, this does not hold for the FIFO policy. This is known as **Bélády's anomaly**, when adding more memory actually increases the number of page faults.

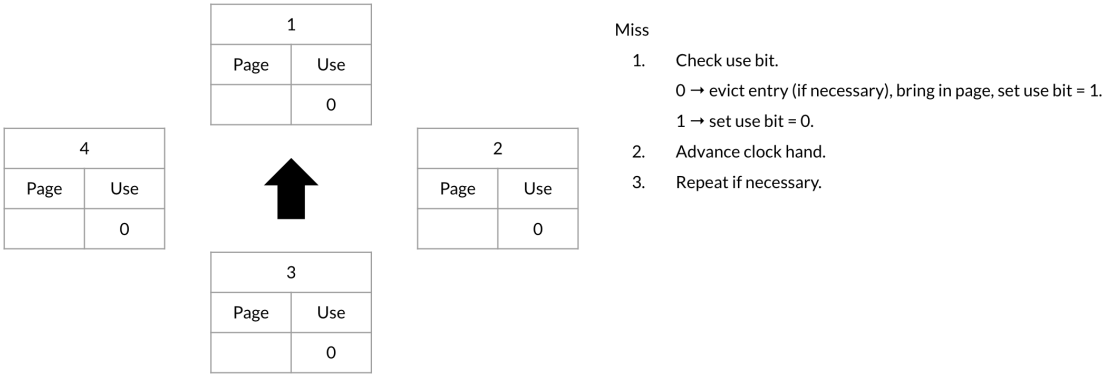
7.4 Approximating LRU

7.4.1 The Clock algorithm

In every page table entry we have some data that gives us information about the page. Some of the bits will tell us if it is writable, is valid, dirty, or has been access recently. A pseudo LRU algorithm we are about to talk about will take advantage of the access bit to approximate LRU. It is called the **clock algorithm**

The clock algorithm works like the following:

- **On a cache hit**, we will want to set the use bit to 1 on the cache entry we hit on. It might already be 1 but we set it anyway
- **On a cache miss** we will go around the clock until we find an empty cache slot or an entry with a use bit of 0. When we find either of these, we will evict the entry (if necessary) and insert the entry we were looking for in the cache, and set the use bit to 1. When going around the clock, when on each cache entry that has a use bit of 1, we will set the use bit to 0.



7.4.2 The Nth Chance clock algorithm

Algorithm is almost exactly the same ad the regular clock algorithm. The only difference is that we will go around N times before we decide to evict an entry, essentially waiting longer to evict an entry, letting it get older and older. This in turn will become a better approximation of LRU, the if N is very large. The downside to this is that it will take longer for the page replacement since we will have to go N times around the clock.

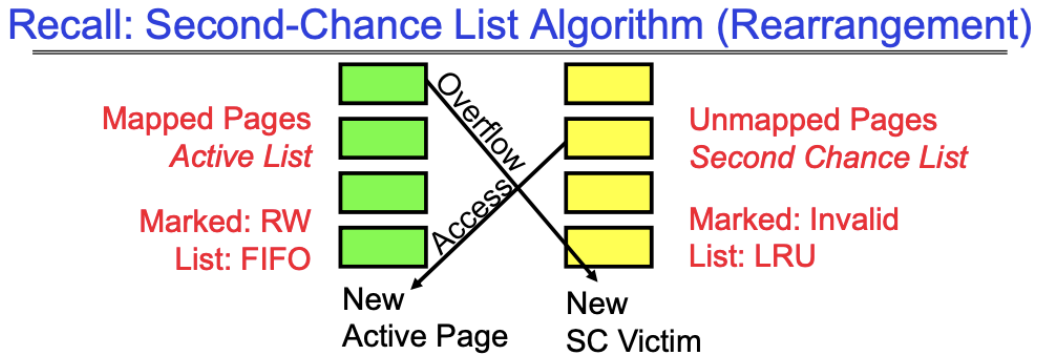
For dirty pages, we will usually go $N + 1$ times around the clock before replacing it. This is because if we decide to kick out this page, we want to give it time to write the its contents to disk, so we will let the clock go one more time around before replacing it.

7.4.3 Clock algorithm variations

- We can use the writable bit instead of the use bit when we have a software database that indicates which pages are writable. Usually this is required anyway. We will essentially at first set every page's writable bit to 0, which marks it as read only, so whenever we try to write to a page, it will result in a page fault, in which we will check with out software database which will tell us whether the page is actually writable or not. We will then set its writable bit to 1, which kind of acts as saying it has been used. Not gon lie this shit hella complicated for no reason idk why you would use this
- Something similar can be done for the use bit, we remove this bit and keep a use bit in the kernel, so in the beginning, all use bits are 0 which results in page faults, so we would go into the kernel and check the bits there. Again why tf you would use this... i don't have the answer

7.4.4 Second Chance List

There is another approximation algorithm called the second chance list. We would use this instead of clock if as mentioned above, we do not have use bits. This is another way to achieving an LRU like behavior.



This scheme splits up the pages into two lists, A green active list and a yellow second-chance list. The active list is sorted in FIFO order and the second-chance list is sorted in proper LRU order. The idea with this is that we will give pages a second chance before we send them out to disk. So if we get a page fault, we will pop off the front of the active list, which remeber is just a FIFO list and add it to the front of the second-chance list which remember is in LRU order. So it is essentially the most recently used out the LRU pages. So we will use this unmapped page as the replacement page. If we need to get the page in from disk, we will and again pop of the front of the active list to put on the second chance list, and throw out the LRU page from the second chance list. Some properties of the second-chance list algorithm are the following:

- If the size of the second-chance list is 0, this is just FIFO replacement policy
- If all the pages are in the second-chance list, then This is LRU replacement, but we will have a page fault every time we access

So from these, we need to pick an intermediate value that will give us the performance of LRU, without having a bunch of page faults.

7.5 Invalidating PTEs

When we want to throw out a page to disk, we need to invalidate every PTE with this page, note many different processes can have this page in its PTE. The best way to invalidate all of these pages is to have a mapping from a page to all of the PTEs it has, so when a page gets evicted, invalidate all the page tables that page is a part of.

7.6 Allocating page frames for different processes

As you can guess, not all processes will need the same amount of pages, since some processes will be a lot bigger than others. So how do we know how many pages to give each process? Well, there is a sense of the minimum number of pages a process will need, and this depends on the architecture, for example for the IBM 370 architecture the minimum number of pages needed for a process was 6. This was because one of their assembly instructions required 6 pages. So if a process did not have at least 6 pages, it would not be able to make forward progress.

When not talking about the minimum allocations, here are some allocation schemes we can use:

- **Equal** we divide the number of pages equally to each process, so if we have 100 pages, and 5 processes, each page gets 20 pages. You can see that this might not be the best scheme since small processes will have many wasted pages and big processes will not have enough pages
- **Proportional** We will give pages to processes proportionaly depending on the size of the process. The formula would be $a_i = \frac{s_i}{S} * m$ where a_i is the number of pages allocated for process i, s_i

is the size of process i, S is the sum of all process sizes, and m is the total number of pages available.

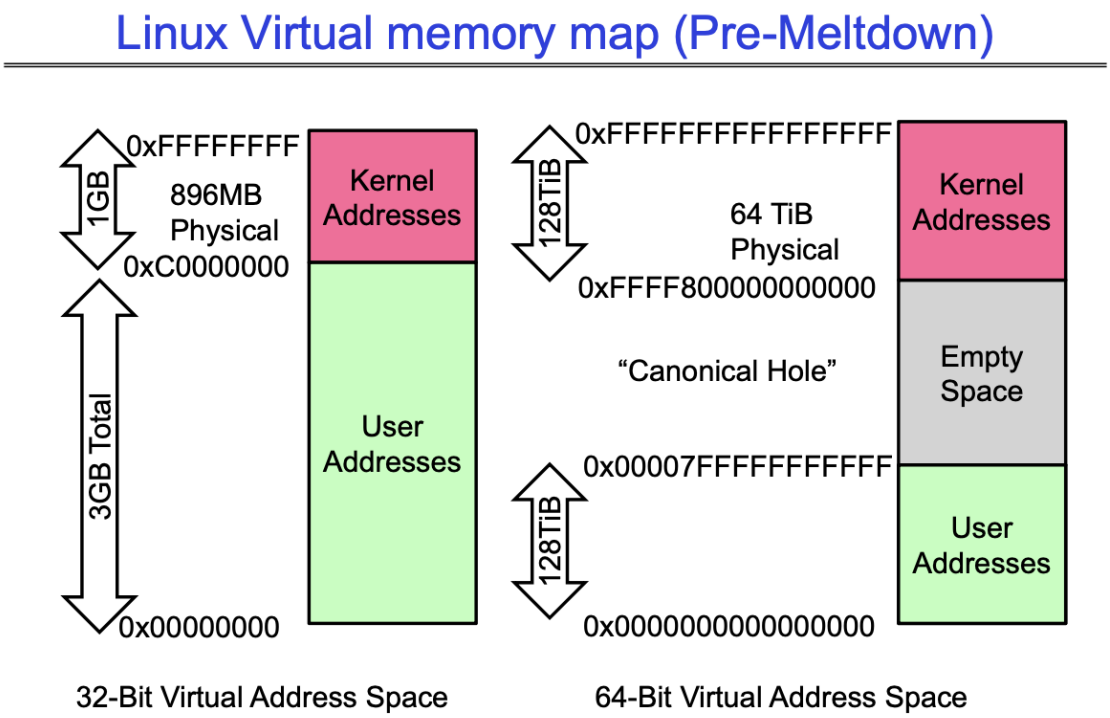
- **Priority** Similar to proportional allocation but we would base it of a processes priority rather than its size.

7.7 Thrashing

Thrashing is when too many pages are being used by all the processes, which results in many request to disk to get free pages. Essentially we can tell if thrashing is happening if the total number of pages being Demanded for a period of time (often called the working set) is greater than the number of pages available in the system, thrashing will occur. When many processes are thrashing, very little forward progress is being made, since most of the time is spent going to disk, which takes about 1 million cycles each time.

7.8 Pre-Meltdown vs Post-Meltdown

The Memory layout for linux is the following:



As you can see, the layout is pretty simple, at the very top of the address space is the kernel memory, and below it is the user memory. Note that for a 64 bit address space, there is a gap between the kernel and user addresses, This is mainly because if we going to use all of the 64 bit address space, there would be too many virtual addresses. So only 48 of the bits are going to be used to map to memory.

This memory layout was able to be exposed in a bug that is now referred to as Meltdown. Here is an example of how it is done.

– Exploit speculative execution to observe contents of kernel memory

```
1: // Set up side channel (array flushed from cache)
2: uchar array[256 * 4096];
3: flush(array);       // Make sure array out of cache

4: try {         // … catch and ignore SIGSEGV (illegal access)
5:   uchar result = *(uchar *)kernel_address;// Try access!
6:   uchar dummy = array[result * 4096];      // leak info!
7: } catch(){;} // Could use signal() and setjmp/longjmp

8: // scan through 256 array slots to determine which loaded
```

– Some details:
  » Reason we skip 4096 for each value: avoid hardware cache prefetch
  » Note that value detected by fact that one cache line is loaded
  » Catch and ignore page fault: set signal handler for SIGSEGV, can use setjump/longjmp….

Essentially, we create an array and make sure it is not in the cache, that is what the flush call is for. Next we try to make a illegal access at a kernel address, this should usually page fault, but due to things such as out of order execution, it might not page fault right away, leading us to the next line where we can access using this result which is a kernel address, putting it into the cache. So even thought the program will catch this illegal access and throw a page fault, which will throw away all the variables we stored this illegal data in, the cache will remain changed. So the last thing we will need to do is go through the whole array and see which access if very fast, that is the entry that we cached and thus have leaked memory from the kernel.

To fix this error, better hardware had to be implemented into these devices, involving better handling of side-channels, which is what was used to expose this bug.

8 IO

IO is the thing about a computer that makes it useful, if we did not have this, there wouldn't actually be a point for computers to exist. Imagine if us humans has nothing but a brain, it is not our mind that makes us useful, but the connection we have between our mind and body that makes us able to do incredible things (wow deep).

Inside a computer, there are many things we can connect to the motherboard that gives the computer functionality. Things such as a keyboard and mouse, graphics cards, WiFi cards and more.

8.1 Bus

A bus is a device that lets us communicate between a hardward devices. A bus is really nice because it lets us connect to n things throughout the computer. That means if we have n devices, each can connect to n other things on the computer, hence we can have n^2 relationships. This can all be accomplished over 1 set of wires.

8.1.1 Parallel Bus vs PCIe

In the early days of Computers and Bus's they were set in a parallel fashion. However there were many limitations to this such as the following:

- The bus must accommodate for the slowest device on the bus, since it needs to know what is happening
- The more things we connect in parallel, the capacitance increases, which slows down the whole device.

PCIe combats the limitation of parallel busses, What makes a PCIe bus different is it is a series of very fast serial channels, basically fast devices do not need to depend on slower ones, since they can have their own connection (you can think of it as lanes on a freeway). With that said, a device can take up multiple lanes depending on the desired bandwidth they require.

8.2 Talking with devices

Given now that we have these IO devices, we need some way for the CPU to talk with these devices. The CPU has what is called a controller. The controller has its own set of registers and memory that are used to request and hold information from devices connected to the computer. Things in the controller might be queues for certain devices.

8.2.1 Port-Mapped and Memory-Mapped IO

In programmed IO, there are 2 different ways to do it. In programmed IO, the CPU is involved in every byte transfer. There is port-mapped IO and memory-mapped IO

- **Port-mapped IO** involves special CPU instructions such as in and out. They also take a region of the physical memory distinct from the space used for regular data. Port-mapped IO is good for usually older devices that do not have much physical memory.
- **Memory-mapped IO** involves regular CPU instructions such as load and store, which simplifies the CPU logic. It is up to the controller to handle the logic. It shares the same memory space as the rest of the computer.

With these approaches to handle device IO, there are a couple limitations, First is that the CPU is involved in every byte of the IO, which consumes cycles that we could be doing something else with. This is where we will introduce direct memory access which allows us to write and read from these devices without cpu intervention

8.3 Direct Memory Access (DMA)

DMA is another way for device IO and is works like the following:

- The CPU will signal to the DMA controller that is wants to communicate with a IO device. This could be a read or write
- The DMA controller will then handle the request from the CPU, this might either involve going to the main memory or pulling it off disk.
- Once it is done, The DMA will get an acknowledgement and interrupt the CPU to signal that is it done handling the IO request.

You might wonder why this is any better than the CPU dealing with handling the IO, since we still need to go to memory or disk to handle our IO request. What makes DMA really good is that we give all the responsibility to the DMA controller to handle all of this, meaning it can handle these requests in parallel to the cpu handling other instructions. Since most of the time the CPU should be accessing the cache for things it needs (if your cache is good) the bus connecting to the main memory is not being used every cycle, making it perfect for the DMA controller to steal cycles from the CPU.

8.4 Notifying the OS

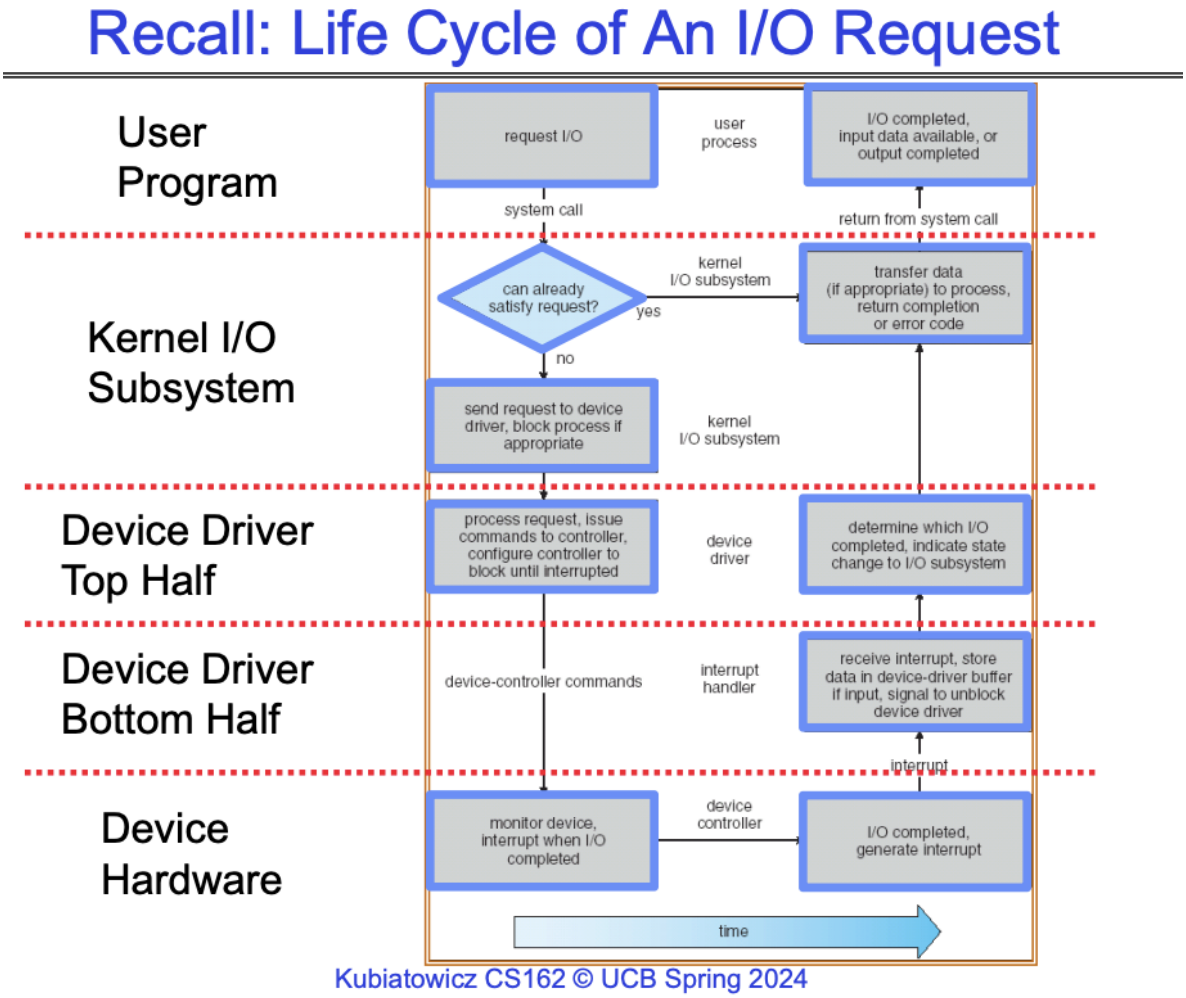
As you could imagine, when the device driver needs to make a request or errors, the OS should know about it so it can be handled properly. There are a couple ways of this being handled:

- **Interrupts** Whenever a device driver is done handling something or needs to request something, they will generate a interrupt for the CPU so they can immediately handle it. An advantage with this is that when unpredictable IO events occur, it is able to be handled no problem. A drawback with this is that we know there is a lot overhead when doing a context switch when an interrupt occurs, so this is not the most efficient.
- **Polling** In polling, the OS will check every so often if there is anything we need to do with regards to device IO, that could be new requests and new data from the device drivers. the advantages and drawbacks of this are pretty much the opposite of using interrupts. It is a lot more efficient since no context switching, but it prone to wasting cycles if there is nothing to receive or if IO devices are undpredictable, it can lead to poor performance.

Recall that IO devices will have what are called device drivers, this is software that is device specific to handle the specifics of that IO device. Here is how it is usually set up:

- The top half of the driver code is meant to handle system calls such as write() and read()
- the bottom hald is the interrupt routine for the IO device

You can see how an IO request is handled, with the driver code mentioned in the graphic below.



8.5 Different interfaces for different devices

As you can imagine, IO devices come in all shapes and sizes and many will have different purposes. For example, a keyboard recognizes what characters you push and a ethernet port will provide a networking signal to your computer, we want different interfaces for each type of device to provide the maximum amount of support and efficiency. Here are a couple different types of interfaces.

- Block devices: reads in data blocks at a time, used for devices such as disk drives and tape drives
- Character devices: reads in data character by character (or byte by byte), used for devices such as keyboards and mice
- Networking devices: Use of sockets and other networking tools. Used by things such as ethernet, WiFi and bluetooth.

8.6 IO device timing

There are different ways we can make a request to an IO device we will discuss them here:

- **Blocking Interface:** Essentially a blocking interface is that when we make a request, we have to wait until the request is fully processed, usually either resulting in a successful request or an error
- **Non-blocking Interface:** In non-blocking, we will make the IO request and very quickly return, essentially there is no guarantee if our request was satisfied, however this allows us to keep going with execution of the process without having to wait
- **Asynchronous** In these operations, we make the request, immediately return, and rely on the kernel to fill the buffer with whatever our request was. Typically asynchronous interfaces are paired with things called callbacks which basically is a piece of code that relies on whatever was returned from our asynchronous request, and will only be executed after we have finish processing that IO request. But it does not have to happen right away, we can make the request and execute other code, when some time passes and the IO request finishes, we then execute the callback.

9 Storage Devices

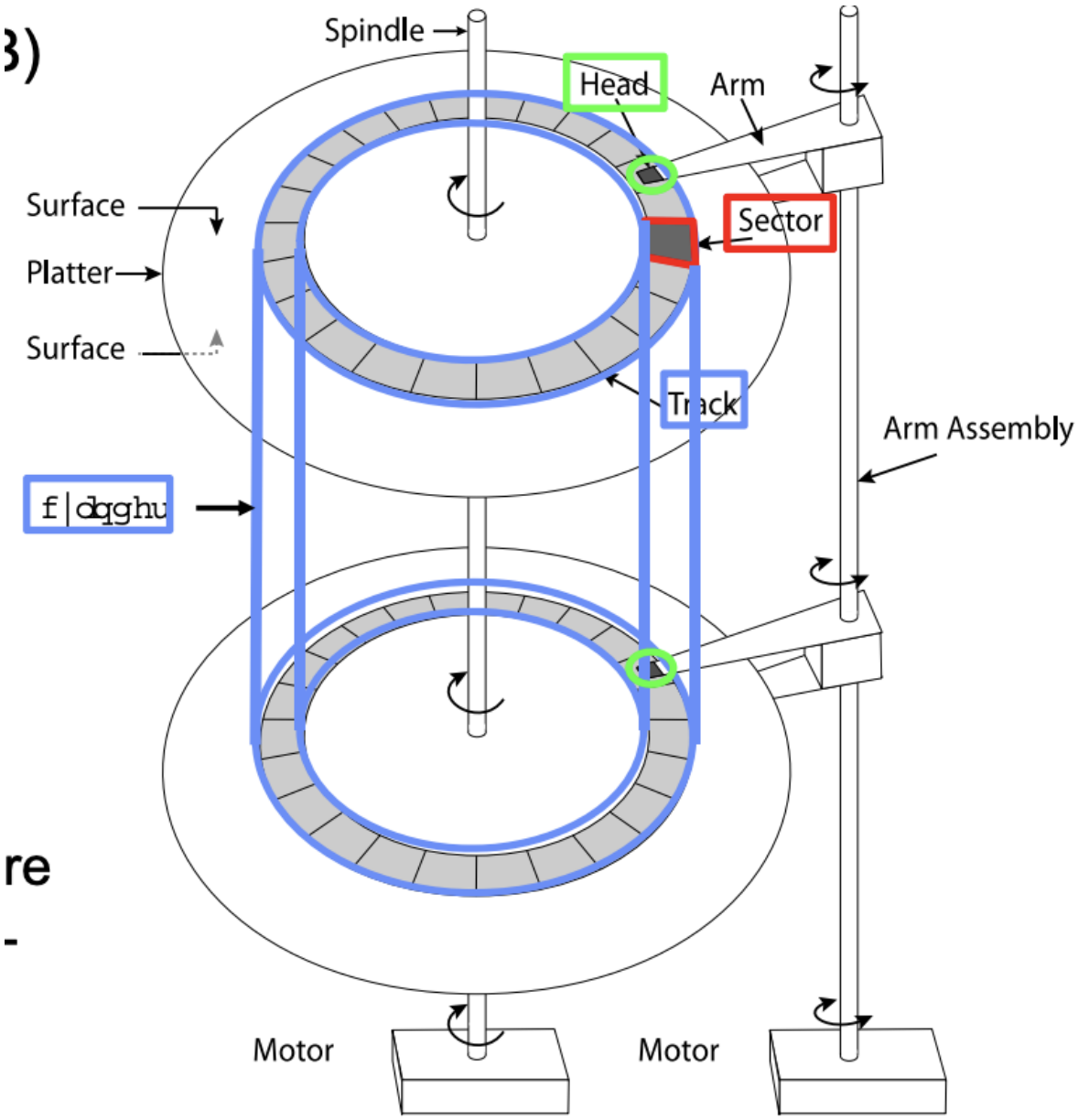
As previously mentioned, we cannot store all the data we need in our DRAM, since there is simply not enough space. This is where we have storage devices that usually can hold much more data, but in turn are a lot slower. There are two main types of storage devices:

- **Magnetic disks** These are decently big sizes boxed that consist of things like a bunch of magnetic disks stacked on top of each other, and arm and more that we will get into later.
- **Flash Memory** These are usually smaller and more expensive than magnetic disks, mainly because they are much faster than disks and smaller in size. We will get into how they work later.

9.1 The magnetic disk

Below is picture of a magnetic disk drive. The technicalities of how it works is beyond the scope of this class but here are some basic things we should now.

- The unit of a magnetic disk is a sector. This usually contains 512B of data for 4KB depending on the disk.
- A ring of sectors form a track
- A stack of tracks forms a cylinder
- The head of the disk is the current position (what sector we are pointing to)
- Typically the OS does not operate in terms of sectors, but rather blocks which usually are multiple sectors. Common block sizes are (4KB or 16KB)



9.1.1 Measuring Performance

As you probably know, disks are hella slow. To measure how slow they are, we have a few metrics to determine the disk latency. That is, how long it takes between going to disk to get the memory, and actually receiving it. There are three main processes that are involved when using disk

- **Seek time:** The time for getting the arm of the magnetic disk under the correct cylinder
- **Rotational Time:** The time to wait for the rotation of the disk so that the head is under the correct sector.
- **Transfer time:** The time to transfer the data from the sector under the head.

The equation for disk latency can be defined as follows:

$$\textbf{Disk Latency} = \text{Queue time} + \text{Seek time} + \text{Rotational Time} + \text{Transfer Time}$$

We will get into more what queue time is later, but it is not involved with the disk specifically, that is why we do not mention it here.

9.1.2 Disk Performance Example

Assume we are given the following;

- The average seek time is $5ms$
- A disk spins at $7200RPM$
- Transfer rate is $50\frac{MB}{s}$

The first thing we should do is convert all of our units, lets to ms since that is the smallest unit we are given

- Rotation time: $(\frac{7200Rev}{min} * \frac{min}{60000ms})^{-1} \approx 8.3\frac{ms}{rev}$ Per revolution
- Transfer rate: $\frac{50MB}{s} * \frac{s}{1000ms} * \frac{10^6B}{MB} * \frac{sector}{4096B} = (\frac{50*10^6\text{sectors}}{1000*4096\text{ms}})^{-1} \approx 0.082\frac{ms}{sector}$

Now we can measure a few things

- Reading a block from a random place on disk:

  $$(5 + 8.3 * \frac{1}{2} + 0.082)ms$$

  Note that we divide the rotaion time in half because it is a random access, meaning the average time it will take it half a spin

- Reading a block from the on the same cylinder:

  $$(8.3 * \frac{1}{2} + 0.082)ms$$

  Since we are on the correct cylinder, we do not need to seek

One thing to take away from this is that we want to strategize how we put things on disk to minimize disk latency.

9.1.3 Minimizing Errors and Latency

Here are some things that are used to either minimize the latency of a disk or to prolong the lifespan of a magnetic disk

- Each sector has error correcting codes, that will fix errors that happen when reading a sector
- When a sector goes bad, they will be remapped. In fact a whole group of sectors could be remapped, to keep sequential access
- You can offset from one sector to get to another sector for sequential operations.

9.1.4 Disk scheduling

As mentioned, the order in which things are seek after on disc can really speed up the performance of your disc. Similar to CPU scheduling, there is disk scheduling for efficiently scheduling things to get off disc. Here are some policies:

- **FIFO:** Yall already know this
- **Shortest seek time first:** Pick the request that is the closest to the head on the disk, which would result in the shortest seek time. The issue with this is it can lead to starvation
- **SCAN:** This is simliar to the shorest seek time first, but we will take the shortest seek time relative to the direction of traveling. This helps with the starvation issue from SSTF
- **C-SCAN:** This is very similar to SCAN, but it only goes in one direction, this is said to be more fair than SCAN, since it will not prioritize sectors that are in the middle.

9.2 Flash Memory:

Flash memory is another way of storing data, instead of using a magnetic disk, transistors that store charge are used to hold data. There are two varieties of FLASH memory:

- NAND: Flash memory using NAND is very dense, allowing lots of memory to be stored in a small space. However all reads and writes must be done in blocks
- NOR: Flash memory using NOR cannot be as dense, but reads and writes are a lot faster.

One thing with NAND Flash memory is that you must read a page at a time and when you write you must erase the whole block which is usually 64 pages.

9.2.1 FLASH latency

There is no seek time nor rotation time when dealing with FLASH. However, there is a controller that will access the flash memory, therefore the disk latency is:

$$\textbf{FLASH latency} = \text{Queuing time} + \text{Controller time} + \text{Transfer Time}$$

9.2.2 NAND writes reads and erases

A general rule of thumbs is an erase takes 10x as long as a write and a write takes 10x as long as a read.

9.2.3 NAND lifetime

Each block in NAND FlASH memory can be used reliable about 10K times before it starts becoming bad.

9.2.4 The Flash Translation Layer FTL

FTL provides accomodations for some of the issues that were previously discusses such as the limited lifetime of blocks and the long time for erases and writes. The FTL provides an abstraction between the OS and the FLASH Here are some things it does and its advantages:

- Maps virtual block numbers to virtual page numbers
- Instead of erasing a block whenever a write happens, write to an empty block and put a mapping so we remember where it is
- Wear out the blocks evenly, meaning give even use for each block for a better lifespan of the FLASH.

9.3 SSD vs HDD

SSD are uses a lot more heavily than HDDs nowadays since they are much faster. Also the storage on some SSDs are becoming comparable to HDDs, as well as the prices.

10 Some Performance Stuff

There are usually 4 metrics to measuring the performance of something. They are the following:

- **Latency:** The time it takes to complete a task.
- **Response Time:** The time it after you initiate a task to get a response
- **Throughput:** The amount of operations you can complete per unit time
- **Startup:** The time it takes to initiate an operation.

10.1 Latency

Latency can be measured by the following equation:

$$L(x) = S + \frac{x}{B_w}$$

Where B_w Represents the bandwidth. We see with this equation, at the task we want to initiate gets larger, the startup time begins to matter less. The startup time plays a big factor if the task we want to initiate is small, since we have to do all this setup just to accomplish a tiny task. Here are a couple more equations that are helpful to know in case they are asked on an exam.

$$\textbf{Effective Bandwidth: } E(x) = \frac{B_w}{\frac{B_w * S}{x} + 1}$$

$$\textbf{Half Power Bandwidth: } E(x) = \frac{B_w}{2}$$

10.2 Bandwidth

Bandwidth is the limit of how much of something we can do per unit of time. So for example we might be able to send 100MB of data per second or something like that. We can determine the bandwidth for an IO device by looking at whatever the bottleneck is. This is because if the device moved faster than what the bottleneck can process, we could end up losing information of corrupting our device. Some things that could lead to this could be the following:

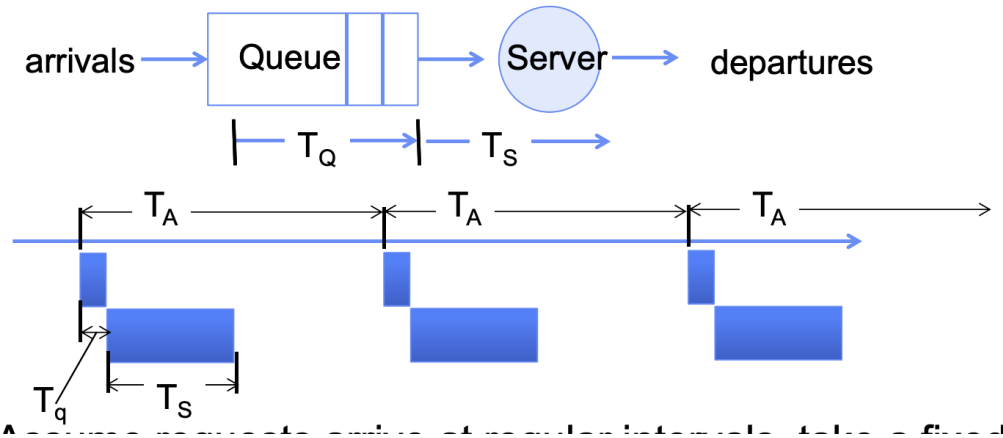
- Bus speed
- Disk rotation speed
- Read write speed of NAND flash
- The speed of a network link

Also the way we set up our system can determine our bandwidth. For example, if we have a server that handles everything sequentially, meaning we have to completely finish one task before starting the next one. This would result in things being pretty slow. However we can do other things to increase our bandwidth.

- **Pipe-lining** if we pipeline our server, we can be accomplishing different stages of different tasks at the same time. So if a stage is of length L and there are k stages. Each stage would take $\frac{L}{k}$ time. We can increase our bandwidth up to $\frac{k}{L}$
- **Multiple servers** Similar to pipe lining, we can have multiple servers each handling a task. So if we have k servers handling tasks of length L we increase our bandwith up to $\frac{k}{L}$

11 Queuing Theory

So far when talking about IO devices, we know we can send request to these devies (for example read and writes). When we request these, they get put in a queue. Everything that gets put into a queue takes a certain time to arrive in the queue and then it takes a certain time to get served in the queue. Takes this idealistic example:



Things have a arrival time T_A and a service time T_s we can measure the rates of taking the reciprocal of these $\lambda = \frac{1}{T_A}$, $\mu = \frac{1}{T_S}$ so if something took 0.5 seconds to arrive and 2 seconds to serve, we 2 tasks could arrive per second and 0.5 tasks could be served a second. We can take the ratio of the arrival rate and the service rate to get the total utilization.

$$U = \frac{\lambda}{\mu}$$

We can note a couple things

- $U < 1$ Things are getting served faster than they arrive meaning the queue is not building up
- $U > 1$ Things are arriving faster than they are getting served, meaning the queue is building up with tasks.

11.1 The Bursty Reality

The reality of things when it comes to queuing is that requests often come in bursts. This will cause some issues. First, this is going to cause large queue delays for tasks. Since they are not coming in at a consistent rate, the average utilization might be low, but we are still experiencing poor performance. We can measure the bursty nature of this by an exponential distribution. We do this because exponential distributions are memoryless, meaning that the probability of something happening right now does not depend at all on the events that came before it.

11.2 Little's Law

One fundamental law of queuing theory is Littles Law. Littles Law says a couple thigns about any stable system.

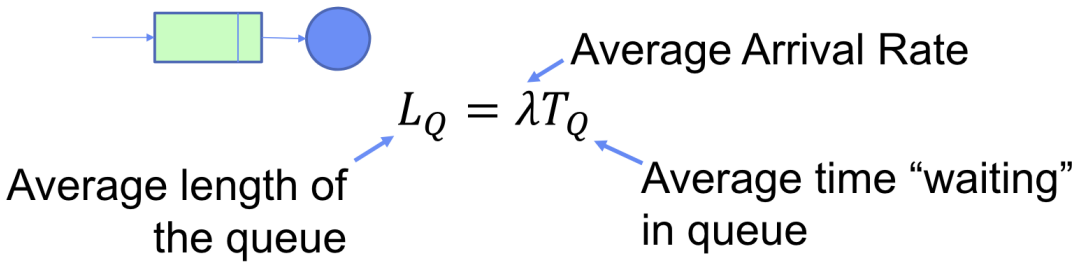
$$A_{avg} = D_{avg}$$

$$N = \lambda * L$$

A = Arrival time, D = Departure time, N = number of jobs λ = Arrival time, L = response time

One thing to note is that N represent the total number of jobs that are currently in the queue, not the total number of jobs. We can apply this knowledge of littles law to a queue to get the following formula

- When Little's Law applied to a queue, we get:



11.3 Solving problems

When solving problems that involve queuing theory, here are some things to know.

- **What is given:**
  - λ: The average arrival time
  - T_{ser}: mean service time
  - C: The covariance
- We can use these to derive:
  - service rate: $\mu = \frac{1}{T_{ser}}$
  - utilization: $u = \lambda * T_{ser}$
- From these parameters, we can derive two major equations to compute the average time spent in the waiting queue T_Q
  - General Equation:

$$T_Q = T_{ser} * \frac{1}{2}(1 + C) * \frac{u}{1 - u}$$

  - Memoryless Equation (When C goes to 1):

$$T_Q = T_{ser} * \frac{u}{1 - u}$$

We see in these two equations, when u approaches 1, T_Q will grow without bound, meaning the time jobs spend in the queue will become infinitely long.

11.4 An Example:

Lets solve the following problem given some information about a disk

- User requests **10***8KB from disk per second
- The request and service is exponentially distributed
- The average time to get serviced is 20ms

Here are some things we can gather from this:

- Since it was mentioned that it was an exponential distribution, C must be 1
- T_{ser} is 20ms
- We can compute the utilization by $u = \lambda T_{ser}$

- the user is making 10 requests a second, so the average number of things arriving per second is $\lambda = 10$

We can solve for our parameters now:

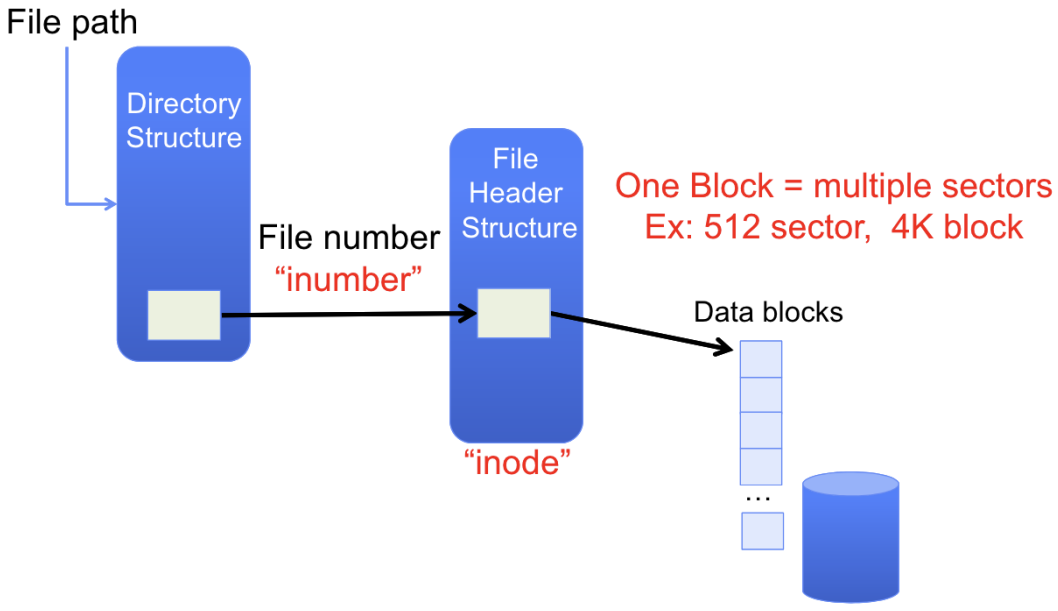
- $\lambda = 10 \frac{arrivals}{s}$
- $T_{ser} = 20ms$
- $u = 0.2s$
- $T_Q = 5ms$ - Use memoryless equation
- $L_Q = 50ms$ - Use Little's Law
- $T_{sys} = 25ms$ - T_{sys} represents the average response time for a job in the system. To compute this we would sum the time a job waits in the queue and the time it takes for the job to get serviced. $T_Q + T_{ser}$

12 File Systems

File systems are a key part of any OS. A file system can either make or break your OS's performance. Since almost everything that is stored is going to use your file system, how you decide to formulate your file system is very important. Some things that any file system needs to do are the following:

- **Track free disk blocks** Lets us know where we can go to get new memory
- **Track which blocks hold which data** Lets us find something that we saved earlier.
- **Track files on disk through a directory** Use some sort of directory design to be able to see all the files in a given directory
- **Store all this information on disk**

Here is a high level picture of how a file is actually stored in the computer.



We will start at the highest level, in which you are probably familiar. You have a directory, and inside if you have files that you gave certain names to. **Name resolution** is then performed to give each file a file inumber, which will point to some metadata about the file. One of those will be what is called an inode. An inode has al the information regarding metadata on the inode and points to the actual datablocks that hold the information about your file.

12.1 What Exactly is a Directory?

It's just a file lol. All a directory is, is a file that will contain vital information on how to get access to everything inside of it. It is usually composed of File Name : File inumber pairs. So if you wanted to get to a file that is inside a directory. You would iterate through the mappings until you found one with a matching file name, you would see its corresponding file inumber and proceed similar to the picture above.

Every file system has what is called a root directory. This is essentially a starting point for every file on your computer. You can get to any file in your file system from the root directory.

File systems also have what is called a relative path. which is a file path that starts from the current directory of your process. Which is usually tracked for you.

12.2 Building a file system

A few studies that were done found out a few things about files in computers.

- Most files on a computer are small in size
- Most bytes that are stored come from large files

Knowing these two things, we need to build a file system that will be efficient for small files, but also be capable to hold large files.

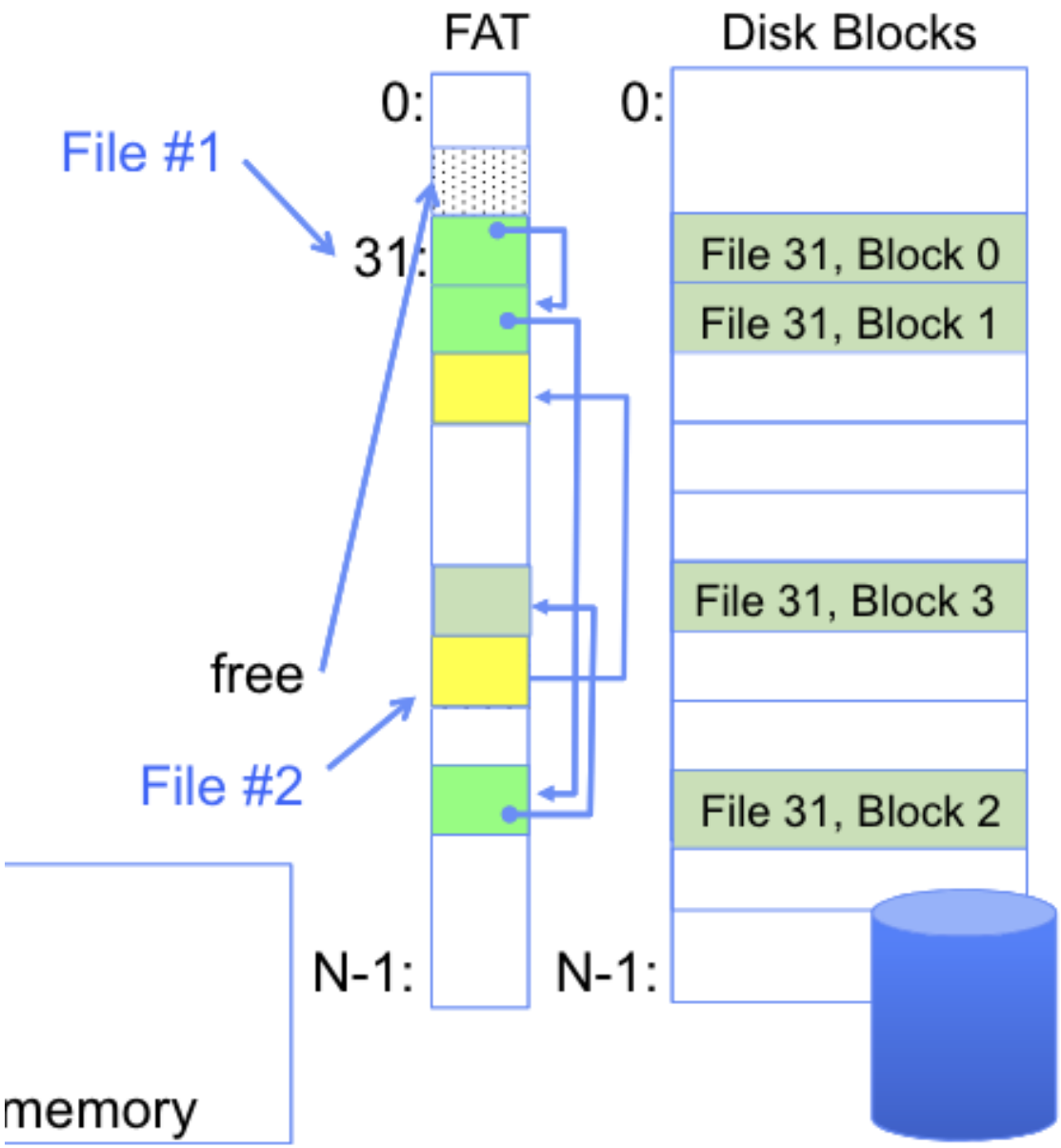
Next, we will go through some different ways of building file systems

12.2.1 File Allocation Table FAT

The FAT file system is probably the most simplistic of the file systems that we will talk about. There is two key points about the FAT file system

- Each file number and the file's starting block must be the same. I.E. if a file has a file number 30, its starting block must be block 30
- int the FAT, each file will have a pointer to the next point to the file.

That is pretty much it to the FAT file system. One thing to note is that since the FAT will have pointers to the next block. File numbers will not go in sequenctial order. For example, in the picture below, there canoot be a file with the file number 32. Here is a picture to visualize this:

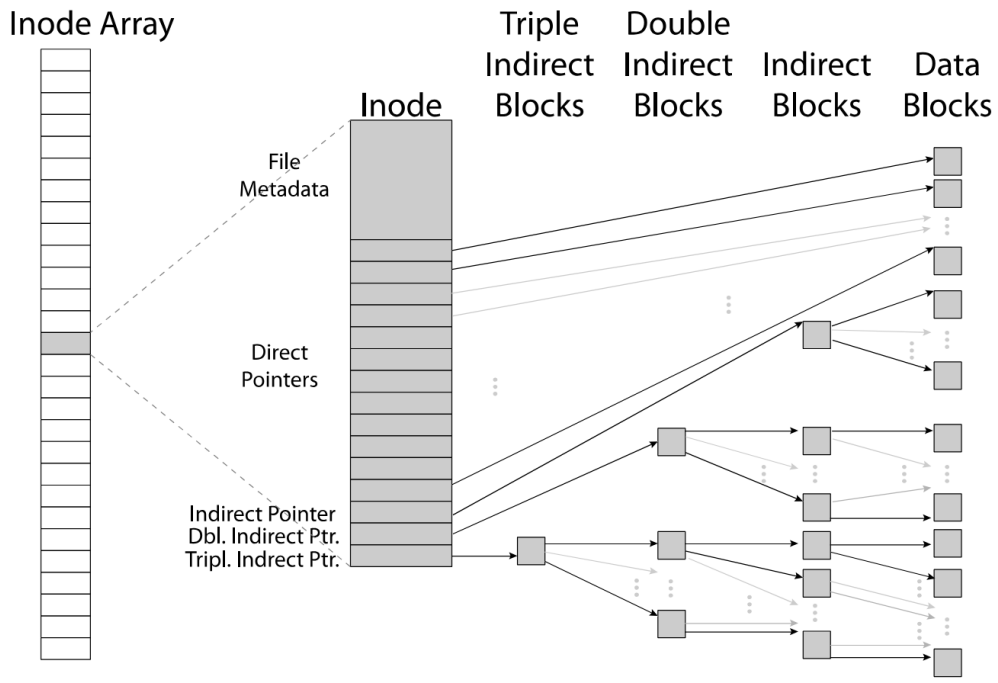


12.3 UNIX FFS

The Fast File System (FFS) is a another design scheme that is meant to help with the some of the limitations of FAT. Here are some key points about the FFS:

- inode structure is always a block in size, This way no fragmentation occurs
- inode will have metadata, direct pointers, indirect pointers, doubly indirect pointers, and so on.
- direct pointers will directly point to data blocks
- indirect pointers will point to an indirect block, which is a block with a bunch of pointers that each point to a data block
- doubly indirect pointers will point to a double indirect block in which each pointer will point to a n indirect block.

Here is a picture of the structure of the FFS.



A couple insights from the FFS are the following

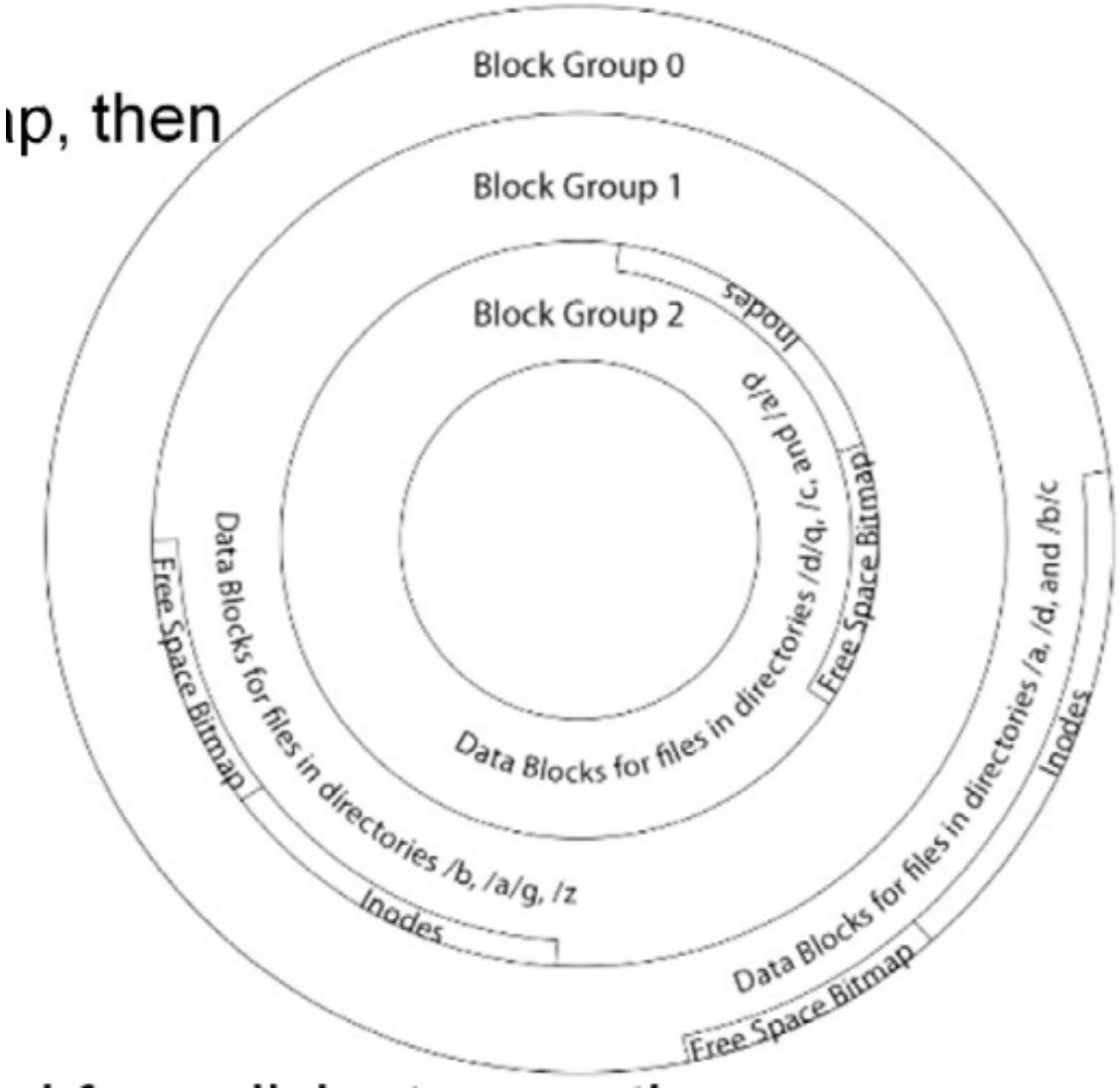
- This scheme works well for small files, since we can access the data fast using the direct pointers
- This scheme has the ability to also support vary large files due to the indirect pointer scheme. Lets see how this file growth works:
  - First, assume, we have $4KB$ blocks and we start off with 12 direct blocks, this means with just the direct blocks, we can hold $4KB * 12 = 48KB$ worth of data
  - One indirect pointer will point to a $4KB$ block which can store 1024 pointers in a 32 bit architecture. each pointer will point to a $4KB$ block meaning we can old $4 * 2^{10}B * 1024 = 4 * 2^{10}B * 2^{10} = 4MB$
  - One doubly indirect pointer will point to a $4KB$ block which can store 1024 pointers, each to a indirect block which also holds 1024 pointers, meaning we can store $4 * 2^{10}B * 2^{10} * 2^{10} = 4GB$
- Any type of indirect block will have $\frac{BLOCKSIZE}{4}$ pointers in the block, if the architecture is 64-bit, it will be $\frac{BLOCKSIZE}{8}$ pointers per block

block index, we can immediately find out where to go in the file, Lets go through an example.

- Imagine we have $1KB$ blocks and 10 direct pointers an indirect pointer and a doubly indirect pointer, for the following blocks, what will we need to access to get to the data we want. We can categorize how many accesses we will need to go through for each block range. Since we know a block of pointers will contain 256 pointers, each pointing to a block. Blocks 0-9 will be one access, blocks 10-265 will be two access and blocks 266 and greater will be 3 accesses
  - block 5: we would do one access through a direct pointer
  - block 23: we would need to do two accesses, one through the indirect pointer and then get the data block
  - block 290: we would need three accesses, double indirect, indirect, and data block.

12.4 Inodes on Disk

Originally, all the inodes on a disk, were stored on the outermost cylinder. A problem that arised with this is if the outermost cylinder failed, we lost all the inodes and all file information. One thing that BSD 4.2 did we spread out the inodes into different places on disk. A picture is provided. This way if some part of the disk failed, we are able to keep most if not all of our file data.



Another thing is each block group has a map of all the free blocks in the region, allowing the allocation of blocks easy. Another thing that is done to make the allocation of blocks fast and always reliable is always keeping at least 10 percent of the space for the file system free. To do this, we essentially lie about how much space we have in the file system, having it only be 90 percent of the actual capactity to mantain this 10 percent.

12.5 Hard Links vs. Soft Links (Symbolic Links)

Hard links are files that are mapped to an inode on file, there can be multiple hard links that point to the same inode, this means if the file is deleted the inode will not be unless the file with the last link to the inode gets deleted.

Symbolic links are files that are linked to another file somewhere in the file system. The contents of the file are usually the path to the file it is linked to. These allow for great flexibility when doing things such as updating libraries or switching to another version of something. All you have to do is change the link and you are done.

13 Buffer Cache

A part of the file system that is placed between the file system and our higher level APIs and system calls. The purpose of the buffer cache is to streamline any accesses that the kernel needs to make to make them fast.

13.1 Replacement policy

LRU will be used as the replacement policy, Previously this was an issue with paging because we could not afford the overhead when it came to pages. We can with the buffer cache so this is not an issue. LRU works very well in most cases, however, there is one case mentioned where it sucks

This command: find . –exec grep foo will search for the word foo in every file in your file system. This is will destroy your buffer cache, with lots of overhead. Since everything that will be put in the cache will never be accessed. A policy that might be better at this is a policy that allows you to say that we are only going to use this file once, so don't cache it.

13.2 Some Design Choices

13.2.1 Memory

One question that arises is how much memory to allocate for your buffer cache. Too much memory and you will have less memory for everything else such as running applications. Too little memory and you will be going to disk a lot since you cannot hold a lot in your cache. The solution to this is dynamically changing the size of the cache so things are balanced.

13.2.2 Pre-Fetching

Pre-fetching can be a great way to speed up your cache during the startup, since your cache will start out cold. There are some things to be aware of. Pre-fetching too much could slow down other applications since we need to go to disk a bunch in the beginning. Pre-fetching too little will cause you to do many seeks on disk.

13.2.3 Delayed Writes

Most buffer caches will implement delayed writes, which will essentially have something to check if the cache entry is dirty, and write it to disk when it gets evicted. This has many advantages:

- **Performance** If entry is in the cache, writing is very fast since we never write to disk
- **Save space** Some files will never make it to the disk, if the file is short lived for example, it will just be in the buffer cache and then if deleted before it gets a chance to be written out to disk.
- **Effective** Will only need to write to disk once entry gets evicted, if many concurrent writes happen, these will all take place in the buffer cache.

One disadvantage that comes with this scheme, is the possibility of the system crashing while dirty entries are in the buffer cache. Since they are never written to disk, when we reboot the system, our files will not be the most up to date, since their most recent changes got lost in the buffer cache. One attempt to combat this could that every so often, say 30 seconds, we flush out the cache.

13.3 Memory mapped files

One small thing you can do is give space in memory for a file. You can do this using the mmap() system call. Which will make access to this file very fast. Another thing you can do with this is share a file between processes

13.4 Durability of File Systems

There are three important ilities:

- **Availability** The probability that a system can accept and process a request. Note this says nothing about if the request will actually be successful or not.
- **Durability** When faults occur, how effectively are we able to recover?
- **Reliability** The ability of a system to perform its required functions.

We will be focusing on durability here.

There are some things that exists in file systems already that help with the durability:

- **Error correcting codes:** Reed-Solomon codes are often used to correc small errors in sectors
- **Surviving in the buffer cache:** To make sure files are the most up to date if a crash occurs, we can either not use delayed writes, or use special ram that will survive even after a system crash, usually using batteries, that can carry out flushing the cache
- **Redundancy** Redundancy is vital in ensuring the durability of files. We will get into this next section

13.5 Redundant Array of Independent Disks (RAID)

RAID is a method of providing redundancy of data There are different types of RAID that we will go through

13.5.1 RAID 1

RAID 1 is the simplest RAID, all it is is that we will have two disks, each will have exactly the same data. here are some things about this:

- There is high overhead to this. Essentially for every write to disk, we will need to perform two writes, to ensure the data on both disks are identical.
- Read's can be optimized by reading from both disks in parallel

This is good for your local machine. So just in case of your disks completely fail, you have a backup with all of your data

13.5.2 RAID 5,6

RAID 5 is a more sophisticated redundancy system, we will essentially split up our data across 4 blocks, one from each disk, then there is one disk which will hold a parity block. The parity block will hold the following data:

$$P = D_0 \oplus D_1 \oplus D_2 \oplus D_3$$

We can construct any block using the same xor property, lets do D_0 for example

$$D_0 = D_1 \oplus D_2 \oplus D_3 \oplus P = D_1 \oplus D_2 \oplus D_3 \oplus D_0 \oplus D_1 \oplus D_2 \oplus D_3 = D_0$$

A similar idea is applied to form RAID 6, the only difference is that we will have 6 disks, and for each stripe, two of them will serve as parity blocks, so we can lose two disks while still being able to recover our data.

13.6 Reliability

13.6.1 Transactions

Transactions are units of action that follow certain properties:

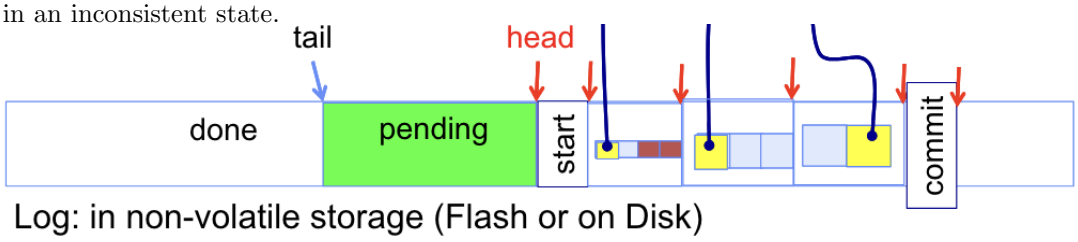
- **Atomicity** Transactions either occur entirely or not at all
- **Consistency** A transaction should take the system from one consistent state to another consistent state. We define a consistent state however we like. An example is for a database of students could be that every student must be assigned a teacher. So if for whatever reason there is a time where student does not have a teacher, that would be an inconsistent state.
- **Durability** A transaction must persist through crashes.
- **Idemptoent** Every transaction must be idempotent meaning if we applied the same transaction twice, we should end up in the exact same state.

To ensure these properties, journals are often used in file systems. Journals use what is called write ahead logging to satisfy these properties we outlined for transactions

13.6.2 Write Ahead Logging

The point of write ahead logging is whenever we want to write to a file, we will first write it in the log, with a series of actions that will compose the transaciton. Every transaction will start with an indicator that a transaction is starting, and will end with a COMMIT message, indicating that the transaction has finished. Note that we will NEVER write a transaction to disk, if the transaction does not end with a COMMIT message. We can assume that this transaction never finished and writing this would have the file system be in an inconsistent state.

The Log has a linked list structure, with a head and a tail, whenever we check the log to see if there is anything that needs to be done, we will start at the tail and walk through until we see a COMMIT message, when we see that message, we will perform the transaction and after the transaction has finished completely, we will move the tail forward to indicate that the transaction has finished, in which it will eventually be garbage collected. It is important to note that we must only move the tail after we have completely finished the transaction. If we move it before we finish, it is possible we crash during the write, then when we boot back up, we will never finish the transaction and the system will be in an inconsistent state.



One thing to note about doing write ahead logging is that it is expensive, We are essentially doubling our overhead, since we are first writing everything to the log and then writing it to disk. However, we are okay with this since it provides a rock solid file system, that is resistant to crashes.

14 Distributed Systems

14.1 Intro

Distributed systems are very common in the real world, due to its performance. If we have a bunch of computers work together to perform tasks, it will be much faster, than having one system do it, no matter how powerful that system it, there is strength in numbers. However, there are issues when working with distributed systems. A main one is coordination.

14.2 General's Paradox

Imagine there are two general's on different hills and they are planning an attack. There are a few rules to this attack:

- If they both attack at the same time, they will **win**
- If they don't attack at the same time, they will **lose**

The issue is that they are communicating over an unreliable network, that could be through a messenger or in the case of distributed systems, through the network. However, there is no guarantee that the general's messages will get through. To combat this, we need to formulat a protocol that will always ensure the complete coordination or not perform on the task at all.

14.3 Two-Phase Commit 2PC

The 2PC protocol is one that will successfully acheive distributed transactions with complete coordination. Assume we are given the following parameters:

- We have one coordinator
- We have N workers
- For a transaction, we must come to a unanimous decision, ie all workers must commit for the transaction to occur.

Here is the protocol:

- Coordinator sends to all workers a vote request. In this vote request, the workers can either COMMIT or ABORT
- Each worker will log their choice and send their vote to the coordinator
- The Coordinator will then log and send a GLOBAL-COMMIT or GLOBAL-ABORT depending on what the workers voted. It will be a COMMIT only if all workers voted COMMIT, and an ABORT otherwise
- Each workers will log what the coordinator sent, when and will either COMMIT or ABORT depending on the coordinator message. After they do this, they will send an ACK to the coordinator
- After the coordinator receives N ACKs, the transaction finishes.

Note in this protocol, it requires that no worker dies. If any worker dies and is not able to send a COMMIT or ABORT, or an ACK, it will lead to blocking, since we must wait until we get every response. To combat this, we can change our parameters to not need a unanimous decision to go through, but maybe a majority. It is up to you when you design your scheme.

14.4 Malicious Workers

In the previous protocol 2PC, we assumed at all the workers were honest, now we need to take into account if there is a malicious worker among us.

14.4.1 Byzantine General's Problem

Through Byzantines General's problem to solve the problem of the General's paradox, if we have n workers, and f of them are malicious, we need n to be at least $3f + 1$