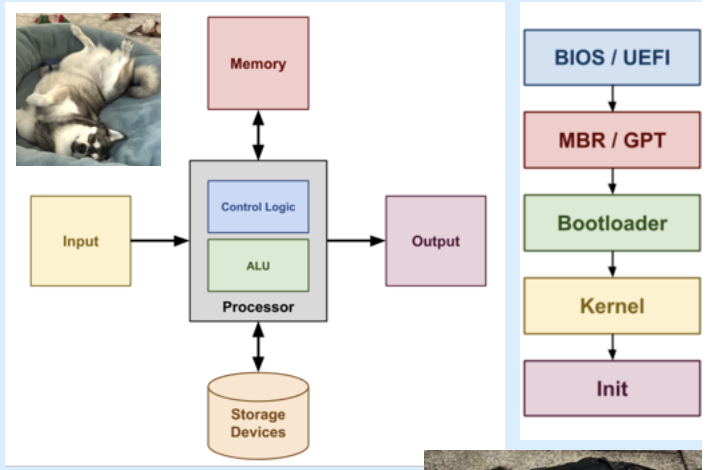
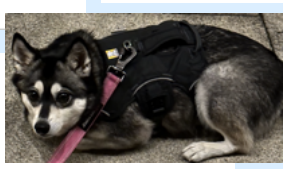
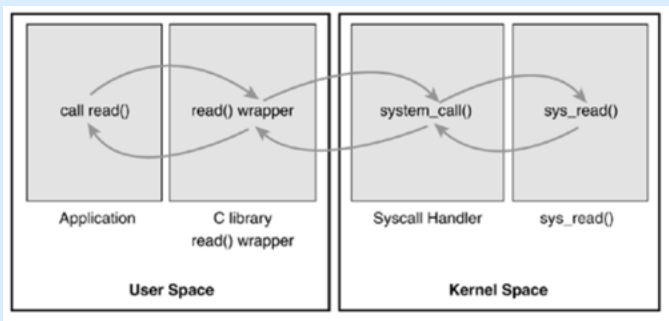


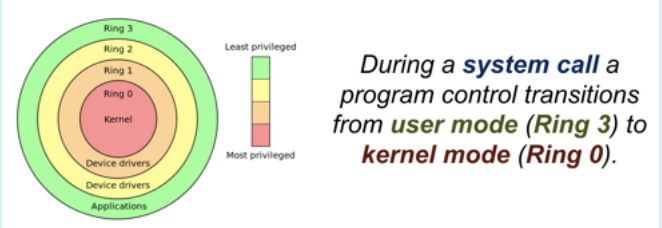


Monolithic A single program (or collection of procedures linked together) ran in kernel mode. **Microkernel** Split the OS into small, well-defined modules, one of which runs in kernel mode and intermediates communication

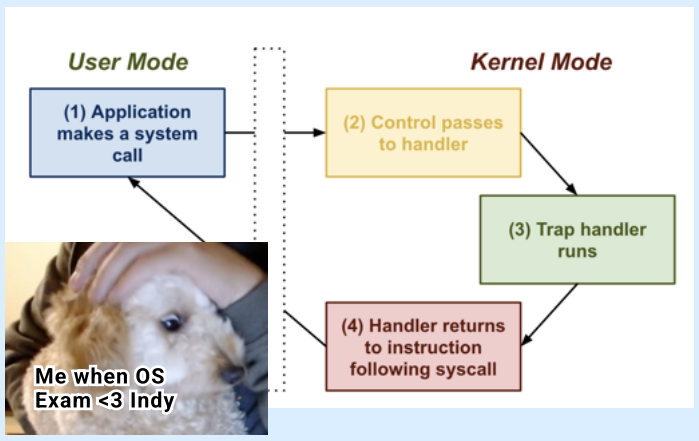
System Call: Request a service from the OS kernel: **File manipulation** Open/ Close, Read/Write, Stat; **Process Control** Fork/Exec, Wait, Signal **Communication** Socket, Listen/Bind, Connect

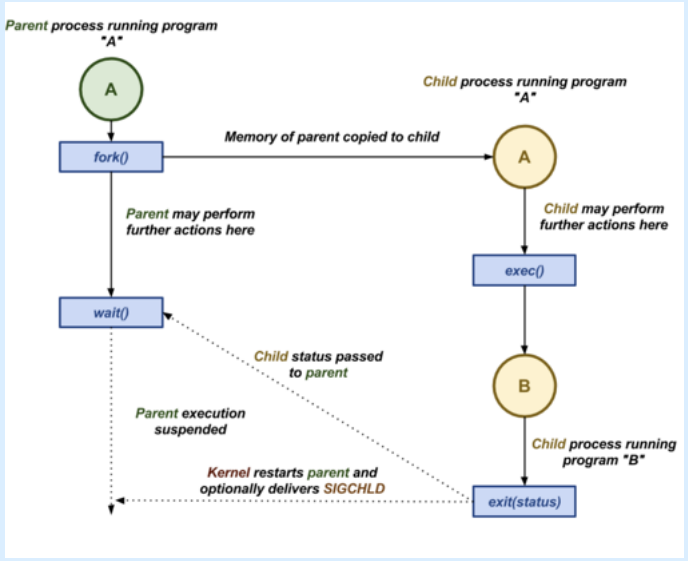


Modern processors have a security model where programs are separated into different protection rings or modes.



A variety of events or exceptions can cause a transition from user mode to kernel mode: Trap = intentional exception; When these events occur, the CPU looks at its interrupt vector table or trap table to determine what to do next. **Traps** To request a service, a user application performs a trap to interrupt the CPU and force it to look up a handler function in the trap table.

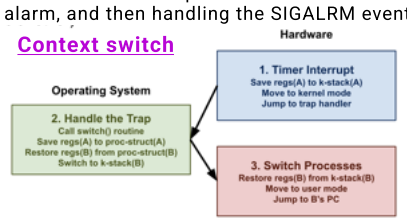


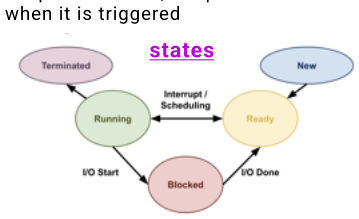


Definitions (Word Bank) One-to-one, One-to-many, Many-to-many, Convoy effect, Time slice, Starvation, Parallelism, Thread, Critical Section, Race Condition, Mutual Exclusion, Lock, Condition variable, Spin lock, Busy waiting, Semaphore, Reader-writer locks, Atomicity violation, Order violation, Deadlock, Livelock, Circular wait, Virtualization, Concurrency, Persistence, Microkernel, Multiprogramming, Timesharing, BIOS, Bootloader, Kernel, System Call, User mode, Trap table, Process, Cooperative multitasking, Pre-emptive multitasking, Signal, Scheduling policy, Turnaround time, Response time, FIFO, Round Robin

Reading 01: Hardware, OS **Os:** A set of abstractions (processes, threads, memory, filesystem) that enable applications to effectively and efficiently utilize hardware resources and interact with each other. **Three key ideas** in Operating Systems: Virtualization, Persistence, Concurrency; **Virtrualization** is when the OS takes a physical resource (such as the processor, or memory, or a disk) and transforms it into a more general, powerful, and easy-to-use form; User applications interact with the OS via **system calls**, which are initiated through a special hardware instruction called a **trap**. **Concurrency:** when you have multiple things executing and utilizing resources at the same time. **Persistence:** the ability to store and share data in a reliable and efficient manner. During one of these service requests, we transition from **user** mode to **kernel** mode; With the advent of **multiprogramming** it became possible to run multiple jobs at a time and thus improve CPU **utilization**. Because the OS now had to manage multiple tasks, it became necessary to provide **memory protection** so one program would not be able to access the memory of another program. Moreover, having multiple applications running at the same time means the OS has to provide mechanisms and policies for dealing with **concurrency** issues. Order of typical x86 **boot sequence**: BIOS/UEFI, Master Boot Record (MBR), bootloader (GRUB), kernel (linux), init System (systemd); **BIOS:** Usually stored in ROM, Performs some basic system integrity checks, searches for device to boot or primary bootloader **MBR/GPT:** This consists of a small primary bootloader program and a partition table; **Bootloader** This is a second bootloader program that loads the kernel and an optional RAM disk. Because this program is not restricted to the MBR, it can be larger and support features such as multiple filesystem support and graphical menus; **Kernel** This is the operating system core and brings support for additional devices and things such as networking, raid, and encryptionOn Linux and macOS, to see boot messages, use:$ dmesg; **Init** This is the first user-space application and is in charge of configuring and managing the user-space daemons and services.

Reading 02: System Calls, Processes Modern operating systems use a basic technique known as **Time sharing**, which allows users to run as many concurrent processes as they would like on a single **CPU**. To do this, the OS will need to employ low-level **mechanisms** or methods such as a **context switch**, which gives the OS the ability to stop one program and start another on a given CPU. Additionally, on top of these methods, the OS will need to provide intelligence in the form of **policies** or algorithms that make decisions within the OS; After a fork(), both the parent and the child processes are almost exactly the same except they have different **Address spaces**, **Registers**, and **PIDs**; When a program is executed, the operating system must: Load code and static data in memory, Create and initialize the stack and heap, Setup I/O or file descriptors, Jump to the main() routine; The **ready** state is where a process can be ran, but for some reason the OS has chosen not to run it at the current moment; The **blocked** state is where a process has performed some kind of operation that makes it unavailable to be ran until another even takes place; The **running** state is where a process is actively executing on a processor; Order events of when user application wishes to **open a file for reading**: User application calls open() system call, Hardware receives trap, Hardware saves registers to kernel stack, hardware moves to kernel mode, hardware jumps to trap handler, OS handles trap by doing work of syscall, os returns-from-trap; On early versions of the Macintosh operating system, the OS used a **cooperative** approach where the OS trusts the processes of the system to behave reasonably. Of course, this was a terrible idea, so modern operating systems now utilize a **timer interrupt** to raise a CPU exception, halt the current process, and load a pre-configured **interrupt handler** in the OS. When this happens, the **scheduler** needs to make a decision on which process to run next. If it chooses to execute another process, then the OS must perform a **context switch**; During a context switch, **General purpose registers**, **program counter**, and **kernel stack pointer** must be saved and restored; We can trace the system calls used by a program by using the **strace** utility; **Machine state:** A running instance of a program; unit of allocation (resources, privileges) that consists of Memory Address Space, Kernel State, and Execution Context; **Data Structure**: In Linux, there is a struct task_struct that represents each process: Each process has a process control block (aka struct task_struct) that stores its information; Process list is a linked list that keeps track of all the processes **Time Sharing:** The operating system virtualizes the CPU via the process abstraction: Each task is associated with a process, which gets a certain slice or share of the CPU time. Multitasking - **Cooperative:** The operating system trusts the processes of the system to behave reasonably. **Pre-Emptive:** The OS uses a timer interrupt to periodically suspend a running process and possibly switch to another process. **Context Switch:** When a timer interrupt goes off, we have the option of performing a context switch (i.e. switch from one process to another); **Process Life Cycle: 1)**Parent forks to create a new process 2)Child performs actions, possibly exec to run another program 3) Parent waits for child process 4) Child exits 5) Parent receives child's exit status **Fork** Creates a new child process based off the current parent; After a successful fork, we have two processes with the same code, but different: Memory address space, Kernel state, Execution context (pid = fork(): pid = 0 = child; -1 = parent error; default = parent success) **Exec** Load a new program into current process' memory address space. **Wait** Suspends parent until one of its children is terminated and retrieves status code **Exit** Terminates a process and sets status code **Signals** are a means of asynchronously notifying a process of an event. Each signal delivers a small integer that represents a particular event; To deliver the event, the kernel will interrupt the normal execution of the process; Processes register handlers to catch certain events; After the handlers are executed, the process will continue executing where it was interrupted **SIGCHILD** When a child process exits, the parent is notified with the SIGCHLD event **SIGALARM** An alarm or timer can be set by first using alarm, and then handling the SIGALRM event when it is triggered





Reading 03: Scheduling In order to create a **scheduling** policy or **discipline**, we first need to consider our typical **workload** or collection of processes running in the system. Additionally, we will need to determine a **scheduling metric** or means of measuring the effectiveness of our decision-making process. For instance, **turnaround time** is the time at which the job completes minus the time at which it arrived in the system. This is in contrast to **response time** which is the time from when a job arrives in a system to the first time it is scheduled. **Assumptions**: 1) Each job runs for the same amount of time 2) All jobs arrive at the same time 3) Once started, each job runs to completion 4) All jobs only use the CPU (no I/O) 5)The run-time of each job is known. When we **relax assumption** 1, FIFO Scheduling can suffer from the convoy effect. When we relax assumptions 1 and 2, SJF Scheduling can also suffer from the Convoy Effect. Whe we relax assumptions 1, 2, and 3, STCF requires the ability to preempt jobs and produces optimal average turnaround times **Convoy effect:** where a large job bottlenecks many smaller jobs. To improve **response time**, the **Round Robin** scheduling policy only runs jobs for a brief period and then switches to the next job in the run queue. This scheduling policy must take care in setting the length of the **time slice** in order to **amortize** the cost of **switching** without making it so long that the system is no longer responsive. In general, any policy that is **fair**, that is evenly divides the CPU among active processes on a small time scale, will perform poorly on metrics such as **turnaround** time but well on metrics such as **response** time. **MLFQ** uses multiple queues with different priority levels, uses round-robin to schedule jobs of the same priority level, lowers the priority of a job once it uses up its allotment at a given level, approximates SJF because it assumes jobs are short and starts them with the highest priority, overcomes the problem of starvation by periodically performing a priority boost **proportional-share scheduling policy** tickets represent the share of a resource a process should receive, The use of randomness is advantageous since it avoids worst case scenarios, is lightweight, and fast. This policy is simple to implement with a random number generator and a linked list that tracks the processes. In systems with multiple CPUs or cores, we must consider multiprocessor scheduling policies such as **MQMS** which use multiple queues to schedule jobs. With these disciplines, we must be careful with **load imbalance** where one queue may get more work than another. To prevent this, we need techniques for performing **migration**, which moves a job from one CPU to another. **Work stealing** is one approach where a queue that is low on jobs will occasionally take work from another fuller queue.Whenever we need to decide which process to run next, we invoke the **scheduler**: A process terminates, A process blocks, A timer interrupt (preemptive multitasking)To make scheduling policies, we need to consider our **workload**, or collection of processes running on our system. **Turnaround time** (throughput) = Tcompletion - Tarrival; **Response Time**(Latency) = Tfirstrun - Tarrival. **Multiprocessor Tradeoffs**: Issues: Cache coherence, synchronization, cache affinity; Tricks: Migration, work stealing; Single Queue: Put all jobs into a single queue and assign jobs to particular CPUs. - poor affinity and stealing; Multi-Queue: Distribute jobs across multiple queues (one per CPU) - Load imbalance and complexity. **FIFO** is relatively straightforward, has good turnaround time, poor response time, and an suffer from problems with the Convoy Effect. **Round Robin:** Instead of running jobs to completion, we use a timer interrupt to rotate through processes; fair policy since it evenly divides the the CPU among active processes; is relatively straightforward, poor turnaround time, good response time, requires preemption, has some overhead. A Multi-Level Feedback Queue (**MLFQ**) tries to optimize both turnaround time and response time:Like FIFO, it tries to complete shortest jobs first, Like Round Robin, it tries to be fair, Unlike either, it will factor incorporate I/O and will adjust priority levels over time.To accomplish this, MLFQ uses multiple queues, where each queue represents a particular priority level. We select jobs from the highest priority levels first. Within a level, we use Round Robin To manage the various priority levels, we use the following rules: 1) If Priority(A) > Priority(B), A runs 2) If Priority(A) == Priority(B), A & B run in RR 3)A job is initially placed in the highest priority level 4) Once a job uses up its time allotment at a given level, its priority is reduced (ie. it is moves down one queue) 5) After some time period S, move all jobs in the system to the topmost queue. A single long job is broken up into discrete time slices. Over time, the job reduces in priority to allow new jobs an opportunity to run. When a short job arrives, it will start in the topmost priority level and be ran first. This allows for short jobs have a fast turnaround and response time. Jobs that are mostly I/O will maintain a higher priority since they do not use up their CPU allotment as quickly as a compute job. This is good for interactive jobs that require good response times. **Problem**: If a process is always in a lower priority level relative to other processes, it will starve because it will not have an opportunity to run. **Solution**: Periodically provide a priority boost by moving all jobs to the topmost queue. Exact time is a voodoo constant because there isn't an objective way to set it. To determine when a job should be moved down a queue (ie. lower its priority), we keep track of how much of a time slice or allotment the job has used. When a time allotment is used up, we lower the priority. We allocate more time for jobs in lower priority queues. MLFQ is an attempt to optimize for both turnaround time and response time; It prioritizes new, short, and I/O heavy jobs over long CPU intensive jobs. **Lottery**: Instead of directly optimizing for metrics such as turnaround time or response time, we can aim to guarantee that each job has a certain percentage of CPU time: Each process is given a set of tickets, which represent a share of a resource; Periodically hold a lottery to determine who gets the resource; Each process has a set of tickets. To select the next process, a lottery is head and the process with that ticket is chosen. We also have a few mechanisms for augmenting the lottery: Ticket currency: allow users to subdivide set of tickets among its jobs Ticket transfer: allow a process to temporarily hand off its tickets to another process; Ticket inflation: allow a process to temporarily raise or lower the number of tickets it owns

Project 01: Process Queue Shell When a **system call** occurs, the process is blocked until the kernel returns a result.A system call always leads to a transition from user mode to kernel mode. One way to implement system calls is to have a user application perform a trap that is caught by the CPU.The code to handle each system call is stored in the CPU's trap table. When a **signal** is delivered to a process, the process is interrupted and then runs the corresponding handler before returning to where it was interrupted. Applications can send a signal to another process by using the kill system call.Signals can interrupt system calls such as read.To unpause another process, you can send it the SIGCONT signal. **Process states**: A process can go from running to blocked when a system call is initiated. A process can go from blocked to ready when a system call completes. A process can go from running to ready when its timeslice is up. A context switch usually occurs when a process goes from running to ready.

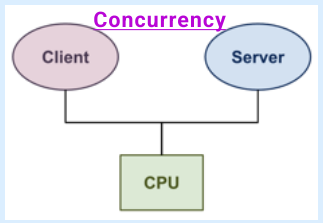
Scheduling

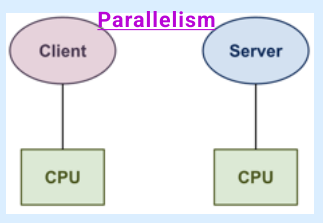
Consider a single-CPU, a timeslice of **10 ms**, and the following processes:

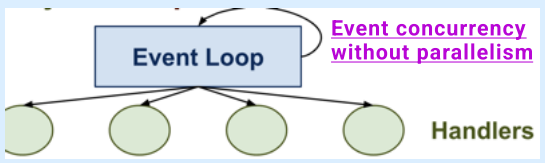
Process	Arrival Time	Run Time
A	0 ms	30 ms
B	10 ms	20 ms
C	20 ms	10 ms

Queue 0	A	B	C				
Queue 1		A	A, C	A, B	A, B	B	
Running	A	B	C	A	A	B	
	0	10	20	30	40	50	60

*Sketch out exactly when each process runs, and for how long given the **MLFQ** scheduling policy. You may assume that **Queue 0** has a timeslice of **10** ms and **Queue 1** has a timeslice of **20** ms.*

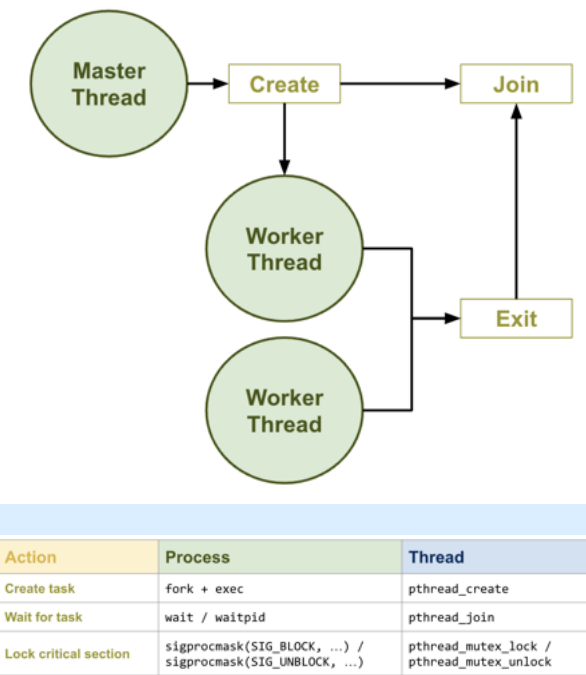






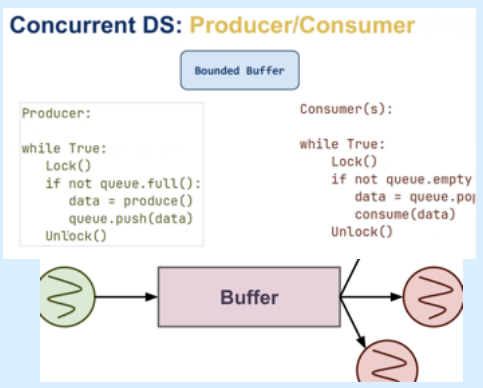
Events: Event Loop (w/ Poll)

```
while (true) {
    // Setup poll structure
    struct pollfd pfd = {STDIN_FILENO, POLLIN|POLLPRI, 0};
    // Wait for either event or timeout
    int    result     = poll(&pfd, 1, TIMEOUT);

    // Check handle events
    if (result < 0) {          // Error

    } else if (result == 0) {// No Events occurred

    } else {                   // Events occurred
        // Handle events
    }
}
```

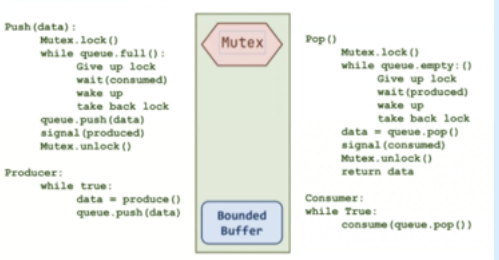


Action	Process	Thread
Create task	fork + exec	pthread_create
Wait for task	wait / waitpid	pthread_join
Lock critical section	sigprocmask(SIG_BLOCK, ...) / sigprocmask(SIG_UNBLOCK, ...)	pthread_mutex_lock / pthread_mutex_unlock
Notify another task	kill / sigaction / signal	pthread_cond_wait pthread_cond_signal

Concurrent DS: Producer/Consumer



Concurrent DS: Monitor



Reading 04: Concurrency In a **multi-threaded** program, there is more than one point of execution. Each **thread** in the process is a separate stream of execution, but still shares the same address space and thus can access the same data. The state of each thread is stored in a thread control block.All threads within a process share the same code, data, and heap, but not stack. To take advantage of modern CPUs with multiple cores, application developers must **parallelize** their **single-threaded** programs such that multiple threads do work on different CPUs simultaneously. One common use of threads is to **overlap** I/O with other activities within a program. With processes, we use fork to create a new process, and wait to wait for the process to finish. With POSIX threads, we use **pthread_create** to start a thread and **pthread_join** to wait for one to finish. If we need some sort of signaling between threads, we can use a **condition variable** To solve the race condition, we either need **atomic** instructions which operate in a "all or none" fashion, or a set of **synchronization primitives** which will allow us to build multi-threaded code that in a synchronized and controlled manner. In **event-based** concurrency, we simply wait for something to occur; when it does, we do some small amount of work based on the type of event. This processing usually happens in a simple construct known as an **event loop**. On a Unix system we can use either the **select** or **poll** system calls to perform non-blocking or **asynchronous** I/O. The advantage of the event-based approach is that we don't need to deal with any locks normally found in threaded programs. **Concurrency:** Whenever we structure a problem such that we have multiple streams of execution **Parallelism**: Whenever we have the hardware resources to simultaneously execute multiple streams of executions **Internal Concurrency** In addition to inter-process concurrency, we can also have concurrency within a single process: Signals, Mixing I/O and Computation, Background vs foreground **Threads** Implement concurrency by dividing a process into multiple streams of execution - enables parallelism and concurrency **Sharing is bad:** Shared Data: Multiple tasks wish to access common resource concurrently; Uncontrolled Scheduling: No control over when a task runs, so access is non-deterministic; Need For Atomicity: Want operation where we will not be interrupted **Threaded Solution** Critical Section: A piece of code that accesses a shared resource; Race ConditionArises if multiple tasks enter critical section at the same time; Mutual Exclusion: A means of guaranteeing only a single thread ever enters a critical section; **Events**: If we only need to overlap I/O and computation, we can use events to provide **concurrency without parallelism**: Register interest in events (callbacks); Event loop waits for events, invoke handlers; Handlers generally short-lived and not preempted; To achieve asynchronous style I/O while still utilizing our traditional read/write system calls, we can use **select** or **poll** to check if I/O is ready or not before performing the blocking call. User threads to kernel threads - **Many-To-One:** Used by many programming languages, in userspace so efficient, but one blocking thread will block all threads, can't split across CPUs **One-To-One:** No problem with blocking or splitting across CPUs, overhead and limited in number of threads **Many-To-Many**: Complex, but can scale (used by Go and Erlang) **Lock Evaluation** To evaluate a lock implementation, we need to consider the following three metrics: 1) Correctness: Does it actually provide mutual exclusion? 2) Fairness: Does it give each thread a fair shot at acquiring the lock? 3)Performance: How much overhead is added by using the lock? realtime os cares more about **fairness, correctness** always number one, **performance** important for linux; **Disabling Interrupts:** prevents from being interrupted mid-stream; we probably want some life-threatening interrupts; This is cooperative - assumes processes are well behaved; Not preemptive - periodically interrupting processes; not correct - we lose interrupts; Need to trust processes to enforce fairness; Poor performance - doesn't scale to multiple proceses; **Spin Lock:** Problem with correctness (race condition) and performance (busy waiting); **Test and set:** hardware solution that provides **atomic** exchanges;, problem with performance (busy waiting) **Yielding spin lock:** reduces cost of spin waiting by yielding (giving CPU to scheduler) as we spin; yield has a high overhead because it uses a context switch; save current ex context, update, load new context; linux kernel favors spinning for a little bit before yielding; Locks can be **tricky** to implement and utilize correctly: To implement them efficiently, we need support from both the hardware and the operating system. To utilize them effectively, we need to identify the critical section and use locks only when absolutely necessary. To avoid spin waiting we will need to utilize condition variables

Reading 05: Locks and Condition Variables When using threads, we can protect a critical section by employing a **lock**. To do so, we must first declare a **lock variable** which holds the state of the object at any instant in time. This object is either **available**, which means no thread holds the object, or **acquired**, which means exactly one thread (aka the **owner**) holds the object and is in the critical section. **coarse vs fine-grained locking:** The big kernel lock (BKL) is an example of coarse-grained locking; Fine-grained locking increases concurrency by protecting different data structures with different locks; **Locking Mechanisms: Test-and-Set** provides a machine instruction that allows us to test the old value at a memory location and while simultaneously setting the memory location to a new value. **Compare-and-Swap** provides a machine instruction that allows us to test a value, update it if it is the expected value, and return the actual value at that memory location. **Load-Linked and Store-Conditional** instructions work together to atomically fetch a value and update it. **Fetch-And-Add** can be used to implement a ticket lock, which guarantees progress for all threads. **Spin Locks**: Spin-waiting is inefficient because a thread wastes CPU resources simply waiting for another thread to release a lock; Yielding allows a thread to deschedule itself when it discovers the lock it wants is being held. Parking involves putting a thread to sleep temporarily and into a queue that will be used to select the next thread to wake up. To make a data structure **thread safe**, that is ensure it provides correct concurrent access to data, we can utilize a **monitor**, where locks are acquired and released automatically as you can and return from object methods. **Concurrent lists, queues, and hash tables**: We should only lock portions of methods that actually access the shared data resources; We should be wary of code that has many returns or exits as that can make it difficult to manage our locks. To avoid spin waiting, we can use a **condition variable** to create an explicit queue that threads can put themselves on when some state of execution is not as desired. When another thread changes said state, it can wake a **waiting** thread and allow them to continue by **signaling** on the object. A **condition variable** is an explicit queue that threads can put themselves on when some state of execution is not as desired. We use a condition variable in conjunction with a lock: 1) Before we wait, must have the lock 2) When we wait, we release the lock and go to sleep 3) When we wake-up from a signal, we re-acquire the lock; We need to use **while** loops around our conditions to deal with spurious wake-ups; Always hold the **lock** when signaling or waiting; **Concurrent Data Structures**: To construct a concurrent data structure, we need to utilize both locks and condition variables; **Producer / Consumer Problem:** In this classic concurrency problem, we have one or more producers generating data that multiple consumers must process: (monitor)

Reading 06: Semaphores A **semaphore** is a synchronization primitive that consists of an **integer** value that we can manipulate with two routines: **sem_wait**: This function decrements the value of the semaphore and waits if the value of the semaphore is negative. **sem_post**: This function increments the value of the semaphore and then wakes up any waiting threads. To make a lock (binary semaphore), we initialize the semaphore to 1. To make a condition variable, we use sem_post to signal and sem_wait to wait. **Reader-writer** locks allow for multiple concurrent readers as long as there are no writers. Reader-writer locks are prone to starvation problems. **Slackline** Get on(): 1) mutex_lock(Lock) 2) while Slackers >= CAPACITY 3) cond_wait(Line, Lock) 4) Slackers++ 5) mutex_unlock(Lock)
Get_off(): 1) mutex_lock(Lock) 2) Slackers-- 3) cond_signal(Line) 4) mutex_unlock(Lock) (Throttling/Rate-limiting problem - only so many people (threads) at once).
A **semaphore** is a synchronization primitive that consists of an integer that we can manipulate with two operations:

A semaphore is a synchronization primitive that consists of an integer that we can manipulate with two operations:

```
sem_wait(s):
    s.value--
    if s.value < 0:
        thread_wait()

sem_post(s):
    s.value++
    if threads_waiting():
        thread_wakeup_one()
```

To use a semaphore as a lock:

- Initialize the semaphore to 1
- Perform a Wait to acquire the lock
- Perform a Post to release the lock

```
// Initialize lock
sem_t m;
sem_init(&m, 0, 1);

// Perform Lock
sem_wait(&m);
// Do critical section
sem_post(&m);
```

To use a semaphore as a condition variable:

- Initialize the semaphore to 0
- Perform Wait to wait on condition variable
- Perform Post to signal condition variable

```
// Initialize cond var
sem_t cv;
sem_init(&cv, 0, 0);

// Thread 1: Wait
sem_wait(&cv);
do_the_thing();

// Thread 2: Signal
do_the_other_thing();
sem_post(&cv);
```

Very helpful when you have a bounded buffer. Unlike locks and condition variables, there is no notion of ownership. Can avoid using locks and condition variables if desired. With semaphores, we can create interesting synchronization objects such as reader-writer locks: Once readers acquire the writelock, as many readers as possible can read the data; Once a writer acquires the writelock, no one else can access the data; Unfortunately, this can add overhead and is prone to starvation

Semaphore: Fair

```
size_t Readers      = 0
sem_t  WriterLock   = 1
sem_t  ReaderLock   = 1
sem_t  ServiceQueue = 1

Writer():
    sem_wait(&ServiceQueue)
    sem_wait(&WriterLock)
    sem_post(&ServiceQueue)

    do_write()
    sem_post(&WriterLock)
```

```
Reader():
    sem_wait(&ServiceQueue)
    sem_wait(&ReaderLock)
    if Readers == 0:
        sem_wait(WriterLock)
    Readers++
    sem_post(&ServiceQueue)
    sem_post(&ReaderLock)

    do_read()

    sem_wait(&ReaderLock)
    Readers--
    if Readers == 0:
        sem_post(WriterLock)
    sem_post(&ReaderLock)
```

Slackline: Locks and Condition Variables

Suppose you and your friends are going **slacklining**. Unfortunately, the **slackline can only support up to three people on it at a time**. Therefore, if there are many people, they will need to **wait before they can get onto the slackline**.

Assuming each person performs the following procedure:

```
get_on()            // Get on slackline if there is enough room
cross_slackline()   // Attempt to walk across slackline
get_off()           // Get off of slackline
```

Solve this concurrency problem using locks and condition variables.

```
22 get_on():
23     mutex_lock(Lock)
24     while Slackers >= CAPACITY:
25         cond_wait(Line, Lock)
26
27     Slackers++
28     mutex_unlock(Lock)
29
30 get_off():
31     mutex_lock(Lock)
32     Slackers--
33     cond_signal(Line)
34     mutex_unlock(Lock)
```

Reading 07: Concurrency Bugs To solve the Dining Philosopher's problem, cange the order in which the last philosopher acquires the forks. **non-deadlock concurrency bugs** are: **atomicity violations**: occur when the assumption of atomicity is incorrect; Solve by locking the resource. **order violations**: occur when the desired order of operations is not enforced during execution.Solve with condition variables (and locks) **Deadlock** occurs when each thread is waiting for another to give up a resource and none can run. Usually this is because there is a cycle in the graph of dependencies; occurs when the following **four conditions** hold: **circular wait:** There exists a circular chain of threads such that each thread holds one or more resources that are being requested by the next thread in the chain. **no preemption**: Resources cannot be forcibly removed from threads that are holding them. **Hold-and-wait**: Threads hold resources allocated to them while waiting for additional resources. **mutual exclusion**: Threads claim exclusive control of resources that they require. To **resolve** deadlocks: Break any one of the four conditions. Provide some form of ordering to circumvent circular waiting, Avoid locks by using atomic hardware instructions. mutex: add a lane (sharing is bad), hold and wait: Implement a condition variable so that they don't hold onto the variable (danger - livelock), no preemption: traffic director (scheduler), circular wait: just use one lock for the whole lane or always use the same ordering. Rather than prevent deadlock, we can instead try to avoid it by having a smart **scheduler** that will consider the dependencies between threads or by having a service that **detects** deadlock and provides **recovery** techniques. **Livelock** occurs when multiple threads are actively attempting to acquire resources, but cannot make progress. **Starvation**: occurs when a thread is prevented from ever accessing a resource; Help prevent Starvation: ordering scheduling thing, Convoy effect (big job runs first) Complete shortest job first

Why a process per tab rather than a thread per tab? Started with threads to share memory; uses less memory; windows - process is very expensive and threads are not very expensive; Not necessary to share memory across tabs; Security sandboxing: because each page was not sandboxed, one tab could mess up all the other tabs and read content in other tabs; Performance - multiple processes better leverage available client computing power; can accidentally make it non sequential with locks and kill concurrency; With processes you dont have to coordinate them; does use a lot more memory

One possible design for a web crawler: 1) Create a thread pool of N threads. 2) Create a concurrent queue (monitor) to store the URLs. 3) Create a concurrent set (monitor) to track which URLs you have seen. 4) Have each thread in thread poll pop one URL from the queue and then process it (download it, find the URLs, update the database). 5) If a thread finds any new URLs, push it to the queue (use the concurrent set to determine if the URL has been seen before).