OPERATING SYSTEMS/KERNEL

- What are the three roles of the Operating System?
    - Multiplexing of hardware resources
        - CPU while finite, is able to handle many processes that want to run
            - OS does through scheduling
        - Many processes load and store to/from memory and disc, even though RAM and disc are finite
            - OS does this through VM, VA spaces, Page table management/TLB management
    - Abstraction of Hardware Platform
        - Written, compiled, and executing programs are abstracted to processes
        - Syscalls provided to access hardware resources
        - VM abstracts physical memory, allows programs to see their own large address spaces
        - Use of threads and locks to abstract low level registers and instructions for a process
        - Turning data storage into the filesystem
    - Protection of software principles from each other through the use of hardware
        - OS guarantees isolation through VM and VA spaces
        - Per process page table structure
        - Access permissions in PTEs (reading memory, writing memory, user bit, etc.)
        - Privileged mode on processor turned on through trap process
            - ‘Protects’ hardware resources from malicious/badly written user programs
        - File system permissions
- What Primitives does hardware provide to the OS?
    - Execution modes on a CPU that can be handled to support protection
    - Interrupts by other devices
    - Exceptions by the CPU for illegal behavior
    - Timers and clock related primitives
    - TLB, MMU, hardware that walks page tables for address translation
        - Access to RAM used to make a multi level page table
        - Implementation of bit manipulation to set access permissions within PTEs
    - (basic) set of registers and instruction set that make up the ISA
        - Register state is saved by the OS during context switches
    - Ways to transition between execution modes:
        - User -> Kernel/Kernel -> Kernel (Traps)
            - Syscalls

            - Set of APIs for user level programmers to access privileged hardware resources via the OS kernel
        - Interrupts
            - External event that needs to be handled
            - Asynchronous (not in accordance with the running program)
            - control+c/any keyboard input, timer interrupts, etc.
        - Exceptions
            - Caused by illegal instruction, so generally synchronous
            - Divide by 0 error, page fault, seg fault, etc.
    - Kernel -> User: Upcalls
        - Upcalls, or signals, are asynchronous notifications from the kernel space to the user space
            - Don't alter execution of a process, not 'callable' in any way
        - Quite useful:
            - Call some kind of signal handler every time a number of timer interrupts occur (lab alarm)
            - Networking: I/O interrupts from receiving a packet can be passed through the application layer, calling user level functionality
            - Signals: SIGINT, SIGTERM implemented through upcalls
            - Interrupts/exceptions that affect the user program could be handled through upcalls

- Implementations of transitioning between execution modes: Traps
    - **Below, I've defined high level concepts and then explained how they are implemented in the RISCV ISA or xv6 kernel**
    - User -> Kernel (Syscalls)
        - We need different execution modes (supervisor vs. user)
            - Set by bits in the **mstatus** register in RISCV
            - Changed when the user thread transitions to the kernel thread
        - Need mapping of virtual to physical address spaces
            - Trampoline and trapframe saved at the top of VA space
                - Allows kernel to execute without making a page table switch
            - SATP register contains the address of the root level page table, changes based on process
        - User mode, via a syscall wrapper, passes arguments down to a kernel level syscall

        - Wrapper calls the **ecall** instruction (the following explains this instruction)
        - Arguments are placed in a0-a6, register a7 contains syscall #
        - users PC register, which keeps track of currently executing instruction saved in **sepc**
        - Caller's mode (user in this case) saved in **sstatus**
        - PC changed to trap handler, which is stored in **stvec**
    - The traphandler is now called inside of the kernel, which determines the cause of the trap, saves the process state, and calls the syscall
        - Jump to the PC (**set by stvec**) which is uservec in trampoline.S
            - Remember, since trampoline is mapped in the user address space in each process, we don't need to make a page table switch in the trap
        - The following is executed in trampoline.S:
            ```
            csrrw a0, sscratch, a0
            ```
            - This is a special instruction that swaps the **sscratch** register with a0, which now holds the address of the trap frame.
            - All user registers are saved in the trapframe now, using offsets of the address in a0.
            - a0 then saved in the trapframe, and we jump to usertrap() in trap.c
        - Usertrap() checks the cause of the trap
            - Any interrupts/exceptions (WHILE IN KERNEL/SUPERVISOR MODE) are sent to kerneltrap()
            - If r_scause() == 8:
                - This is a system call
                - Set saved **sepc** in trapframe to +4, to go to the next instruction when the kernel returns
                - Call syscall() in syscall.c
        - syscall() in sycall.c determines function to call
            - Get syscall number from a7 register in the trapframe
            - Use function pointer array to get address of the syscall function
            - Call syscall function in sysproc.c, and put the result in the a0 register.

            - usertrapret() called in trap.c, after syscall() finishes executing
                - Sets trapframe values to values needed for future traps
                - Sets privilege mode to user
                - Resets pc
                - Jump back to trampoline.S
            - trampoline.S now switches back to the user page table, restores the register state from the trapframe, and calls the sret instruction to return to user space
    - With user -> kernel traps that AREN'T syscalls, such as exceptions that can occur from illegal instructions in user programs, the flow is relatively similar, although r_scause will be different. (This is dealt with in usertrap()). The hardware will be doing the context switch rather than a wrapper via ecall
    - With the case of an interrupt that occurs during a user process (think lab alarm), the code to deal with this is usually implemented in trap.c (specifically usertrap()
        - In lab alarm, we implemented a handler when the timer interrupts reached a certain amount of ticks
    - Kernel -> Kernel (Exceptions, Interrupts)
        - The main difference here is that instead of going to usertrap() in trap.c, it goes to kerneltrap() in the same file. Before going to kerneltrap in trap.c, the control flow goes to kernelvec in kernelvec.s, rather than uservec in trampoline.s. This assumes that stvec points to kernelvec in supervisor mode.
        - Device interrupts, timer interrupts, and kernel exceptions are handled here.
        - I found this to be a good resource: https://clownote.github.io/2021/03/16/xv6/Xv6-traps-and-system-calls/
- Implementations of transitioning between execution modes: Upcalls/Signals
    - A good example of this is in lab alarm, where when making the sigalarm syscall, we pass in a function handler. This is stored in the proc structure. When the timer interrupt occurs (and enough ticks are reached), the kernel "upcalls" to the user program by setting epc to the address of the handler, effectively "calling" the user function

PROCESSES AND THREADS

- What is a program? A process? A thread?
    - A program is a list of instructions written in some kind of programming language

        - It also has global variables stored
        - Stored in the disk
    - A process is an abstraction created by the operating system to represent an instance of a currently running program.
        - Processes have their own ids, address spaces (with a stack, heap, code, data), file descriptor table, and state
        - Processes can create other processes and be child processes themselves (meaning that they have a parent process)
    - A thread is another abstraction to represent execution contexts within a process
        - Specifically, a thread is some execution sequence for a task that is schedulable by the OS
        - Processes contain threads which share resources such as memory
- Processes vs. Threads
    - While processes represent an instance of an entire running program, threads represent a specific execution context within a process.
    - Processes have their own address space, parent process, and file descriptor table. Threads borrow from this address space, but they have their own stacks and register state, and general state
    - While processes are an abstraction to help programmers and programs run, threads are an abstraction to support concurrency
- States for Processes:
    - Running: A process that is currently being executed by the CPU is in this state
    - Ready: A process has been descheduled and placed in the ready state. The scheduler selects from processes in the ready state when choosing a process to run. A process in the ready state is NOT currently running, but isn't blocked either
    - Blocked: Processes are placed here because they are waiting for some event to occur, such as an I/O operation. Once the event finishes, it moves into the ready state
    - Zombie: This state is entered when the child exits
- How do transitions occur between process states?
    - Running -> ready
        - Descheduling by the scheduler
    - Running -> Blocked
        - Process has to wait on I/O operation (Disc read/write, network operation)
        - Parent process calls wait()
    - Blocked -> ready
        - Process finished waiting on I/O operation
        - Child exited
    - Ready -> Running

        - Scheduler chooses to schedule the process
- States for Threads:
    - Running: Currently running Thread
    - Ready: A created thread that isn't running yet, or a descheduled thread (not blocked)
    - Blocked: Thread that is waiting for some event to occur can be blocked, or a thread that called join()
    - Zombie: Could occur when a thread that was "joined to" exits
- How do transitions occur between thread states?
    - Running -> ready
        - Thread called yield() and gave up the processor
        - Was simply descheduled (via timer interrupt, for instance)
    - Running -> blocked
        - Thread called join()
        - Waiting on an I/O operation
    - Blocked -> Ready
        - I/O operation finished
        - Joined thread exited
    - Ready -> Running
        - Scheduler schedules the thread
- How do processes and threads support concurrency?
    - Processes can be descheduled/blocked while other processes run
    - Similarly, threads can be descheduled/blocked while other threads run
        - This is very useful for I/O operations that are blocking
    - Threads can be run at the same time on separate cores
        - This is very useful for parallel computation
    - What are the implications of 1:1 and N:1 threading models?
        - In the 1:1 model, there is one kernel thread for every user thread
            - OS has control over all threads via the scheduler (within each processor)
            - Used by xv6 and linux
            - In xv6, there can only be 1 user/kernel thread pair per process. In linux, there can be more
            - Syscalls/traps related to the user thread are handled by the corresponding kernel thread
            - Can require a lot of state (specifically a lot of kernel stacks)
            - Need OS kernel for context switch between user threads and user -> kernel thread
        - In the N:1 model, the OS sees a single unified thread

            - Various user level threads that are implemented through a software library
                - These run over one kernel thread
            - User level version of “swtch()” defined to switch between threads
            - Preemptive scheduling via timer interrupt by cpu
                - Timer interrupt triggers kernel to do a signal to the current user thread
                    - Similar to sigalarm
            - If a syscall blocks, we will go into the kernel and all other threads will be blocked (the switching occurs in the user level)
                - The support for I/O operations which often are blocking is therefore poor under this model
- What is the implementation for processes and threads in xv6?
    - Processes:
        - Process control block (struct proc) that contains the trapframe, process state, pid, page table, file descriptors, etc.
        - Processes have a per process page table and address space
        - Process control block also contains struct context, which stores the context for the kernel thread for that process
        - The context for the kernel scheduler thread is stored in the cpu struct
    - Threads:
        - Each process has a user thread (the executing program) and a kernel thread (handles traps)
        - Uservec and userret handle transitions between user and kernel thread
            - Saving user context (registers in the trapframe)
            - Restoring kernel stack pointer and page table
            - Restoring user context (in userret)
        - Another kernel level thread exists for the scheduler (per CPU core)
            - yield() switches to this thread
            - The scheduler thread determines next user level process to run
            - The scheduler also switches to the new process’ kernel thread
            - These switches happen through the swtch() function, which is more lightweight then user -> kernel/kernel -> user switches
                - No need to save caller-saved registers
                - No need to switch page table
                - No need to clear TLB (same address space in the kernel)
            - Kernel mode threads all share the same page table
            - User mode threads have a per process page table
        - Relationship between switch and yield? Timer interrupt and yield syscall?

PROGRAMMING INTERFACE

- Why are syscalls needed?
    - Syscalls both abstract the process of accessing/using hardware resources by making a simple to use API to the kernel, and also protect hardware resources from potentially malicious or badly written programs. They give the kernel flexibility and power to handle user requests for hardware resources
- How are syscalls implemented?
    - High level: User level wrapper calls ecall, which saves the status of the user program, the current pc, loads in arguments, and essentially jumps to the trampoline and in turn the trap handler. The trap handler then calls the syscall, which loads the return value into the a0 register. The trap returns, and the trampoline code restores the user state. Finally, we are back in user space
    - For a more low level explanation of how syscalls are implemented in xv6, check the section above
- What are file descriptors?
    - On a high level: “handles” to resources
        - i.e. 0 points to stdin, 1 to stdout, 2 to stderr, etc.
    - Syscalls often take file descriptors as arguments
    - The process control block has a file descriptor table
- What do common syscalls do?
    - Fork():
        - Creates a copy of the parent process that called fork, including the entire parent’s address space
        - The child has its own PID, its own address space (but the memory is copied), and its own copy of the file descriptor table
        - The child starts execution as if it had just called fork itself
        - From the parent’s perspective, fork returns the child PID. From the child’s perspective, fork returns 0. This can be used to differentiate parent output with child output. When further combined with the wait() syscall and exec() syscall, a powerful, deterministic model for process creation arises
    - Wait():
        - Waits for a child process to finish, effectively blocking the parent process until the child finishes running. Using this with fork() makes the output deterministic.
        - When the child process finishes, it goes into a “zombie” state before being cleaned up
    - Exec():
        - Changes the program being run by the process that called exec
        - Commonly used with fork:
            - fork() to create a child process

            - exec() to replace the child process with a specific program and arguments
    - Pipe(fd[2]):
        - Creates a two ended “pipe”
        - Each end = file descriptor
        - Fd[1] WRITES into the pipe, while Fd[0] READS from the pipe
        - Internal buffer maintained by OS holds data to be read
        - Information flows in one direction
            - Want two directions? Make two pipes
        - This can be used for interprocess communication
    - Dup(fd) -> fd or Dup2(fd, fd) -> fd:
        - Duplicates a file descriptor for the same resource
        - Pass in an old fd, return a new fd
            - New FDs are always the lowest available number
            - However, if using dup2, the second argument is used for the new file descriptor rather than lowest available
                - If the newFD was being used, close(FD) is called
        - Allows multiple file descriptors to reference the same resource
        - Can be used to replace stdout/stdin with a pipe (when used with close)
            - Simply close fd1, and dup(p[1]). Now, writing to stdout (write(1, __, __)) really writes to the pipe
            - Note: the read() syscall is blocking when no data is provided. This means that if a child calls exec on a program that uses the read syscall (and the child was scheduled before the parent), the child will be blocked, and execution will go to the parent first
    - Close():
        - Closes what a file descriptor references. For instance, if you close the file descriptor 1, it will no longer reference a resource.
        - This opens up the file descriptor to new resources via dup

VIRTUAL MEMORY

- How are memory allocations managed?
    - An initial allocation, through a syscall such as mmap() or sbrk(), is made
    - This allocation of virtual memory is the initial size of the heap, and represents the entire “free list” of memory, condensed into one block
    - Each block in the freelist has a fixed-length header that describes the size of the block, and could also have a magic number for data integrity purposes
    - Allocations and frees from memory take from the free list and add back to it
- Mechanisms for memory allocations:
    - Splitting:

        - When memory is malloced, the free list is traversed. Each header is checked for a block size big enough to fit the request. If the block is big enough, it could potentially be split.
        - For instance, assume a call of malloc(32 bytes), and assume 24 byte headers. When traversing the free list, we come across a header that gives a size of 64 bytes. Based on policy, we select this block. It can then be split - 32 + 24 = 56 bytes split off for the allocated block (including the header), leaving 88 - 56 = 32 bytes left. This 32 byte chunk is left in the freelist, and the 56 byte is split off and returned to the caller of malloc
        - Remember, in the above example, the header gives a size of 64 bytes, but the true size is 64 + 24 to include the header
    - "Normal Allocation"
        - It is possible that due to policy/layout of the current freelist, a node could be exactly the size of what the allocation demands. This would lead to a normal allocation, where the node is simply removed from the list and returned to the caller of malloc()
    - Coalescing
        - When nodes are returned to the freelist via free(), they could potentially be "coalesced" if the memory address aligns with the node on the left, on the right, or on both sides
        - Essentially, nodes can combine with adjacent freelist nodes when being inserted back into the list, which reduces external fragmentation
        - Otherwise, they are just inserted normally
- Policies for memory allocations:
    - There are various policies for allocating:
        - First fit: choose the node that "fits" the allocation first in the traversal
            - Can be efficient in terms of searching
        - Best fit: choose the node that "fits" the allocation the best in the free list traversal, i.e. it is smallest node that fits the allocation
        - Worst fit: choose the node that is the largest node to fit the allocation, opposite of best fit
            - Can reduce external fragmentation in the freelist
        - Each of these policies can perform differently based on the situation, neither of them are truly the best or optimal
- What is Fragmentation? How can it occur?
    - Fragmentation, generally, is inefficient memory use or waste that can occur with allocated memory.
    - Fragmentation can occur after splitting nodes too much and being unable to coalesce them, having wasted memory (too much was allocated and now a remaining small portion is wasted), etc.

- What are the different flavors of fragmentation?
    - External (More common)
        - External fragmentation occurs when memory that is to be allocated is fragmented, i.e. too many small chunks.
        - An allocation may occur of 48 bytes, for instance. There may be a total of 64 bytes available in the freelist, but divided into two chunks of 32. Therefore, the allocation would fail, and memory is wasted
    - Internal
        - Internal fragmentation occurs when too much memory is given to the process that is allocating. In other words, they might only use 20 bytes of a 64 byte chunk, leading to 44 bytes of waste
        - This could occur under policy such as worst fit
- How does Base and Bounds work? Segmentation? Paging?
    - Base and Bounds:
        - Each process, like always, gets their own address space. This address space is managed by two registers, which contain physical addresses: a base register and a bound register (Base = lower end, base + bound = higher end)
        - To get a physical address from a virtual address, simply add VA + base. If VA >= bound, throw an error
        - Pros:
            - This is a simple implementation
        - Cons
            - Large internal fragmentation, as lots of space is allocated to support stack and heap growth
                - This leads to a large chunk in the middle of the address space that is allocated but not in use
                - This occurs because address spaces must be placed in fixed-sized slots
                    - Why fixed sized? The OS has to set the base and bounds register to point to free physical memory when the process begins. The contents of these registers can't change, so the address space remains fixed
            - Very bad at multiplexing RAM
                - How can you run a process that takes up the entire address space? How can you run thousands of processes that collectively have address spaces larger than RAM?
    - Segmentation

    - Segment table maintained for each process, with each entry representing a arbitrarily sized segment with base and bounds
    - Segments can be located anywhere in memory, rather than in a fixed-sized slot
    - Segments have access permission bits
    - Register points to the segment table
    - Two processes can share the same physical segment
    - Virtual addresses are used to index into the segment table, and locate the specific byte of memory in the mapped physical address segment.
        - VA divided into ID and offset
        - ID indexes into the segment table
        - Segment table entry used to get base and bounds
        - Offset + base gets us to the actual address in physical memory
        - PA = Seg.base + VA.offset
    - Scenarios where exceptions should be thrown:
        - VA.ID not in segment table
        - Bad access permissions (i.e. trying to write data to a read only segment)
        - VA.offset >= bound
    - Pros:
        - Less internal fragmentation (no wasted space between stack and heap)
        - Can now share code (libraries) and have access permissions
    - Cons:
        - Segment tables take up space in memory
        - Because we have to go to memory to access these tables, address translation is much slower than the base and bounds approach
        - More complex
- Paging
    - Uses fixed size “pages” of memory (typically 4KB)
    - Each segment of memory is made up of a number of pages
    - Virtual pages map to physical pages
    - Uses a per process page table instead of segment table
    - Register points to page table
    - Virtual addresses are used to index into the page table, and determine the offset of which byte to access within a page
        - VA divided into VPN and offset
        - VPN indexes into the page table to find the PTE and PFN
        - Check to see if PTE is valid (checks valid bit)
        - PFN * Pagesize = base

        - PFN + 1 * Pagesize = bound
        - PA = PTE.PFN + VA.offset
    - Scenarios where exceptions should be thrown:
        - PTE.valid = 0
        - Invalid access permissions
    - Pros
        - Less external fragmentation due to fixed sized pages
        - Simpler due to fixed sizes
    - Cons
        - Still very slow due to accessing memory
        - Still takes up a lot of space due to each process having a full page table that spans the address space
- How does the TLB work? What problem does it solve?
    - The TLB is a cache inside the MMU that greatly speeds up memory translations
    - This solves the problem of paging being too slow because the page tables are located in physical memory
    - Maintains VPN to PFN mappings for valid pages
        - There isn't one row per VPN, so VPNs are actually in the TLB
    - Perform lookups on the cache -> if it's in the cache and the correct access permissions, return the PFN. Otherwise, we have to go into memory
    - What about context switches? There are two approaches
        - Flush the TLB on a context switch via CPU instruction
            - SFENCE.VMA on RISCV
        - Use tagged TLB with ASID field
- How do multi-level page tables work? What problem does it solve?
    - Multi level page tables divide the original, fully spanning address space page table into multiple levels.
    - VPNs split into indices for each level, and PFNs in higher level tables point to the addresses of lower level tables.
    - L0, the lowest table, directly points to main memory
    - Lower level tables are simply omitted and not allocated if the higher level entry is not valid (This saves space)
    - VA split into indexes for each level, and an offset
        - Assume a 3 level page table
        - L2 index indexes into the L2 table (the address of the L2 table is stored in the **satp** register in RISCV)
        - The entry in L2 points to the address of L1
        - L1 index indexes into L1, getting the address of L0
        - L0 index indexes into L0, getting the PFN

            - The offset then indexes into the physical page to get the exact memory location
        - Multi level page tables solve the problem of paging taking up too much space/overhead
- How is an access to a virtual address handled end-to-end?
    - Find the VPN within the the virtual address
    - Lookup the VPN in the TLB:
        - If it is a hit:
            - Check protection bits:
                - If they are properly set:
                    - Find the offset
                    - Get the physical address using the PFN and offset
                    - Access the memory
                - If they aren't:
                    - Raise an exception (protection fault)
        - If it is a miss:
            - Get the address of the root level page table via the **satp** register
            - Traverse the multilevel page table using the VPN as index bits into each table
            - If at any point there is a page table entry with invalid protection bits:
                - Raise an exception (protection fault)
            - Else if at any point the page table entry's present bit is true
                - Insert the vpn and pte into the TLB
                - Retry the instruction
            - Else if the present bits are false (not in main memory)
                - Raise an exception (page fault)
    - Page faults are handled like this:
        - Find a free physical page in memory
            - This requires a policy if there is no free page and you need to evict a page, such as LRU, FIFO, etc.
            - Could potentially check bits such as accessed (for LRU), or dirty to see if you need to write the evicted page back into the disc swap space
        - Perform an IO operation to move memory from disc into RAM
            - To get the memory, use the disk address found in the page table entry corresponding to the original virtual address
                - Since it is not present in physical memory, the PFN will have the disc address rather than RAM address
            - Additionally, pass in the PFN of the newly found free page

        - Set the present bit to true in the page table entry
        - Set the PFN to the new PFN
        - Retry the instruction (the OS does not do this, the application issues it)
        - If the page is NOT found in disc, a segfault will be thrown
    - After getting the physical address, the actual caches are checked for memory: L1, then L2, then L3, then DRAM
- How can we apply virtual memory indirection in useful ways?
    - This usually involves modifying the mappings in some way (changing which VAs map to which PAs or the general memory layout) or modifying the page fault handlers
    - Null pointer debugging:
        - Make the user's virtual address layout start one page above 0, effectively making 0 → PAGESIZE a guard page
            - This is a mapping modification
        - If the user accesses this guard page, we can throw a null pointer exception (but there is no need to heavily modify the page fault handler here)
    - Lazy Allocation:
        - Lazy allocation (in regards to heap allocation) is a bit tricky. On a high level:
            - When a process calls malloc in xv6, the sbrk() syscall is used to grow the heap by a certain amount. Normally, pages would actually be allocated in the process' page table
            - However, in lazy allocation, the sbrk() syscall will only increase the "size" of the heap without actually allocating memory
            - Everytime an access is made to memory that has been allocated on the heap, there will now be a page fault
                - We can modify the page fault handler to check if these are valid accesses, and then allocate on demand
                - This potentially saves space and improves performance
            - In a way, you are modifying the mappings by not allocating anything via sbrk, and you are definitely modifying the page fault handler by choosing to allocate memory there
    - Demand Paging
        - Set up swap space in the disk
        - Add a present bit in the PTEs
        - If the present bit is 0, that means the memory is in disk. In order to locate this memory, the PTE should have a disk address, rather than a PFN/PPN
        - In this case, we are modifying the mappings. However, when there is a page fault (valid but present bit == 0), we should modify the handler to properly copy the memory from disk to RAM, and evict data if necessary

- Copy-on-write
    - When a child process is created via fork, a copy of the parent's memory is given to the child. This can be wasteful if exec() is called immediately after fork(), because all the copied memory is immediately overwritten by a new program
    - In copy on write, we modify all entries in both the parent and child page table to be readable, and to point to the same memory (so there isn't really a copy anymore)
    - If a write is made, the page fault handler makes a copy of the memory on demand, and the associated mappings are allocated
        - Protection bits also are updated
    - We are changing the mappings (protection bits, no longer making a copy) and the page fault handler here

CONCURRENCY/SYNCHRONIZATION

- Why do we need synchronization for concurrent programs (e.g., multiple threads)?
    - Threads share the same address space (besides a per thread stack). If multiple threads access shared memory at the same time with at least one write, undefined behavior can occur. This is known as race conditions or a data race.
        - The segment of code that is subject to race conditions is known as the "critical section"
    - In order to fix this issue and make output more deterministic for multithreaded programs, we need synchronization
        - We need to make sure that only one thread accesses a critical section at a time, and no interleavings between threads of this critical section occur
        - This principle is known as mutual exclusion: critical sections should be mutually exclusive, meaning that they can't be accessed at the same time by multiple threads
    - Keep in mind, the thread scheduler is non deterministic
        - We need to come up with a method that makes sure that only one thread is in the critical section. If the scheduler deschedules it, the method should still ensure that no other threads can access the critical section until the original thread no longer is in it.
- What primitives do hardware and compilers provide to build on top of?
    - Hardware offers **atomic instructions**
        - These are instructions that condense what are normally multiple operations into one operation that appears to "fully run" or "not run at all".
        - They can be used to do something in a single atomic step
    - The CPU can also reorder instructions, which can make synchronization tricky. The hardware offers "fences" or barriers
        - Draws lines around the critical section

            - Instructions can be reordered on either side of these lines (or even within them), but they can't be reordered across the lines
            - This makes sure that code before the critical section executes before it, and code after executes after it
    - The compiler can also reorder instructions. asm volatile("" ::: "memory") can be used as a sort of barrier to prevent this
    - __sync_synchronize()
        - This provides the barrier solution for both the compiler and the CPU
- What is a lock?
    - A lock is an abstraction (around atomic instructions and barriers) provided by the OS that creates mutual exclusion around data
        - It prevents multiple threads from writing to the data at the same time, when it is held
    - What is the API?
        - Acquire: Attempts to set lock variable to 1 (in the lock struct) through an atomic instruction. If it is already set to 1, it "spins" in a while loop until the lock variable is set to 0. After setting the variable to 1, it calls a barrier() function to prevent moving a critical section ahead of a lock access
            - High level: attempt to acquire a lock, wait if you can't
        - Release: First, the barrier created in acquire is called again to keep the critical section contained between lock acquire and release. Then, the lock is released (variable set to 0) in an atomic instruction
            - High level: release the lock
    - What are lock properties/rules?
        - Properties:
            - Safety: Something bad NEVER happens
                - Make sure the program doesn't go into an invalid or illegal state
            - Liveness: Eventually, something good WILL happen
                - For instance, the program, regardless of what happens, will successfully return an input
        - Rules:
            - Lock data, not code
                - Locks should be related to state and not functions/instructions
            - Acquire before entering a critical section
                - Should wait if the lock is currently held
            - Release after finishing a critical section
                - Should be called by the thread that is holding the lock

- How do we build a simple lock (spinlock) leveraging the above primitives? What are the pros/cons?
    - Acquire:
        - Disable interrupts (will go over this below)
        - Make a while loop with the atomic instruction to check if the lock is acquired:
            - If it isn't acquired yet, create a barrier to end the acquire function.
                - At this point, the critical section can execute with the guarantee of mutual exclusion, and won't be reordered outside the boundary
            - If it is already acquired, do nothing (spin) and continue to the next iteration
    - Release
        - Call the barrier function to "close out" the critical section
        - Release the lock with an atomic instruction
        - Reenable interrupts
    - Pros:
        - Relatively simple and lightweight
        - Good for simple operations/short critical sections
    - Cons:
        - Thread starvation (one thread may never be scheduled and get access to critical section)
        - Lots of wasted potential for work via spinning
        - Long critical sections can cause lots of waste via spinning
            - Also, interrupts will be disabled for a while
    - What is the issue with using spinlocks in both threads and interrupt handlers (why disable interrupts?)
        - Let's say a thread acquires a lock. Upon entering the critical section, a timer interrupt occurs
        - Let's also say the timer interrupt has a critical section and attempts to acquire the same lock
            - Since it has been acquired from the previous thread, the interrupt handler will spin indefinitely, and we will be stuck
        - To prevent this issue:
            - disable interrupts after acquiring a lock OR
            - Make sure the interrupt handler doesn't use the same lock as other kernel contexts
- Yield Locks
    - Basically the same as a spin lock, but instead of spinning (just waiting in a for loop), we yield the processor in acquire() if something else already has the lock

        - Remember, yielding will make the process go from RUNNING → READY
        - Pros: More efficient due to increased "goodput"
            - We can actually do something while we wait
        - Cons:
            - Starvation still exists, whatever thread runs depends on the whims of the scheduler
                - Threads could potentially be completely shut out
            - Also, the time to acquire the lock once the other thread releases it could be much higher, because the thread needs to get scheduled again
- Sleep Locks
    - Instead of spinning or yielding, what if we block the thread, and wake it up later?
    - High Level Steps:
        - Maintain a queue of blocked threads in the lock structure
        - Acquire(): Attempt to acquire the lock. If another thread has it, add the current thread to the queue, and block the current thread
        - Release(): Release the lock. Then, if the queue is non empty, pop() and wake up the thread
            - This supports fairness and alleviates starvation problem
        - There are small critical sections around the locked state and queue themselves. These will be protected via spinlock
- How do we build a sleeplock using spinlocks? What are the pros/cons?
    - You can think of the sleeplock acquire() and release() functions as wrappers around lower level locking primitives (spinlocks), with some added functionality
    - Spinlocks are used to lock the queue, and the actual locked state
    - In sleep_acquire():
        - Pass in a spinlock
        - Immediately try to acquire the spinlock
        - If you have acquired it:
            - **While** it is locked, some other thread is holding the sleep lock. Add the current thread to the queue and lock
            - When nothing is holding it anymore, set the locked attribute to 1, and release the spinlock.
                - Other threads can now enter the sleep_acquire, but since the locked attribute will be set to 1, they will be blocked
    - In sleep_release():
        - Acquire the spinlock again
        - Set the locked attribute of the spinlock to 0
        - Wakeup a waiting process in the queue if there is one