Begin with an overview of xv6

Objective: To introduce the fundamental design choices and components of the xv6 kernel, highlighting its simplicity as an educational tool for understanding operating systems.

- It's designed to run on a **multiprocessor system**, meaning it can manage multiple CPU cores that all share the same main memory
- **CPU / Core / HART (Hardware Thread):** In the context of xv6, these terms are used interchangeably. They all refer to a processing unit that can execute instructions.
- xv6 is a modern reimplementation of **Unix Version 6 (v6),** developed at **MIT for educational purposes**. Its primary design goal is not performance or feature richness, but clarity and pedagogical value.
- Unlike production operating systems like Linux or BSD, whose codebases are measured i**n millions of lines,** the xv6 kernel is small enough to be understood in its entirety within a single academic semester.
- This deliberate simplicity makes it an unparalleled tool for learning the fundamental principles of operating system engineering

  ### Hardware World of Xv6

- Main Memory (RAM):

  ```
        - Fixed at **128 MB**. Unlike a real OS that detects and adapts to available RAM, xv6 assumes this fixed amount.
        - **Caching is ignored.** The kernel doesn't deal with complex L1/L2 cache management, which simplifies the code significantly.
  ```

- *Devices:** xv6 interacts with a small set of emulated devices:

  ```
         - **Timer Interrupts:** Each core has its own timer to ensure processes can be preempted.
         - **Disk Drive:** A single emulated disk for the file system.
  ```

Memory Management: Pages and Lists

![[Screenshot 2025-08-22 at 10.18.27 PM.png]] **Physical Memory:** - The kernel allocates memory for itself from a **free list**—a simple linked list of available pages. When the kernel needs memory, it takes a page off the list; when it's done, it puts it back. There's no complex `malloc` for the kernel. - Divided into **4 KB pages**. A page is the smallest unit of memory that the OS manages. - **Virtual Memory:** - Managed using **three-level page tables**. A page table is like a map that translates the virtual addresses a program uses into physical addresses in RAM. - Each process gets its own page table. - The kernel has its own master page table that maps all of physical memory, which is shared across all cores.

Source Code arrangement

A preliminary tour of the xv6 source code reveals a logical separation that mirrors its conceptual architecture. The codebase is primarily divided into two main directories: `kernel/`, which contains all the privileged kernel code, and `user/`, which contains the user-level programs, including the shell and standard Unix utilities.

What is a kernal

The **xv6 kernel** is the core of the operating system. Think of it as the master control program that has privileged access to all the computer's hardware. Its primary job is to manage the computer's resources (CPU, memory, disk) and provide essential services to user programs, creating a safe and efficient environment for them to run.

---

1. The Kernel as a Resource Manager

The central idea is that the kernel is the **sole manager of the system's resources**. User programs can't directly access hardware or interfere with each other. They must ask the kernel for permission and services.

- **Analogy:** The kernel is like the manager of an apartment building. It decides which tenant (process) gets which apartment (memory), controls access to shared utilities like the laundry room (disk), and ensures tenants don't enter each other's apartments.
- **Protection:** This management is enforced by the CPU's hardware protection mechanisms. The CPU can operate in two modes:
  - **Kernel Mode:** Privileged mode where all instructions are allowed. The kernel runs in this mode.
  - **User Mode:** Restricted mode for user programs. Sensitive operations (like accessing hardware) are forbidden.

---

2. Core Responsibilities of the xv6 Kernel

The kernel's job can be broken down into four main areas.

****Process Management:**

This is about managing the execution of programs.

- **What is a process?** A process is simply a program in execution. The kernel's job is to create, schedule, and terminate these processes.
- **Multitasking:** xv6 creates the illusion of running multiple programs simultaneously by rapidly switching the CPU's attention between different processes. This is called **context switching**.
- **How it works in xv6:** The kernel maintains a **process table** to keep track of every process's state (e.g., `RUNNING`, `SLEEPING`, `ZOMBIE`). The **scheduler** decides which process to run next.

****Memory Management: ****

This is about allocating and protecting memory for the kernel and each process.

- **The Challenge:** Every process needs its own memory space, and it must be prevented from corrupting other processes or the kernel itself.
- **Virtual Memory:** xv6 gives each process its own private **virtual address space**, which is a clean, continuous view of memory starting from address 0.
- **How it works in xv6:** The kernel uses the CPU's **page tables** to map these virtual addresses to actual physical addresses in RAM. This mapping is unique for each process, ensuring isolation.

****File System:**

This is about organizing data on a persistent storage device, like a disk.

- **The Goal:** Provide a structured way to store and retrieve information through files and directories.
- **How it works in xv6:** xv6 implements a simple, Unix-like file system using:
    - **Inodes:** Data structures that describe a file (its size, location on disk, etc.), but not its name.
    - **Data Blocks:** The actual blocks on the disk that hold the file's content.
    - **Directories:** Special files that contain a list of names and their corresponding inode numbers.

****System Calls:**

This is the interface between user programs and the kernel.

- **The Bridge:** A user program cannot directly call kernel functions. Instead, it must make a **system call** to request a service. This is the only legitimate way to cross the boundary from user mode to kernel mode.
- **How it works in xv6:** A program executes a special instruction (`int`) that triggers a **trap**. This trap forces the CPU to switch into kernel mode and jump to a pre-defined kernel function (a trap handler), which then identifies and executes the requested service.
- **Key xv6 System Calls:**
    - `fork()`: Creates a new process.
    - `exec()`: Loads and runs a new program.
    - `read()` / `write()`: Performs file input/output.
    - `sbrk()`: Requests more memory for a process.
    - `exit()`: Terminates the current process.

Putting It All Together: The Life of a Command

When you type `ls` in the xv6 shell, this is what happens:

1. The **shell** (a user program) calls the `fork()` system call to create a new child process.
2. The kernel's **process manager** duplicates the shell process.
3. The new child process calls the `exec("ls", ...)` system call.
4. The kernel's **memory manager** sets up a new address space and the **file system** loads the `ls` program from the disk into that memory.
5. The `ls` program runs. It uses the `open()`, `read()`, and `write()` system calls to interact with the **file system** and display the directory contents on the console.
6. While `ls` waits for data from the disk, the kernel's **scheduler** may perform a context switch to run another process.
7. Once finished, `ls` calls the `exit()` system call, and the kernel cleans up all its resources (memory, open files, etc.).

spinlocks

Why do we need locks

We need locks to safely manage access to **shared data** in multi-threaded applications.

Without locks, multiple threads could try to read and write to the same memory location simultaneously. This creates a **race condition**, where the final state of the data is unpredictable and often incorrect, leading to data corruption and crashes.

A lock acts like a key to a room, ensuring only one thread can access the shared data at a time. This principle is called **mutual exclusion**.

A thread "acquires" the lock, enters the "critical section" to modify the data, and then "releases" the lock, allowing another waiting thread to take its turn.

Spinlocks are a specific type of lock where waiting threads repeatedly check in a tight loop ("spin") until the lock becomes available. This is efficient for locks that are held for extremely short durations, as it avoids the overhead of putting a thread to sleep.

In essence, locks enforce order and guarantee that your program behaves predictably and reliably. ![[Screenshot 2025-08-22 at 2.58.30 PM.png]] ![[Screenshot 2025-08-22 at 2.55.27 PM.png]]![[Screenshot 2025-08-22 at 2.59.46 PM.png]]

Spinlock Fundamentals

- [00:07] **Spinlocks** are synchronization primitives used in operating systems to protect shared data from concurrent access.
- They are represented by a single word in memory, which can have two states:
    - **0:** The lock is **free** (unlocked or released).
    - **1:** The lock is **held** (acquired or locked).

- [00:54] In the **xv6 operating system**, the spinlock structure contains three fields:
    - `locked`: The current state of the lock (0 or 1).
    - `name`: A string for debugging purposes.
    - `cpu`: A pointer to the CPU core that currently holds the lock.

Core Spinlock Functions

- [01:38] There are four key functions for managing spinlocks:
    - **Initialize**: Sets the lock's name, the `cpu` field to null, and the `locked` value to 0.
    - **Acquire**: Attempts to obtain the lock.
    - **Release**: Releases the lock.
    - **Holding**: Checks if the current core holds the lock, used for error checking.

The "Acquire" Function: Challenges and Solutions

- [02:13] A simple implementation of the `acquire` function that first checks if the lock is free and then sets it to 1 is vulnerable to **race conditions** in a multi-core environment [02:55].
- [03:46] To address this, the **RISC-V architecture** provides the `amo.swap` (atomic memory operation swap) instruction. This instruction atomically swaps a value in a register with a value in memory, preventing any other instruction from interrupting the operation.
- [04:28] The `acquire` function uses a loop that repeatedly executes the `amo.swap` instruction, attempting to write a 1 to the lock's `locked` field. The loop continues until the value returned by `amo.swap` is 0, indicating that the lock was successfully acquired.

The "Release" Function

- [05:12] The `release` function is simpler than `acquire`. It sets the `locked` field back to 0. While a standard store operation is often sufficient, xv6 uses `amo.swap` for this as well.

Code Walkthrough (spinlock.c)

- **`init` function** [05:40]: This function initializes the spinlock by setting the `name`, setting the `cpu` to null, and the `locked` status to zero.
- **`acquire` function** [06:04]: This function uses `__sync_lock_test_and_set` (which compiles to `amo.swap`) inside a `while` loop. It also records which `cpu` is holding the lock [07:02], checks for any attempts to acquire the lock twice

[07:26], and uses a `__sync_synchronize` memory fence [07:46] to prevent the compiler from reordering memory operations.

- **`release` function** [09:06]: This function checks to make sure the current core is the one holding the lock, sets the `cpu` field to null, and then uses `__sync_lock_release` to set the `locked` field to zero. It also includes a `__sync_synchronize` [09:50].
- **`holding` function** [10:19]: This function checks to see if the lock is held and if the current CPU is the one holding it.

Spinlock Usage Principles

- [10:49] Spinlocks should be held for the shortest possible duration to avoid wasting CPU cycles.
- The thread holding the lock should not be put to sleep or be preempted.
- [11:34] Spinlocks protect shared data by enforcing "constraints" on that data.
- [11:46] The typical usage pattern is: `acquire_lock` -> `access_shared_data` (critical section) -> `release_lock`.

Deadlocks and Interrupts

- [13:17] A **deadlock** can occur if a thread holding a spinlock is interrupted, and the interrupt handler tries to acquire the same lock.
- [14:43] To prevent this, interrupts are disabled before acquiring a spinlock and re-enabled after releasing it. This ensures that the critical section is executed without interruption.

Handling Nested Calls and Interrupt Status

- [15:43] Simply disabling and enabling interrupts is not sufficient for nested `acquire`/`release` calls or when interrupts are already disabled.
- [16:41] A counter, `n_off`, is used to track nested calls.
- [17:39] A variable, `int_ena`, stores the interrupt status before the first `acquire` call.

`push_off` and `pop_off` Functions

- [18:05] These functions are used to manage the interrupt status during nested `acquire` and `release` calls:
    - **`push_off`** (called by `acquire`): Disables interrupts. If it is the first call (`n_off` is zero), it saves the current interrupt status in `int_ena` and then increments `n_off`.
    - **`pop_off`** (called by `release`): Decrements `n_off`. If `n_off` becomes zero and `int_ena` was true, it re-enables interrupts.

Why do we need locks

We need locks to safely manage access to **shared data** in multi-threaded applications.

Without locks, multiple threads could try to read and write to the same memory location simultaneously. This creates a **race condition**, where the final state of the data is unpredictable and often incorrect, leading to data corruption and crashes. ▯

A lock acts like a key to a room, ensuring only one thread can access the shared data at a time. This principle is called **mutual exclusion**.

A thread "acquires" the lock, enters the "critical section" to modify the data, and then "releases" the lock, allowing another waiting thread to take its turn.

Spinlocks are a specific type of lock where waiting threads repeatedly check in a tight loop ("spin") until the lock becomes available. This is efficient for locks that are held for extremely short durations, as it avoids the overhead of putting a thread to sleep.

In essence, locks enforce order and guarantee that your program behaves predictably and reliably.

Trap and context switching

Trap Handling Fundamentals

- [00:20] The video discusses **traps**, which are events that interrupt the normal flow of program execution. There are two main types:
    - **Exceptions**: Synchronous events caused by the instruction stream, such as system calls or illegal instructions.
    - **Interrupts**: Asynchronous events from external devices, like a timer or a network card.
    - ![[Screenshot 2025-08-21 at 11.36.09 PM.png]]![[Screenshot 2025-08-21 at 11.36.09 PM.png]]![[Screenshot 2025-08-21 at 11.37.38 PM.png]]![[Screenshot 2025-08-22 at 4.28.53 PM.png]]

Supervisor Mode Traps

- [01:03] The `stvec` **(Supervisor Trap Vector Base Address) register** holds the memory address of the trap handler code.
- [01:22] In xv6, there are two primary trap handlers: `kernelvec` for traps occurring in supervisor mode and `uservac` for traps from user mode.
- [01:43] The `sstatus` **(Supervisor Status) register** contains crucial bits for trap handling:
    - `sie`: Supervisor Interrupt Enable.
    - `spie`: Previous Supervisor Interrupt Enable (saves the value of `sie` during a trap).

    - A bit to record the previous execution mode (user or supervisor).

- **Trap Process**: [04:22] When a trap occurs, the hardware automatically performs several steps:
    1. [04:36] Saves the current program counter (PC) in the `sepc` (Supervisor Exception Program Counter) register.
    2. [04:46] Jumps to the trap handler by loading the address from `stvec` into the PC.
    3. [04:56] Records the cause of the trap in the `scause` register.
    4. [05:34] Saves the previous mode and interrupt status in `sstatus`.
    5. [05:51] Switches to supervisor mode and disables interrupts.
- **Returning from a Trap**: [06:20] The `sret` (supervisor return) instruction is used to exit the trap handler, which restores the PC, mode, and interrupt status.

What `sret` Does

1. Privilege Level Transition

```
# Before sret: CPU in supervisor mode (kernel mode)
sret
# After sret: CPU in user mode
```

`sret` switches from supervisor mode back to user mode, restoring the privilege level that was active before the trap occurred.

2. Program Counter Restoration

```
# sret jumps to the address stored in sepc register
# sepc was set by usertrapret() before calling userret
```

The CPU jumps to whatever address is stored in the `sepc` (Supervisor Exception Program Counter) register, which contains the user program address where execution should resume.

3. Interrupt/Exception Status Restoration

```
# sret restores interrupt enable state from sstatus register
# sstatus.SPIE bit → sstatus.SIE bit
# This re-enables interrupts in user mode if they were enabled before
```

Context Before `sret`

Just before `sret` executes, the system state is:

CPU State:

- **Mode**: Supervisor (kernel) mode
- **Page table**: User page table (switched back by `csrw satp, a0`)
- **Stack**: Still using kernel stack
- **Registers**: All user registers restored from trapframe

Key Registers:

- **`sepc`**: Contains user program counter (set by `usertrapret()`)
- **`sstatus`**: Contains user mode status (set by `usertrapret()`)
- **`satp`**: Contains user page table (already switched)

What Happens During `sret`

Hardware Actions:

1. **Set privilege mode to user** (clear supervisor bit)
2. **Jump to address in `sepc`** (resume user program)
3. **Restore interrupt state** from `sstatus.SPIE` → `sstatus.SIE`
4. **Update status registers** appropriately

Before and After `sret`:

```
BEFORE sret:
├── Mode: Supervisor (kernel)
├── PC: userret function
├── Registers: User values restored from trapframe
├── Stack: Kernel stack
├── Page table: User page table
└── sepc: User program counter to resume

AFTER sret:
├── Mode: User
├── PC: Value from sepc (user program)
├── Registers: User values (unchanged)
├── Stack: User stack (sp restored from trapframe)
├── Page table: User page table (unchanged)
└── sepc: Unchanged
```

Complete User Return Flow

The full sequence in `userret`:

```
userret:
    # 1. Switch to user page table
    csrw satp, a0                  # Load user page table
    sfence.vma zero, zero          # Flush TLB

    # 2. Restore all user registers from trapframe
    li a0, TRAPFRAME
    ld ra, 40(a0)                  # Restore user return address
    ld sp, 48(a0)                  # Restore user stack pointer
    # ... restore all other registers ...
    ld a0, 112(a0)                 # Restore a0 last
    # 3. Return to user mode and user program
    sret                           # ← THE MAGIC HAPPENS HERE!
```

What `usertrapret()` Set Up Before Calling `userret`:

```
void usertrapret(void) {
    // ...

    // Set up for sret instruction:
    w_sstatus(sstatus_bits);       // Configure user mode, interrupts
    w_sepc(p->trapframe->epc);     // Where user program should resume

    // Call assembly code that ends with sret
    uint64 trampoline_userret = TRAMPOLINE + (userret - trampoline);
    ((void (*)(uint64))trampoline_userret)(satp);
}
```

Why `sret` is Special

1. Atomic Transition

- **Cannot be interrupted** during the mode switch
- **Hardware guarantees** the transition completes safely

2. Privilege-Protected

- **Only available in supervisor mode**
- **User programs cannot execute `sret`**

3. Complete State Restoration

- **Restores execution exactly** where the user program left off
- **Maintains user program illusion** that nothing happened

Example: System Call Return

```
// User program makes system call
int fd = open("file.txt", O_RDONLY);
//                         ↑
//                    User program paused here by ecall

// After kernel processes system call:
// 1. usertrap() handles the system call
// 2. usertrapret() prepares for return
// 3. userret restores user state
// 4. sret jumps back to user program

printf("fd = %d\n", fd);  // ← User program resumes HERE
//                           sret jumped to this instruction
```

Summary

`sret` performs the final step in returning from kernel to user mode:

1. **Switches CPU from supervisor → user mode**
2. **Jumps to user program address** (stored in `sepc`)
3. **Restores user interrupt state**
4. **Completes the kernel → user transition atomically**

It's the **exit ramp** from kernel mode back to user space, restoring the user program's execution exactly where it was interrupted by the trap!

Context Switching

![[Screenshot 2025-08-21 at 11.55.33 PM.png]] ![[Screenshot 2025-08-21 at 11.57.56 PM.png]]

User and Kernel Modes

- [00:34] Code execution happens in two modes: **user mode** for applications and **kernel mode** for the operating system.

- [00:53] A **trap** is the mechanism that switches the CPU from user mode to kernel mode.

- [01:01] Traps can be triggered by:

  - **Interrupts**: An external device, like a timer, needs attention.

  - **System Calls**: The user program intentionally requests a service from the kernel.

  - **Exceptions**: The user program performs an illegal operation, like dividing by zero.

- [0.01:24] The **`sret` (system return)** instruction is used to switch from kernel mode back to user mode.

- [01:50] During a trap, the state of the user thread (its registers) is saved. During an `sret`, this state is restored, allowing the thread to resume exactly where it left off.

Time Slices and Scheduling

- [02:15] A user thread runs for a period of time called a **time slice**.

- [02:22] A **timer interrupt** signals the end of a time slice, causing a trap into the kernel.

- [02:52] The kernel's **scheduler** then decides which thread gets to run next. A system return (`sret`) starts a time slice for a chosen thread, and a trap ends it.

Trap Handling Process

1. [05:11] When a trap occurs, the user thread's registers and program counter are saved.

2. [05:33] The kernel identifies the cause of the trap (interrupt, system call, etc.) and runs the appropriate handler code.

3. [05:18] After handling the trap, the kernel restores the user registers and executes `sret` to return to user code.

Multi-Core Systems

- [06:10] On a multi-core system, different cores can run different threads simultaneously.
- [08:26] A process's state (its saved registers) is stored in shared memory, allowing it to be paused on one core and resumed on another.
- [08:49] **Locks** are crucial to protect this shared memory from being accessed by multiple cores at the same time, which would cause a race condition.

Context Switching Mechanisms

- [11:26] A context switch isn't just a simple trap and return. It often involves switching from the context of a user process's kernel thread to a separate scheduler thread, and then to the kernel thread of the next process to run.
- [12:45] The assembly function `swtch` is used for these **kernel-to-kernel** context switches. It saves and restores only the registers that the kernel threads themselves use, which is a smaller set than the full user register state saved during a trap.

Trap Handling

Trap Handling Overview

- [00:00] The following section the complete lifecycle of a **trap** in the xv6 operating system, from the moment it occurs in user mode to the final return.
- [00:42] Traps are caused by either **asynchronous interrupts** (like the timer or I/O devices) or **synchronous exceptions** (like system calls or program errors).

Key Data Structures

- **Trap Frame** [08:09]: A structure that holds all the saved state of a user process during a trap, including its registers, program counter, and kernel-related pointers.
- **CPU Structure** [09:35]: An array (one entry per CPU core) that keeps track of the currently running process on that core.
- **Proc Structure** [12:01]: The "Process Control Block". This large structure contains all the essential information about a single process, such as its state (running, sleeping, etc.), process ID, memory space, open files, and a pointer to its trap frame.

- ![[Screenshot 2025-08-22 at 7.46.45 PM.png]]![[Screenshot 2025-08-22 at 7.47.34 PM.png]]

  ### The Trap Handling Process

1. **Hardware Actions** [00:57]: When a trap occurs, the CPU hardware automatically:
   - Disables interrupts.
   - Switches from user mode to supervisor mode.
   - Saves the user's program counter (PC) and the cause of the trap.
   - Jumps to the trap handler function, `usertvec`.
2. **The Trampoline Page** [01:40]: Before the main kernel C code is executed, a special "trampoline" page is used. This page contains assembly code that saves all the user's general-purpose registers into a data structure called a **trap frame**.
3. **Kernel Entry (`usertrap`)** [02:12]: After the user state is saved, the code switches to the kernel's virtual address space and jumps to the C function `usertrap`.
4. **`usertrap` Logic** [03:05]: This C function is the main hub for trap handling. It examines the cause of the trap and decides what to do:
   - **System Call**: Executes the requested system call.
   - **Device Interrupt**: Calls the appropriate device driver.
   - **Timer Interrupt**: Yields the CPU to the scheduler, allowing another process to run.
   - **Program Exception**: If the process caused an error or was marked as "killed", the kernel terminates it.
5. **Returning to User Mode (`usertrapret`)** [05:43]: When the kernel has finished handling the trap, the `usertrapret` function begins the process of returning to user mode. It prepares the necessary information in the trap frame for the final return.
6. **User Return Assembly** [06:43]: The execution jumps back to the trampoline page, which runs assembly code to:
   - Switch back to the user's address space.
   - Restore all the user's registers from the trap frame.
   - Execute the `sret` instruction to switch the CPU back to user mode and re-enable interrupts, resuming the user program.

![[Screenshot 2025-08-22 at 10.30.14 PM.png]] ---![[Screenshot 2025-08-22 at 10.31.48 PM.png]]![[Screenshot 2025-08-22 at 10.32.56 PM.png]]

Scheduling

![[Screenshot 2025-08-21 at 11.55.33 PM.png]] ![[Screenshot 2025-08-21 at 11.57.56 PM.png]] ![[Screenshot 2025-08-22 at 8.32.30 PM.png]] ![[Screenshot 2025-08-22 at 9.25.13 PM.png]]

Scheduling and Context Switching in xv6

This video provides a deep dive into the **scheduling mechanism** of the xv6 operating system. It explains how the kernel switches between different processes to create the illusion of simultaneous execution. The core components discussed are the `yield`, `sched`, and `scheduler` functions, and the `swtch.S` assembly code.

Helper Functions

Before diving into the main scheduling logic, the video introduces a few essential helper functions that identify the current execution context:

- **`cpuid`**: Returns the ID of the CPU core the code is currently running on.
- **`mycpu`**: Retrieves the `cpu` structure for the current core, which holds core-specific information.
- **`myproc`**: Gets a pointer to the `proc` structure of the process currently running on that core.

The Scheduling Flow: From Trap to Switch

The entire process of switching from a user process to the scheduler is a carefully choreographed sequence:

1. **A Trap Occurs**: A **timer interrupt** fires, causing a trap into the kernel. From this point until the user process is resumed, interrupts remain disabled to prevent race conditions.
2. **`yield()` is Called**: The trap handler eventually calls the `yield()` function.
3. **`sched()` Prepares the Switch**:
   - `yield()` calls `sched()`, which acts as a staging area for the context switch.
   - It acquires a **lock** on the current process's `proc` structure.
   - It changes the process's state from **`RUNNING`** to **`RUNNABLE`**, indicating it's ready to run but not currently on a CPU.
   - Crucially, it then calls `swtch()`.

The `scheduler()` Function: The Heart of Scheduling

The `scheduler()` function runs as a dedicated kernel thread on each CPU core, and its job is simple: find and run a process.

- **Infinite Loop**: It runs in an infinite loop, constantly scanning the process table for a process with the state **`RUNNABLE`**.
- **Finding a Process**: When it finds a runnable process:
  1. It acquires a lock on that process.
  2. It changes the process's state to **`RUNNING`**.
  3. It updates the current CPU's state to point to this new process.
  4. It calls **`swtch()`** to switch the context *to* the new process's kernel thread.
- **No Runnable Processes**: If no processes are ready to run, the scheduler temporarily re-enables interrupts. This is a critical step to prevent deadlock, as it allows device interrupts to come in and potentially wake up a sleeping process, making it runnable.

`swtch.S`: The Assembly Magic

The `swtch` function is where the low-level context switch actually happens. It's written in assembly language because it needs to directly manipulate CPU registers.

- **Two Contexts**: It takes two arguments: a pointer to the "old" context to save and a pointer to the "new" context to load.
- **Save and Restore**:
  - It **saves** the callee-saved registers (like the stack pointer and return address) of the current thread into its `context` structure.
  - It **restores** the registers from the new thread's `context` structure.
- **The "Return"**: The magic of `swtch` is that when it "returns," it doesn't return to where it was called. Instead, it returns to the instruction that the *new* thread was about to execute, effectively completing the context switch.

This switch happens in two key places:

1. From a process's kernel thread (`sched`) **to** the `scheduler` thread.
2. From the `scheduler` thread **to** a new process's kernel thread.

Code walkthrough

Timer Interrupt During User Mode Execution

Let's say we have this user program running:

```c
// Process A running in user mode
int main() {
    int sum = 0;
    for (int i = 0; i < 1000000; i++) {
        sum += i;  // ← TIMER INTERRUPT HAPPENS HERE
    }
    return 0;
}
```

Complete Flow: Timer Interrupt → Process Switch

Step 1: Timer Interrupt Occurs

```
User Process A → Timer Hardware → RISC-V Hardware Actions:
1. Save current PC in sepc register (points to "sum += i" instruction)
2. Switch to supervisor mode
3. Jump to uservec (address in stvec register)
```

Step 2: Save Process A's Context (`uservec` in trampoline.S)

```
uservec:
    # Save ALL of Process A's registers in its trapframe
    li a0, TRAPFRAME
    sd ra, 40(a0)       # Save Process A's return address
    sd sp, 48(a0)       # Save Process A's stack pointer
    sd t0, 72(a0)       # Save Process A's temporary registers
    # ... save all 32 registers of Process A ...

    # Switch to Process A's kernel stack and page table
    ld sp, 8(a0)        # Load Process A's kernel stack
    ld t1, 0(a0)        # Load kernel page table
    csrw satp, t1       # Switch to kernel page table

    # Jump to C trap handler
    jr t0               # Jump to usertrap()
```

Step 3: Handle Timer in `usertrap()`

```
uint64 usertrap(void) {
    struct proc *p = myproc();  // p = Process A

    // Save the interrupted PC (where Process A was interrupted)
    p->trapframe->epc = r_sepc();  // Save "sum += i" instruction address

    // Determine interrupt type
    int which_dev = devintr();

    if(which_dev == 2) {  // Timer interrupt
        yield();          // ← KEY: This switches to scheduler!
    }

    // If we get here, Process A was rescheduled and resumed
    prepare_return();     // Prepare to return to Process A
}
```

Step 4: `yield()` Switches to Scheduler

```c
void yield(void) {
    struct proc *p = myproc();  // p = Process A
    acquire(&p->lock);

    p->state = RUNNABLE;        // Mark Process A as runnable (not running)
    sched();                    // ← Switch to scheduler

    // When Process A is rescheduled later, execution resumes HERE
    release(&p->lock);
}
```

Step 5: `sched()` Context Switch

```c
void sched(void) {
    struct proc *p = myproc();  // p = Process A

    // Save Process A's kernel context (registers, stack pointer)
    swtch(&p->context, &mycpu()->context);  // ← Context switch happens here

    // When Process A is rescheduled, it will resume HERE
}
```

Step 6: `swtch()` Assembly Magic

```
# swtch(&p->context, &mycpu()->context)
# a0 = &Process A's context, a1 = &scheduler context

swtch:
    # Save Process A's kernel registers in p->context
    sd ra, 0(a0)        # Save Process A's kernel return address
    sd sp, 8(a0)        # Save Process A's kernel stack pointer
    sd s0, 16(a0)       # Save Process A's saved registers
    # ... save all callee-saved registers of Process A ...

    # Load scheduler's registers from mycpu()->context
    ld ra, 0(a1)        # Load scheduler's return address
    ld sp, 8(a1)        # Load scheduler's stack pointer
    ld s0, 16(a1)       # Load scheduler's saved registers
    # ... load all scheduler registers ...

    ret                 # Return to scheduler (ra points to scheduler code)
```

Step 7: Scheduler Runs

```c
void scheduler(void) {
    struct cpu *c = mycpu();
    c->proc = 0;  // No process running
    for(;;) {
        // Look for a RUNNABLE process
        for(p = proc; p < &proc[NPROC]; p++) {
            acquire(&p->lock);
            if(p->state == RUNNABLE) {  // Found Process B (or A later)
                p->state = RUNNING;
                c->proc = p;

                // Switch to Process B
                swtch(&c->context, &p->context);  // ← Switch to Process B

                c->proc = 0;
            }
            release(&p->lock);
        }
    }
}
```

Step 8: Resume Different Process (Process B)

When scheduler finds Process B and calls `swtch(&c->context, &p->context)`:

```
swtch:
    # Save scheduler registers in c->context
    # Load Process B's registers from p->context

    ret  # Return to where Process B was interrupted (could be yield, sleep, etc.)
```

Key Points About Context Switching

1. Two Levels of Context

- **User Context**: Saved in trapframe (user registers, user PC, user stack)
- **Kernel Context**: Saved in proc->context (kernel registers, kernel stack)

2. Process A's Complete State After Timer

```
struct proc *processA = ...;

// User context (saved in trapframe)
processA->trapframe->epc = 0x1000;  // Address of "sum += i"
processA->trapframe->sp = 0x2000;   // User stack pointer
processA->trapframe->a0 = 5;        // User register values
// ... all 32 user registers saved ...

// Kernel context (saved in context)
processA->context.ra = yield+offset;  // Where to resume in yield()
processA->context.sp = 0x3000;        // Kernel stack pointer
// ... callee-saved kernel registers ...

processA->state = RUNNABLE;           // Ready to run again
```

3. When Process A Runs Again

Later, when the scheduler picks Process A again:

1. **Scheduler calls `swtch()`** → Resume in `yield()`
2. **`yield()` returns** → Resume in `sched()`
3. **`sched()` returns** → Resume in `usertrap()`
4. **`usertrap()` calls `prepare_return()`** → Return to user space
5. **`userret` in trampoline** → Restore user registers from trapframe
6. **`sret` instruction** → Return to "sum += i" instruction
7. **Process A continues** exactly where it was interrupted

Summary

When a timer interrupt causes a process switch:

1. **User context saved** in trapframe (where user program was)
2. **Kernel context saved** in proc->context (where kernel was)
3. **Scheduler runs** and picks a different process
4. **New process resumes** from where it was last interrupted
5. **Original process waits** in RUNNABLE state until rescheduled

The magic is that **both user and kernel execution contexts are preserved**, allowing processes to be interrupted and resumed seamlessly at any point in their execution!

Locking and Multi-Core Safety

The video emphasizes the importance of locks throughout this process. Locks on the proc structures ensure that a process's state isn't corrupted if another CPU core tries to interact with it during a context switch. A key point is that a lock can be acquired on one core (before a swtch) and released on a completely different core (after that process is scheduled again).

System calls

![[Screenshot 2025-08-22 at 8.26.26 PM.png]]

The User/Kernel Divide

The kernel operates with full hardware privileges, enabling it to manage memory, control I/O devices, and orchestrate program execution. User processes, by contrast, are confined to their own memory space and are forbidden from performing privileged operations directly. This raises a critical question: How can a user program, such as the shell, perform a task like opening a file when it lacks the necessary privileges? The answer lies in a controlled transition known as a **system call**, which is a type of trap.

Page tables and virtual addresses if time

Address Translation and the SATP Register

![[Screenshot 2025-08-22 at 12.03.59 AM.png]]

- [00:01] This video explains **address translation** and the **page table architecture** in the **RISC-V processor**, using the **xv6 kernel** as an example.
- [00:26] The **SATP (Supervisor Address Translation and Protection) register** is a key component. It stores the physical memory address of the **root of the current page table**.
- [01:12] Virtual address translation is always active in **supervisor and user modes** but is turned off in **machine mode**.

Page Tables in RISC-V

- **Kernel Page Table:** [01:29] The xv6 kernel maintains a single, shared page table that maps the kernel's virtual addresses directly to physical addresses. This allows the kernel to have a simple, one-to-one mapping for all of physical memory, including memory-mapped I/O devices [01:53].
- **User Mode Page Tables:** [02:00] Each user-level process has its own separate page table, providing isolation and protection.
- **sv39 Paging Scheme:** [02:09] RISC-V supports several paging schemes (sv32, sv39, sv48). xv6 uses **sv39**, which is a three-level page table architecture for 64-bit processors.

Translation Lookaside Buffers (TLBs)

- [03:17] To speed up address translation, processors use a **TLB (Translation Lookaside Buffer)**, which is a cache for recently used page table entries.
- [03:44] When the page table is changed (i.e., when the SATP register is updated), the TLB must be flushed to ensure that the processor uses the new translations. This is done with the `sfence.vma` instruction.

Virtual Addresses in sv39

- [04:19] An sv39 virtual address is **39 bits** long.
- [04:30] It is divided into two parts:
    - A **12-bit offset**, which is used to select a byte within a 4KB page.
    - A **27-bit index**, which is further divided into three 9-bit fields, one for each level of the page table.

Page Table Entry (PTE)

- [05:03] Each entry in a page table is called a **Page Table Entry (PTE)**.
- A PTE contains several important bits:
    - **Permission bits**: Control whether the page can be read, written, or executed.

    - **U-bit**: Determines if the page is accessible from user mode.
    - **V-bit**: Indicates whether the PTE is valid.
    - **Physical Page Number (PPN)**: [05:41] The physical address of the page in memory.

Page Table Structure

- [06:21] The SATP register points to the root of a **three-level tree** structure.
- [06:30] Each node in this tree is a 4KB page in physical memory.
- [06:54] Each interior node of the page table contains **512 entries**, with each entry being 64 bits in size.
- [07:32] The hardware uses the three 9-bit index fields from the virtual address to traverse this tree and find the correct PTE.
- [05:51] The **physical page number** from the PTE is then combined with the **12-bit offset** from the virtual address to form the final **56-bit physical address**.