Xv6 Traps, System Calls, and Scheduling

CMPU 334 – Operating Systems
Jason Waterman

How to Efficiently Virtualize the CPU with Control?

- The OS needs to share the physical CPU by **time sharing**
- Issues
  - **Performance**: How can we implement virtualization without adding excessive overhead to the system?
  - **Control**: How can we run processes efficiently while retaining control over the CPU?

Direct Execution

Just run the program directly on the CPU

OS	**Program**
1. Create entry for process list	
2. Allocate memory for program	
3. Load program into memory	
4. Set up stack with `argc` / `argv`	
5. Clear registers	
6. Execute call `main()`	
	7. Run `main()`
	8. Execute `return` from `main()`
9. Free memory of process	
10. Remove from process list	

Without *limits* on running programs,
the OS wouldn't be in control of anything and
thus would be "just a library"

Problem 1: Restricted Operation

- What if a process wishes to perform some kind of restricted operation such as …
    - Issuing an I/O request to a disk
    - Gaining access to more system resources such as CPU or memory

- **Solution**: Using protected control transfer
    - User mode: Applications do not have full access to hardware resources
    - Kernel mode: The OS has access to the full resources of the machine

System Call

- Allow the kernel to carefully expose certain key pieces of functionality to user program, such as …
  - Accessing the file system
  - Creating and destroying processes
  - Communicating with other processes
  - Allocating more memory
- **Trap** instruction
  - Jump into the kernel
  - Raise the privilege level to kernel mode
- **Return-from-trap** instruction
  - Return into the calling user program
  - Reduce the privilege level back to user mode

Traps

- A trap is a generic term for an event that forces the CPU to stop normal execution and transfer control to special kernel code.
- Three main types of traps
  - System call: A user program deliberately requests a kernel service using the ecall instruction
  - Exception: An instruction performs an illegal action, like accessing an invalid memory address
  - Device Interrupt: Hardware, like a disk controller, signals that it needs attention
- The goal is for traps to be transparent, allowing the interrupted code to resume later as if nothing happened.

Trap Handling Flow

- Event Occurs: A trap forces a transfer of control into the kernel
- Save State: The kernel saves registers and other machine state
- Execute Handler: The kernel runs the appropriate code to handle the event (e.g., a device driver)
- Restore & Return: The kernel restores the saved state and returns from the trap
- Resume: The original code continues where it left off

- In xv6, all traps are handled entirely within the kernel

RISC-V Hardware for Traps

- The RISC-V CPU uses a set of special, supervisor-only control registers to manage traps.
  - stvec: The kernel writes the address of its trap handler code here. The CPU jumps to this address when a trap occurs
  - sepc: When a trap happens, the CPU saves the current program counter (pc) here. The sret instruction later restores it
  - scause: The CPU puts a number here to indicate the cause of the trap
  - sscratch: A special register the kernel handler uses to save registers without corrupting user data
  - sstatus: Contains bits that control interrupts (SIE) and track whether the trap came from user or supervisor mode (SPP)

What the CPU Does During a Trap

- When a trap is triggered, the RISC-V hardware automatically performs these steps:
  - Disables further interrupts
  - Copies the program counter (pc) to sepc
  - Saves the current mode (user/supervisor) in sstatus
  - Sets scause with the reason for the trap
  - Switches the CPU to supervisor mode
  - Jumps to the address held in the stvec register

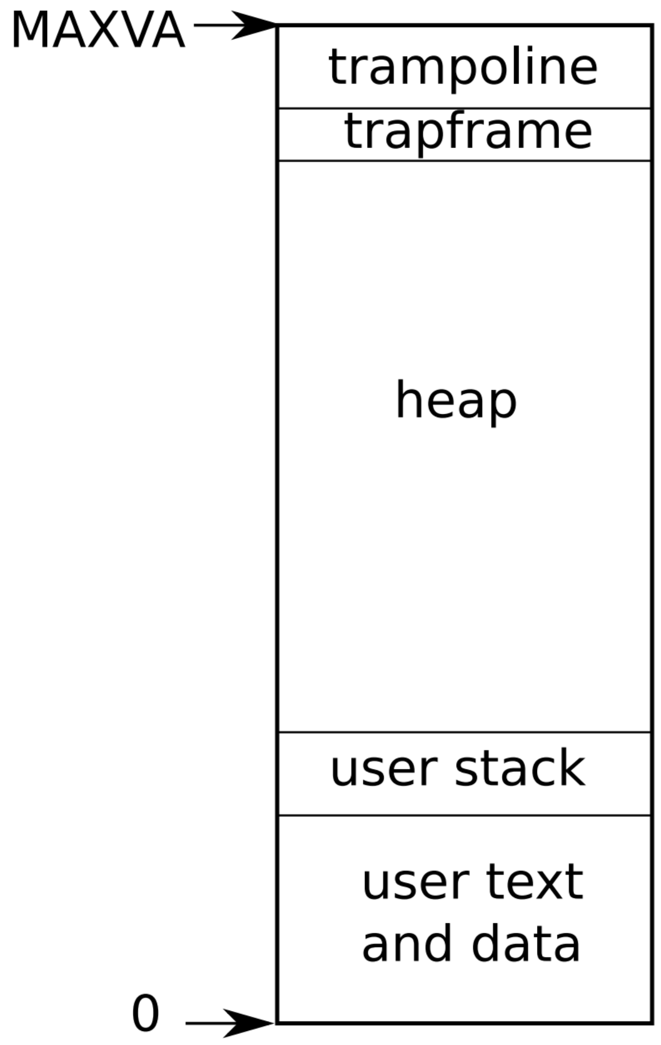
- Notably, the hardware **does not** switch page tables or save general-purpose registers; this is the kernel's responsibility

Userspace Trap Challenges

- A major constraint in trap handling is that the RISC-V CPU does not switch page tables when a trap occurs
  - **The Problem**: The CPU is still using the *user's page table* when the trap handler begins executing
  - **The Requirement**: Therefore, the handler's address (stvec) must be mapped in the user's address space
  - **The Complication**: The handler also needs to switch to the *kernel's page table* and continue running. This means the handler's address must *also* be mapped in the kernel's address space

Xv6 Trampoline Page

- xv6 solves this challenge with a **trampoline page**.
  - It contains the initial trap handler code, uservec
  - It is mapped at the **same high virtual address** in every user's page table *and* in the kernel's page table
  - This mapping allows the trap handler to start executing, switch page tables from user to kernel, and continue running without interruption



uservec and usertrap Flow

- **uservec (Assembly)** entry point
  - Uses the sscratch register to temporarily save one register, making it available for use
  - It saves all user registers to a pre-allocated trapframe page
  - It retrieves kernel information (like the kernel page table address) from the trapframe
  - Finally, it switches the page table to the kernel's and jumps to the C function usertrap
- **usertrap (C)**: This C function takes over
  - It determines the cause of the trap (system call, device interrupt, etc.)
  - It calls the appropriate handler and then prepares to return to user space

Traps from Kernel Space

- xv6 handles traps that occur in kernel mode differently
  - When entering the kernel, usertrap changes stvec to point to a different handler: kernelvec
  - kernelvec can assume it's already using the kernel's page table and a valid kernel stack
  - It saves all registers on the current kernel thread's stack and jumps to the kerneltrap C function
  - kerneltrap is simple: it is only designed to handle device interrupts. Any other kind of trap in kernel mode must be a kernel bug, which causes a system panic()

Switching Between Processes

- xv6 switches from one process to another in two main situations
- **Voluntary Switches:** Occur when a process makes a system call that must wait for an event, such as read or wait
- **Involuntary Switches:** Periodically forced by the OS to handle processes that compute for long periods without blocking
  - **A timer interrupt**
    - During the boot sequence, the OS start the <u>timer</u>
    - The timer <u>raise an interrupt</u> every so many milliseconds
    - When the interrupt is raised:
      - The currently running process is halted
      - A pre-configured interrupt handler in the OS runs

Multiplexing Challenges

- **The Switch Mechanism:** How to save the state of one process and restore another's
  - This involves low-level register manipulation that cannot be expressed in C
- **Transparency:** How to force switches on long-running processes without their knowledge
  - xv6 uses hardware timer interrupts for this
- **Concurrency:** How to prevent multiple CPUs from trying to run the same process simultaneously
  - Requires a robust locking plan
- **Process Identity:** How does a CPU know which process it is currently executing so system calls affect the correct kernel state?

Context Switch Overview

- xv6 does not switch directly from one process to another
    - All switches go through a central **scheduler thread** on each CPU
- Steps
    - A process traps into the kernel
    - The kernel thread switches to the CPU's scheduler thread
    - The scheduler thread picks a new process to run
    - The scheduler thread switches to the new process's kernel thread
    - The new kernel thread returns to user space

swtch()

- The low-level context switch is performed by the swtch() function
  - Saves the current thread's CPU registers in its struct context
  - Restores the registers of the next thread from its struct context

- Thread state
  - Memory and registers
  - Memory doesn't need to be swapped because each thread has its own memory regions (not a shared resource)
  - The single set of CPU registers must be shared, so they are saved and restored

The scheduler

- Each CPU has its own dedicated scheduler thread that runs the scheduler() function
  - The scheduler loops through the process table looking for a runnable process (p->state == RUNNABLE)
  - Once found, it sets the process state to RUNNING
  - Calls swtch() to start running the chosen process
- A process yields the CPU by calling sched(), which calls swtch() to save its context and switch to the scheduler
- The scheduler implements Round Robin scheduling
  - The scheduler implements a simple policy that runs each process in turn, cycling through the process table

Locking

- xv6 uses an unusual locking convention: p->lock is held across the call to swtch
- The thread calling swtch acquires the lock, but the thread that resumes execution after the switch is the one that releases it
- This ensures that a process's state (p->state) and its saved registers (p->context) are updated atomically
    - releasing the lock earlier could cause another CPU to run the process with partially saved registers