CS350: Operating Systems

Lecture 6: System Calls and Interrupts

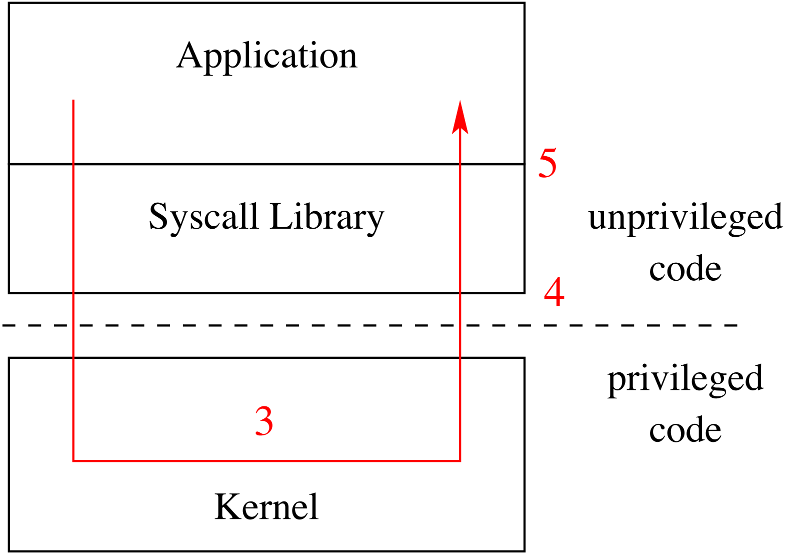
Ali Mashtizadeh

University of Waterloo

Outline

1. Kernel API
2. Calling Conventions
3. System Calls
4. Switching Threads/Processes

System Software Stack



System Call Interface

System Calls: Application programmer interface (API) that programmers use to interact with the operating system.

- Processes invoke system calls
- Examples: `fork()`, `waitpid()`, `open()`, `close()`, ...
- System call interface can have complex calls
  - `sysctl()` Exposes operating system configuration
  - `ioctl()` Controlling devices
- Need a mechanism to safely enter and exit the kernel
  - Applications don't call kernel functions directly!
  - Remember: kernels provide protection

Privilege Modes

- Hardware provides multiple protection modes
- At least two modes:
  - *Kernel Mode* or *Privledged Mode* – Operating System
  - *User Mode* – Applications
- Kernel Mode can access privileged CPU features
  - Access all restricted CPU features
  - Enable/disable interrupts, setup interrupt handlers
  - Control system call interface
  - Modify the TLB (virtual memory ... future lecture)
- Allows kernel to protect itself and isolate processes
  - Processes cannot read/write kernel memory
  - Processes cannot directly call kernel functions

Mode Transitions

- Kernel Mode can only be entered through well defined entry points
- Two classes of entry points provided by the processor:
- *Interrupts*
  - Interrupts are generated by devices to signal needing attention
  - E.g. Keyboard input is ready
  - More on this during our IO lecture!
- *Exceptions*:
  - Exceptions are caused by processor
  - E.g. Divide by zero, page faults, internal CPU errors
- Interrupts and exceptions cause hardware to transfer control to the *interrupt/exception handler*, a fixed entry point in the kernel.

Interrupts

- Interrupt are raised by devices
- *Interrupt handler* is a function in the kernel that services a device request
- Interrupt Process:
  - Device signals the processor through a physical pin or bus message
  - Processor interrupts the current program
  - Processor begins executing the interrupt handler in privileged mode
- Most interrupts can be disabled, but not all
  - Non-maskable interrupts (NMI) is for urgent system requests

Exceptions

- Exceptions (or faults) are conditions encountered during execution of a program
  - Exceptions are due to multiple reasons:
  - Program Errors: Divide-by-zero, Illegal instructions
  - Operating System Requests: Page faults
  - Hardware Errors: System check (bad memory or internal CPU failures)
- CPU handles exceptions similar to interrupts
  - Processor stops at the instruction that triggered the exception (usually)
  - Control is transferred to a fixed location where the exception handler is located in privledged mode
- System calls are a class of exceptions!

MIPS Exception Vectors

- Interrupts, exceptions and system calls use the same mechanism
- Some processors use a special path for system calls for performance (e.g., x86)

```
EX_IRQ 0 /* Interrupt */
EX_MOD 1 /* TLB Modify (write to read-only page) */
EX_TLBL 2 /* TLB miss on load */
EX_TLBS 3 /* TLB miss on store */
EX_ADEL 4 /* Address error on load */
EX_ADES 5 /* Address error on store */
EX_IBE  6 /* Bus error on instruction fetch */
EX_DBE  7 /* Bus error on data load or store */
EX_SYS  8 /* Syscall */
EX_BP   9 /* Breakpoint */
EX_RI  10 /* Illegal instruction */
EX_CPU 11 /* Coprocessor unusable */
EX_OVF 12 /* Arithmetic overflow */
```

System Calls

- System calls are performed by triggering the `EX_SYS` exception:

1. Application loads the arguments into CPU registers
2. Load the system call number into register `$v0`
3. Executes `syscall` instruction to trigger `EX_SYS` exception
4. Kernel processes the system call through the exception handler
5. Returns to userspace using `rfe`, return from exception instruction

- Many processors include similar instructions (e.g., `syscall` in x86)

Hardware Handling in MIPS R3000 (Sys/161)

- Exception handlers in MIPS R3000 are at fixed locations

- Processor jumps to these addresses whenever an exception is encountered
  - `0x8000_0000` User TLB Handler (virtual memory)
  - `0x8000_0080` General Exception Handler
- TLB exceptions are frequent
  - Handler is usually hand optimized assembly, unlike general exceptions

- Remember that `0x8000_0000`–`0x9FFF_FFFF`:
  - Mapped to the first 512MBs of physical memory
  - Where the OS resides

Hardware Handling: the MIPS Coprocessor

- Kernel accesses exception and processor state through the MIPS coprocessor
  - MIPS CP0: system control coprocessor
  - MIPS CP1 floating point coprocessor

- System Control Coprocessor (CP0) contains exception handling information
  - Use the mfc0/mtc0 (Move from/to co-processor 0) instructions
  - `c0_status`: CPU status include kerner/user mode flag
  - `c0_cause`: Cause of the exception
  - `c0_epc`: Program counter (PC) where the exception occurred
  - `c0_vaddr`: Virtual address associated with the fault
  - `c0_context`: Used by OS/161 to store the CPU number

System Call Operation Details

- Application calls into the C library (e.g., calls `write()`)
- Library executes the `syscall` instruction
- Kernel exception handler `0x8000_0080` runs
  - Switch to kernel stack
  - Create a *trapframe* which contains the program state
  - Determine the type of exception
  - Determine the type of system call
  - Run the function in the kernel (e.g., `sys_write()`)
  - Restore application state from the trap frame
  - Return from exception (`rfe` instruction)
- Library wrapper function returns to the application

Outline

1. Kernel API
2. Calling Conventions
3. System Calls
4. Switching Threads/Processes

How are values passed?

- Application Binary Interface (ABI) defines the contract between functions an application and system calls.
- Operating Systems and Compilers must obey these rules referred to as the *calling convention*
- MIPS + OS/161 Calling Convention
  - System call number in `v0`
  - First four arguments in `a0, a1, a2, a3`
  - Remaining arguments passed on the stack
  - Success/fail in `a3` and return value/error code in `v0`

System Call Numbering

- System calls numbers defined in kern/include/kern/syscall.h

```
#define SYS_fork  0
#define SYS_vfork 1
#define SYS_execv 2
#define SYS__exit 3
#define SYS_waitpid 4
#define SYS_getpid 5
...
```

MIPS Calling Conventions

- *Caller-saved registers* are saved before calling another function
  - `$t0`–`$t9`: Temporary registers
  - `$a0`–`$a3`: Argument registers
  - `$v0`–`$v1`: Return values
- *Callee-saved registers* are saved inside the function
  - `$s0`–`$s7`: Saved registers
  - `$ra`: Return address

- Instructions:
  - `jal`: Jump and link – Call function and save return address in `$ra`
  - `jr $ra`: Jump Register – Return from function

Functions in MIPS

- Review MIPS function calls
- Functions are called with the `jal` instruction
- `jal`: Jump-and-link, calls a function and saves the return address in `$ra`

```
foo:
        li $a0, 1

        /* Save caller-save registers */

        jal bar /* Call bar */
        nop /* Delay slot */

        /* Restore registers */

        jr $ra /* Return */
        nop /* Delay slot */
```

Functions in MIPS Continued

- Simple functions may not need to save any registers
- We save callee-saved registers if needed for performance

```
int bar(int a) {
        return 41 + a;
}

bar:
        li $v0, 41
        add $v0, $v0, $a0

        jr $ra
        nop /* Delay slot */
```

Where are registers saved?

- Registers are saved in memory in the per-thread stack
- A *stack frame* is all the saved registers and local variables that must be saved within a single function
- Our stack is made up of an array of stack frames

```
/* Push stack element */
      subi $sp, $sp, 8
      sw $t1, 4($sp)
      sw $t2, 0($sp)

/* Pop stack element */
      lw $t1, 4($sp)
      lw $t2, 0($sp)
      addi $sp, $sp, 8
```

Outline

1. Kernel API
2. Calling Conventions
3. System Calls
4. Switching Threads/Processes

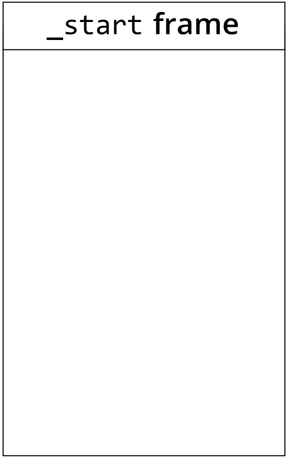
Execution Contexts

Execution Context: The environment where functions execute including their arguments, local variables, memory.

- Context is a unique set of CPU registers and a stack pointer
- Multiple execution contexts:
  - *Application Context*: Application threads
  - *Kernel Context*: Kernel threads, software interrupts, etc
  - *Interrupt Context*: Interrupt handler
- Kernel and Interrupts usually the same context
- Context transitions:
  - *Context switch*: a transitions between contexts
  - *Thread Switch*: a transition between threads (in OS/161 between kernel contexts)

Application Stack

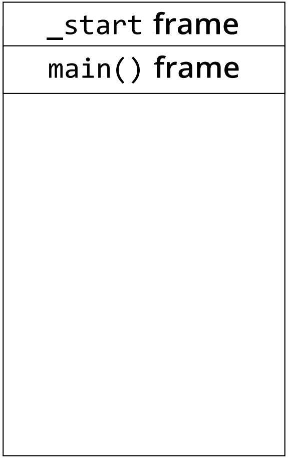
- Stack made of up *frames* containing locals, arguments, and spilled registers
- Programs begin execution at `_start`



User Stack

Application Stack

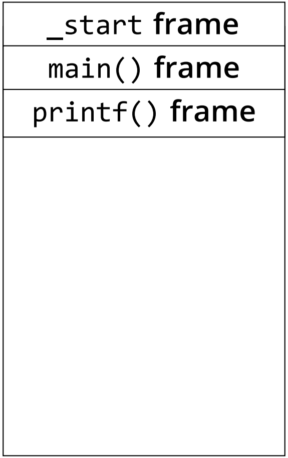
- Stack made of up *frames* containing locals, arguments, and spilled registers
- Programs begin execution at `_start`



User Stack

Application Stack

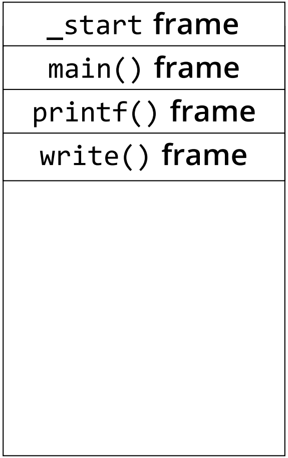
- Stack made of up *frames* containing locals, arguments, and spilled registers
- Programs begin execution at `_start`



User Stack

Application Stack

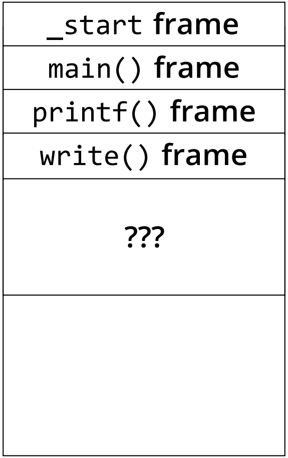
- Stack made of up *frames* containing locals, arguments, and spilled registers
- Programs begin execution at `_start`



User Stack

Application Stack

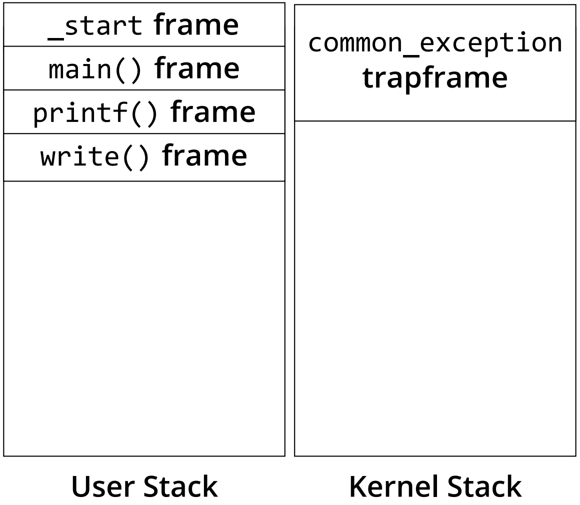
- Stack made of up *frames* containing locals, arguments, and spilled registers
- Programs begin execution at `_start`



User Stack

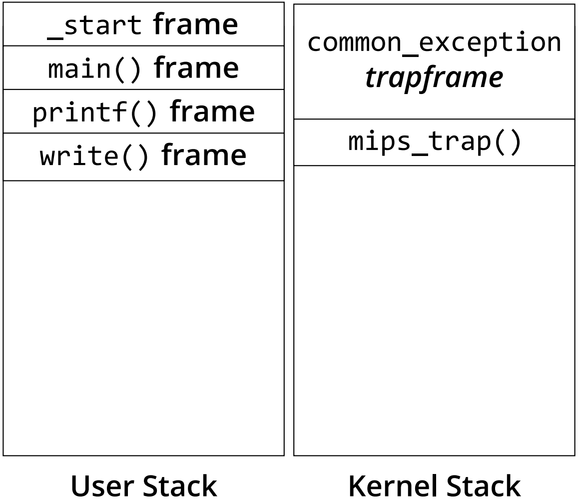
Context Switch: User to Kernel

- *trapframe*: Saves the application context
- `syscall` instruction triggers the exception handler



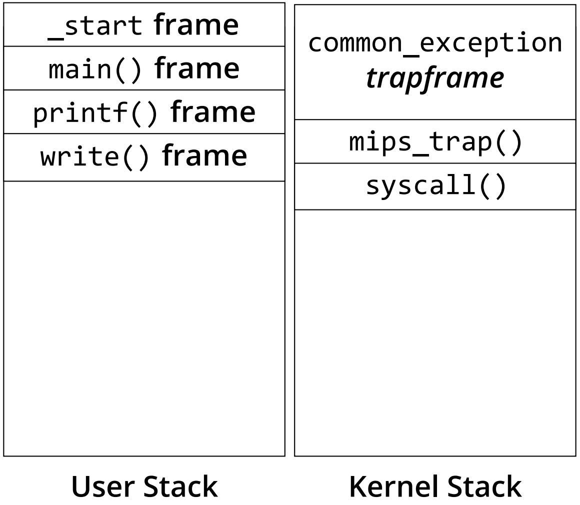
Context Switch: User to Kernel

- *trapframe*: Saves the application context
- `common_exception` saves trapframe on the kernel stack!



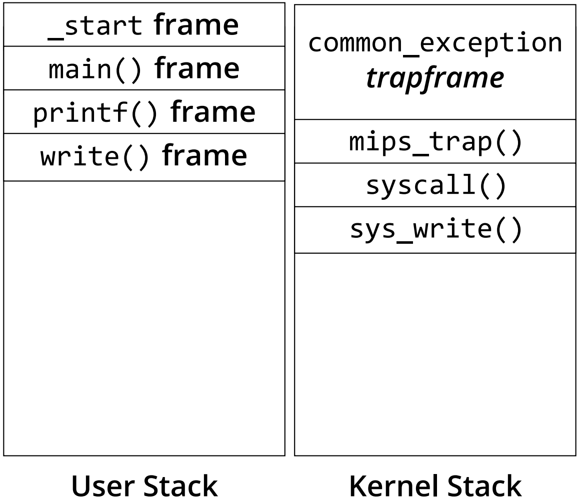
Context Switch: User to Kernel

- *trapframe*: Saves the application context
- Calls `mips_trap()` to decode trap and `syscall()`



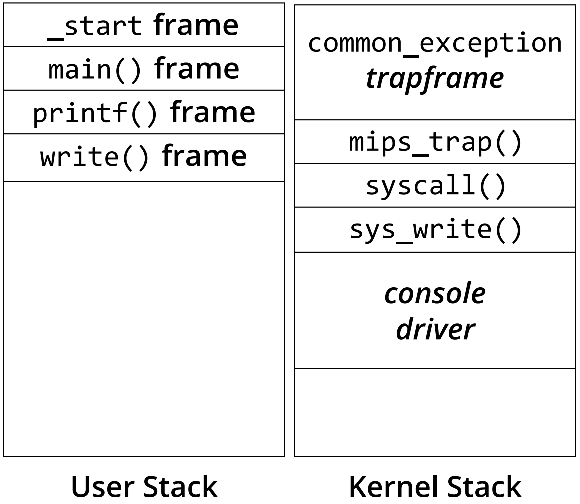
Context Switch: User to Kernel

- *trapframe*: Saves the application context
- `syscall()` decodes arguments and calls `sys_write()`



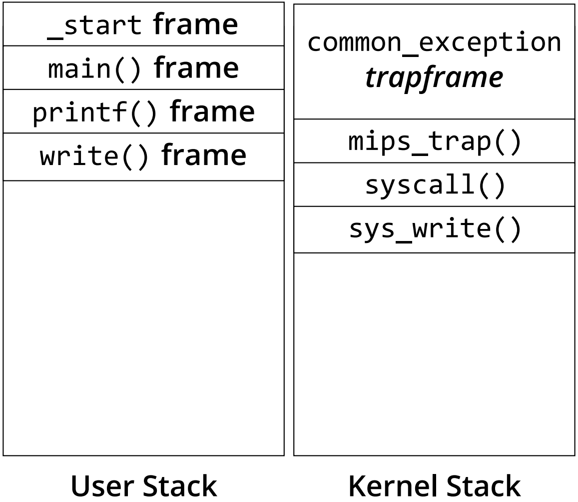
Context Switch: Returning to User Mode

- *trapframe*: Saves the application context
- `sys_write()` writes text to console



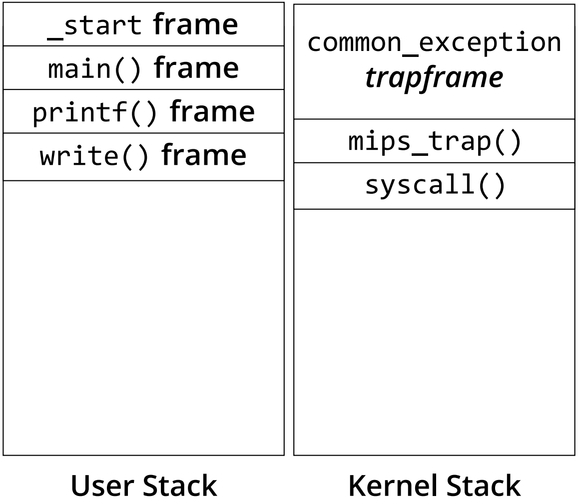
Context Switch: Returning to User Mode

- *trapframe*: Saves the application context
- Return from `sys_write()`



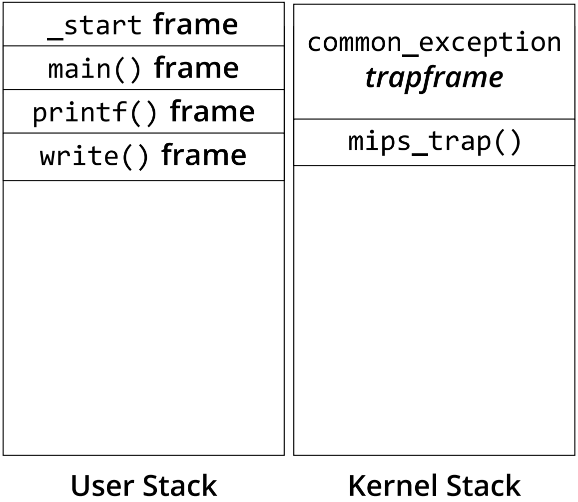
Context Switch: Returning to User Mode

- `syscall()` stores return value and error in trapframe
- `v0`: return value/error code, `a3`: success (1) or failure



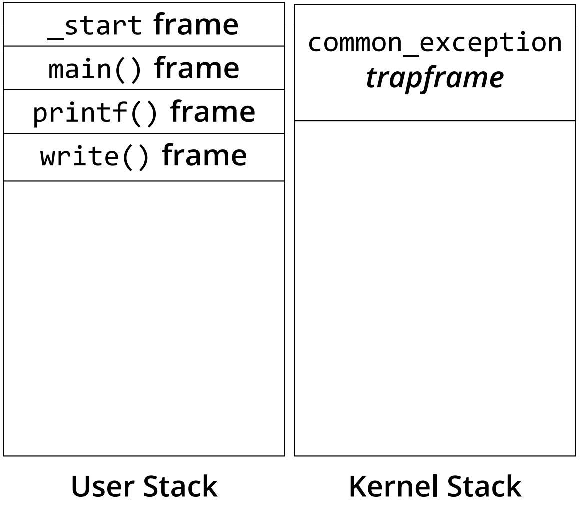
Context Switch: Returning to User Mode

- `mips_trap()` returns to the instruction following `syscall`
- `v0`: return value/error code, `a3`: success (1) or failure



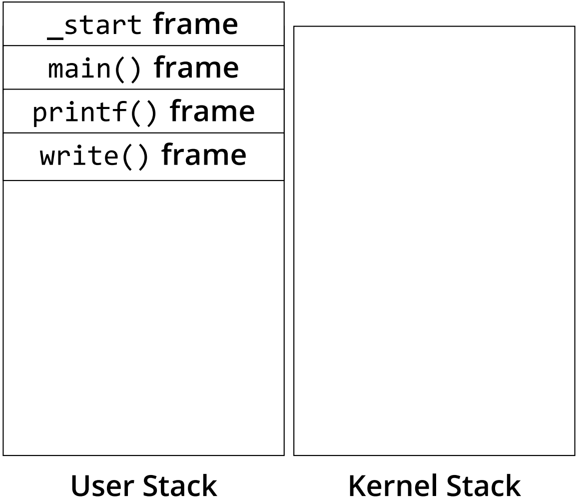
Context Switch: Returning to User Mode

- `common_exception` restores the application context
- Restores all CPU state from the trapframe



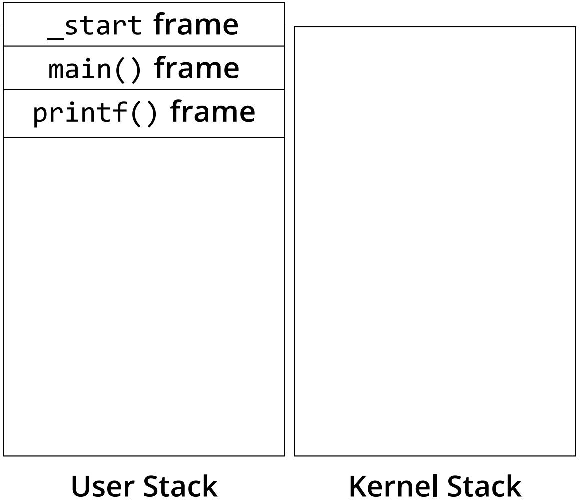
Context Switch: Returning to User Mode

- `write()` decodes `v0` and `a3` and updates `errno`
- *errno* is where error codes are stored in POSIX



Context Switch: Returning to User Mode

- *errno* is where error codes are stored in POSIX
- `printf()` gets return value, if -1 then see `errno`



Outline

1. Kernel API
2. Calling Conventions
3. System Calls
4. Switching Threads/Processes

Scheduling

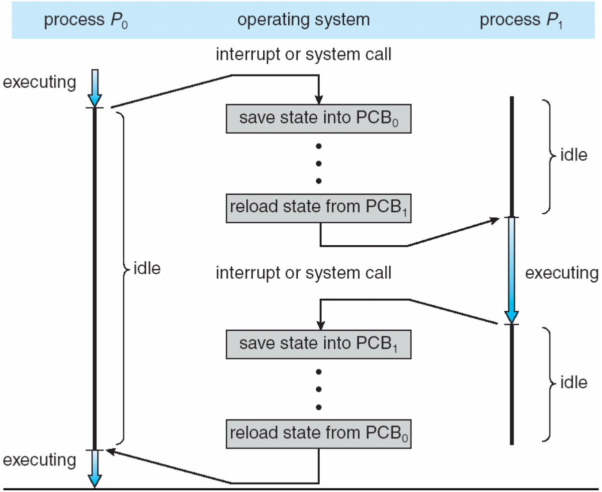
- How to pick which process to run
- Scan process table for first runnable?
  - Expensive. Weird priorities (small pids do better)
  - Divide into runnable and blocked processes
- FIFO/Round-Robin?
  - Put threads on back of list, pull them from front

(OS/161 `kern/thread/thread.c`)
- Priority?
  - Give some threads a better shot at the CPU

Preemption

- Can preempt a process when kernel gets control
- Running process can vector control to kernel
  - System call, page fault, illegal instruction, etc.
  - May put current process to sleep—e.g., read from disk
  - May make other process runnable—e.g., fork, write to pipe
- Periodic timer interrupt
  - If running process used up quantum, schedule another
- Device interrupt
  - Disk request completed, or packet arrived on network
  - Previously waiting process becomes runnable
  - Schedule if higher priority than current running proc.
- Changing running process is called a *context switch*

Context switch



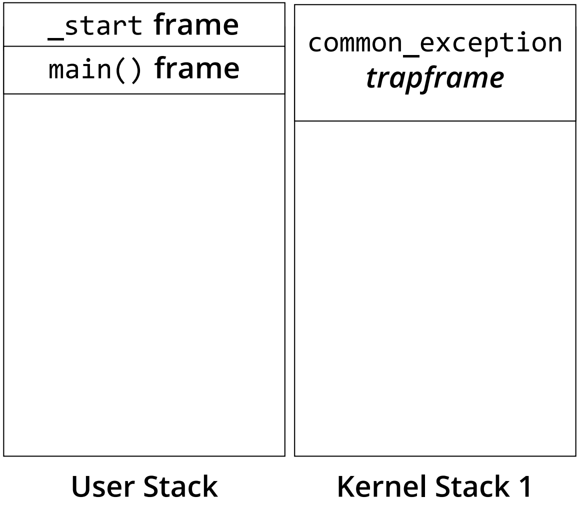
Context switch details

- Very machine dependent. Typical things include:
  - Save program counter and integer registers (always)
  - Save floating point or other special registers
  - Save condition codes
  - Change virtual address translations

- Non-negligible cost
  - Save/restore floating point registers expensive
    - Optimization: only save if process used floating point
  - May require flushing TLB (memory translation hardware)
    - HW Optimization 1: don't flush kernel's own data from TLB
    - HW Optimization 2: use tag to avoid flushing any data
  - Usually causes more cache misses (switch working sets)

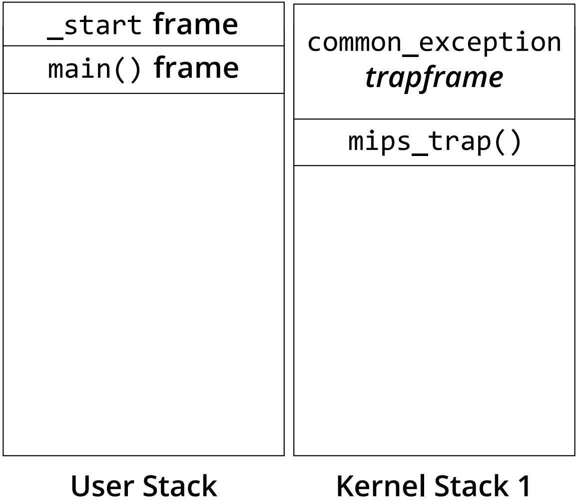
Switching Processes: Timer Interrupt

- Starts with a timer interrupt or sleeping in a system call
- Interrupts user process in the middle of the execution



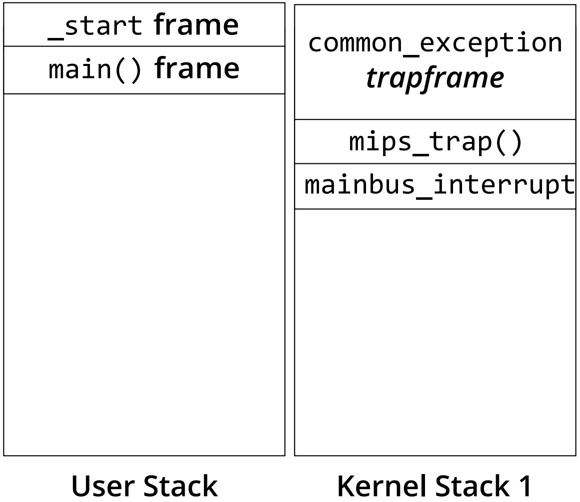
Switching Processes: Timer Interrupt

- `common_execution` saves the trapframe
- `mips_trap()` notices a `EX_IRQ` from the Timer



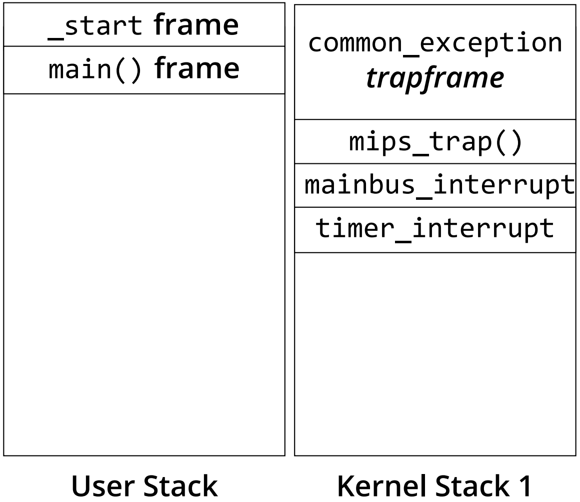
Switching Processes: Timer Interrupt

- Calls `mainbus_interrupt` to handle the IRQ
- On many machines there are multiple IRQ sources!



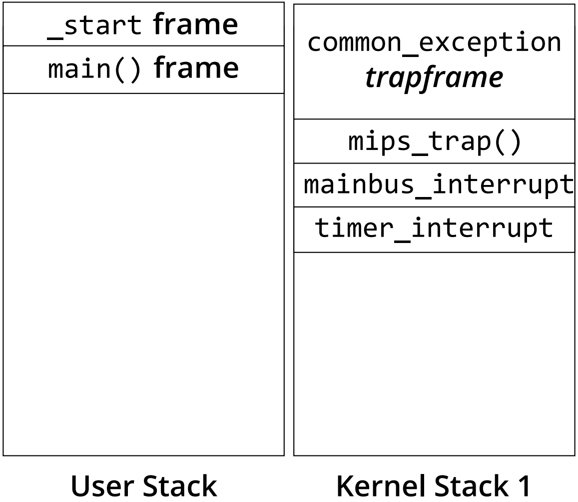
Switching Processes: Timer Interrupt

- `mainbus_interrupt` reads the bus interrupt pins
- Determins the source, in this case a timer interrupt



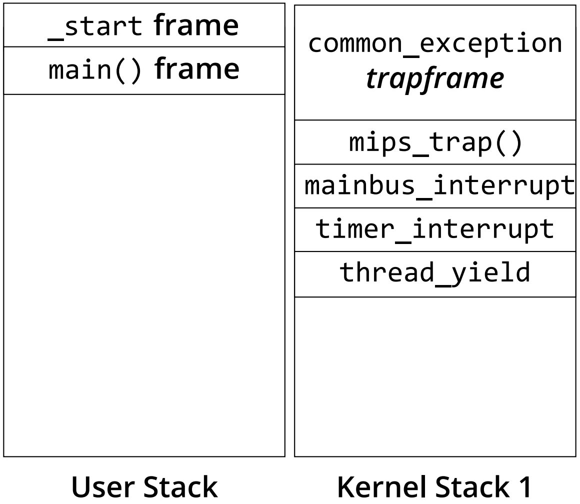
Switching Processes: Timer Interrupt

- Timers trigger processing events in the OS
- Most importantly, calling the CPU scheduler



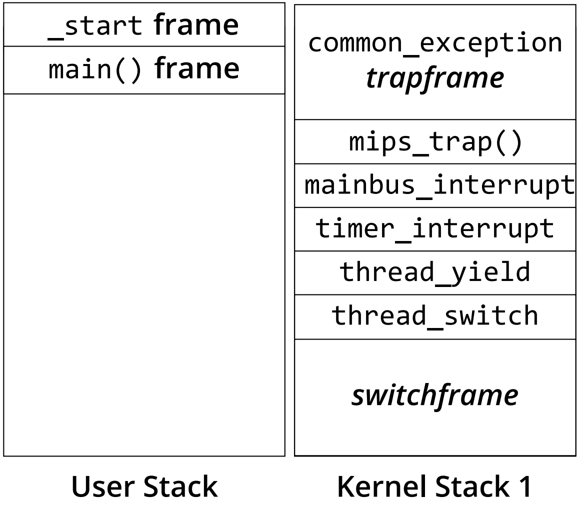
Switching Processes: CPU Scheduler

- `thread_yield()` calls into scheduler to pick next thread
- Calls `thread_switch()` to switch threads



Switching Processes: Thread Switch

- `thread_switch`: saves and restores kernel thread state
- Switching processes is a switch between kernel threads!



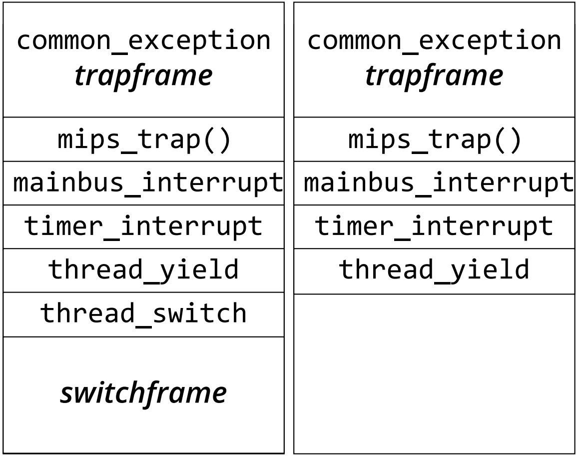
Switching Processes: Thread Switch

- `thread_switch` saves thread state onto the stack
- *switchframe*: contains the kernel context!

Kernel Stack 1	Kernel Stack 2
common_exception *trapframe*	common_exception *trapframe*
mips_trap()	mips_trap()
mainbus_interrupt	mainbus_interrupt
timer_interrupt	timer_interrupt
thread_yield	thread_yield
thread_switch	thread_switch
switchframe	*switchframe*

Switching Processes: Thread Switch

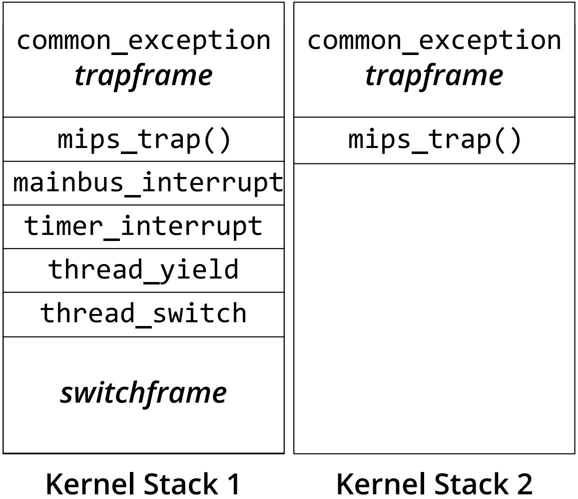
- `thread_switch` restores thread state from the stack
- *switchframe*: contains the kernel context



Kernel Stack 1 Kernel Stack 2

Switching Processes

- Returns from the device code
- `mips_trap()` returns



Switching Processes

- `common_exception` restores the trapframe
- *trapframe*: contains the application context!