CS350: Operating Systems
Lecture 6: System Calls and Interrupts

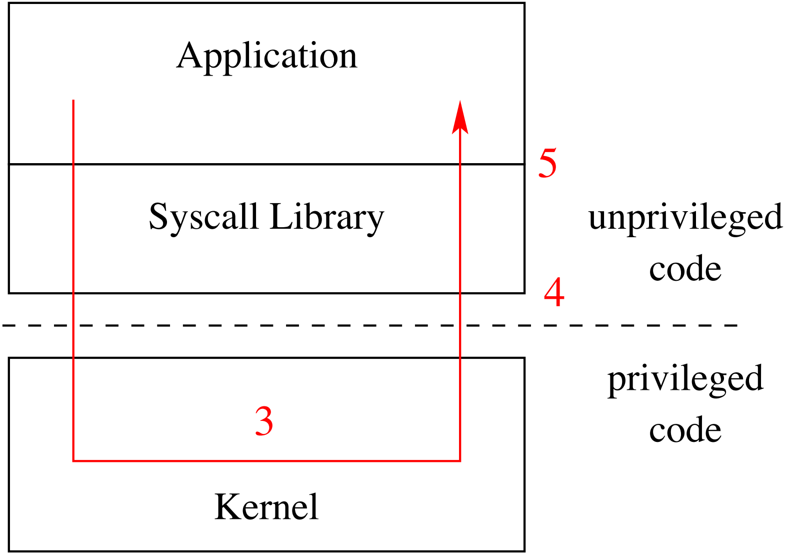
Ali Mashtizadeh

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!

x86-64 Exception Vectors

- Interrupts, exceptions and system calls use the same mechanism
- x86–64 offers a high performance path for system calls (not used in COS)

```
#define T_DE        0       /* Divide Error Exception */
#define T_DB        1       /* Debug Exception */
#define T_NMI       2       /* NMI Interrupt */
#define T_BP        3       /* Breakpoint Exception */
#define T_OF        4       /* Overflow Exception */
#define T_BR        5       /* BOUND Range Exceeded Exception */
#define T_UD        6       /* Invalid Opcode Exception */
#define T_NM        7       /* Device Not Available Exception */
#define T_DF        8       /* Double Fault Exception */
#define T_TS        10      /* Invalid TSS Exception */
#define T_NP        11      /* Segment Not Present */
#define T_SS        12      /* Stack Fault Exception */
#define T_GP        13      /* General Protection Exception */
#define T_PF        14      /* Page-Fault Exception */
#define T_MF        16      /* x87 FPU Floating-Point Error */
#define T_AC        17      /* Alignment Check Exception */
#define T_MC        18      /* Machine-Check Exception */
...
```

System Calls

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

1. Application loads the arguments into CPU registers
2. Load the system call number into register `rdi` (first arg)
3. Executes `int 60` instruction to trigger `T_SYS` exception
4. Processor looks up the interrupt vector
5. Processor jumps to the kernel exception handler
6. Returns to userspace using `iret`, return from exception instruction

Hardware Interrupt Handling in x86–64

- Interrupt descriptor table: defines the entry point for interrupt vector.
- Configuring the IDT:

1. OS initializes IDT with entry point of interrupt vectors (1-255)

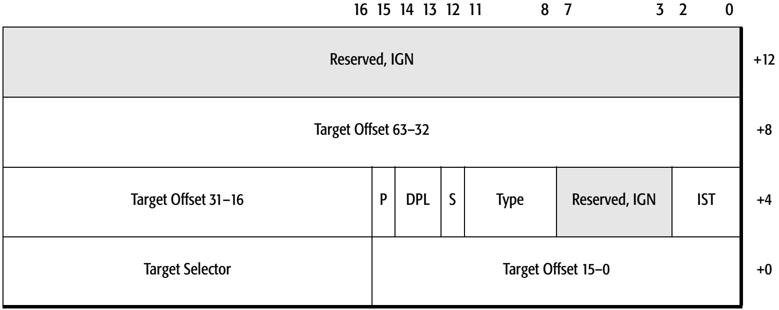


Figure 4-24. Interrupt-Gate and Trap-Gate Descriptors—Long Mode

Interrupt Gate Descriptor (x86–64)

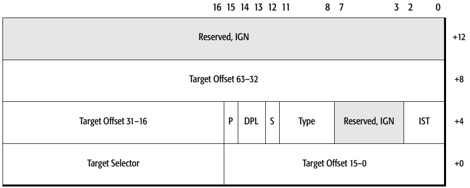
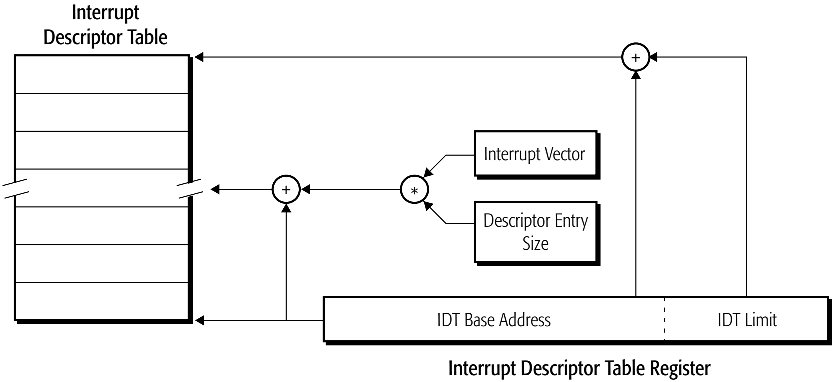


Figure 4-24. Interrupt-Gate and Trap-Gate Descriptors–Long Mode

- *Target Offset*: First instruction of the interrupt handler
- *Target Selector*: Code segment – sets priviledge level (user/kernel mode)
  - More on this later
- *P*: Present (i.e. valid)
- *DPL*: Minimum priviedge level that can trigger it
  - Prevents user programs from triggering device interrupts
- *Type*: Constant for 64-bit IDT entry
- *IST*: Kernel stack to use

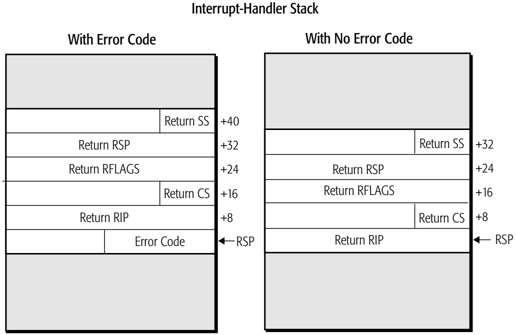
Configuring Interrupt Handling (x86–64)

1. OS initializes IDT with entry point of interrupt vectors (1-255)
2. OS initializes the IDT descriptor containing address and length of IDT
3. OS uses `lidt` instruction to load the IDTR



Hardware Interrupt Handling Process (x86–64)

1. Finds the IDT through the IDTR register
2. Read the IDT descriptor entry
3. Look up the kernel stack in the TSS (Task State Segment)
4. IST field specifies which stack to use
5. CPU pushes the interrupt stack frame



Hardware Interrupt Handling Process (x86–64)

1. Finds the IDT through the IDTR register
2. Read the IDT descriptor entry
3. Look up the kernel stack in the TSS (Task State Segment)
4. IST field specifies which stack to use
5. CPU pushes the interrupt stack frame
6. Kernel pushes the trap frame
7. Kernel sets up CPU to known state to run C code

Bit Offset 31 – 16	Bit Offset 15 – 0	Byte Offset
I/O-Permission Bitmap (IOPB) (Up to 8 Kbytes)		IOPB Base
I/O Map Base Address	Reserved, IGN	+64h
Reserved, IGN		+60h
		+5Ch
IST7[63:32]		+58h
IST7[31:0]		+54h
IST6[63:32]		+50h
IST6[31:0]		+4Ch
IST5[63:32]		+48h
IST5[31:0]		+44h
IST4[63:32]		+40h
IST4[31:0]		+3Ch
IST3[63:32]		+38h
IST3[31:0]		+34h
IST2[63:32]		+30h
IST2[31:0]		+2Ch
IST1[63:32]		+28h
IST1[31:0]		+24h
Reserved, IGN		+20h
		+1Ch
RSP2[63:32]		+18h
RSP2[31:0]		+14h
RSP1[63:32]		+10h
RSP1[31:0]		+0Ch
RSP0[63:32]		+08h
RSP0[31:0]		+04h
Reserved, IGN		+00h

Figure 12-8. Long Mode TSS Format

System Call Operation Details

- Application calls into the C library (e.g., calls `write()`)
- Library executes the `syscall` instruction
- Kernel exception handler 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 (`iret` 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*

System Call Numbering

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

```
#define SYSCALL_NULL     0x00
#define SYSCALL_TIME     0x01
#define SYSCALL_GETPID   0x02
#define SYSCALL_EXIT     0x03
#define SYSCALL_SPAWN    0x04
#define SYSCALL_WAIT     0x05

// Memory
#define SYSCALL_MMAP>--->-------0x08
#define SYSCALL_MUNMAP>->-------0x09
#define SYSCALL_MPROTECT>-------0x0A
...
```

x86–64 Calling Conventions

- *Caller-saved registers* are saved before calling another function
  - `r10`, `r11`: Scratch registers
  - `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`: Argument registers
  - `rax`, `rdx`: Return values
- *Callee-saved registers* are saved inside the function
  - `rbx`, `r12–r15`: Saved registers
- *Stack registers*
  - `rsp`: Stack pointer
  - `rbp`: Frame pointer (assuming -fno-omit-framepointer)
- Instructions:
  - `call`: Call function and save return address on stack
  - `ret`: Return from function

Functions in x86–64

- Functions are called with the `call` instruction
- `call` pushes the return address to the stack and jumps to the target

```
foo:
      push %rbp # Save the frame pointer
      mov %rsp, %rbp # Set the frame pointer to TOS

      # Save caller-save registers (if needed)

      call bar # Call bar

      # Restore registers (if needed)

      pop %rbp
      ret # Return
```

Functions in x86–64 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:
      mov %edi, %eax # Move 1st arg to eax (lower 32-bits of rax)
      add $41, %eax # Add 41 to eax
      ret
```

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
push %rax
# Equivalent to:
mov %rax, -8(%rsp) # Store into the top of stack
sub $8, %rsp

# Pop stack element
pop %rax
# Equivalent to:
mov 0(%rsp), %rax # Load from the top of stack
add $8, %rsp
```

Outline

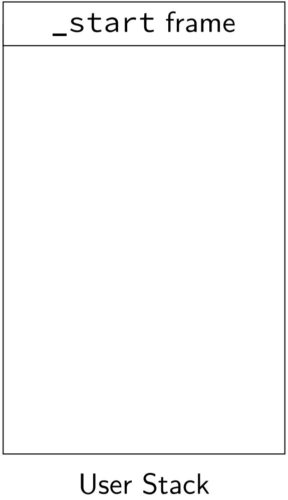
Execution Contexts

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 (usually between kernel contexts)

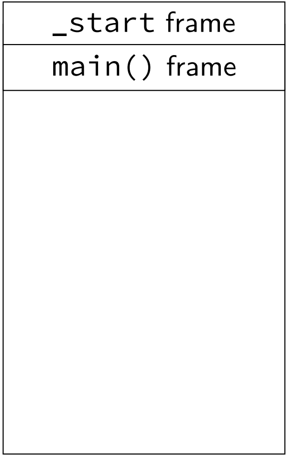
Application Stack

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



Application Stack

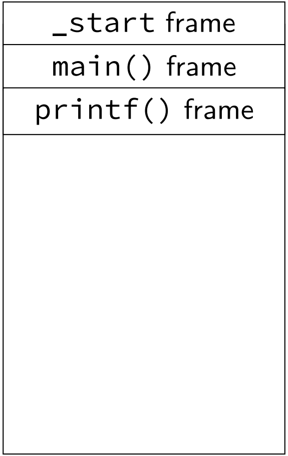
- Stack made of up *frames* containing locals, arguments, and spilled registers
- Programs begin execution at `_start`



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`

`_start` frame
`main()` frame
`printf()` frame
`write()` frame

User Stack

Application Stack

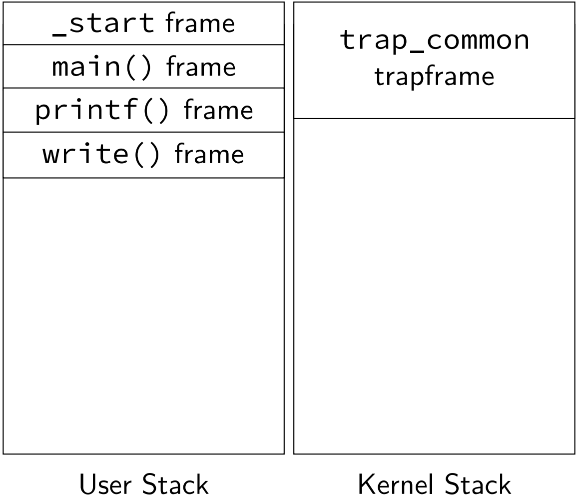
- Stack made of up *frames* containing locals, arguments, and spilled registers
- Programs begin execution at `_start`

`_start` frame
`main()` frame
`printf()` frame
`write()` frame
???

User Stack

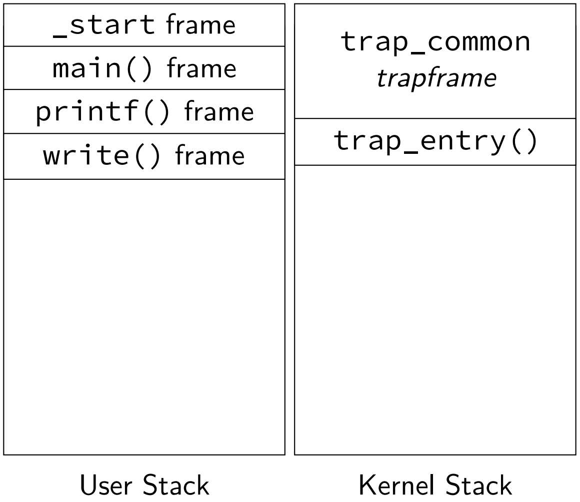
Context Switch: User to Kernel

- *trapframe*: Saves the application context
- `int $60` instruction triggers the exception handler (vector 60)



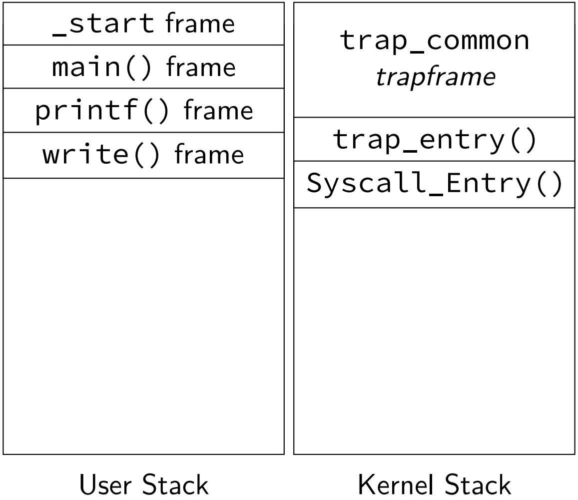
Context Switch: User to Kernel

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



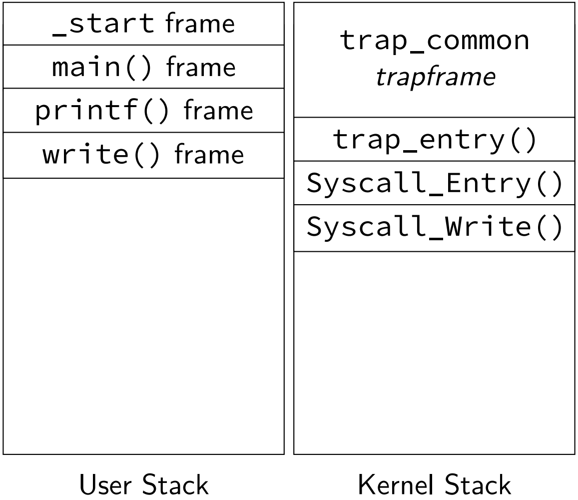
Context Switch: User to Kernel

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



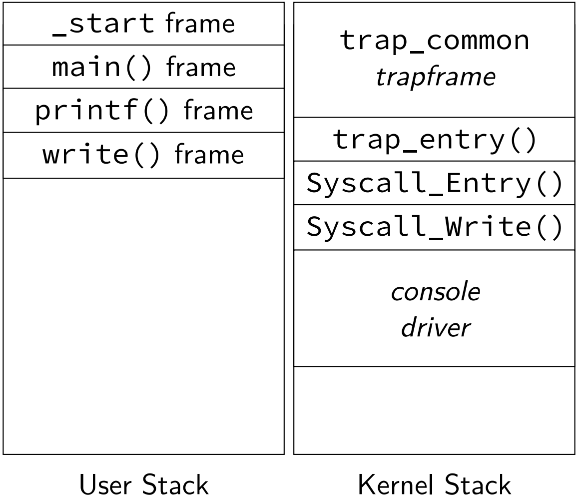
Context Switch: User to Kernel

- *trapframe*: Saves the application context
- `Syscall_Entry()` decodes arguments and calls `Syscall_Write()`



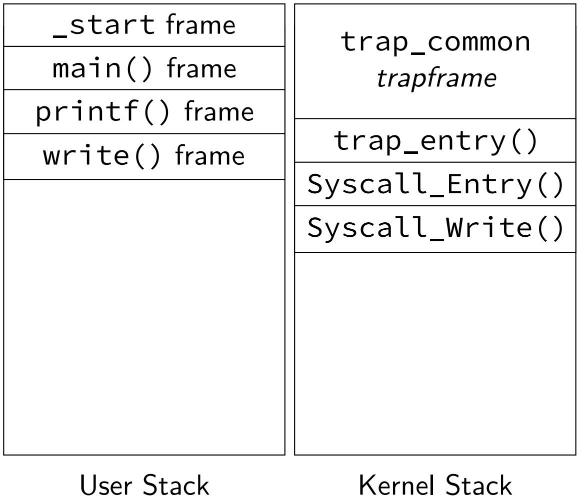
Context Switch: Returning to User Mode

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



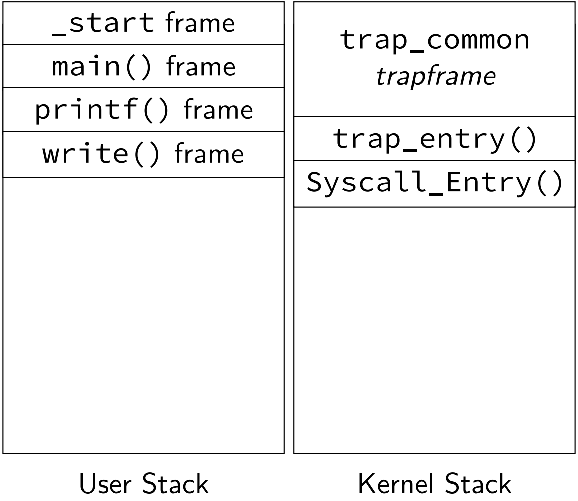
Context Switch: Returning to User Mode

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



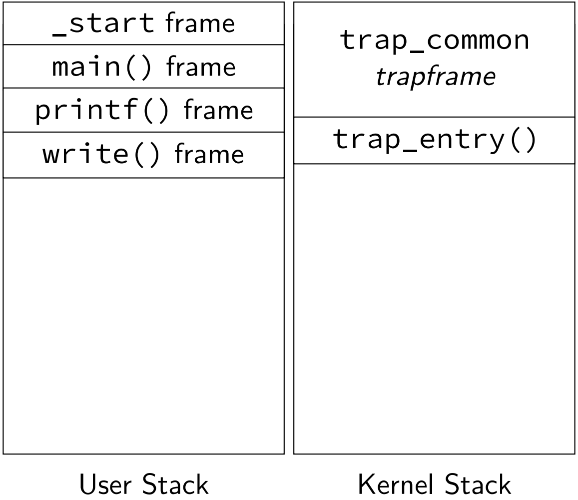
Context Switch: Returning to User Mode

- `Syscall_Entry()` stores return value and error in trapframe
- `rax`: return value/error code



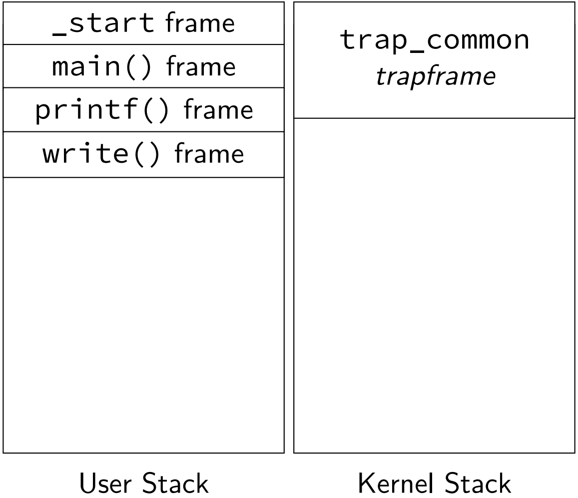
Context Switch: Returning to User Mode

- `trap_common()` returns to the instruction following `int $60`
- `rax`: return value/error code



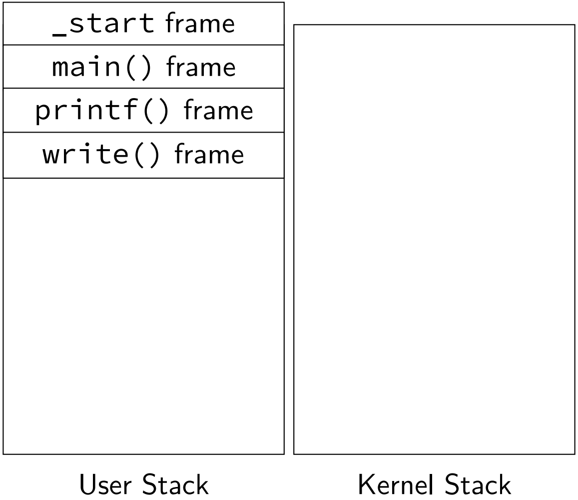
Context Switch: Returning to User Mode

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



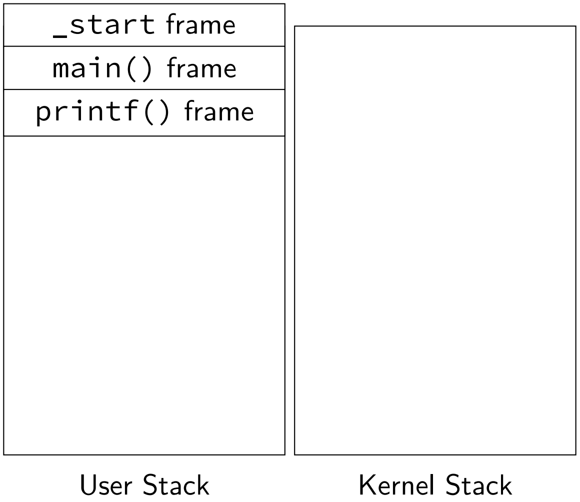
Context Switch: Returning to User Mode

- `write()` decodes `rax` 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 sets `errno`



Outline

Scheduling

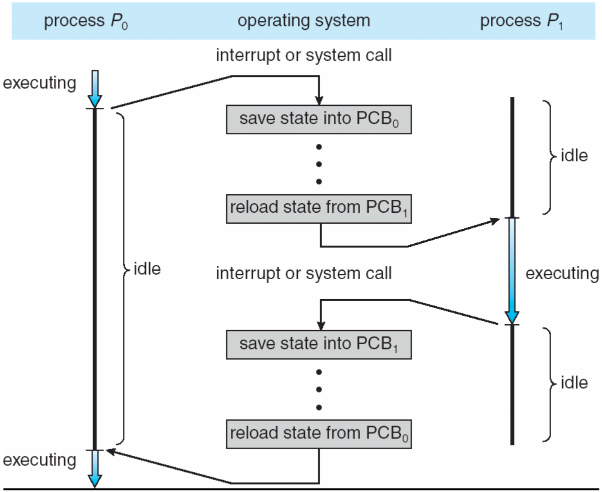
- How to pick which process to run
- Scan process table for first runnable?
  - Expensive. Weird priorities (small pids do better)
  - Divide into runnable and blocked processes
- FIFO/Round-Robin?
  - Put threads on back of list, pull them from front

    (see `kern/sched.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



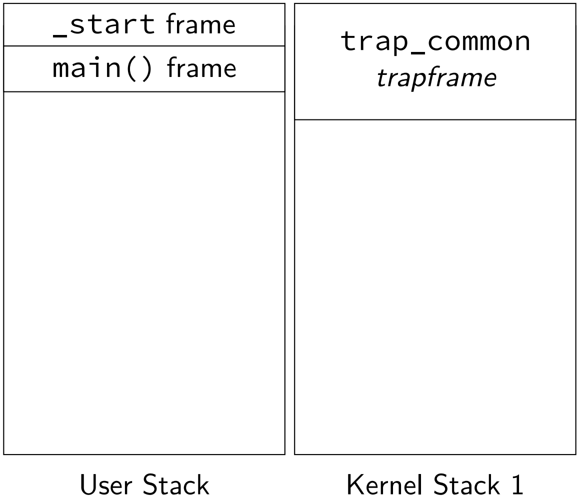
Context switch details

- 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)

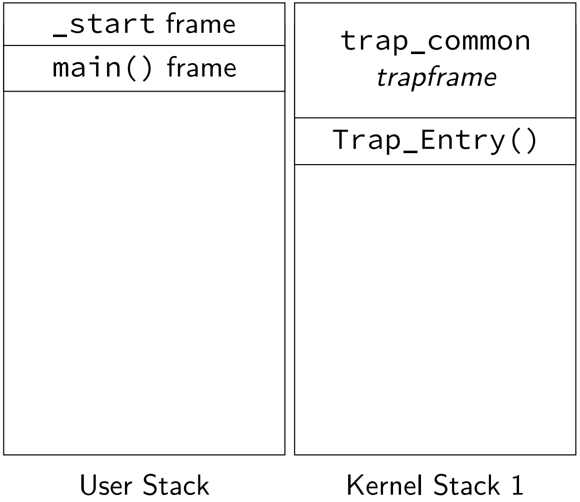
Switching Processes: Timer Interrupt

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



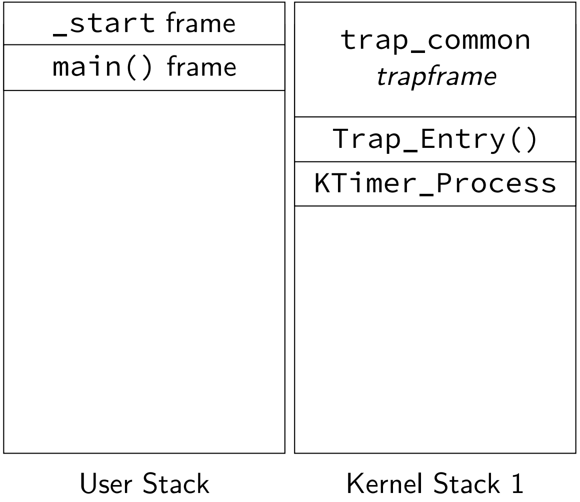
Switching Processes: Timer Interrupt

- `trap_common` saves the trapframe
- `Trap_Entry()` notices a `T_IRQ_TIMER` from the Timer



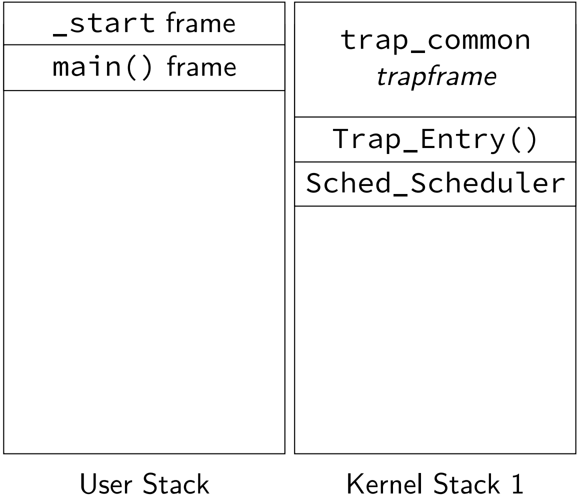
Switching Processes: Timer Interrupt

- Calls `KTimer_Process` to process any scheduled timer events



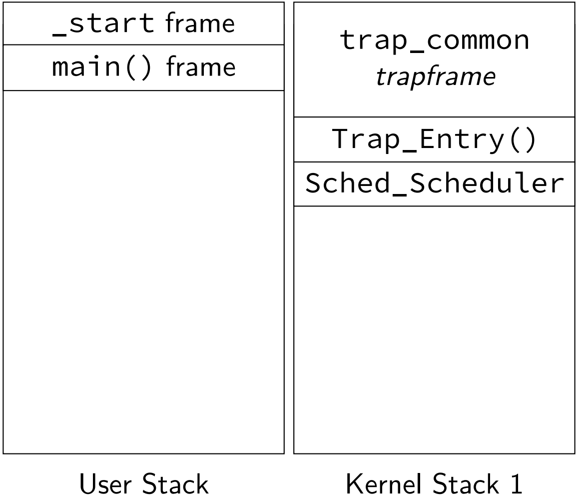
Switching Processes: Timer Interrupt

- Calls `Sched_Scheduler` to switch to a new process



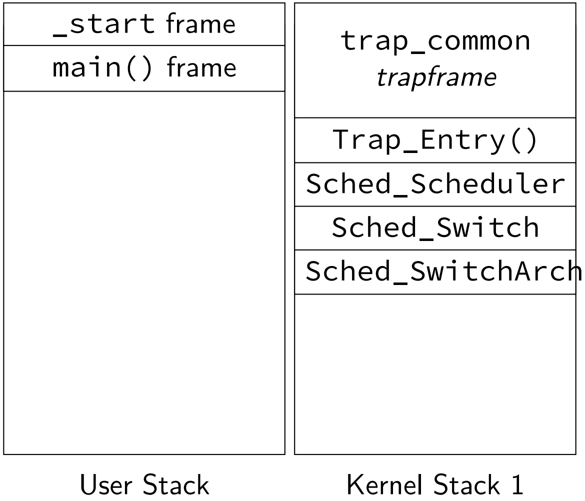
Switching Processes: Timer Interrupt

- Timers trigger processing events in the OS and the CPU scheduler



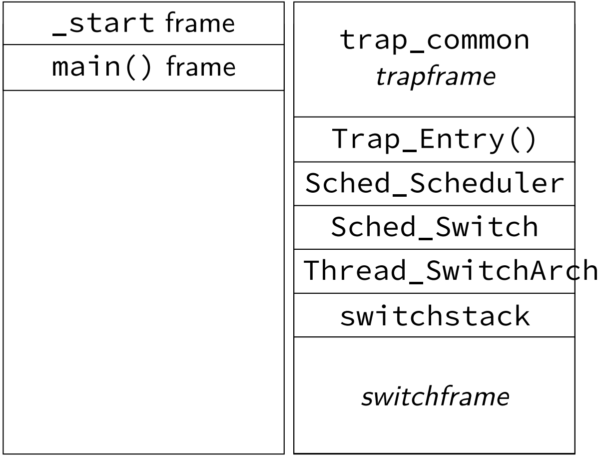
Switching Processes: CPU Scheduler

- `Sched_Scheduler()` calls into scheduler to pick next thread
- Calls `Sched_Switch()` to switch threads



Switching Processes: Thread Switch

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



User Stack   Kernel Stack 1

Switching Processes: Thread Switch

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

`trap_common` *trapframe*	`trap_common` *trapframe*
`Trap_Entry()`	`Trap_Entry()`
`Sched_Scheduler`	`Sched_Scheduler`
`Sched_Switch`	`Sched_Switch`
`Thread_SwitchArc`	`Thread_SwitchArch`
`switchstack`	`switchstack`
switchframe	*switchframe*
Kernel Stack 1	Kernel Stack 2