Today's Lecture

- **System Calls and Trap Frames**
- **Switching Processes or Threads**

Outline

1. System Calls
2. Switching Threads/Processes
3. System Call Implementation

Execution Contexts

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

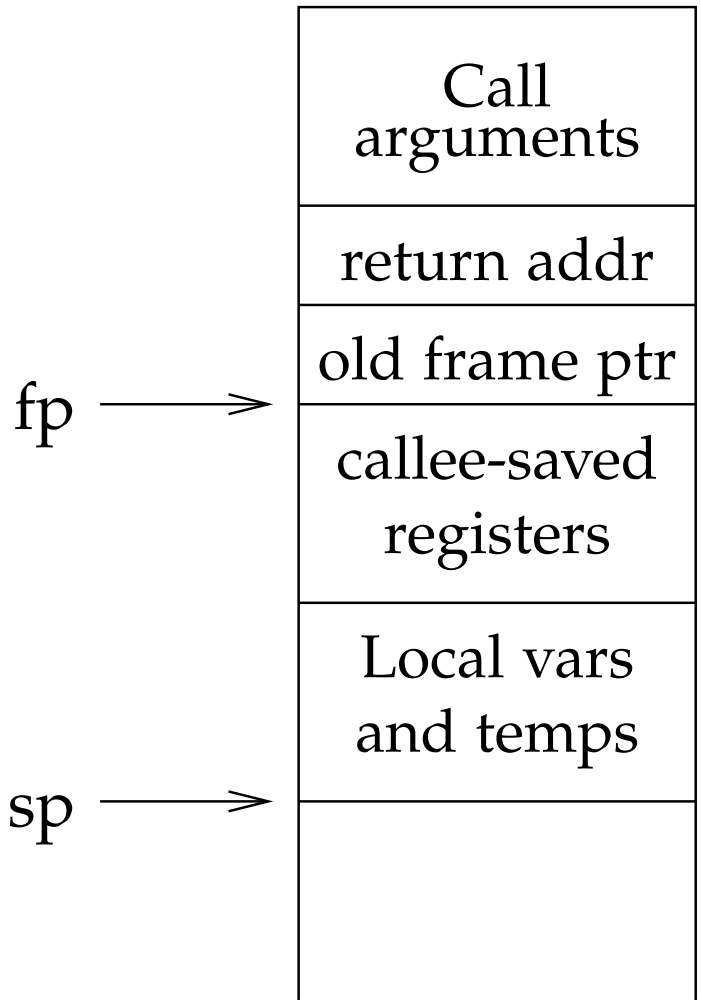
- **Many different execution contexts!**
- *Application Context*: **Application threads**
- *Kernel Context*: **Kernel threads, software interrupts, etc**
- *Interrupt Context*: **Interrupt handler**
- **Kernel and Interrupts usually the same context**
- **Today's Lecture: transitioning between, saving and restoring contexts**

Application Context

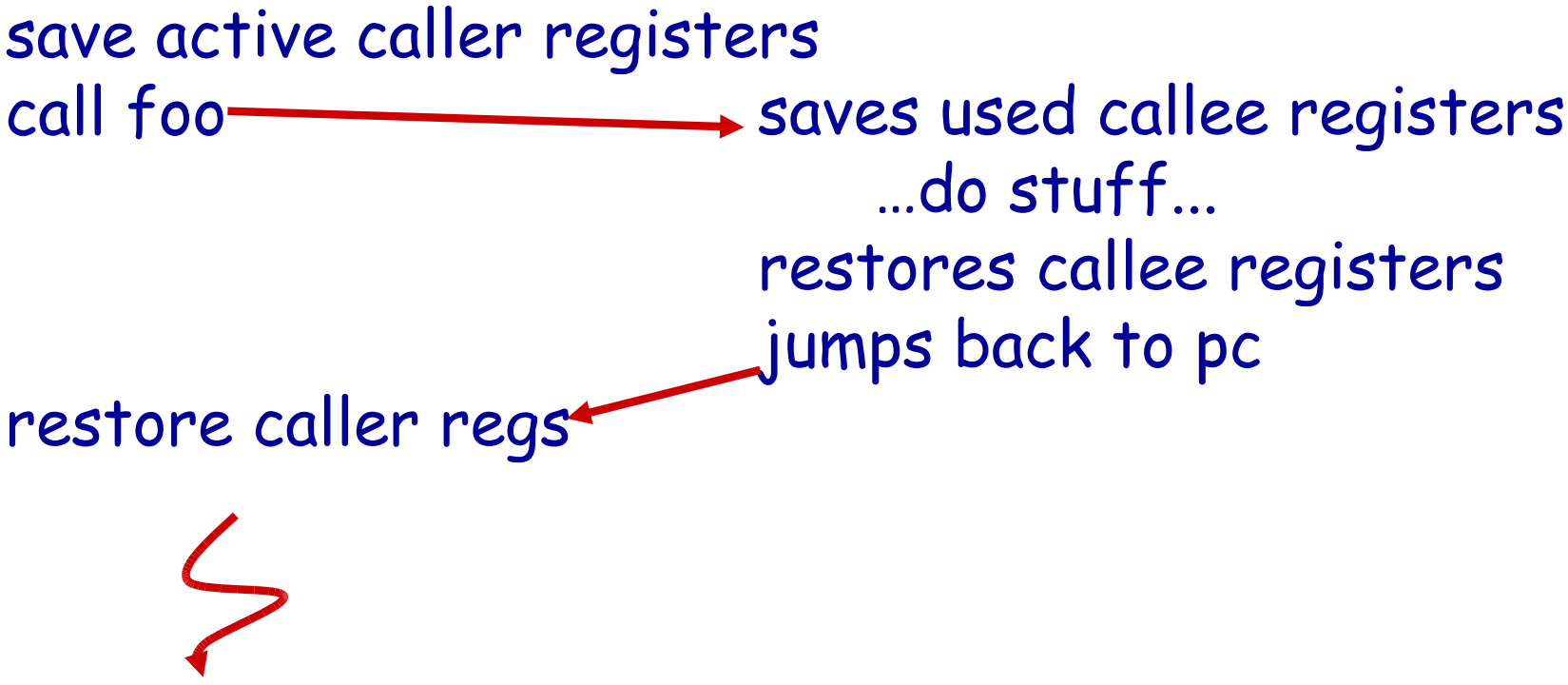
- **Application context consists:**
  - CPU Registers and Stack: arguments, local variables, return addresses

Calling Conventions

- **Registers divided into 2 groups**
  - Functions free to clobber *caller-saved* regs (`%eax` [return val], `%edx`, & `%ecx` on x86)
  - But must restore *callee-saved* ones to original value upon return (on x86, `%ebx`, `%esi`, `%edi`, plus `%ebp` and `%esp`)
- ***sp* register always base of stack**
  - Frame pointer (*fp*) is old *sp*
- **Local variables stored in registers and on stack**
- **Function arguments go in caller-saved regs and on stack**
  - With x86, all arguments on stack



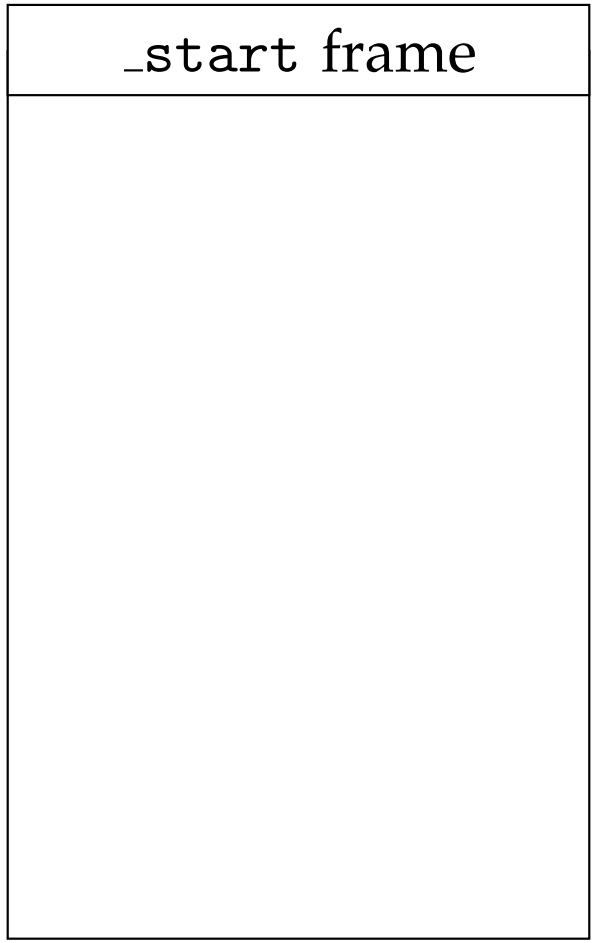
Procedure Calls



- **Some state saved on stack**
  - Return address, caller-saved registers
- **Some state not saved**
  - Callee-saved regs, global variables, stack pointer

Application Stack

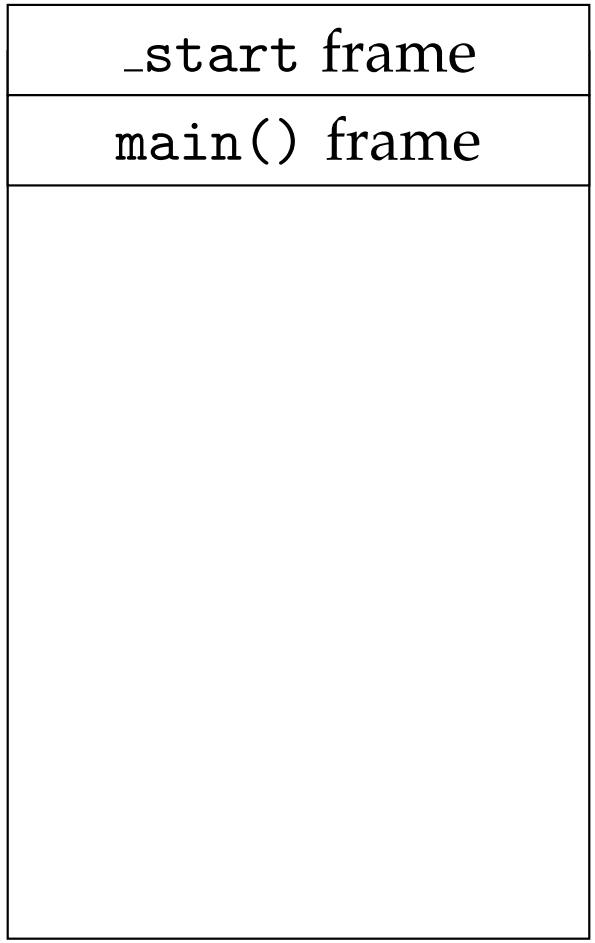
- **Application stack is made of up *frames* containing locals, arguments, and spilled registers**
- **Programs begin execution at `_start`**



User Stack

Application Stack

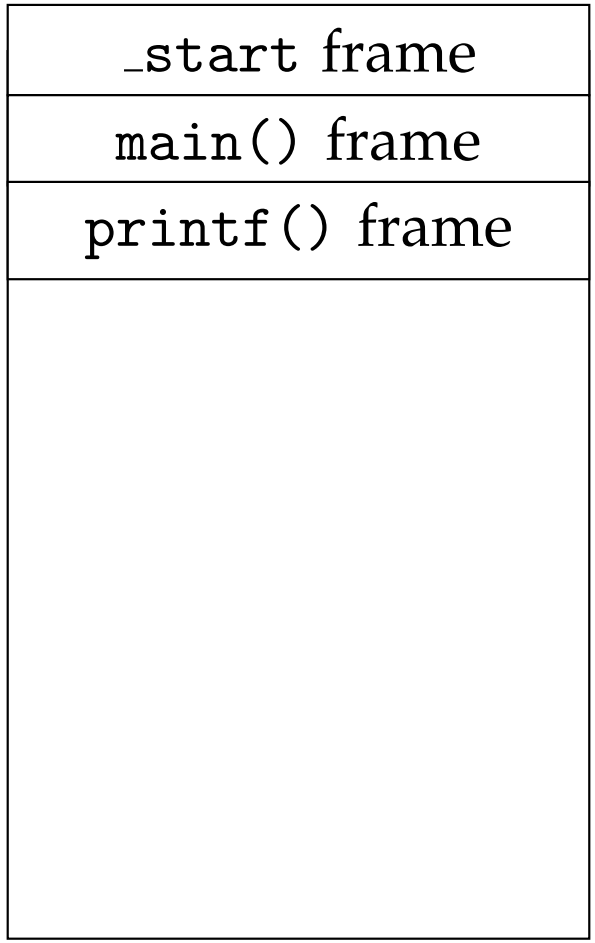
- **Application stack is made of up *frames* containing locals, arguments, and spilled registers**
- **Programs begin execution at `_start`**



User Stack

Application Stack

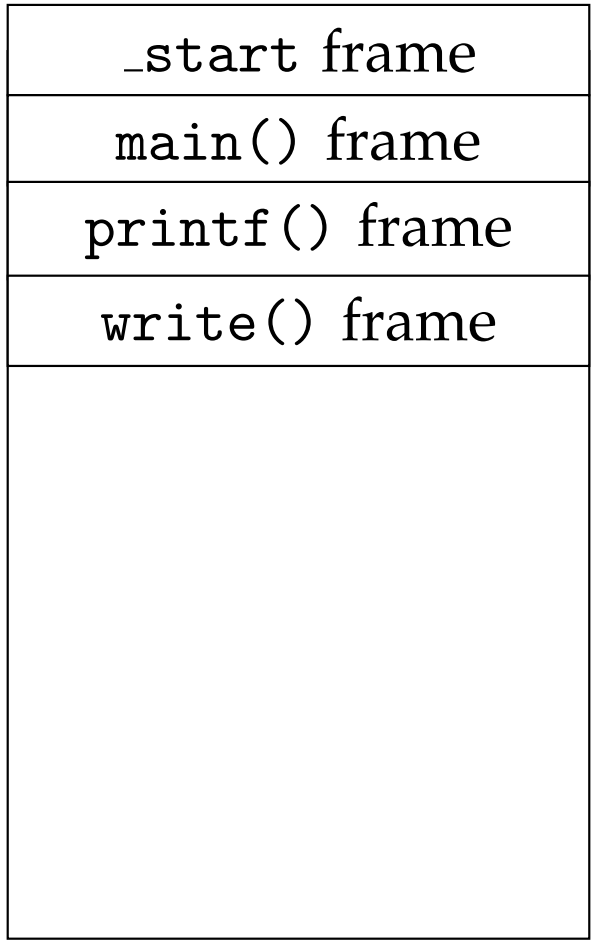
- **Application stack is made of up *frames* containing locals, arguments, and spilled registers**
- **Programs begin execution at `_start`**



User Stack

Application Stack

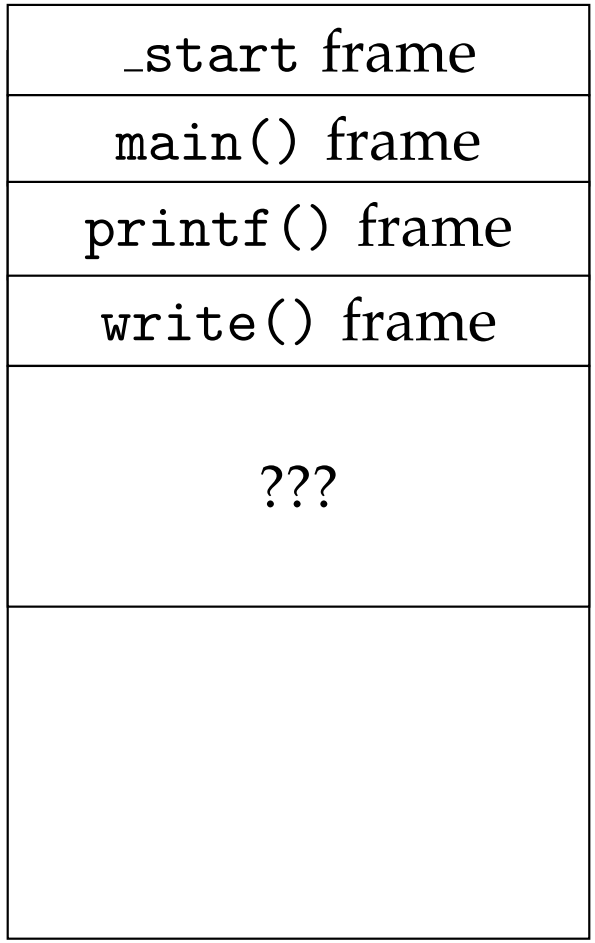
- **Application stack is made of up *frames* containing locals, arguments, and spilled registers**
- **Programs begin execution at `_start`**



User Stack

Application Stack

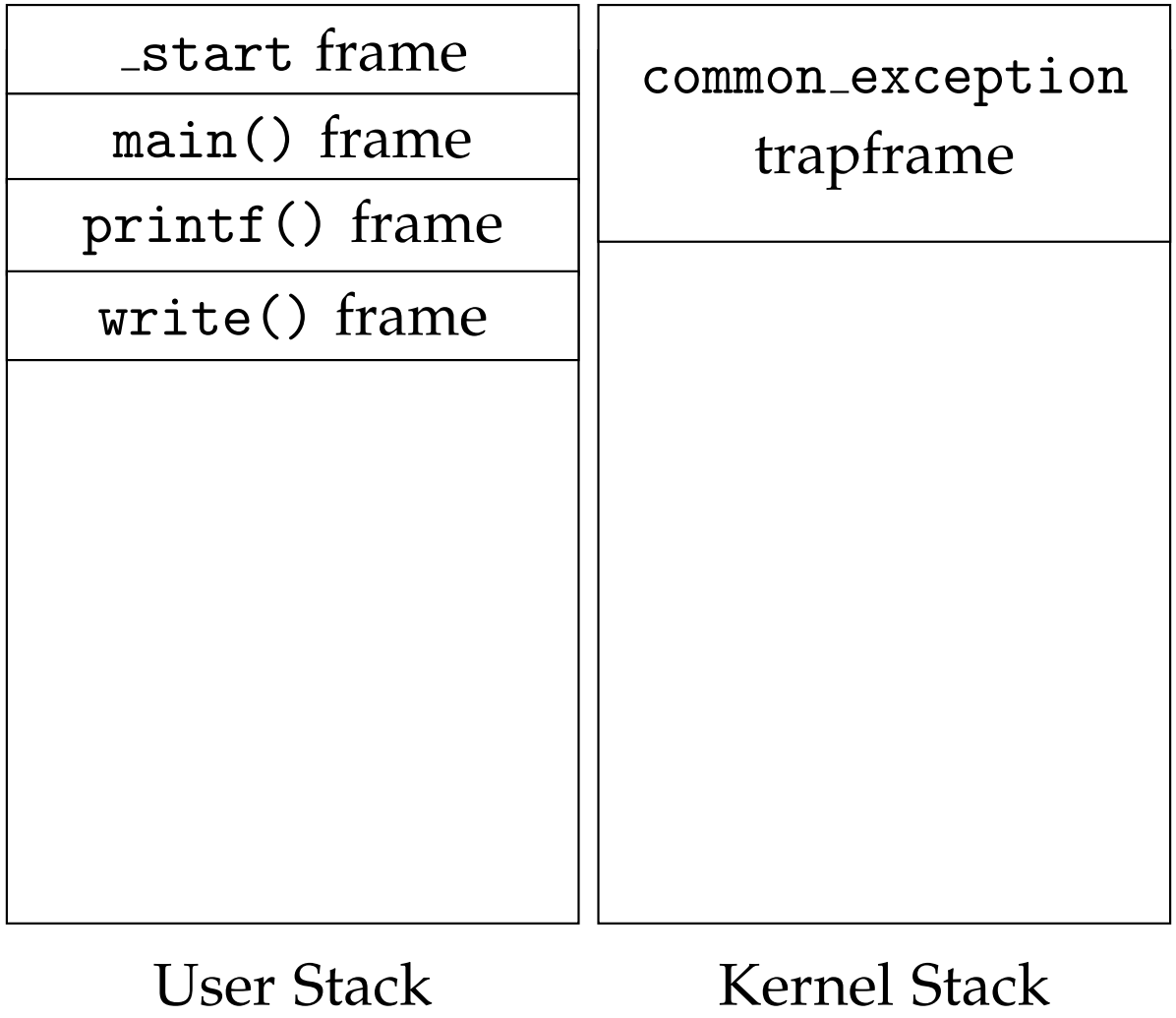
- **Application stack is made of up *frames* containing locals, arguments, and spilled registers**
- **Programs begin execution at `_start`**



User Stack

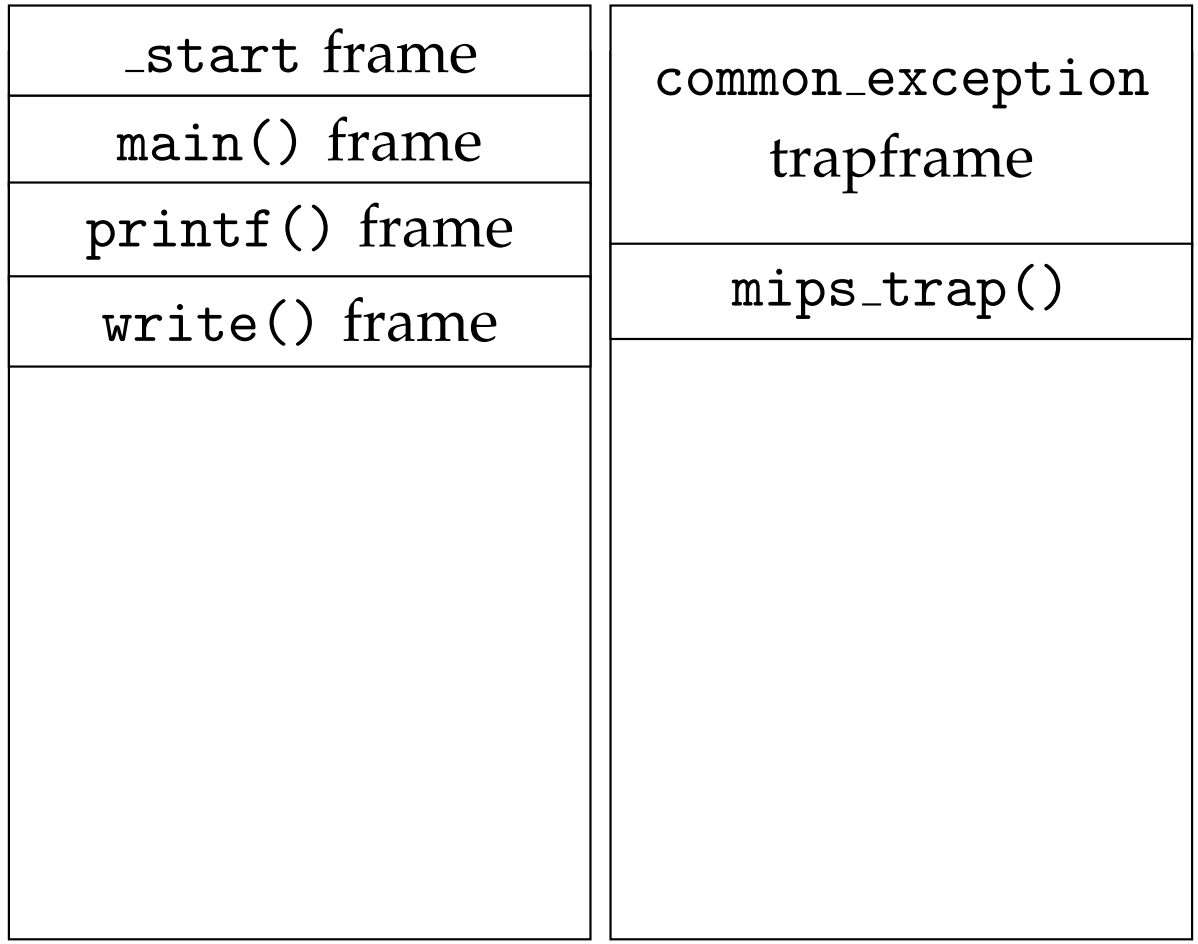
Mode Switching: User to Kernel

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



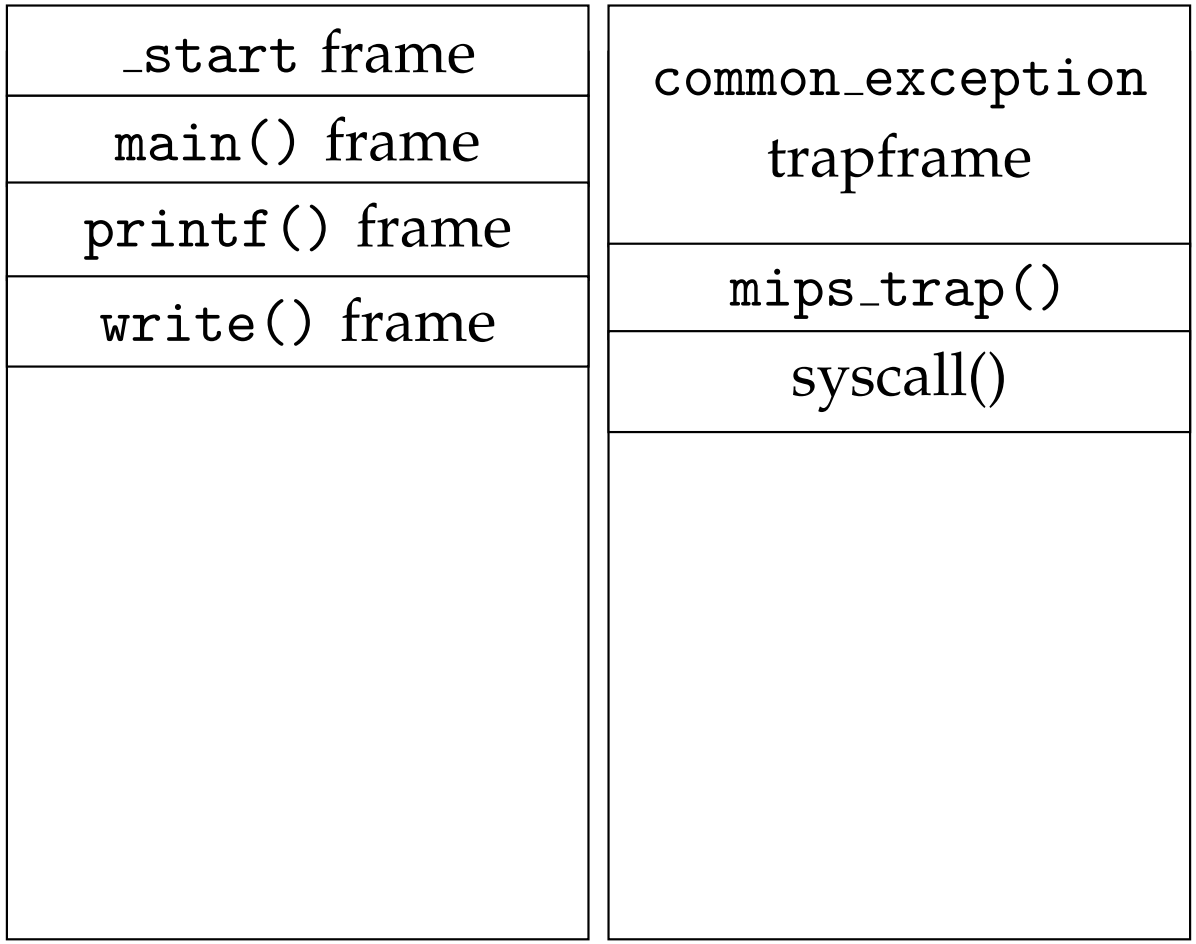
Mode Switching: User to Kernel

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



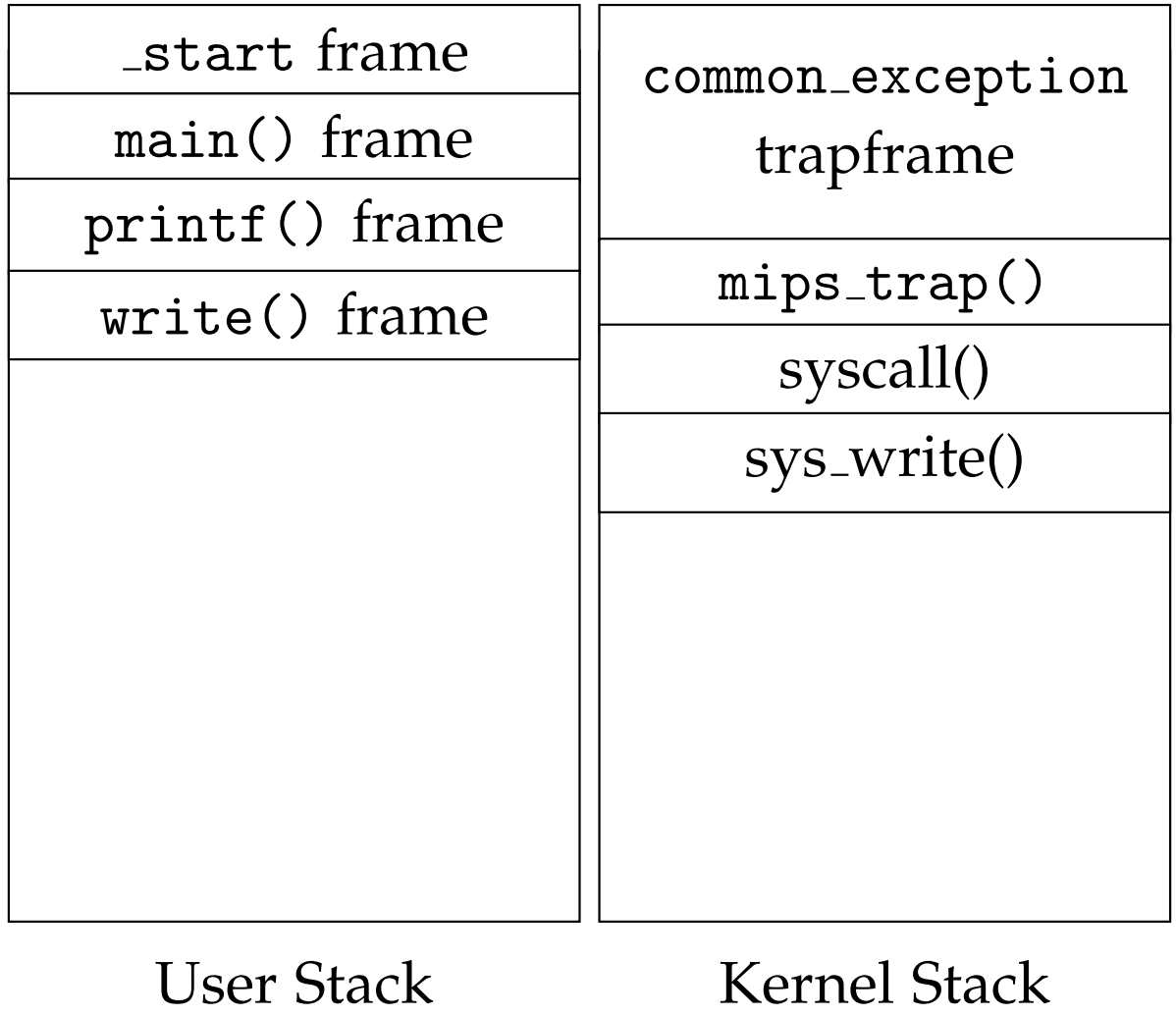
Mode Switching: User to Kernel

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



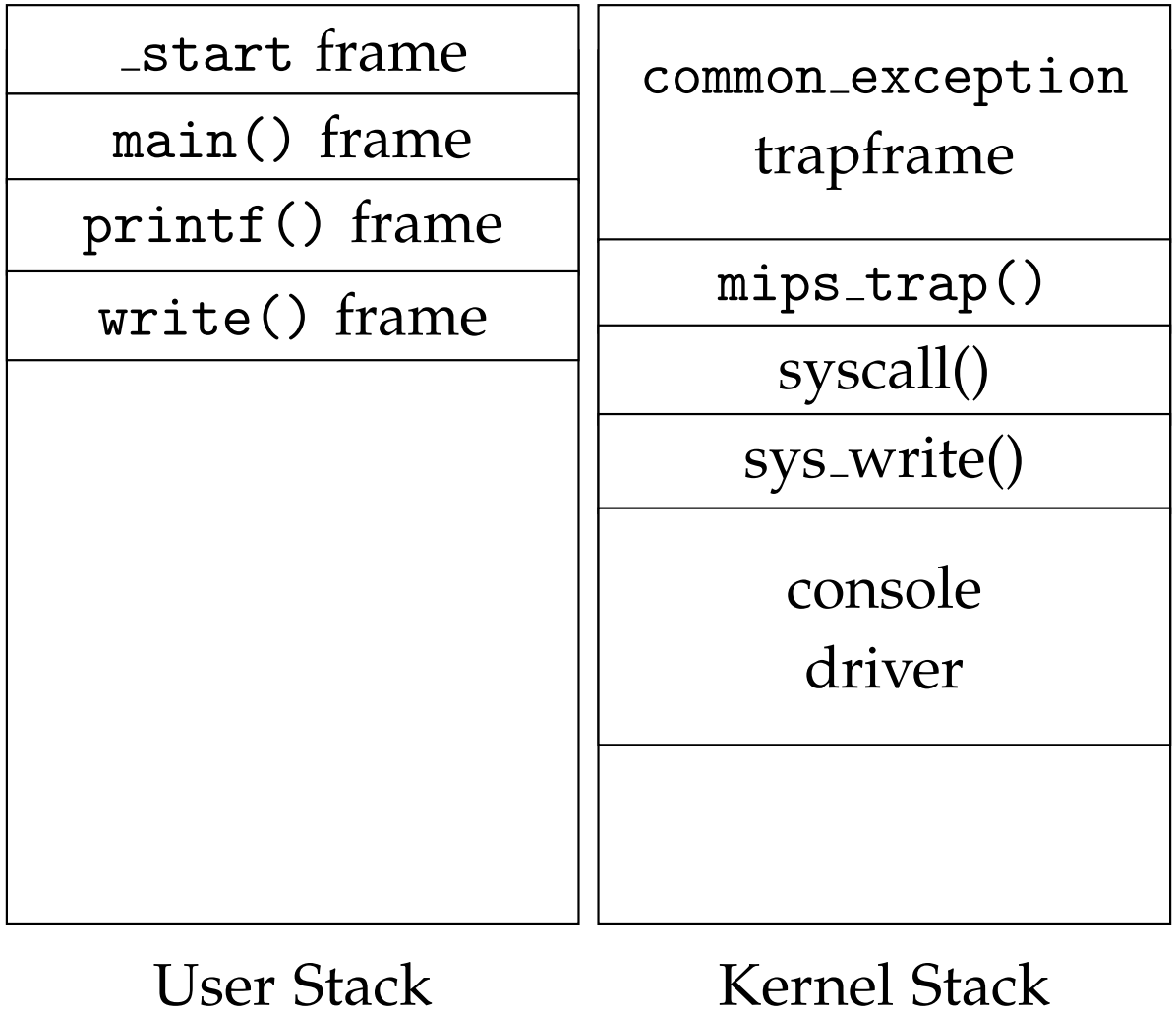
Mode Switching: User to Kernel

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



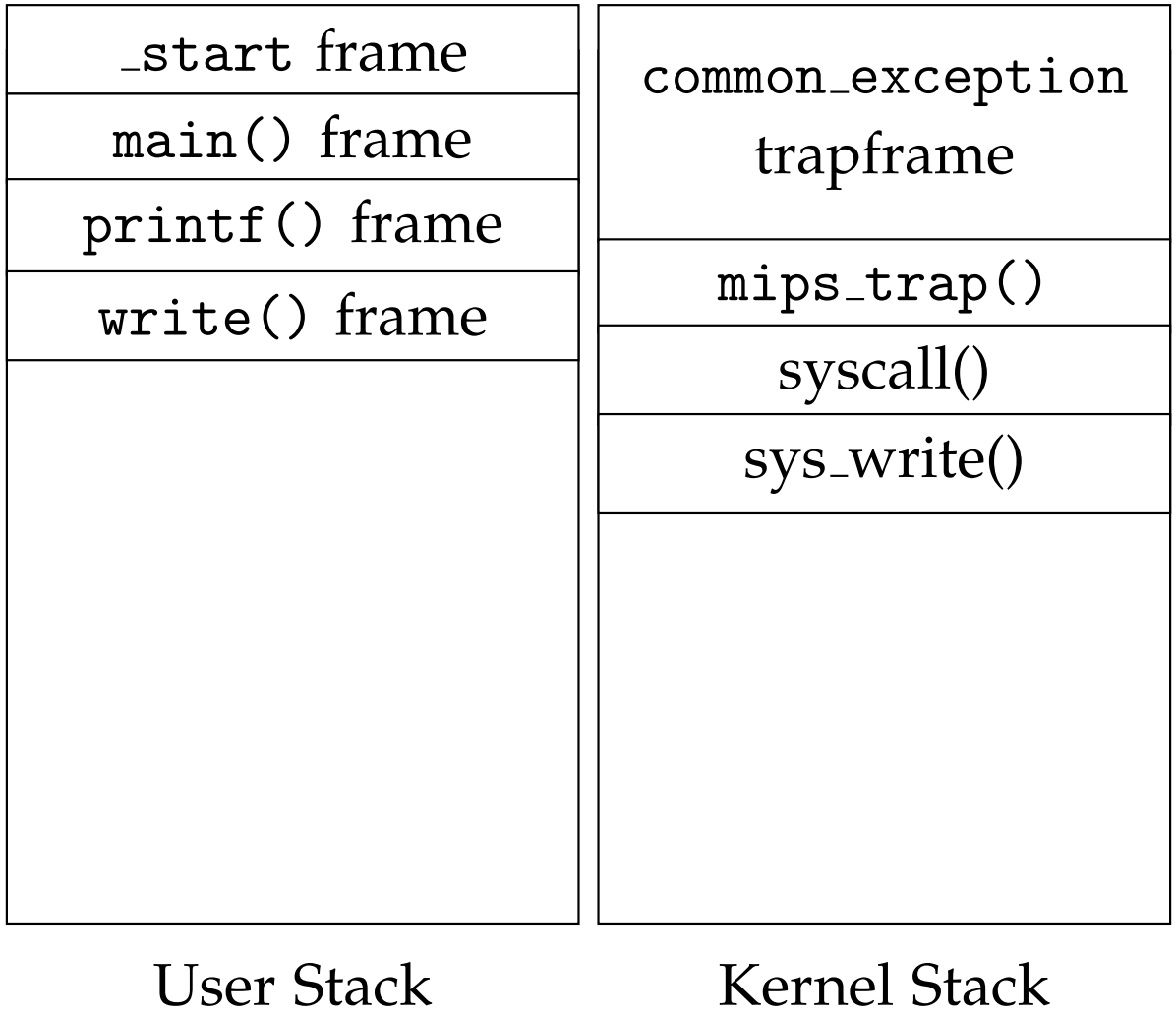
Returning to User Mode

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



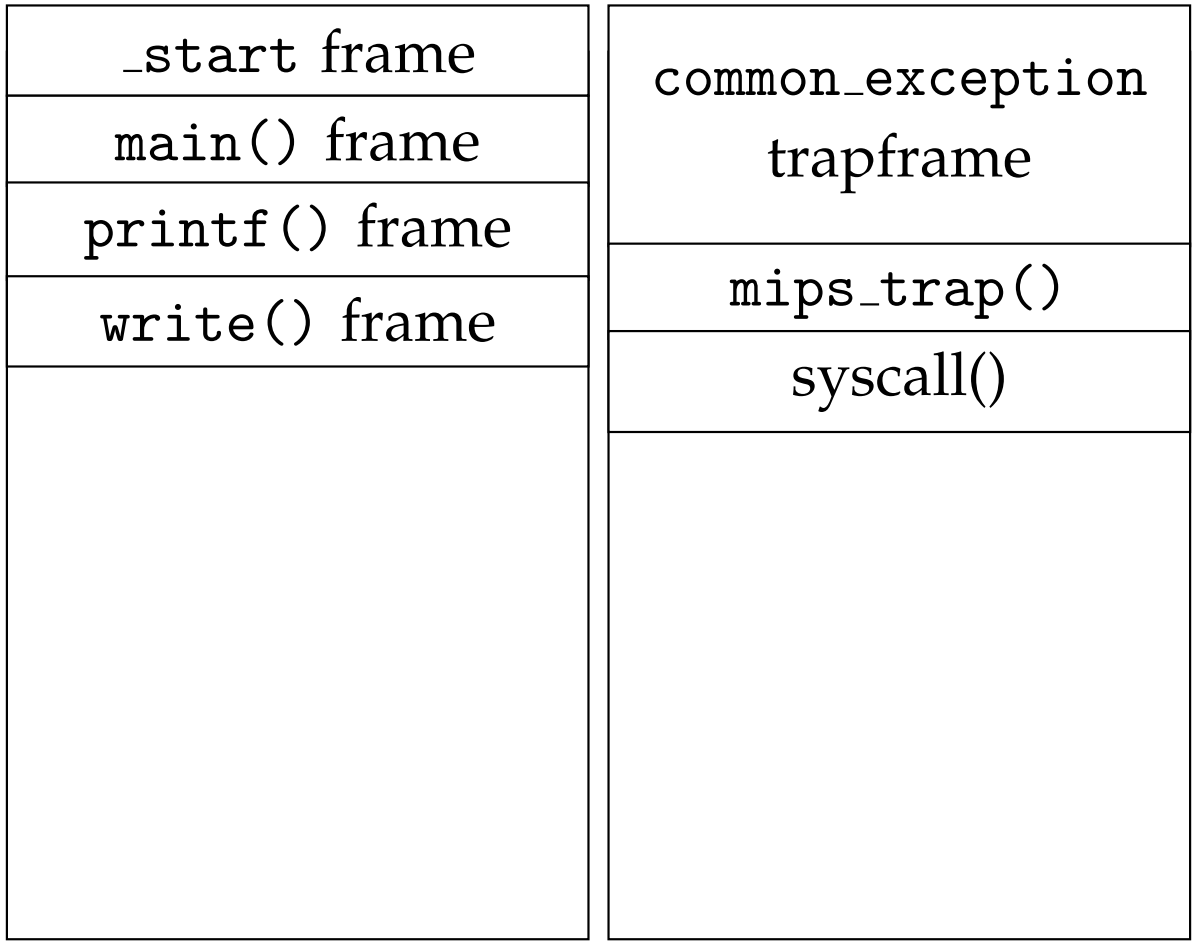
Returning to User Mode

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



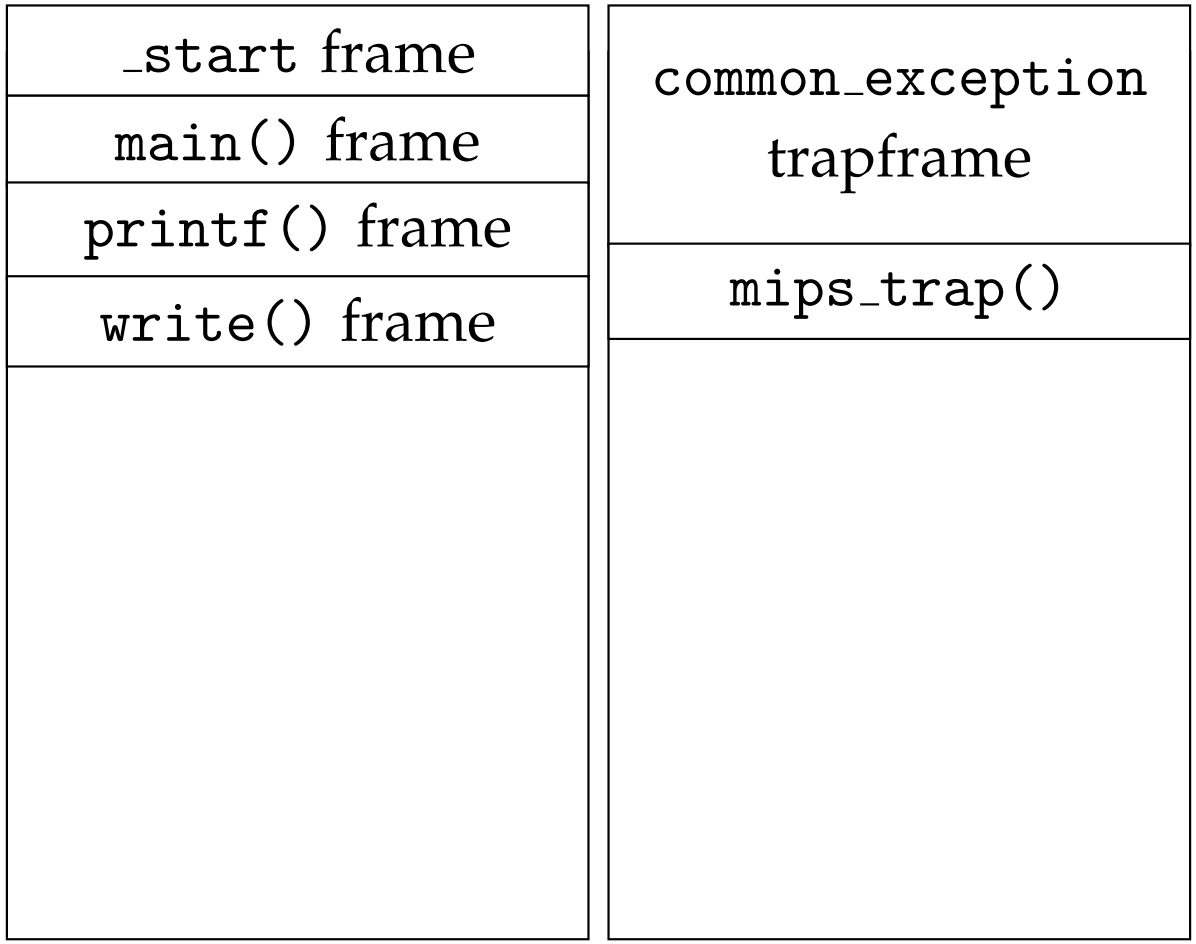
Returning to User Mode

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



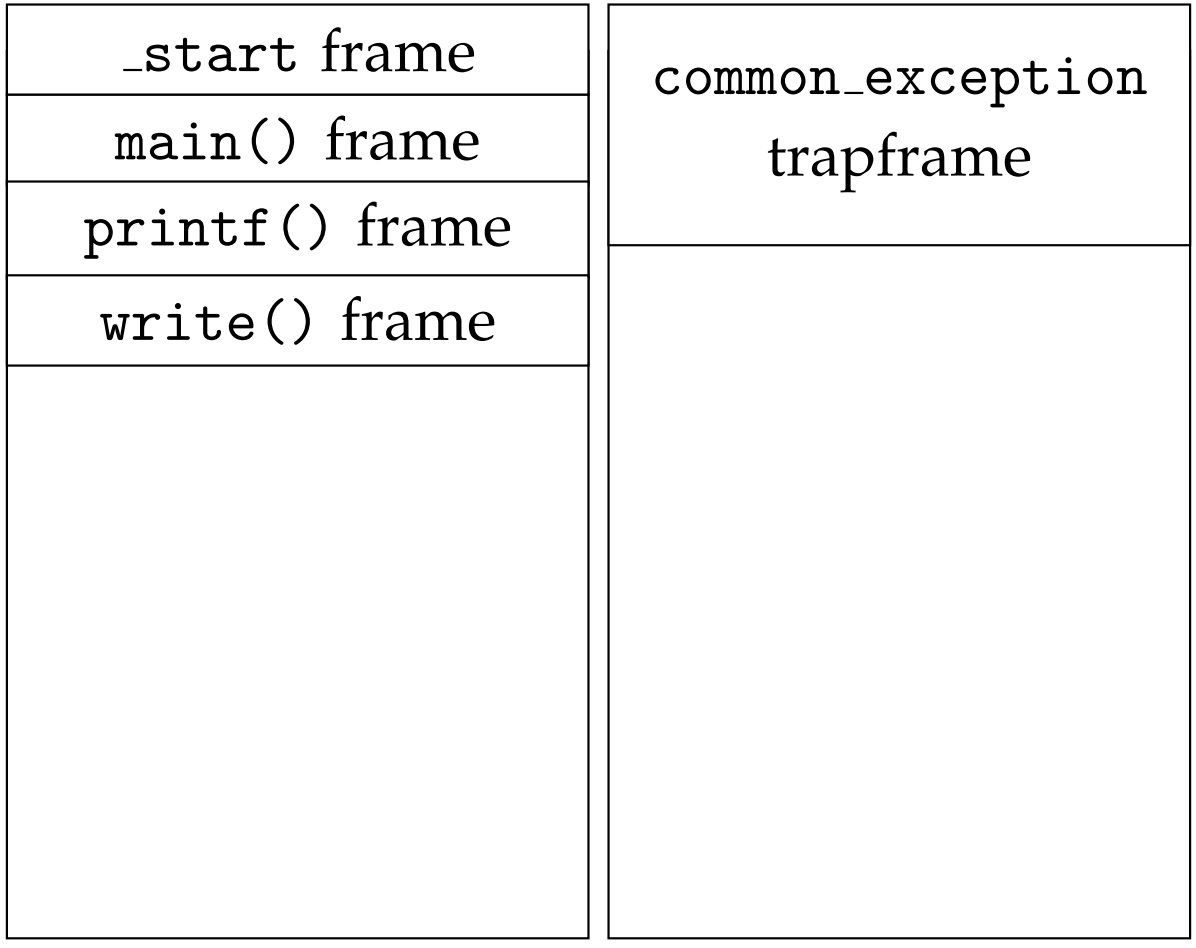
Returning to User Mode

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



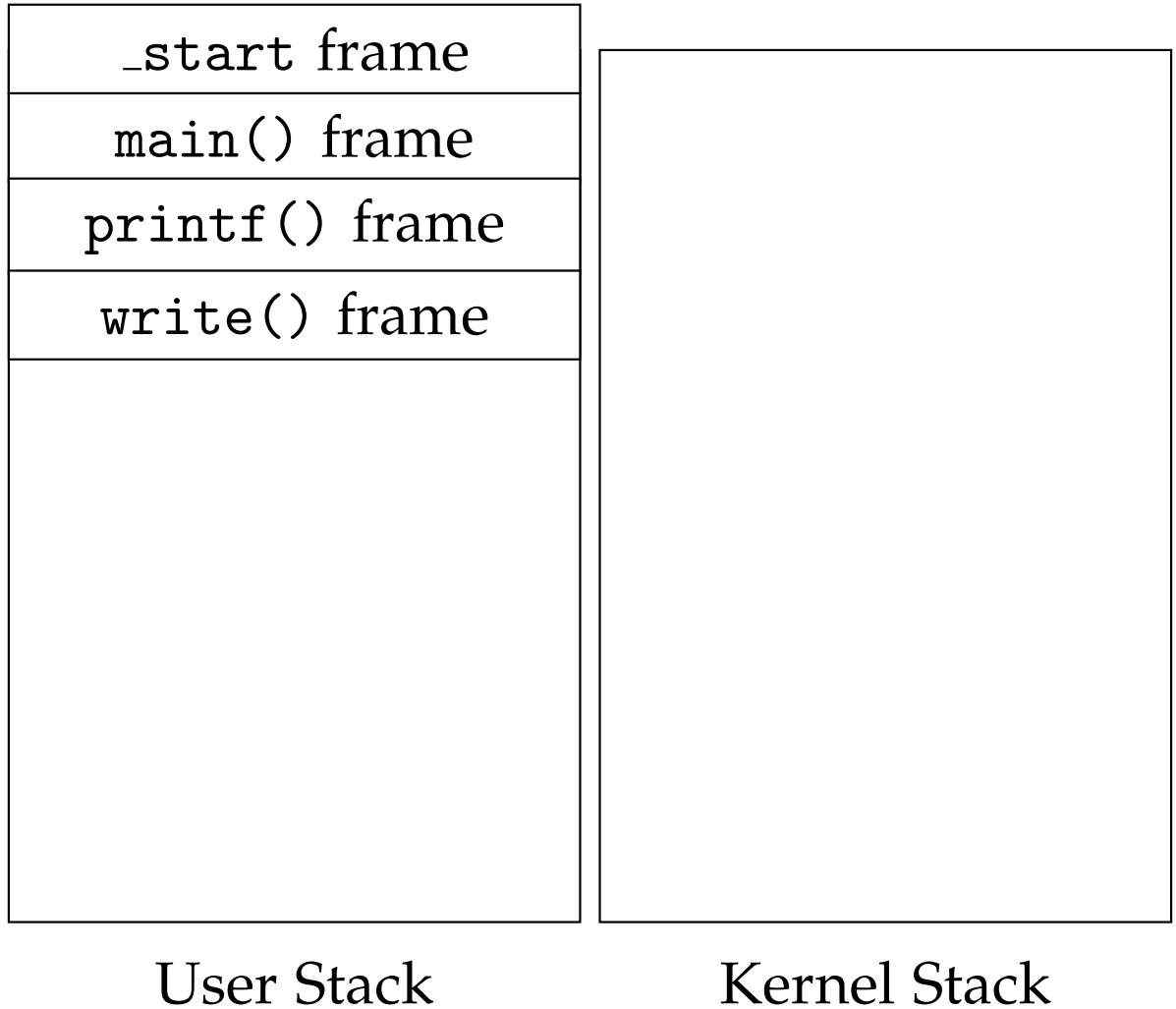
Returning to User Mode

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



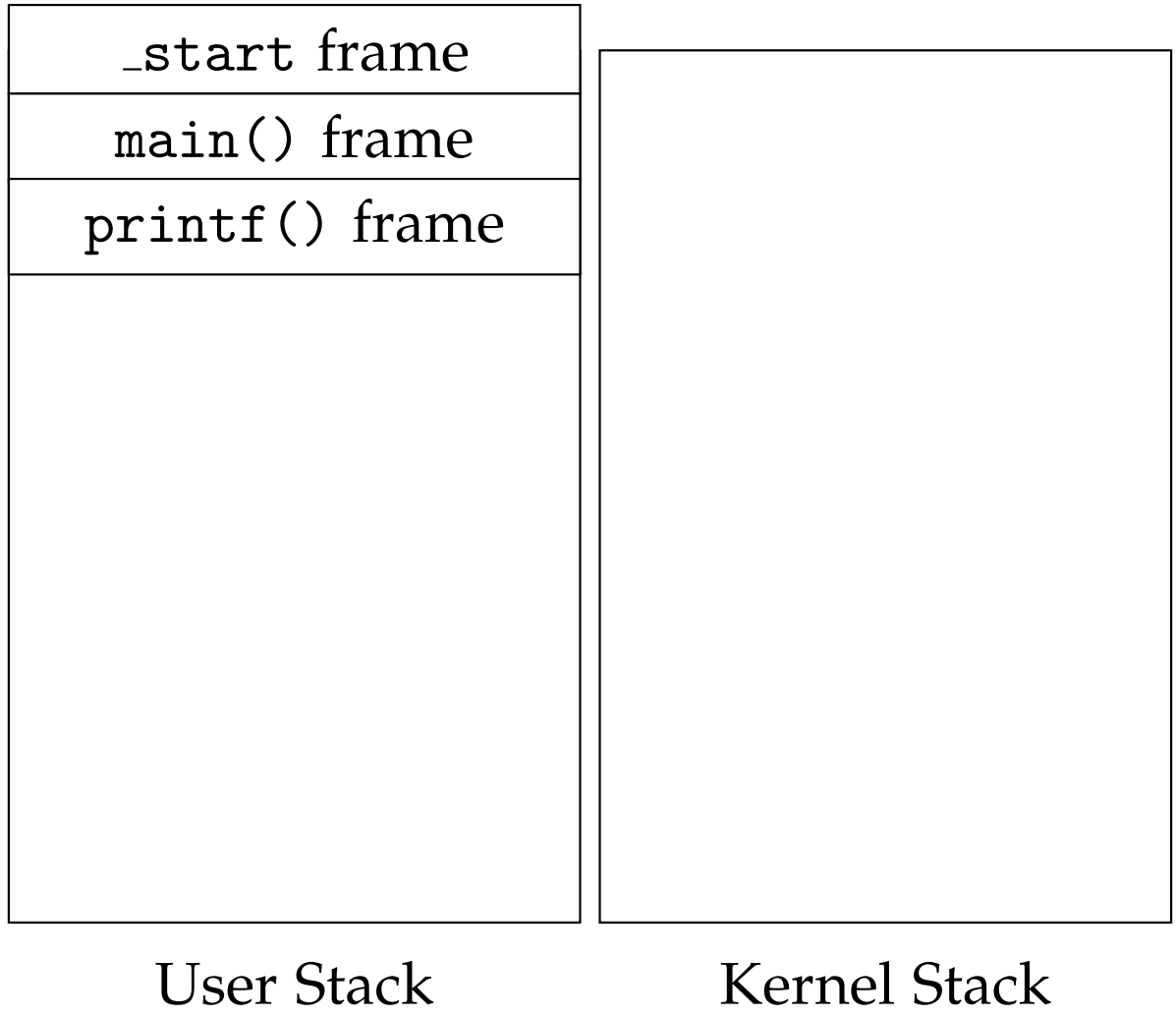
Returning to User Mode

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



Returning to User Mode

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



Outline

1. System Calls
2. Switching Threads/Processes
3. System Call Implementation

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

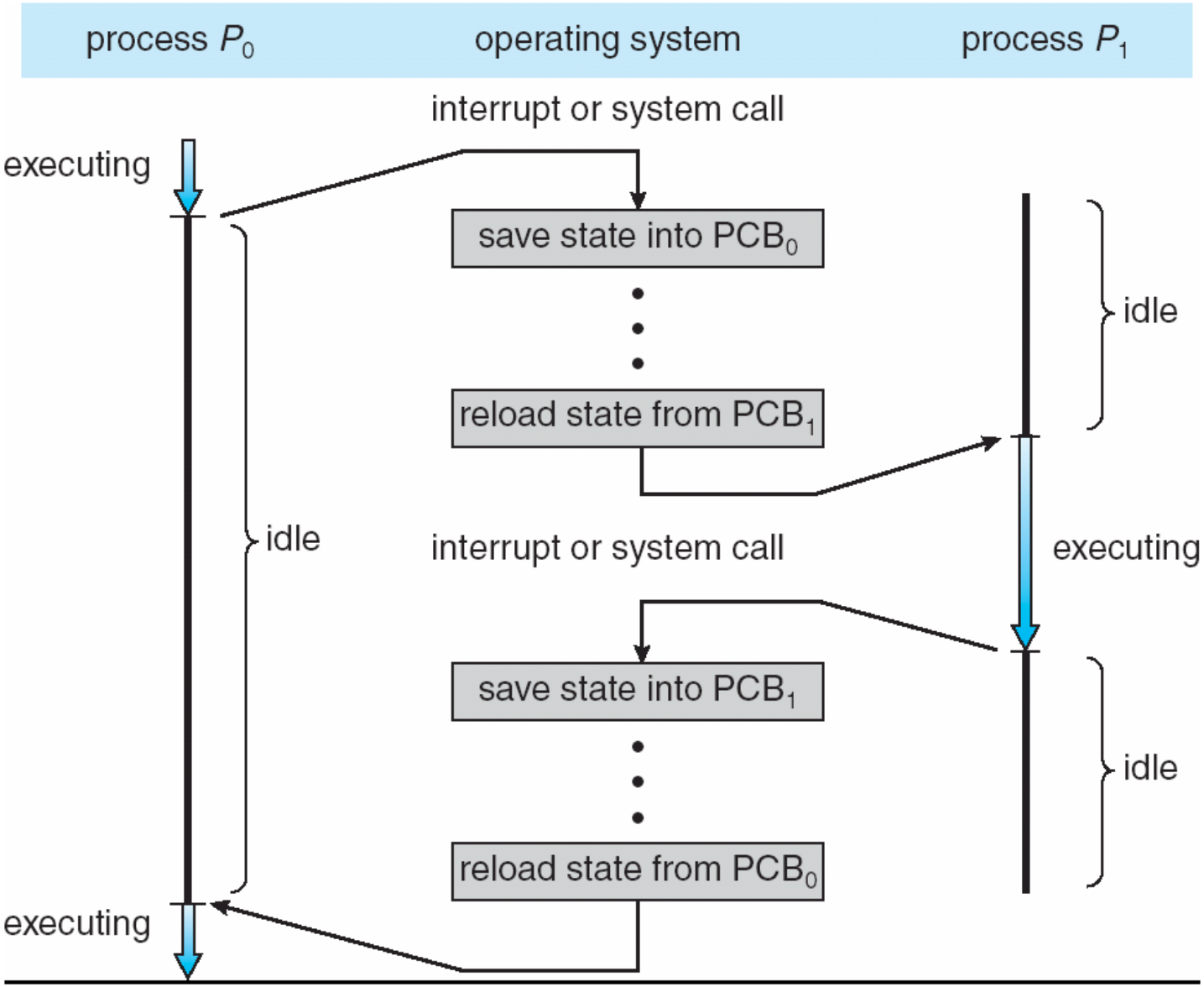
(OS/161 `kern/thread/thread.c`)

- **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

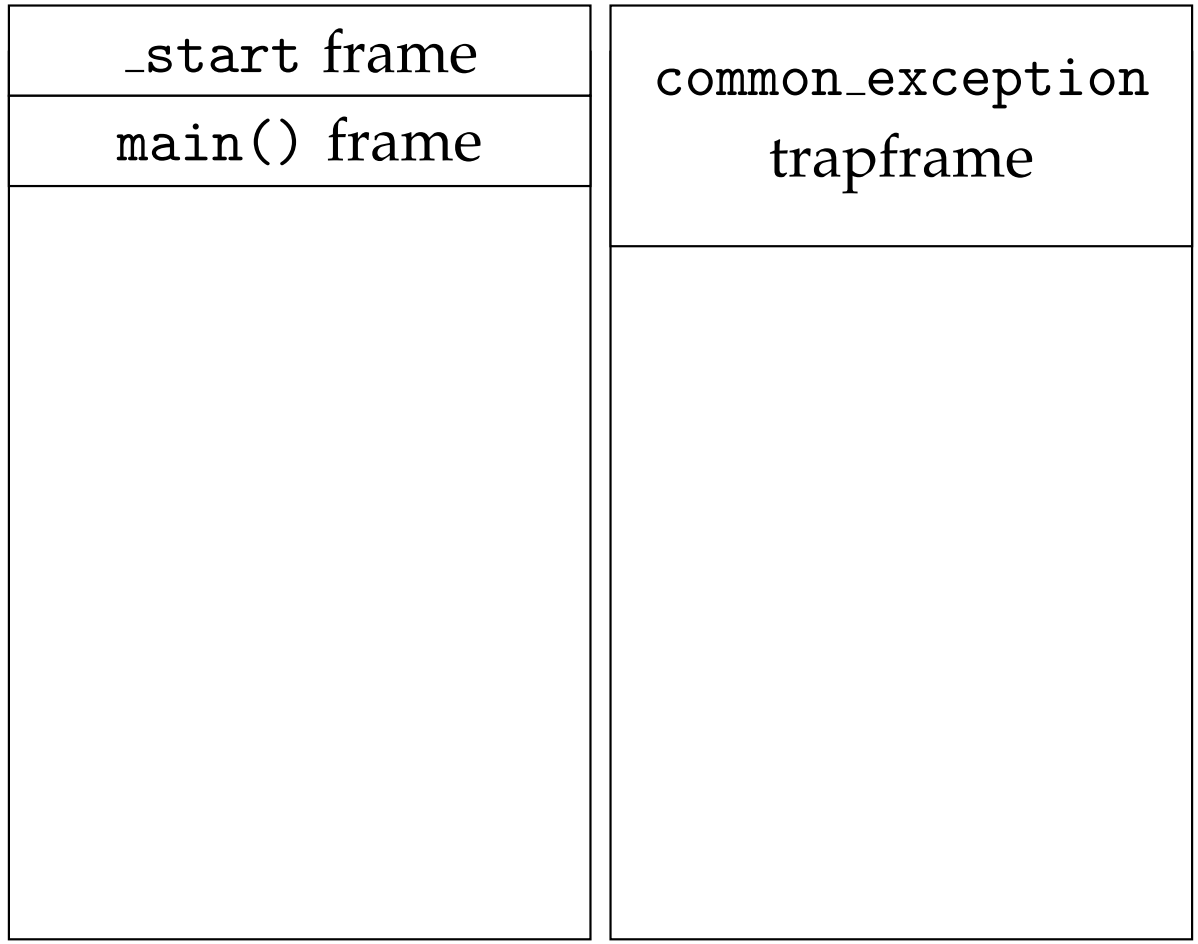


Context switch details

- **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)

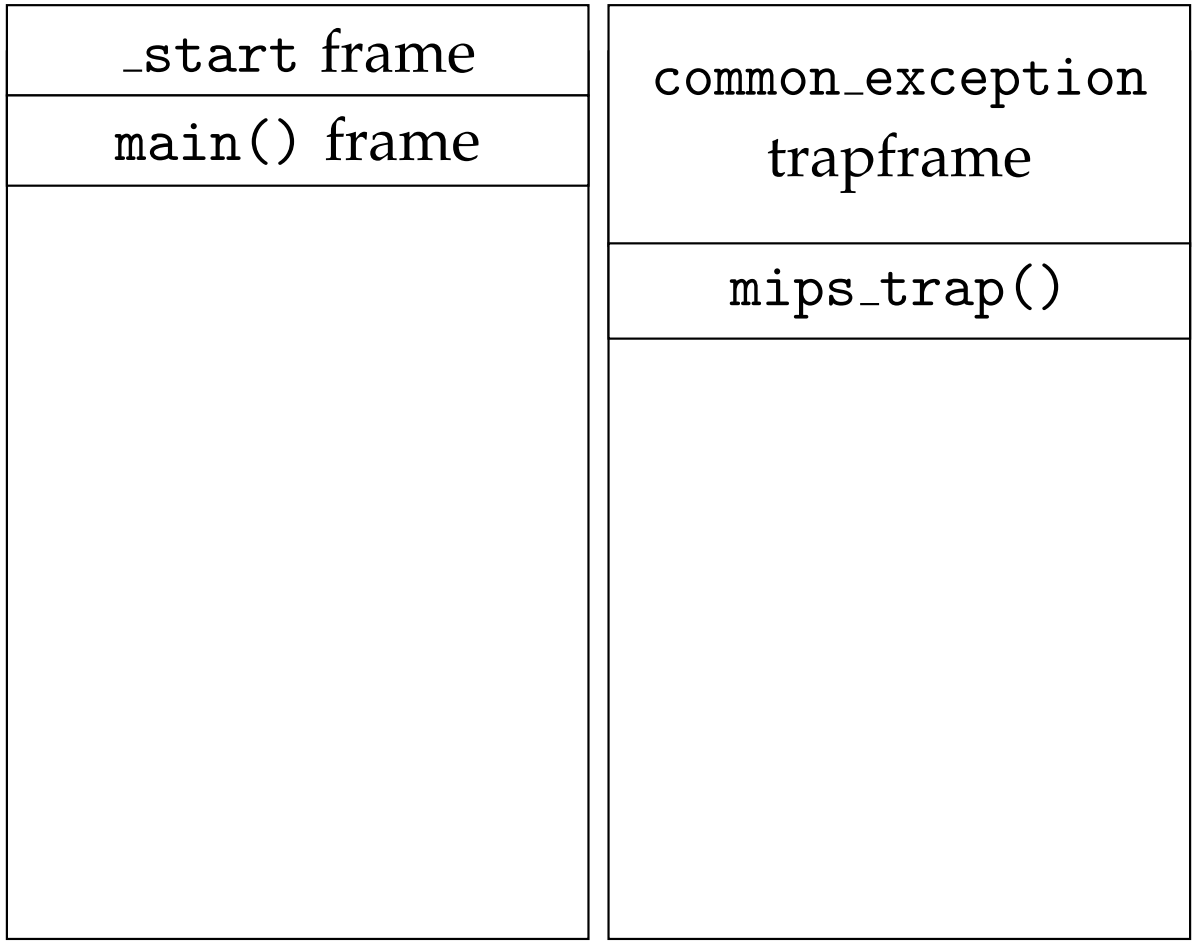
Switching Processes

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



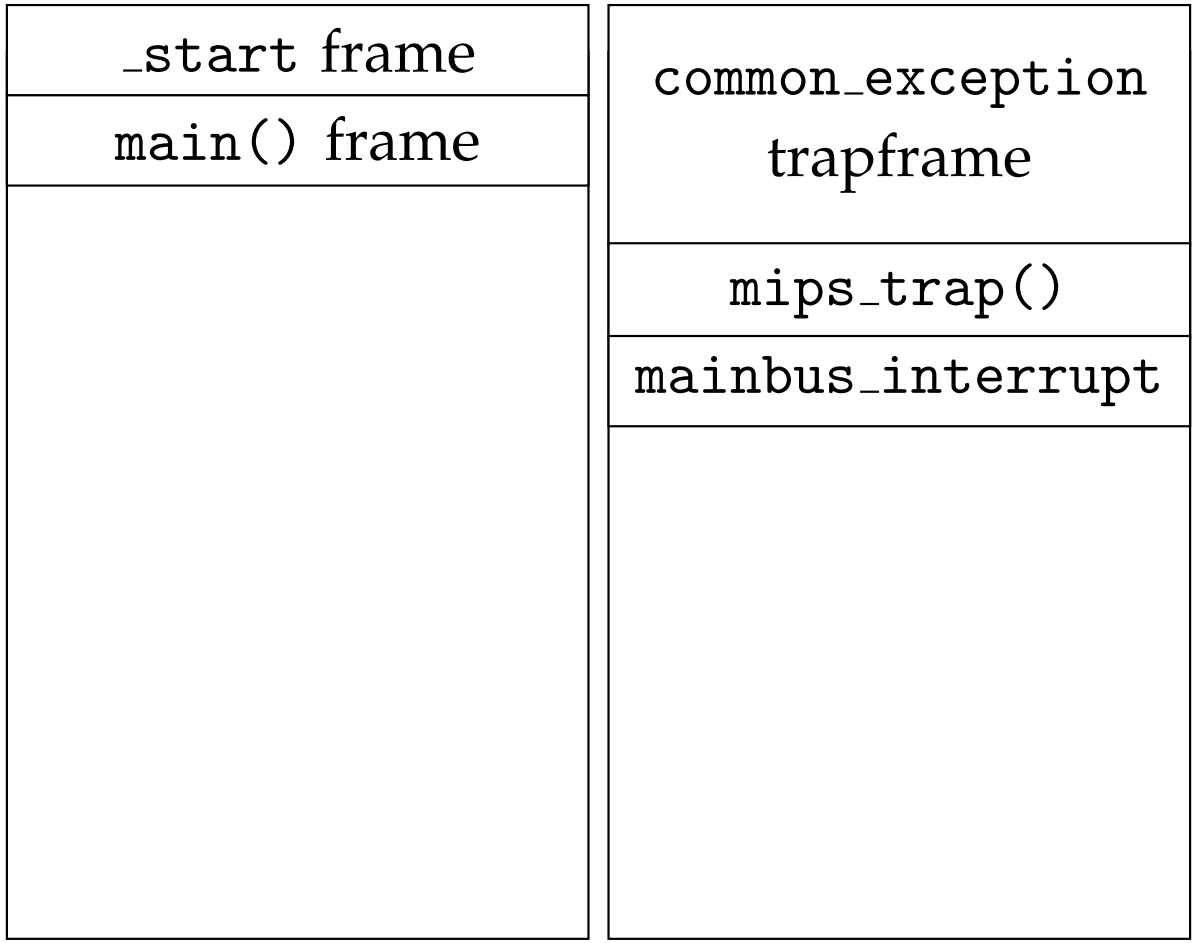
Switching Processes

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



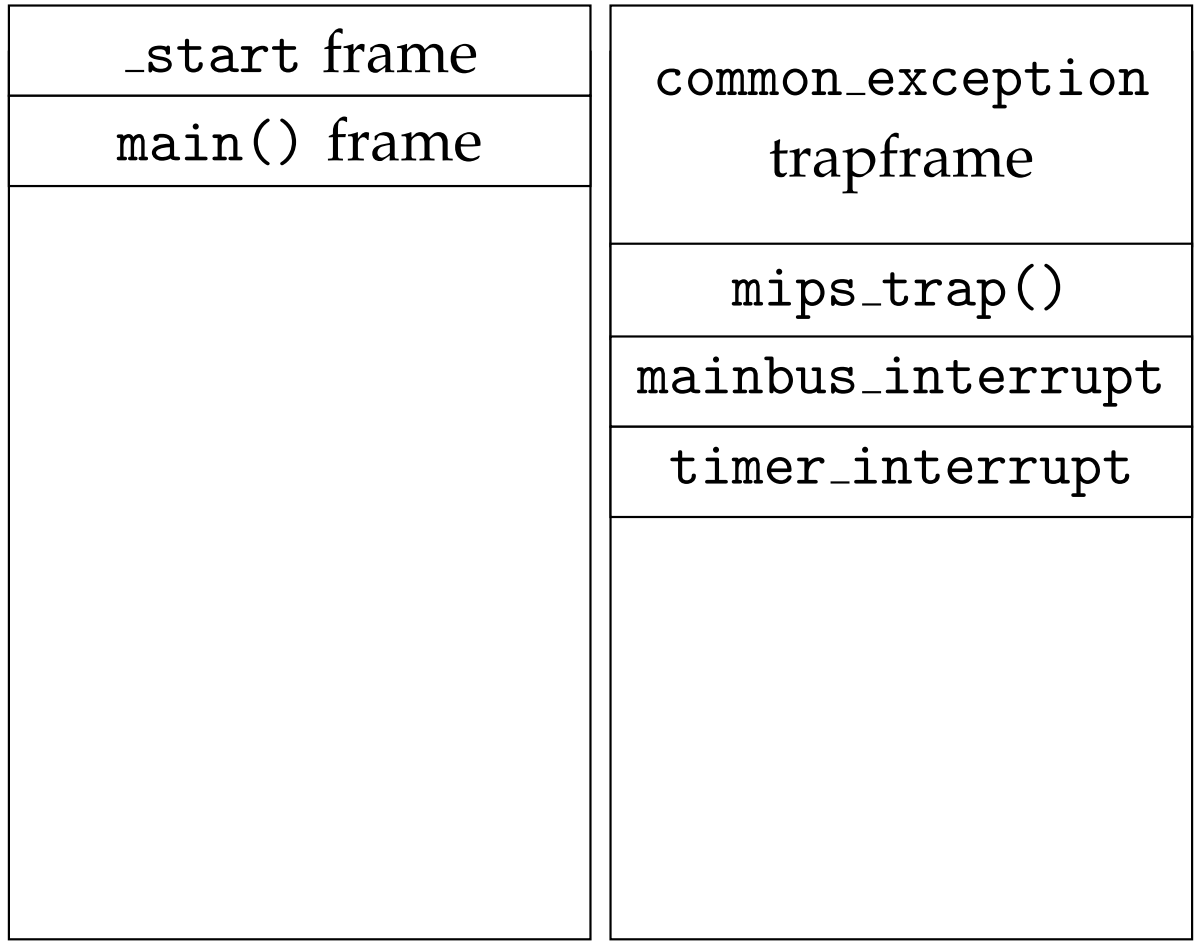
Switching Processes

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



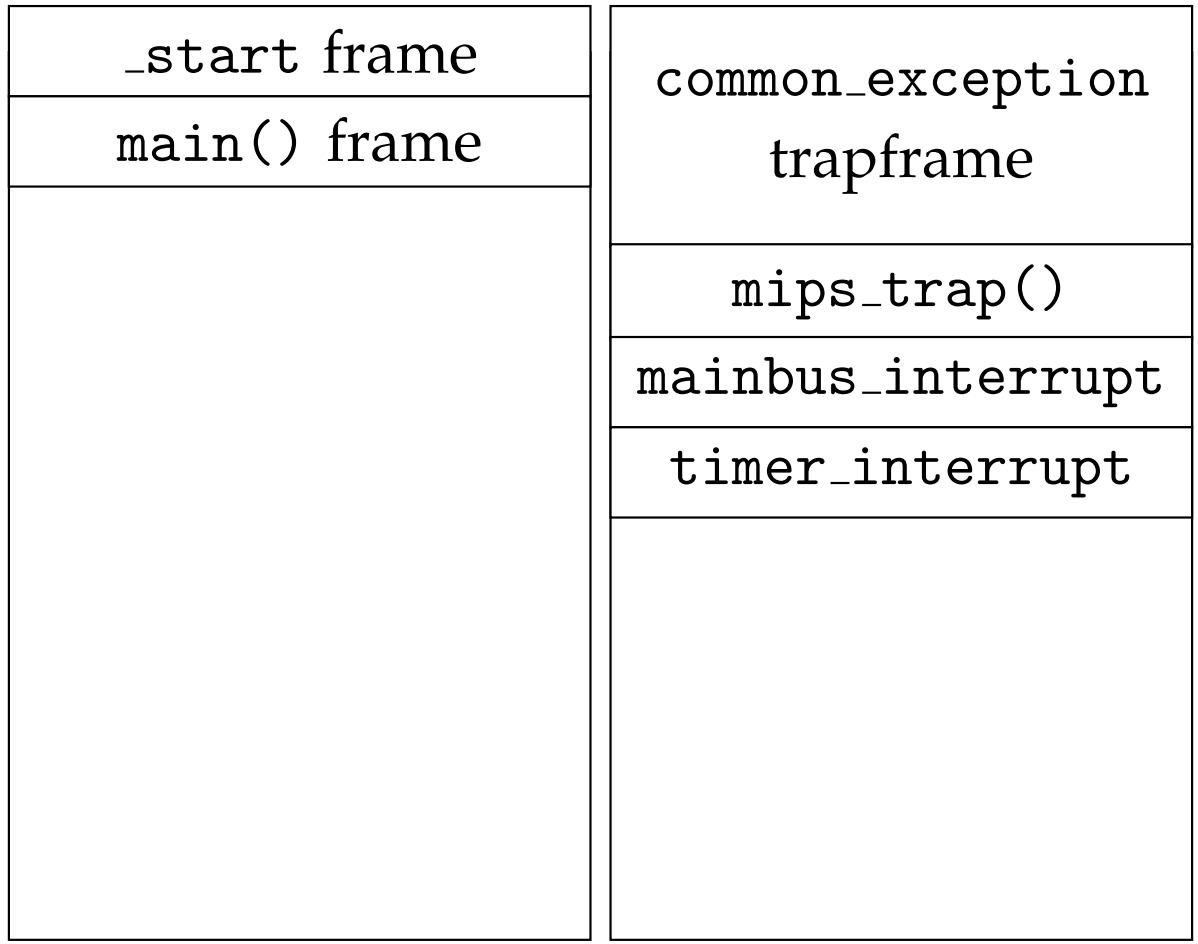
Switching Processes

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



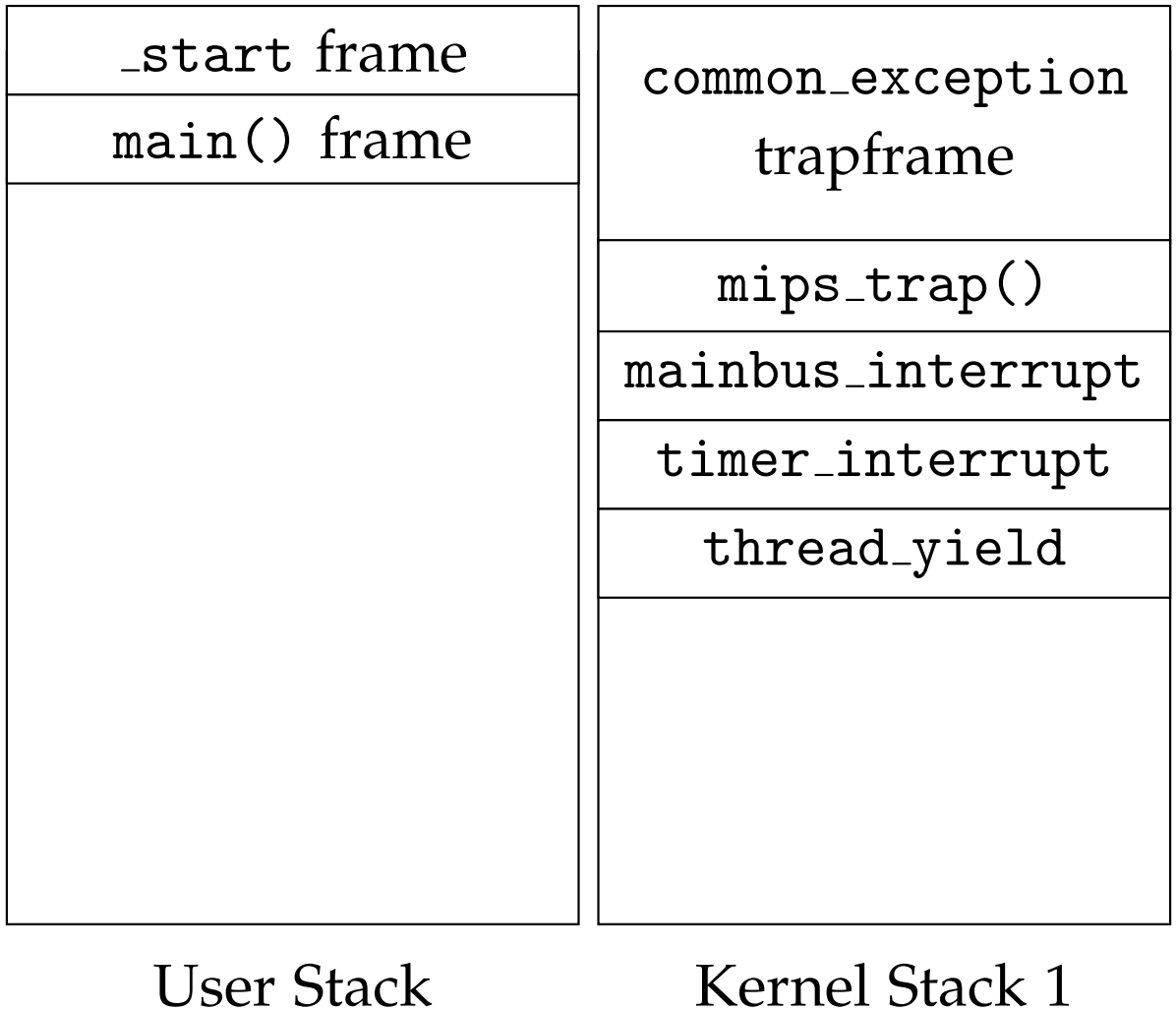
Switching Processes

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



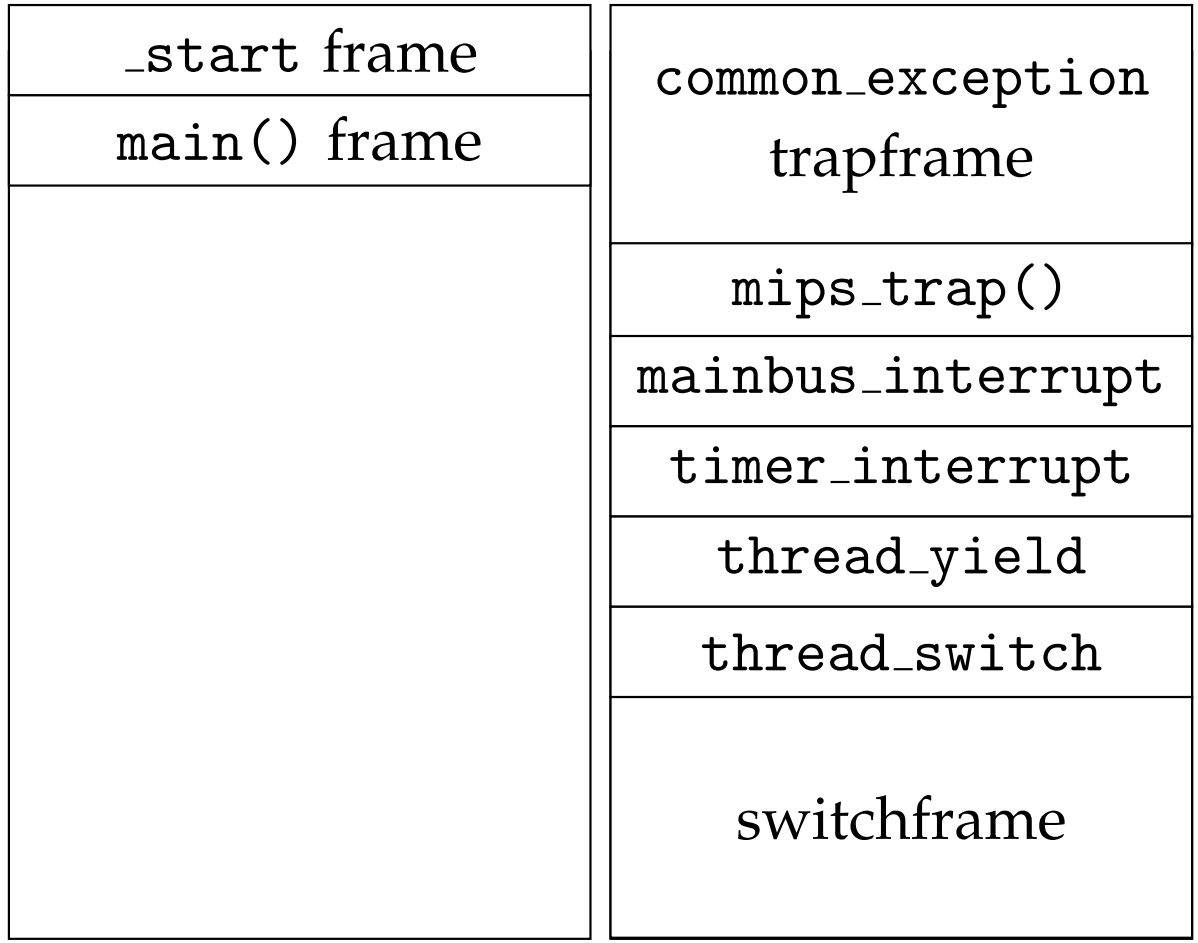
Switching Processes

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



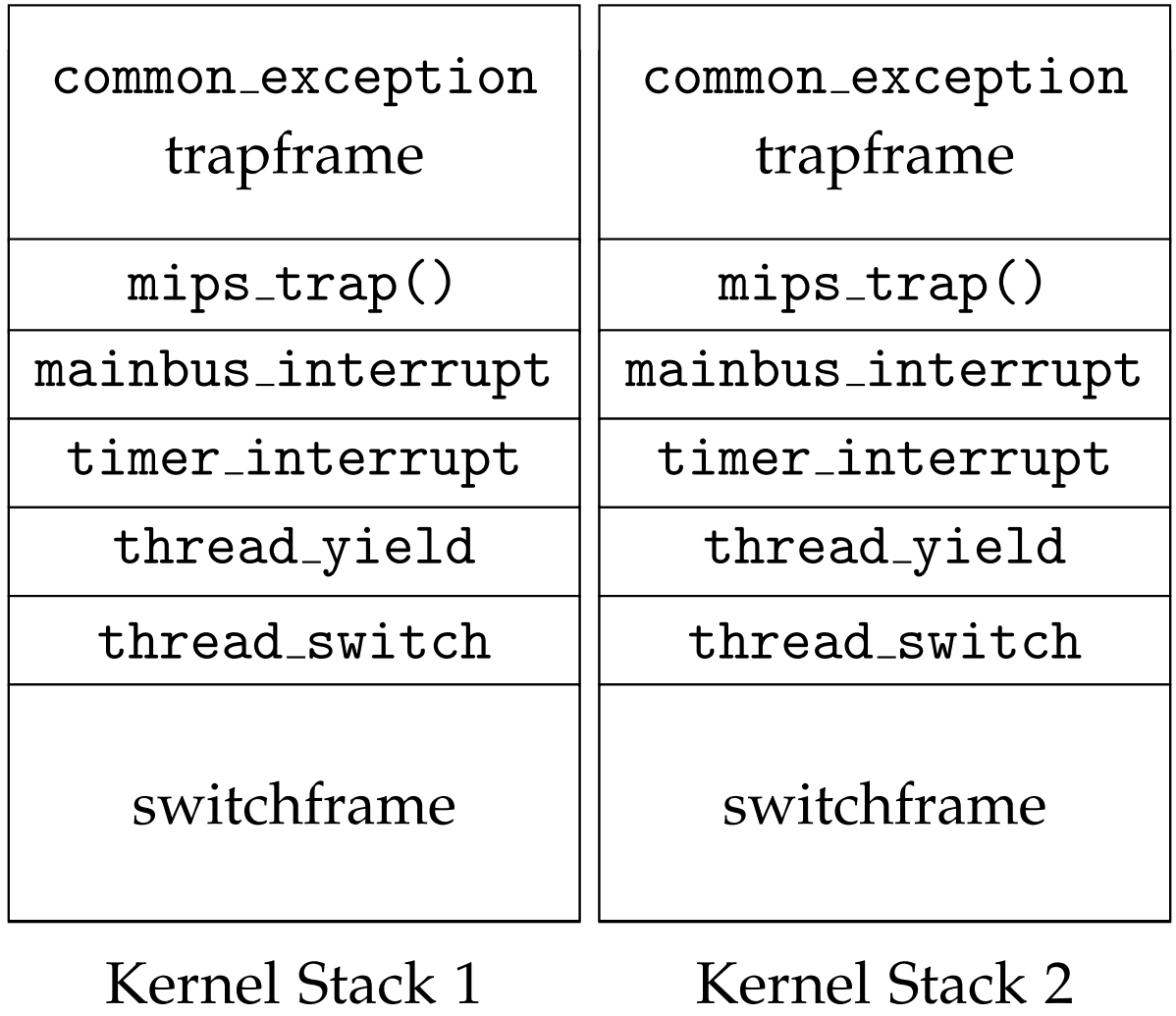
Switching Processes

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



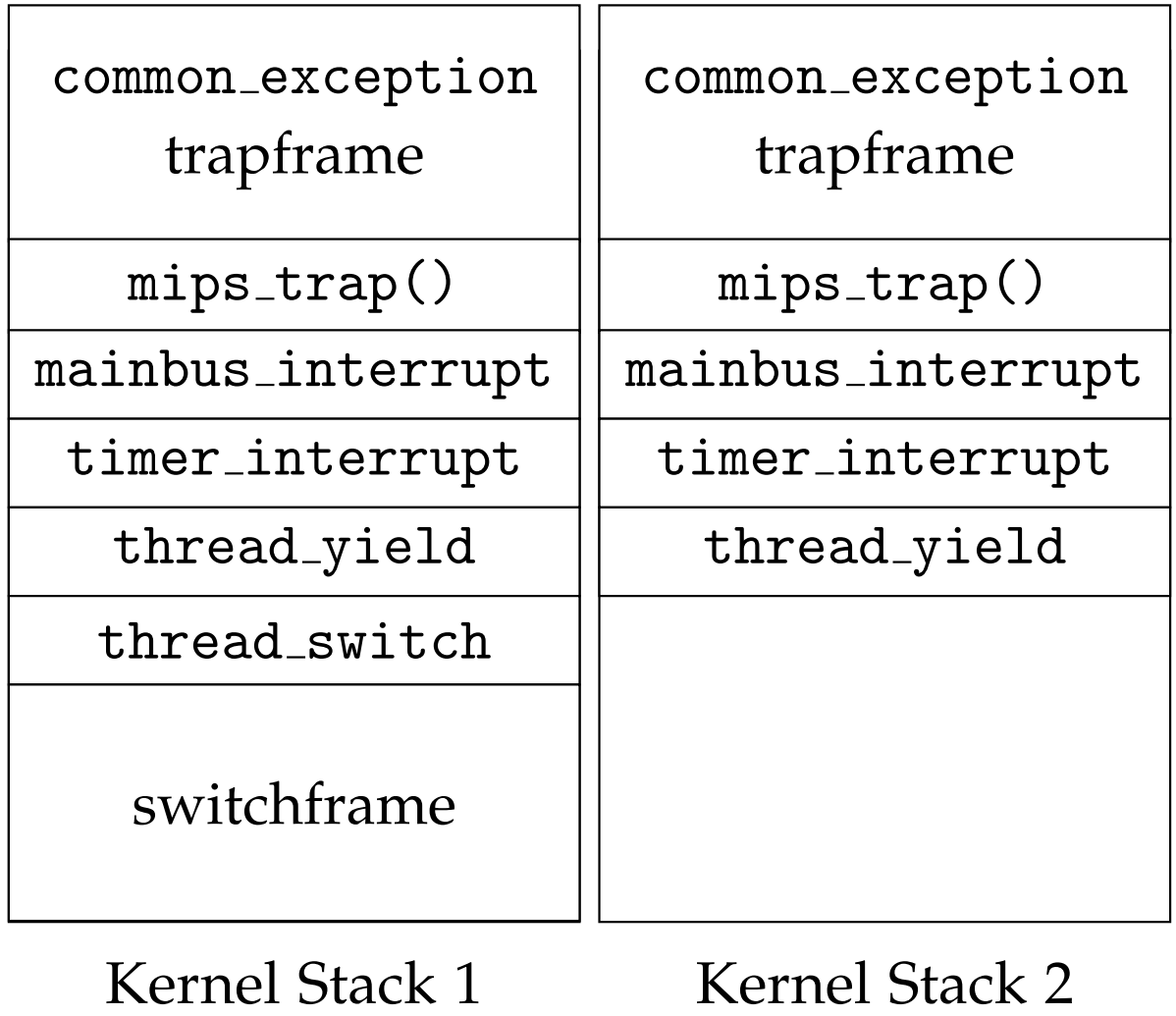
Switching Processes

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



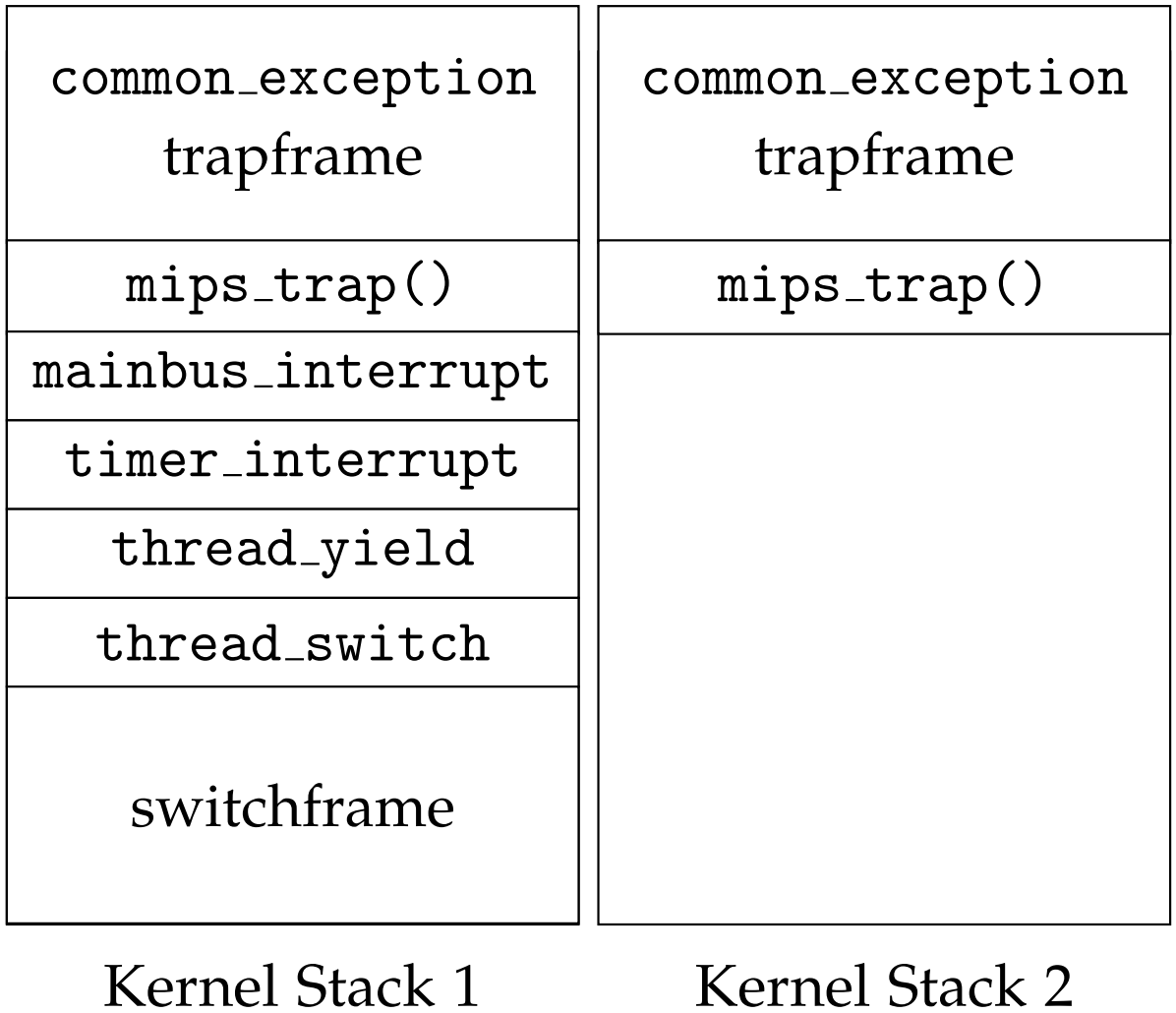
Switching Processes

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



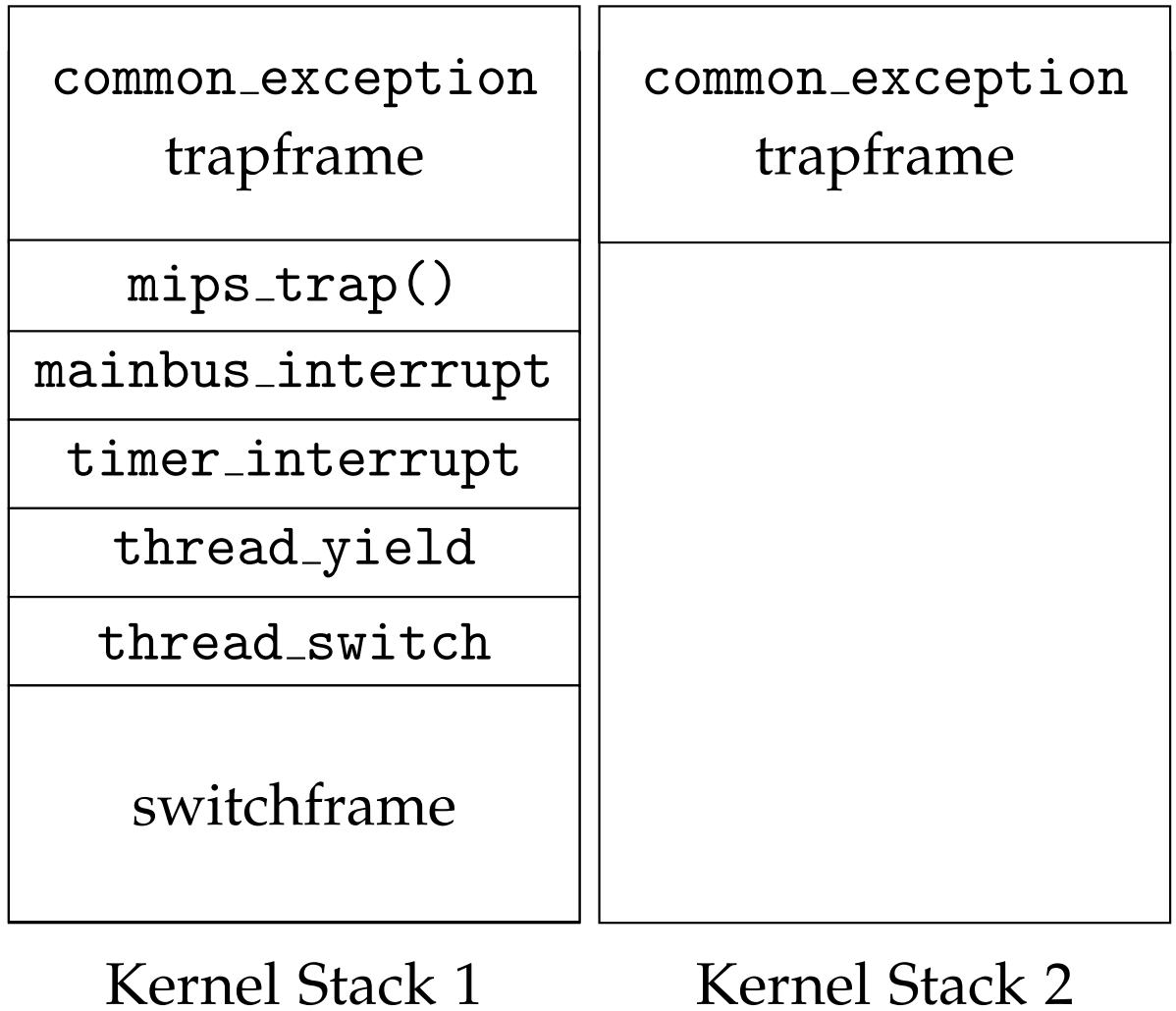
Switching Processes

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



Switching Processes

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



Outline

1. System Calls
2. Switching Threads/Processes
3. System Call Implementation

Creating processes

- `int fork (void);`
  - Create new process that is exact copy of current one
  - Returns *process ID* of new process in "parent"
  - Returns 0 in "child"
- **Creates a new kernel thread** `thread_fork()`
- **Duplicates all process structures**
- **Duplicates trapframe with modified return value**
- **Calls** `mips_usermode()` **to restore trapframe**

Deleting processes

- `void exit (int status);`
  - Current process ceases to exist
  - `status` shows up in `waitpid` (shifted)
  - By convention, `status` of 0 is success, non-zero error
- **Cleans up memory and most resources**
- **Set state to zombie process (no longer runnable)**

Cleaning up processes

- `int waitpid (int pid, int *stat, int opt);`
  - `pid` – process to wait for, or -1 for any
  - `stat` – will contain exit value, or signal
  - `opt` – usually 0 or `WNOHANG`
  - Returns process ID or -1 on error
- **Searches for zombie processes**
- **Retrieves exit status code and frees proc struct**