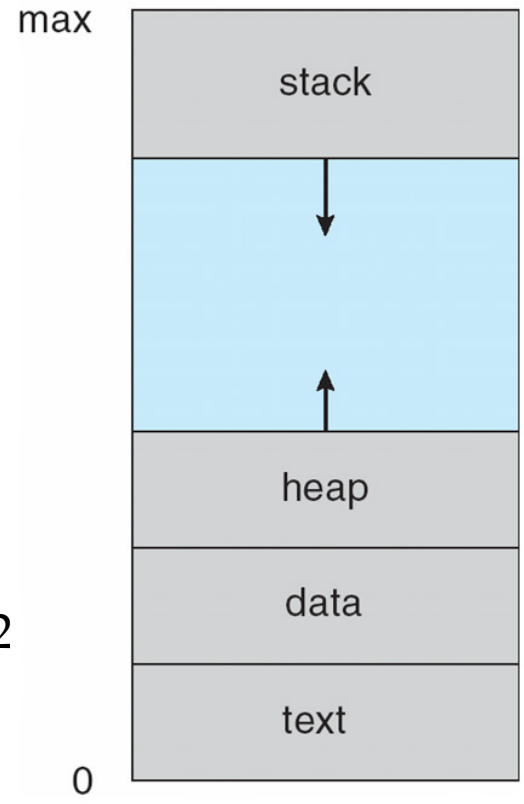
Processes

- **A *process* is an instance of a program running**
- **Modern OSes run multiple processes simultaneously**
- **Examples (can all run simultaneously):**
  - `gcc file_A.c` – compiler running on file A
  - `gcc file_B.c` – compiler running on file B
  - `emacs` – text editor
  - `firefox` – web browser
- **Non-examples (implemented as one process):**
  - Multiple firefox windows or emacs frames (still one process)
- **Why processes?**
  - Simplicity of programming
  - Higher throughput (better CPU utilization), lower latency

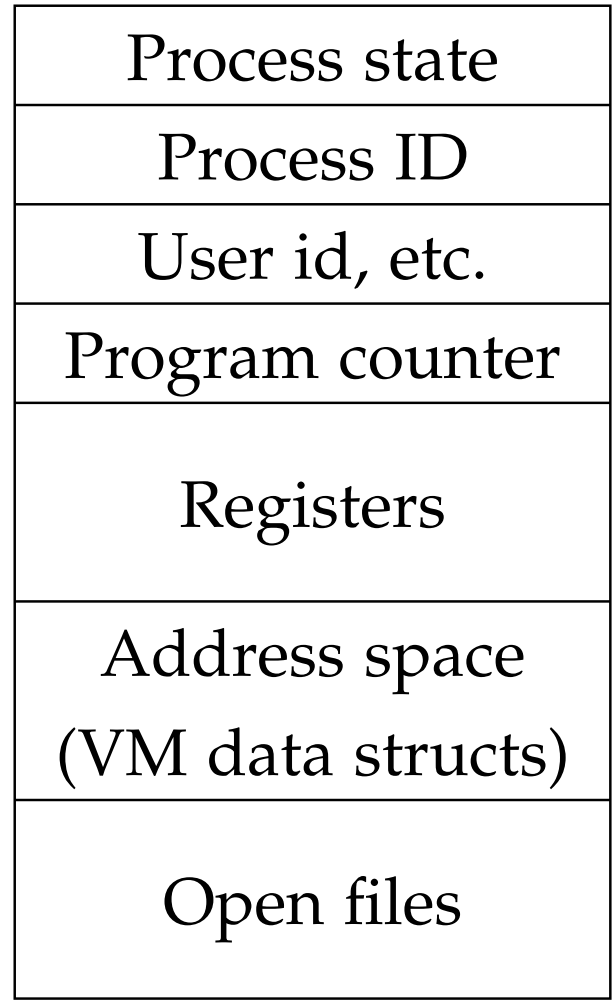
A process's view of the world

- **Each process has own view of machine**
  - Its own address space
  - Its own open files
  - Its own virtual CPU (through preemptive multitasking)
- `*(char *)0xc000` **different in** P_1 **&** P_2
- **Simplifies programming model**
  - `gcc` does not care that `firefox` is running
- **Sometimes want interaction between processes**
  - Simplest is through files: `emacs` edits file, `gcc` compiles it
  - More complicated: Shell/command, Window manager/app.



Implementing processes

- **OS keeps data structure for each proc**
  - Process Control Block (PCB)
  - Called `proc` in Unix, `task_struct` in Linux, and just `struct thread` in OS/161
- **Tracks *state* of the process**
  - Running, ready (runnable), blocked, etc.
- **Includes information necessary to run**
  - Registers, virtual memory mappings, etc.
  - Open files (including memory mapped files)
- **Various other data about the process**
  - Credentials (user/group ID), signal mask, controlling terminal, priority, accounting statistics, whether being debugged, which system call binary emulation in use, ...



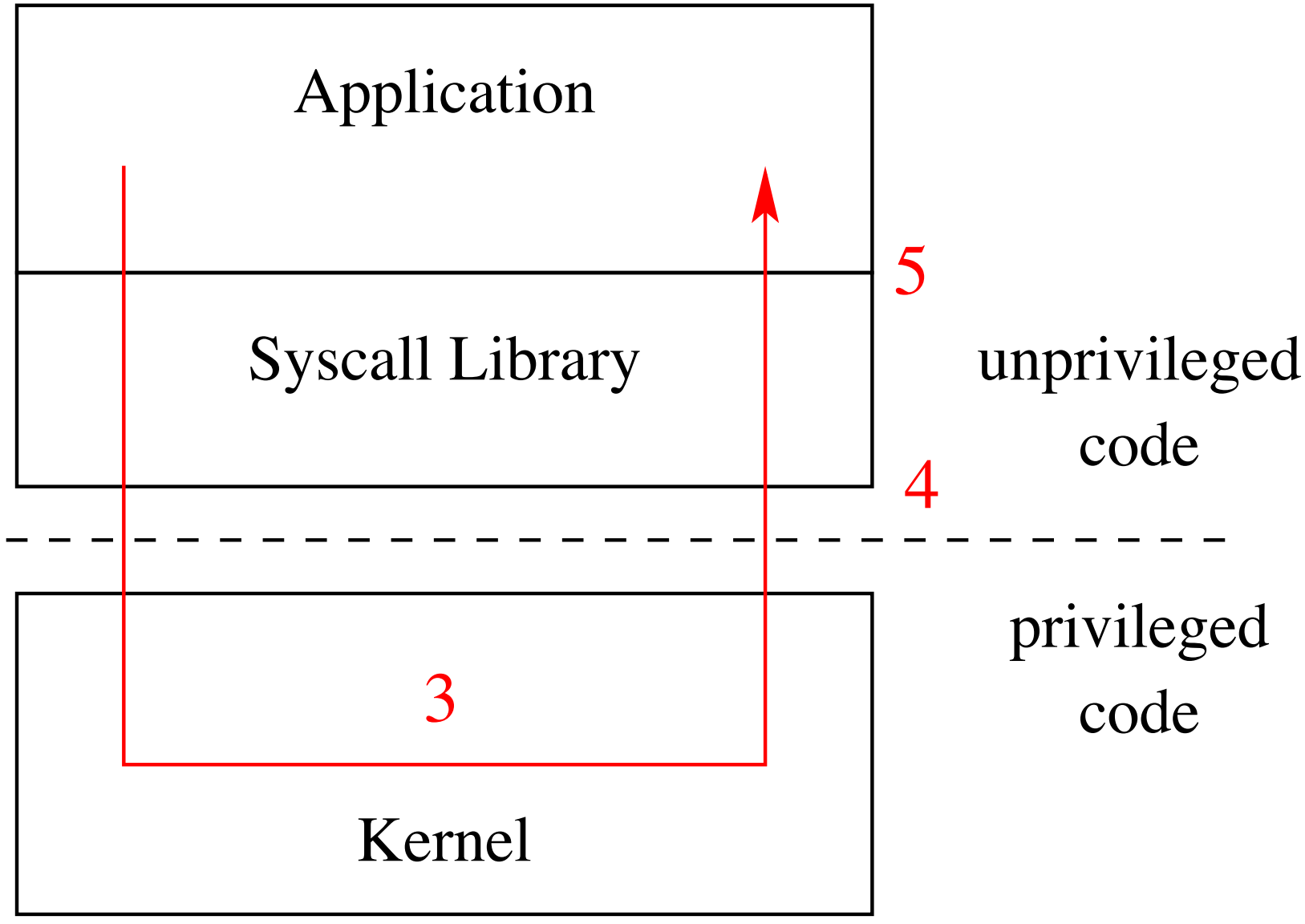
PCB

Next few lectures...

- **Today: Application/Kernel Interface**
- **Details of process and interrupt handler contexts**
- **Virtual Memory Hardware**
- **Virtual Memory Software**

Outline

System Software Stack



System Call Interface

System Calls: The 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
- **We need a mechanism to safely enter and exit the kernel**
  - Applications can't call a syscalls directly!
  - Remember: kernels provide protection

Privilege Modes

- **Hardware provides multiple protection modes (or domains)**
- **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 (e.g., Co-processor 0 on MIPS)
  - 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 are handled through the same mechanism**
- **Some processors specially handle system calls for performance reasons**

```
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**
- **First, an application loads the parameters of the system call into CPU registers**
- **Second, it specifies the system call number in a specific CPU registers**
- **Finally, executes the `syscall` instruction to trigger the `EX_SYS` exception**
  - Many processors include similar instructions
  - For example, x86 contains the `syscall` and/or `sysenter` instruction, but without using the normal exception handler path

Hardware Handling in Sys/161

- **Exception handlers in 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 so frequent that they are typically written in hand optimized assembly, unlike general exceptions**
- **Remember that `0x8000_0000`–`0x9FFF_FFFF` is mapped to the first 512MBs of physical memory**

Hardware Handling Continued

- **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 *trap frame* 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
- **Library wrapper function returns to the application**

Outline

1. Kernel API

2. Calling Conventions

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
  - Result 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
- **Calls are made with the `jal` instruction**
- **Returns are made with the `jr` instruction**

Functions in MIPS

- **A quick review of function calls in MIPS**
- **Functions are usually called with the `jal` instruction**
- **`jal`: Jump-and-link, calls a function and saves the retrn address in `$ra`**

```
foo:
        li $a0, 1

        /* Save caller-save registers */

        jal bar /* Call bar */

        /* Restore registers */

        jr $ra /* Return */
```

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
```

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
```

- **Next time we visualize the stack behavior**