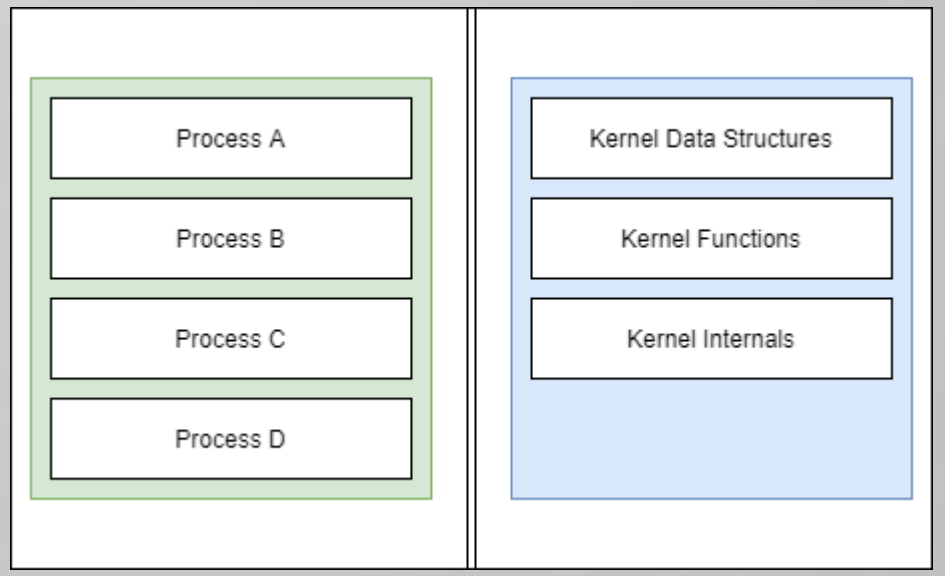
SYSTEM CALLS AND KERNEL SERVICES

CPE / CSC 159: OPERATING SYSTEM PRAGMATICS

GREG CRIST (CRIST@CSUS.EDU)

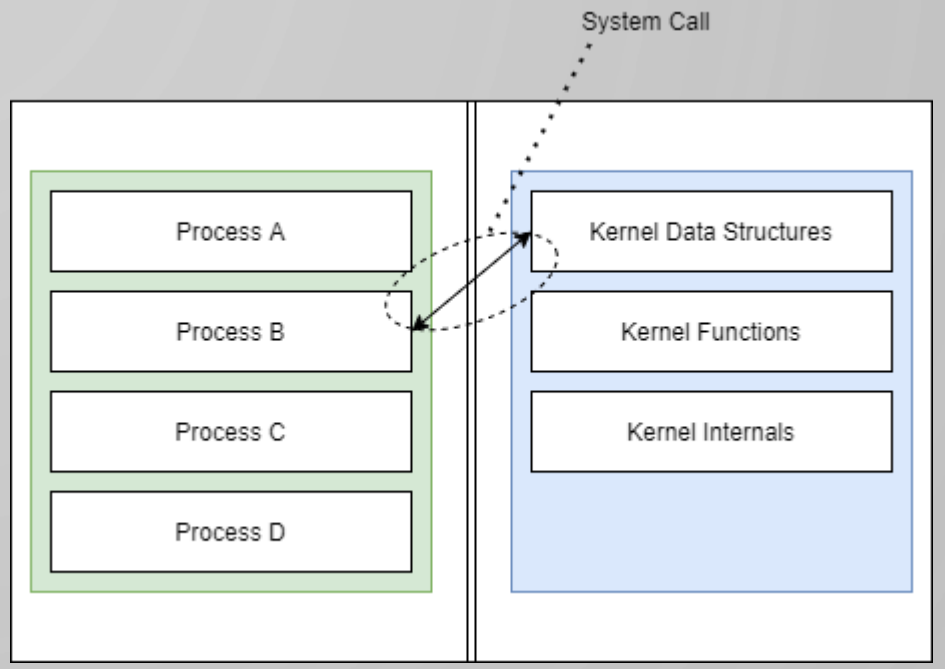
THE KERNEL/USER-SPACE BARRIER

- Maintains a separation between kernel-space and user-space
- The kernel has visibility into the operation of the processes that run, but processes shouldn't have visibility into the kernel operation



THE KERNEL/USER-SPACE BARRIER

- All processes interact with the kernel services through the use of **system calls.**



WHAT ARE KERNEL SERVICES?

- The Kernel maintains a set of services that it provides, such as:
  - Process management
  - Memory management
  - System information
  - Communication
  - Hardware interfacing / drivers

EXAMPLES OF SYSTEM CALLS

- Process Management
  - Creating a new process
  - Exiting a process
  - Putting a process to “sleep”
- Memory Management
  - Memory allocation (ex: malloc, free)
- Communication
  - Interprocess communication
  - Networking
- File management
  - Open / close a file
  - Read from / write to a file
- Retrieving kernel/system information
  - Obtaining the system time
  - Obtaining the running process id
- Hardware Interfacing
  - Input/Output operations

HIGH-LEVEL OVERVIEW OF SYSTEM CALLS

- System calls provide mechanisms to:
  - Exchange data between kernel/runtime processes
  - Enter the kernel context to access the kernel service
  - Return to the user process
- How might this be performed?
  - How to enter the kernel context?
  - When in the kernel context, how to exchange data?
  - How to return to the user process?

SYSTEM CALLS: ENTERING KERNEL CONTEXT

- Kernel context can be triggered via interrupts
  - Hardware driven interrupts (such as the timer interrupt) enter the kernel for processing
  - Software driven interrupts can similarly be used to enter the kernel context
- Software instructions may be executed to enter kernel context as well

SYSTEM CALLS: DATA EXCHANGE

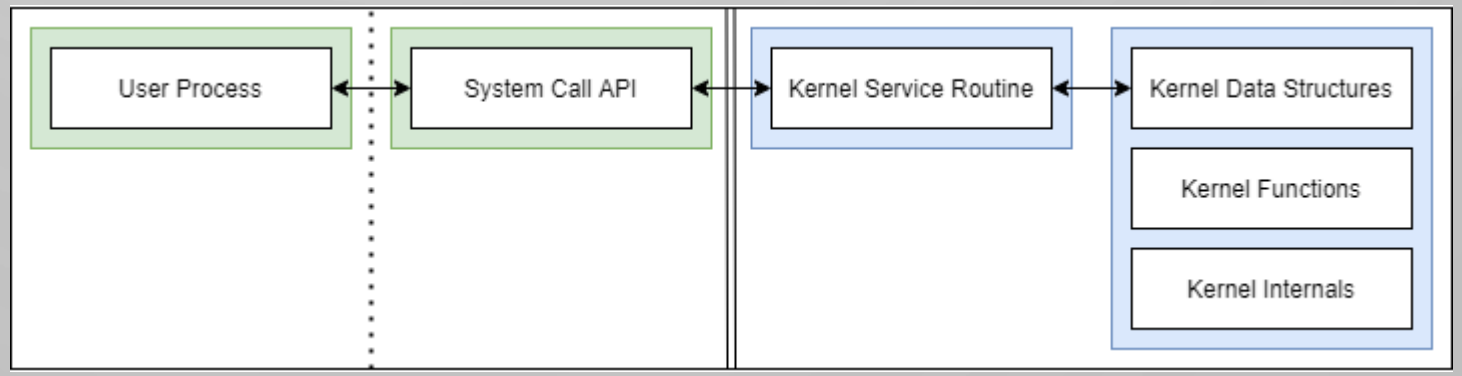
- When an interrupt is triggered, the CPU state is saved for the interrupt service routine to run
- This is leveraged to exchange data using CPU registers
  - Data can be sent from user space to kernel space
  - Data can be received from kernel space to user space

SYSTEM CALLS: RETURNING TO USER SPACE

- Just as a hardware interrupt is triggered and processed, a software interrupt is triggered and processed
- When the interrupt servicing is complete, the user process may be loaded and continue executing, completing the system call

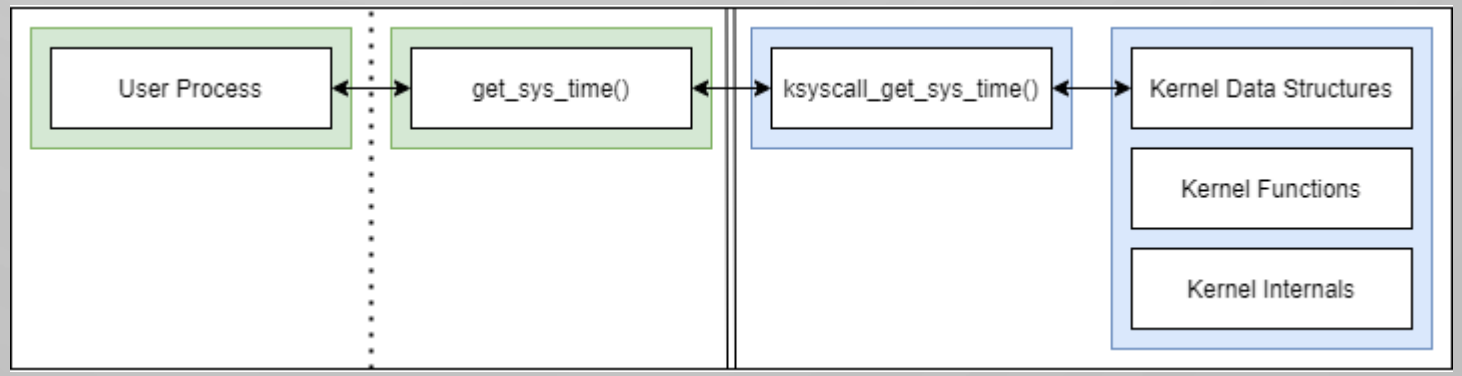
WHAT MAKES UP A SYSTEM CALL?

- System calls are made up of various components:
  - The system call API (function called from user space)
  - The interrupt to trigger the system call
  - Interrupt service routine to enter kernel context
  - System call handler



EXAMPLE SYSTEM CALL: GET_SYS_TIME()

- User process calls an API (function) to retrieve the current system time
  - get_sys_time() function triggers the system call interrupt
  - ISR results in kernel context being entered (kernel_run) and is handled
  - System call is dispatching to the specific system call handler (ksyscall_get_sys_time())
  - Process scheduler loads the process and resumes to user context
  - System call API completes and returns to the calling process



GET_SYS_TIME() – USER CONTEXT

- User process calls the system call API that:
  - Sets registers to indicate to the kernel the system call to perform
  - Triggers an interrupt to enter kernel context
  - When the interrupt returns, sets data and then finally returns to the caller

```
void user_proc(void) {
    int current_time;
    current_time = get_sys_time();
    printf("System time is: %d\n", current_time);
}
```

```
int get_sys_time() {
    int time;
    asm("movl %1, %%eax;"
        "int $0x80;"
        "movl %%ebx, %0;"
        : "=g" (time)
        : "g" (SYSCALL_GET_SYS_TIME)
        : "eax", "ebx");
    return time;
}
```

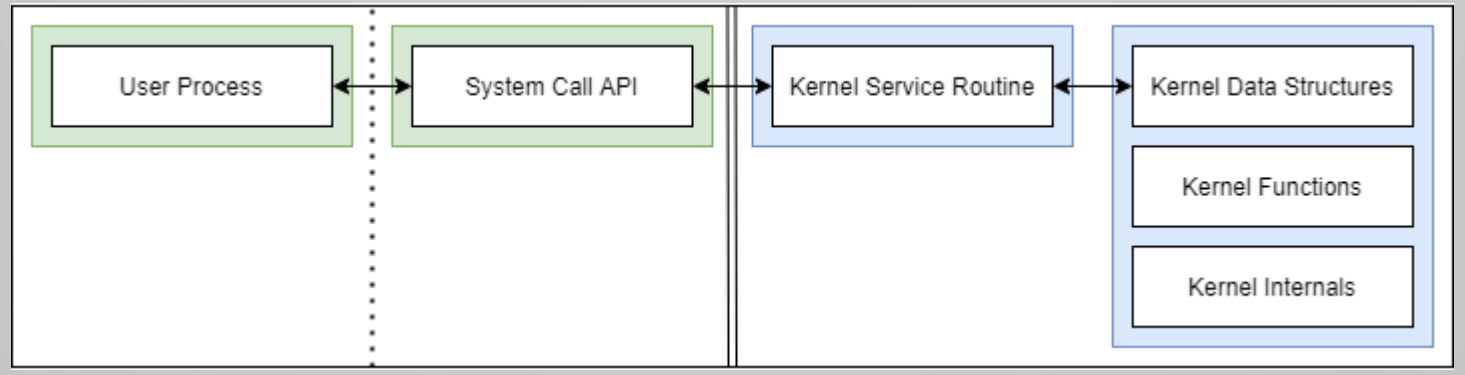
GET_SYS_TIME(): KERNEL CONTEXT

- In the kernel:
  - Syscall Interrupt Service Routine decodes the system call
  - Dispatches to the kernel service routine to process
  - Continues kernel run loop
  - Process is loaded / restored -> CPU registers are set

```
void kisr_syscall(void) {
    int syscall = pcb[active_pid].trapframe_p->eax;
    if (syscall == SYSCALL_GET_SYS_TIME) {
        ksyscall_get_sys_time();
    }
}
```

```
void ksyscall_get_sys_time() {
    pcb[active_pid].trapframe_p->ebx = system_time;
}
```

REVISITING THE SYSTEM CALL CONTEXT SWITCH



- All system calls trigger the context switch from user context to kernel context and then back to user context
- Only mechanism for data exchange is via CPU registers
  - Consider: what can you pass through registers?

SYSTEM CALLS: BEYOND 80386

- Different CPU architectures provide different mechanisms; we are looking at 80386 (subset of larger x86 instruction set)
  - x86: use a software defined interrupt to enter kernel context
  - IA-32 (Intel Pentium) introduced the SYSENTER instruction to trigger a system call
    - SYSEXIT instruction to return to the caller
  - x64 introduced the SYSCALL instruction to trigger a system call
    - SYSRET instruction to return to the caller
  - ARM64: uses the SVC instruction (supervisor call) to trigger a system call
- Dedicated instructions are used to for speed/simplicity
  - Interrupts are generally considered expensive
  - Dedicated instructions can perform additional work versus needing to write additional code