What is the difference between Kernel Mode and User Mode?

1. Faults in user space can be captured by the kernel
2. Privileged instructions, e.g., I/O instructions, can only be issued (when the CPU is) in the kernel mode
3. Some memory address space (i.e., the kernel space) can only be accessed in the kernel mode; this portion of memory stores the kernel code and data
Why are Protection Rings needed?

- **Fault isolation**: the program crash can be captured and handled by a lower ring
- **Privileged instructions** can only be issued in a privileged ring (e.g., ring 0), which makes resource management, isolation and protection possible
- “Privileged” memory address space (e.g., the kernel space) can only be accessed in a privileged ring
Given an X86 CPU, how do you tell whether the CPU is in the kernel mode or user mode

The lowest two bits in the CS (Code Segment) register indicate the Current Privilege Level of the CPU. E.g., 00 means that the CPU is in ring 0.
Previous class...

What if the user mode code tries to execute privileged instructions?

Whenever a privileged instruction is executed, the CPU checks whether it is in the kernel mode; if not, an exception (e.g., in x86, a General Protection exception) is triggered to end the current process.
Given that I/O instructions can only be executed in the kernel mode, how does a user program perform I/O?

System calls. When a system call is invoked, the CPU mode switches to kernel mode and CPU can thus execute privileged instructions, such as I/O instructions.
System calls in Linux

INT 0x80/SYSENTER are instructions used to issue system calls.

System call dispatch table, which is an array of addresses of system service functions.
Outline

• Process State Transition
• Call Stack and Calling Convention
• Execution Context
• Process Control Block
• Process Switch and Timer Interrupts
• Processes vs. Threads
Three basic process states and the transitions

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

The transitions 1, 2 & 3 involve context switch (or, process switch), which will be discussed next.
Call stack

• A call stack is a stack data structure that store information of the active function calls
• A call stack is composed of stack frames (also called activation records or activation frames). Each function call corresponds to a stack frame, which consists of
  – Arguments passed to the routine
  – The return address
  – Saved register values (in order to restore them at return)
  – Local variables
• The call stack grows when a new call is issued, and shrinks when a function call returns
Calling Convention

• A Calling Convention stipulates important information about how a function calls another, and how the call returns. It consists of
  – How parameters are passed (e.g., through registers or the call stack)
  – The order of parameters if there are multiple ones
  – Who (caller or callee) is to recover the environment after the call returns
  – How the return value is to be passed from the callee to the caller
Call stack and calling convention (x86-32 as an example)

- `Bar(42, 21, 84) // invoked by Foo()`
Execution context

- The execution context (or context; or processor state) is the content of the CPU registers at any point of time, e.g.
  - Program counter: a specialized register that indicates the current program location
  - Call stack pointer: indicates the top of the kernel-space call stack for the process
  - Location of memory allocated for the process
  - Other register values
Caution: the meaning of “state” is ambiguous

• *Process* state: Running/Blocked/Ready
• *Processor* state, i.e., the execution context
• Don’t mistake them
Where is the context information stored?

- A Process Control Block (PCB) is an instance of a data structure in the kernel memory containing the information needed to manage a particular process. It includes
  - The stored execution context information
  - Process ID
  - Process control information, such as the scheduling state, opened file descriptors, accounting information
Linux’s PCB: task_struct

```c
#include/linux/sched.h

1344 struct task_struct {
1346     void *stack;
1444     pid_t pid; // thread id
1445     pid_t tgid; // thread group id
1527     /* open file information */
1528     struct files_struct *files;
1531     /* signal handlers */
1532     struct signal_struct *signal;
1780     struct thread_struct thread;
```

The execution context is stored in the thread_struct structure
How to locate the PCB task_struct?

#define current get_current()
static inline struct task_struct * get_current(void) {
    return current_thread_info()->task;
}

thread_info and the kernel stack together reside in 8k (2^13=8k) space (two pages)
static inline struct thread_info *current_thread_info(void) {
    __asm( "
        movl $0xfffffffe000, %eax
        andl %esp, %eax
    ")
}

Mask off 13 bits of esp, and you can get the addr of thread_info, which has a field task pointing to task_struck
Process switch

• A Process Switch (or Task Switch) occurs when the OS scheduler suspends the execution of one process on a CPU and runs another
  – Store the context of the current process into its PCB
  – The scheduler picks a process in the “ready” list
  – Use the PCB of the picked process to restore the contents of the CPU registers

• The effect is that the first process is scheduled *out* and the second process is scheduled *in*
When does a Process Switch occur?

- A process blocks (e.g., due to I/O)
- A process exits
- The CPU time slice is used up
How does the kernel know when the CPU time slice of the current process is used up?

- Assume the timer interrupt has a frequency of 1000hz, i.e., it occurs once per 1ms
- Assume the CPU time slice for a process is 10ms; thus, a counter of the process is set to 10 when it is scheduled in
- Each time the timer interrupt occurs, the interrupt handler (in kernel) will decrement the counter
- When the counter is 0, scheduling occurs: the current process is scheduled out and another is scheduled in
Whenever a timer interrupt is issued, the interrupt handler in the kernel is invoked to determine whether scheduling should happen.

Timer interrupts ensure that the CPU time allocation is under the control of the kernel; i.e., no user process can occupy the CPU longer than it is supposed to.
If the CPU time slice is 10ms, why not just set the timer interrupt frequency as 100?

Timer interrupts are not only used by the scheduler, but also timing purpose (e.g., games, multimedia player)
Linux/x86 – How to locate kernel stack during mode switch?

• The x86 CPU has a Task Register (TR), that points to the Task State Segment (TSS) descriptor
• Linux kernel creates a Task State Segment (TSS) for each CPU, which contains
  – Kernel stack address (TSS.esp0)
• TSS is updated during context switch
Hardware vs. Software context switch

- Intel makes Task State Segment (TSS) complex to support h/w context switch; it is supposed to work as PCB
- Linux, however, uses s/w context switch, because it is easier to port for other CPUs, and has more opportunities for optimizations.
- Linux only uses TSS for saving the kernel stack address and IO permission checking. Other registers are saved in the kernel stack and `task_struct`, which is the PCB in Linux
Linux/x86 – Task switch

- Switch from process prev to next
  - Switch the page table (using task_struct.mm_struct.pgd)
  - Save ebp, eflags, and general purpose registers at prev’s stack
  - Save eip onto the stack of prev
  - Save and restore esp (using thread_struct of prev & next)
  - Current CPU’s TSS is updated (using next’s thread_struct)
  - Restore eip using the value saved for next // through “ret” instruction

- eflags register
  - CF: carry flag (unsigned arithmetic)
  - OF: overflow flag (signed arithmetic)
  - ZF: zero flag
  - TF: trap flag (for debugging)
  - IF: interrupt enable flag (cli: turn off interrupts)
Process vs. Thread

Process
• A process is an executing instance of a program
• Different processes have different memory address spaces
• Resource-heavyweight: significant resources are consumed when creating a new process

Thread
• A thread is the entity within a process that can be scheduled for code execution
• A process has at least one thread
• Threads of a process share a lot of information, such as memory address space, opened files, etc.
  – Thus, resource-lightweight to create a new thread
Using Threads over Processes

Pros

- Cheaper to create a new thread
- Cheaper task switch: when you switch the threads of the same process, you only need to update part of the context (e.g., esp)
- Cheaper data sharing between threads of a process

Cons

- One bug in one thread may render the whole process unstable
- Once one thread is attacked (more precisely, compromised), the whole process is in danger
  - The main reason why one-process-per-tab browsers become so popular
Summary

• Process state transition
  – Ready, blocked, running
• Call Stack and Calling Convention
• Execution Context
• Process switch
  – CPU time slice is used up
  – Process is blocked (e.g., waiting for user input)
  – Process has exited
• Process vs. Thread
Interesting Readings

• Threads vs. Processes:
• Timer interrupt frequency
  – https://lwn.net/Articles/145973/
• Assembly using RDTSC; faster than clock_gettime()
  – Constant TSC rate is one every CPU now: https://stackoverflow.com/questions/7935518/is-clock-gettime-adequate-for-submicrosecond-timing
  – http://oliveryang.net/2015/09/pitfalls-of-TSC-usage/
• clock_gettime: portable method for time measurement