CIS 5512 - Operating Systems
Processes & Threads

Professor Qiang Zeng

Some graphs are courtesy of Bovet and Cesati
Previous class…

What are the two different CPU modes?

Kernel mode and user mode
What is the difference between kernel mode and user mode?

1. Privileged instructions, e.g., I/O instructions, can only be issued (when the CPU is) in the kernel mode
2. 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.
What if privileged instructions are executed in user mode?

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 Fault) 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.
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.
- Resource-lightweight
Three basic process states and the transitions

- Running
- Blocked
- Ready

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

The transitions 1, 2 & 3 involves 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
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 contents of the CPU registers at any point of time
  – 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 (will be covered soon)
  – A specialized register that indicates the page table (will be covered in the memory section of the course)
  – Other register values
Caution: the meaning of “state” is ambiguous

• It may refer to process state: Running/Blocked/Ready
• Or, the processor state, i.e., the execution context
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
  - Stored execution context
  - 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

struct task_struct {
    void *stack;
    pid_t pid; // thread id
    pid_t tgid; // thread group id
    /* open file information */
    struct files_struct *files;
    /* signal handlers */
    struct signal_struct *signal;
    struct thread_struct thread;
};
```

The execution context is stored in the thread_struct structure.
Context switch (or, process switch)

- Context switching, also called task switch or process switch, is to suspend the execution of one process on a CPU and to resume execution of another process
  - Store the context of the current process into its PCB
  - The scheduler picks a process in the “ready” list
  - Retrieve the context of the picked process from its PCB and restore the contents of the CPU registers
- The first process is scheduled out and the second process is scheduled in
When does context switching occur?

• A process blocks (due to I/O or synchronization) or exits
• The CPU time slice of the current process is used up
How does the kernel track 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
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
Mode switch

• Mode switch means program execution switches between different CPU modes
  – User -> kernel
  – Kernel -> user
  – Other: kernel <-> hypervisor

• When does the user -> kernel mode switch occur?
  – System calls
  – Interrupts (if CPU is in user mode)
  – Exceptions (if CPU is in user mode)

• We have covered system calls; next, we will introduce interrupts and exceptions
## Interrupts, Exceptions and Signals

<table>
<thead>
<tr>
<th>Type</th>
<th>Triggered by</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exceptions (or, s/w interrupts)</td>
<td>Instruction execution</td>
<td>breakpoint; page fault; divide-by-zero; system calls</td>
</tr>
<tr>
<td>Signals (sent by kernel; handled in userspace)</td>
<td>kill(); sent by exception handler</td>
<td>SIGTRAP; SIGSEGV; SIGFPE</td>
</tr>
<tr>
<td>Interrupts (or, h/w interrupts)</td>
<td>interval timer and I/O</td>
<td>timer; input available</td>
</tr>
<tr>
<td>Language exceptions (as in C++ and Java)</td>
<td>throw</td>
<td>throw std::invalid_argument(“”);</td>
</tr>
</tbody>
</table>
Exceptions

• Programmed exceptions
  – int 0x80 // old method of issuing system calls
  – int 3 // single-step debugging

• Anomalous executions
  – a/0 // divide by zero
  – p = NULL; a = *p // a kind of page fault
Signals due to exceptions

<table>
<thead>
<tr>
<th>#</th>
<th>Exception</th>
<th>Exception handler</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide error</td>
<td>divide_error()</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>1</td>
<td>Debug</td>
<td>debug()</td>
<td>SIGTRAP</td>
</tr>
<tr>
<td>2</td>
<td>NMI</td>
<td>nmi()</td>
<td>None</td>
</tr>
<tr>
<td>3</td>
<td>Breakpoint</td>
<td>int3()</td>
<td>SIGTRAP</td>
</tr>
<tr>
<td>4</td>
<td>Overflow</td>
<td>overflow()</td>
<td>SIGSEGV</td>
</tr>
<tr>
<td>5</td>
<td>Bounds check</td>
<td>bounds()</td>
<td>SIGSEGV</td>
</tr>
<tr>
<td>6</td>
<td>Invalid opcode</td>
<td>invalid_op()</td>
<td>SIGILL</td>
</tr>
<tr>
<td>7</td>
<td>Device not available</td>
<td>device_not_available()</td>
<td>None</td>
</tr>
<tr>
<td>8</td>
<td>Double fault</td>
<td>doublefault_fn()</td>
<td>None</td>
</tr>
<tr>
<td>9</td>
<td>Coprocessor segment overrun</td>
<td>coprocessor_segment_overrun()</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>10</td>
<td>Invalid TSS</td>
<td>invalid_TSS()</td>
<td>SIGSEGV</td>
</tr>
<tr>
<td>11</td>
<td>Segment not present</td>
<td>segment_not_present()</td>
<td>SIGBUS</td>
</tr>
<tr>
<td>12</td>
<td>Stack segment fault</td>
<td>stack_segment()</td>
<td>SIGBUS</td>
</tr>
<tr>
<td>13</td>
<td>General protection</td>
<td>general_protection()</td>
<td>SIGSEGV</td>
</tr>
<tr>
<td>14</td>
<td>Page Fault</td>
<td>page_fault()</td>
<td>SIGSEGV</td>
</tr>
<tr>
<td>15</td>
<td>Intel-reserved</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>16</td>
<td>Floating-point error</td>
<td>coprocessor_error()</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>17</td>
<td>Alignment check</td>
<td>alignment_check()</td>
<td>SIGBUS</td>
</tr>
<tr>
<td>18</td>
<td>Machine check</td>
<td>machine_check()</td>
<td>None</td>
</tr>
<tr>
<td>19</td>
<td>SIMD floating point</td>
<td>simd_coprocessor_error()</td>
<td>SIGFPE</td>
</tr>
</tbody>
</table>

An exception is usually converted to a user space signal
Signals due to IPC

Tips: “kill –l” to list all signal names

• cmd “kill –<signname> <pid>” or function “kill(<pid>, <sig>)”: send signals to a process
• The parameter “<pid>” above is very expressive
  – >0: a real pid
  – 0: all processes in the same process group as the sender process
  – -1: all processes for which the sender has permission to send signals
  – When a signal is sent to a process, it can be handled by any thread of the process
• pthread_kill(): send signals to a specific thread within the sender’s process
• tgkill(<pid>, <tid>, <sig>): send a signal to thread <tid> in process <pid>. Warning: This is Linux specific
• sigqueue(<pid>, <sig>, <value>): send a signal and an associate value to process <pid>
Signal handler

- Signal handlers are shared among threads of a process, while the signal mask is per thread.
- If you want to change the default signal handling behaviors, use `sigaction()` to install your own handlers; don’t use `signal()`
  - `signal()` is not reliable in the sense that, upon the invocation of your handler, the default signal handler is restored as default
  - With `signal()`, when your handler is being invoked, the same type of signals are not blocked
  - Plus, `sigaction()` is more capable. For example
    - It supports blocking other signals when your handler is invoked
    - If a blocking system call, e.g., `read/write()`, is interrupted by the signal handling, the system call can be restarted automatically
Interrupts and the hardware

- Each interrupt and exception is identified by a number in $[0, 255]$. Intel calls this number vector
- IRQ: Interrupt ReQuest line
- PIC: Programmable Interrupt Controller
- NMI: Non-Maskable Interrupt
- IPI: Inter-Processor Interrupt (through local APIC)
Interrupt Descriptor Table (IDT)

• Used by both Interrupt and Exception handling
• Each entry is a descriptor that refers to an Interrupt or Exception handler
• Difference between the Interrupt entry and the Exception entry
  – CPU will clear the IF flag to disable local interrupts upon handling of an interrupt (using cli instruction)
  – IF flag will not be disabled when handling exceptions
Interrupt/exception handling

- CPU uses the IDT to jump to a handler automatically. Below shows interrupt handling
Interrupt/exception handling

• Basic steps:
  – **Mode switch to kernel mode** if the current mode is user mode
  – Save the current context
  – Invoke the corresponding handler function
  – Restore the context
  – **Mode switch back to user mode** if the original mode was user mode
What happens upon a keystroke?

- **Interrupt handling**
  - **Hardware part**
    - CPU refers to IDT to locate the handler
  - **Software part**
    - Execution of the handler according to the interrupt number
Which processes have PID 0 and PID 1

- Try command “ps -eaf”
- PID 0: idle process
  - The first process
  - Invoke hlt instructions when being scheduled to save power
- PID 1: init process
  - Initially it is a kernel thread created by idle
  - Then exec(init) to become a regular process

<table>
<thead>
<tr>
<th>UID</th>
<th>PID</th>
<th>PPID</th>
<th>C</th>
<th>STIME</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Aug27 ?</td>
<td></td>
<td>00:00:01</td>
<td>/sbin/init</td>
</tr>
<tr>
<td>root</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>Aug27 ?</td>
<td></td>
<td>00:00:00</td>
<td>[kthread]</td>
</tr>
<tr>
<td>root</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>Aug27 ?</td>
<td></td>
<td>00:00:00</td>
<td>[migration/0]</td>
</tr>
<tr>
<td>root</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>Aug27 ?</td>
<td></td>
<td>00:00:04</td>
<td>[ksoftirqd/0]</td>
</tr>
<tr>
<td>root</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>Aug27 ?</td>
<td></td>
<td>00:00:00</td>
<td>[watchdog/0]</td>
</tr>
<tr>
<td>root</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>Aug27 ?</td>
<td></td>
<td>00:00:10</td>
<td>[events/0]</td>
</tr>
<tr>
<td>root</td>
<td>7</td>
<td>2</td>
<td>0</td>
<td>Aug27 ?</td>
<td></td>
<td>00:00:00</td>
<td>[cpuset]</td>
</tr>
<tr>
<td>root</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>Aug27 ?</td>
<td></td>
<td>00:00:00</td>
<td>[khelper]</td>
</tr>
<tr>
<td>root</td>
<td>9</td>
<td>2</td>
<td>0</td>
<td>Aug27 ?</td>
<td></td>
<td>00:00:00</td>
<td>[netns]</td>
</tr>
</tbody>
</table>
How is a process created? Userspace view

- `fork()`: create a new process

```c
int pid = fork();
if (pid < 0) {
    // error; no process created;
} else if (pid > 0) {
    // this is the parent process
} else { // pid == 0
    // this is the child process
}
```
```c
Parent
int main()
{
    pid_t pid;
    char *message;
    int n;
    pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    if (pid == 0) {
        message = "This is the child\n";
        n = 6;
    } else {
        message = "This is the parent\n";
        n = 3;
    }
    for(; n > 0; n--) {
        printf(message);
        sleep(1);
    }
    return 0;
}

Child
int main()
{
    pid_t pid;
    char *message;
    int n;
    pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    if (pid == 0) {
        message = "This is the child\n";
        n = 6;
    } else {
        message = "This is the parent\n";
        n = 3;
    }
    for(; n > 0; n--) {
        printf(message);
        sleep(1);
    }
    return 0;
}
```
Some APIs critical for implementing shell

- The **exec**() family of functions (execl, execlp, execle, …) changes the program being executed.
  - E.g., execl("/bin/ls","ls","-l",NULL);
  - “/bin/ls” determines the program to be executed, while “ls”, “-l” form argv[]

- The **wait**() system call suspends execution of the calling process until one of its children terminates.
How is shell implemented?

```c
char *prog, **args;
int child_pid;

// Read and parse the input a line at a time
while (readAndParseCmdLine(&prog, &args)) {
    child_pid = fork();
    if (child_pid < 0)
        exit(-1);
    if (child_pid == 0) {
        exec(prog, args);
        // NOT REACHED
    } else {
        wait(child_pid);
    }
}
```

Q: How is `system(char* cmd)` implemented?
A: Just remove the loop wrapper of the left
How is fork() implemented in kernel?

- Kernel stack
  - Copied; each has its own
- Address space
  - “Copied”
  - Copy-on-write (later classes)
- PCB
  - Copied with PID changed
  - Including signal mask/handling and file descriptors
  - A file descriptor is an integer pointing to a file description
  - Thus, file description is shared
Zombie Process in Linux/Unix

• Once a child process exits, it becomes a zombie process with its exit state to be queried by its parent. A zombie process is cleaned up if
  – Its parent calls `wait()` to retrieve the exit state, or
  – Its parent has expressed no interest in that exit state by installing handler for `SIGCHLD`

• If a parent process exits, its zombie child processes become children of the `init` (pid = 1) process, which periodically reaps zombies
  – Zombie processes occupy precious kernel resources (e.g., PCB), which you want to reclaim ASAP; don’t defer it to the `init` process
Take away…

- Process state transition
  - Ready, blocked, running
- Context switch
  - Process switch
- Mode switch
  - System calls
  - Interrupt/exception handling
- Interrupt vs. exception vs. signal
- Calling convention
- fork() and Shell
Interesting Readings

• [https://superuser.com/questions/1052231/does-32-or-64-bits-cpu-use-segmentation-addressing-on-linux](https://superuser.com/questions/1052231/does-32-or-64-bits-cpu-use-segmentation-addressing-on-linux)