CSCE 311 - Operating Systems
Interrupts, Exceptions, and Signals

Qiang Zeng, Ph.D.
Fall 2018
Previous Class

- Process state transition
  - Ready, blocked, running
- Call Stack
- Execution Context
- Process switch
  - CPU time slice is used
  - Process is blocked (e.g., waiting for user input)
  - Process has exited
- Process vs. Thread
The three basic process states and the four transitions between them

**Three states:** Running, Blocked, Ready

1. **Running -> Blocked:** I/O; wait for a lock
2. **Blocked -> Ready:** I/O is ready; the lock is acquired
3. **Ready -> Running:** a ready process is picked to run
4. **Running -> Ready:** time slice used up
What is a Call Stack?

A stack data structure that stores the information for the active function calls; it consists of stack frames, each of which corresponds to an active function call.
What is Execution Context? Give two examples that are part of the context

The contents of the CPU registers at any point of time.

**Program counter**: a register that points to the instruction to be executed

**Call stack pointer**: a register that points to the top of the call stack, e.g., esp in x86 CPU
What is Process Switch? When does it occur?

Also called task switch. Suspending one process and resuming another.

The current process has used up its time slice, blocks (due to I/O or requesting lock), or exits.
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 some of the registers (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
Outline

• Mode Switch
• Interrupt vs. exception vs. signal
• fork() and Shell
• Zombie
What is Mode Switch? When does it occur?

Program execution switches between different CPU modes
User -> Kernel: system calls, interrupts, exceptions
Kernel -> User: finish handling of sys calls/interrupts/exceptions
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
  – Exceptions

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

• Programmed exceptions (*or traps*)
  – `int 0x80` // old method of issuing system calls
  – `int 3` // single-step debugging

• Anomalous executions (*or faults*)
  – `a/0` // divide by zero
  – `p = NULL; a = *p` // dereference null pointer
How “Division by Zero” is handled?

• What happens upon “a/0”?
  – The CPU that is executing the division operation triggers an exception
  – Mode switch to the kernel, and the divided-by-zero handler is invoked automatically
  – The handler sends SIGFPE to the faulty process, and mode switch back to the user space
  – Has the process registered a SIGFPE handler?
    • Yes: execute the SIGFPE handler
    • No: kills the process
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 by the kernel exception handlers.
Signals causes

- cmd “kill –signame 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. This is Linux specific
- sigqueue(pid, sig, value): send a signal and an associate value to process pid
- When a process exits, its parent process receives SIGCHLD
- …
**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.

---

Linux’s new `signal()` implementation supports reliable signals. The implementation actually invokes `sigaction()`. Don’t rely on that; other Unix OSes may not be that way.
Interrupts in x86 Processors

- 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
## 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></td>
<td></td>
<td></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></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Language exceptions (as in C++ and Java)</td>
<td>throw</td>
<td>throw std::invalid_argument(&quot;&quot;);</td>
</tr>
</tbody>
</table>
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
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
  }
  ```
Parent

```c
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

```c
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
Zombie Process in Linux/Unix

- Once a child process “exits”, it enters the zombie state process. A zombie process is cleaned up if
  - Its parent calls wait() to retrieve the exit state, or
  - if the parent *explicitly* ignores SIGCHLD by setting its handler to SIG_IGN
- 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), so you want to reclaim them ASAP
  - If a process has created a lot of child processes, and the parent does not reap them, what will happen?
A more complete Process-State Transition Graph

- READY
  - Create process (fork)
  - Finish blocking
  - Selected to run
  - Quantum ends
  - RUN
  - Execution ends (exit)
- BLOCKED
  - I/O operation Waiting for event
- ZOMBIE
Summary

• Interrupt vs. exception vs. signal
• fork() and Shell
• Zombie
Interesting Readings

- Causes of signals in Linux/Unix
  - [https://unix.stackexchange.com/a/6337/58273](https://unix.stackexchange.com/a/6337/58273)
- ld (compile-time linker) vs ld-linux.so (load-time linker)
  - [http://www.linuxjournal.com/article/6463](http://www.linuxjournal.com/article/6463)
- ldd
  - [https://stackoverflow.com/questions/1488527/hierarchical-ldd](https://stackoverflow.com/questions/1488527/hierarchical-ldd)
- What happens when you run a program
  - [https://stackoverflow.com/a/1220211/577165](https://stackoverflow.com/a/1220211/577165)
- Good slides