Previous class...

- Functions of an OS
  - Services, resource allocation, protection, and IPC

- Protection Rings
  - Fault isolation
  - Privileged instructions at Ring 0: it makes resource allocation and protection possible.

- Hardware-level Virtual machines
  - Classification based on hypervisor:
    - Type 1 and type 2
  - Classification based on virtualization mechanism:
    - Para-virtualization, Full virtualization, hardware-assisted virtualization

*Some graphs in the following slides are credits of Bovet and Cesati*
Process and its three states

- An instance of program execution
- **Idea behind**: virtualize the CPU, so that each program execution thinks it owns a CPU

![Diagram showing the three states of a process: Running, Blocked, and Ready. The transitions are numbered 1 to 4.]

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

- A process of storing/restoring the execution context
  - Stack pointer, program counter, registers, address space
  - Saved in Process Control Block and kernel stack
- When to switch?
  - Task switch (or, process switch)
    - A process is blocked due to I/O, synchronization
    - The CPU time slice of the current process is used up
  - Interrupt and exception handling

![Context Switch Diagram]

CIS 5512 – Operating Systems

4
Example: CPU timer interrupt

- A timer interrupt indicates that it’s time to check whether the process’s current time slice has been used up
- Step 1: context switch due to interrupt handling
- Step 2: context switch due to process switch
- Difference between the two types of context switch?
  - Context switch due to interrupt/exception handling is lighter, as it doesn’t change the current process, which means the memory address space and kernel stack are not changed
- Critical thing: locate the kernel stack of the current process to save user space states and issue kernel function calls
Linux/x86 – How to locate kernel stack?

- Task Register (TR) 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)
  - I/O permission bitmap
Hardware vs. Software context switch

- Intel makes 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 – task_struct and thread_struct

• Since only one TSS is allocated for a CPU, during Process Switch where are the values used to update TSS?

```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;
1787 };
```

thread_struct: arch/x86/include/asm/processor.h
Linux – How to locate task_struct?

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

(1) `thread_info` and the kernel stack together reside in 8k (2^13=8k) space (two pages)

```c
static inline struct thread_info *current_thread_info(void) {  
    __asm( "movl $0xfffffe000, %eax  
            andl %esp, %eax"
    )
}
```

(2) Mask off 13 bits of esp, and you can get the addr of `thread_info`, which has a filed `task` pointing to `task_struck`
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)
calling convention

• Bar(42, 21, 84) // invoked by Foo()

<table>
<thead>
<tr>
<th>Calling Convention</th>
<th>Cleans Stack</th>
<th>Arguments</th>
<th>Arg Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdecl</td>
<td>Caller</td>
<td>On the Stack</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>fastcall</td>
<td>Callee</td>
<td>ECX, EDX, then stack</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td>stdcall</td>
<td>Callee</td>
<td>On the Stack</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td>VC++ thiscall</td>
<td>Callee</td>
<td>EDX (this), then stack</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>GCC thiscall</td>
<td>Caller</td>
<td>On the Stack (this pointer first)</td>
<td>Right-to-left</td>
</tr>
</tbody>
</table>

Diagram:

1. Stack beginning with return address.
2. Saved ebp.
4. Local variable.
5. Argument 1 (42).
6. Argument 2 (21).
7. Argument 3 (84).
8. Saved eax.

Foo() function calls Bar().
# Interrupts, Exceptions and Signals

<table>
<thead>
<tr>
<th>Type</th>
<th>Triggered by</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupts (or, h/w interrupts)</td>
<td>interval timer and I/O (asynchronous)</td>
<td>timer; input available</td>
</tr>
<tr>
<td>Exceptions (or, s/w interrupts)</td>
<td>Instruction execution (synchronous)</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>Language exception (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 execution
  – a/0 // divide by zero
  – p = NULL; a = *p // a kind of page fault

• Valid page fault
  – A page that has been swapped out is accessed
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.
Sending signals

- cmd “kill –signum 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

Tips: “kill –l” to list all signal names
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 `signall()`, 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 and the hardware

- Each interrupt and exception is identified by a number in $[0, 255]$. Intel calls this number a 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 a 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 – h/w part

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

![Diagram showing interrupt handling process]

In the diagram:
- `Device 1` and `Device 2` are connected to an `INT` signal which triggers the interrupt.
- The `INT` signal is processed by the `PIC` (Programmed I/O Controller).
- The IDT table is accessed at `IDT[32+n]` to determine the appropriate handler.
- The handler `interrupt[n]` is called, followed by `do_IRQ(n)`.
- Two interrupt service routines, `Interrupt service routine 1` and `Interrupt service routine 2`, are handled separately.
Interrupt/exception handling – s/w part

• Three basic steps:
  – Save the current context such as registers
  – Invoke the corresponding handler function
  – Restore the context

• Interrupt handling is divided into two stages
  – Top half: short; the IRQ line is usually disabled
  – Bottom half: takes time; Softirqs (reentrant and statically allocated), Tasklets (serialized execution and dynamic), or work queues (sleepable) are used
  – Why two stages? Make the IRQ disabled time as short as possible to keep IO running
What happens when you type a char?

- **Interrupt handling**
  - **Hardware part**
    - CPU refers to IDT to locate the handler
  - **Software part**
    - Top half
    - Bottom half
Which processes have PID 0 and PID 1

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

```
qiang@ubuntu:~$ ps -eaf | head -10
UID  PID  PPID  C   STIME TTY          TIME CMD
root 1    0    0  Aug27 ?    00:00:01 /sbin/init
root 2    0    0  Aug27 ?    00:00:00 [kthreadd]
root 3    2    0  Aug27 ?    00:00:00 [migration/0]
root 4    2    0  Aug27 ?    00:00:04 [ksoftirqd/0]
root 5    2    0  Aug27 ?    00:00:00 [watchdog/0]
root 6    2    0  Aug27 ?    00:00:10 [events/0]
root 7    2    0  Aug27 ?    00:00:00 [cpuset]
root 8    2    0  Aug27 ?    00:00:00 [khelper]
root 9    2    0  Aug27 ?    00:00:00 [netns]
```
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
  }
  ```

- `exec*()`: replace the current whatever process with another program
  ```c
  execl("/bin/ls","ls","-l",NULL); // "/bin/ls" determines the program to be executed, while "ls", "-l" form argv[]
  ```
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;
}
```
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);
    }
}
```

How is `system(char* cmd)` implemented?
Just remove the loop of the left code
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

• 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 SIG_INT (SIG_DFL doesn’t work) 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., PID and PCB), which you want to reclaim ASAP; don’t defer it to the init process
IPC

• Windows IPC
  – http://www.codeproject.com/Articles/13724/Windows-IPC

• Linux IPC
  – http://www.tldp.org/LDP/lpg/node7.html
Take away…

• Process state transition
  – Ready, blocked, running

• Context switch
  – Process switch
  – Interrupt/exception handling

• Interrupt vs. exception vs. signal

• Calling convention

• fork() and Shell