CS153: Architecture Support 2

Chengyu Song

Slides modified from
Harsha Madhyvasta, Nael Abu-Ghazaleh, and Zhiyun Qian
Administrivia

- Waiting list
  - New labs created, email us if there's any problem

- Lab
  - Ask questions, piazza is preferred as other students may have the same one
Recall major types of arch support

- Isolation
  - Protects hardware configuration and direct access
- Virtualization
  - Efficient multiplexing
- Events generation and dispatch
  - Improves resources utilization
- Parallelism and security
Events

• An event is an “unnatural” change in control flow
  • Events immediately stop current execution
  • Changes mode, context (machine state), or both
• The kernel defines a handler for each event type
  • Event handlers always execute in kernel mode
  • The specific types of events are defined by the machine
• Once booted, OS is one big event handler
  • All entry to the kernel occurs as the result of an event
Types of events

- Two kinds of events: **synchronous** and **asynchronous**
  - Sync events are caused by executing instructions
  - Async events are caused by an external event
Causes of events

- Two *reasons* for events: *unexpected* and *deliberate*
  - Unexpected events are, well, unexpected
    - Examples?
  - Deliberate events are scheduled by OS or application
    - Why would this be helpful?
Matrix of events

<table>
<thead>
<tr>
<th></th>
<th>Unexpected*</th>
<th>Deliberate</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronous</td>
<td>fault</td>
<td>syscall</td>
</tr>
<tr>
<td>asynchronous</td>
<td>interrupt</td>
<td>signals</td>
</tr>
</tbody>
</table>

* from the hardware perspective

- **NOTE**: Terms may be used slightly differently by OS and arch
  - For example, POSIX signals, async system traps, async or deferred procedure calls (APC or DPC)
Faults

• Hardware defines, detects, and reports "exceptional" conditions
  • Divide by zero, memory access violation (unaligned, permission, not mapped, bounds), illegal or privilege instructions, etc
• Upon exception, hardware "faults" (verb)
  • Must save state (PC, regs, mode, etc.) so that the faulting process can be restarted
  • Invokes the registered handler
HW events handling

```plaintext
handleTimerInterrupt() {
    ...
}

handleDivideByZero() {
    ...
}

handleSystemCall() {
    ...
}
```
Fault handler

switch fault-mode:
  case USER:
    if expected:
      invoke handler and resume execution
    elif process-has-registered-handler:
      invoke user-handler
    else:
      kill the process;
  case KERNEL:
    if expected:
      invoke handler and resume execution
    else:
      kernel panic
User faults handling

• Expected faults
  • OS may "abuse" hw faults to implement some functionalities
    • Lazy page allocation, virtual memory, copy-on-write, etc.
    • Transparently handled (from the app perspective)
User faults handling (cont.)

- Truly unexpected faults
  - If the process has registered a handler for corresponding faults (e.g., `SIGSEGV`), OS invokes the handler with fault information
    - Try `man sigaction`
  - Otherwise the process is terminated
Kernel fault handling

- Some faults in kernel mode is expected, e.g., `copy_from_user`, paging
  - These faults can be recovered
- Most other faults are not recoverable and will result in system crash
  - Famous blue screen of death (BSOD)
  - Kernel panic
System calls

• For requesting the more privileged program to perform tasks that are not doable under current mode, e.g., I/O

• Hardware provides special instruction(s) to
  • Causes an exception, which invokes a kernel handler
  • Saves caller state (PC, regs, mode) so it can be restored
    • Why save mode?
  • Returning from system call restores the state
System call questions

• There are hundreds of syscalls, how do we let the kernel know which one we intend to invoke?
  • Before issuing \texttt{int \$0x80} or \texttt{sysenter/syscall}, set \texttt{%eax/%rax} with the syscall number

• System calls are like function calls, but how to pass parameters?
  • Instead of stack, syscall parameters are passed through registers
  • Try \texttt{man syscall}
System call questions (cont.)

• How to reference kernel objects (e.g., files, sockets)?
  • Name vs ID

```c
char buffer[128];
int fd = open("file");
read(fd, buffer, 128);
```

• Why can’t we reference the kernel objects by memory address?
System calls in xv6

- Look at `traps.h` and `trap.c`
  - Interrupt handlers are initialized in two arrays (`idt` and `vectors`)
    - `Tvinit()` function does the initialization
  - Syscalls have a single trap handler (`T_SYSCALL`, 64)
  - `Trap()` handles all exceptions, including system calls
    - If the exception is a system call, it calls `syscall()`
- Keep digging from there to understand how system calls are supported
  - You will be adding a new system call in Lab 1
Interrupts

- Interrupts signal asynchronous events
  - I/O hardware interrupts
  - Timers
  - Inter-processor interrupt (IPI)
    - On multicore/multiprocessor systems
Flavors of interrupts

• From hardware perspective
  • **Precise**: interrupt leaves the CPU is a *well-defined state*
  • **Imprecise**: otherwise
    • How is this possible?
  • OS designers like precise interrupts, CPU designers like imprecise interrupts
    • Why?
Flavors of interrupts (cont.)

- From OS perspective
  - **Maskable** interrupt (IRQ)
    - Why does OS want to mask interrupt
  - **Non-maskable** interrupt (NMI)
    - Highest priority tasks
I/O using interrupts

- Recall polling vs. interrupt
- Interrupts are the basis for asynchronous I/O
  - OS initiates I/O
  - Device operates independently of CPU
  - Device sends an interrupt signal to CPU when done
  - CPU invokes the interrupt handler registered by OS
I/O example

1. Ethernet receives packet, writes packet into memory
2. Ethernet signals an interrupt
3. CPU stops current operation, switches to kernel mode, saves machine state (PC, mode, etc.)
4. CPU reads address from vector table indexed by interrupt number, branches to address (Ethernet device driver)
5. Ethernet device driver processes packet
6. Upon completion, restores saved state
Timer

- The key to an timesharing OS
- The fallback mechanism by which the OS reclaims control
  - Timer is set to generate an interrupt after a period of time
    - Setting timer is a privileged instruction
  - Basis for OS scheduler (more later ...)
    - Ensures **fairness**
- Also used for time-based functions (e.g., `sleep()`)
Synchronization

- Interrupts cause difficult problems
  - An interrupt can occur at any time
  - A handler can execute that interferes with code that was interrupted
- OS must be able to synchronize concurrent execution
- Need to guarantee that short instruction sequences execute atomically
  - Temporarily disables interrupts
  - Special atomic operations
Additional architecture support

- Parallelism
  - Data consistency (cache coherent)
  - Intel TSX (Transactional Synchronization Extensions)
- Security
  - Memory access permission (WRX)
  - SMEP (Supervisor Mode Execution Protection), SMAP
  - MPX, SGX, CET, MPK
## Summary

<table>
<thead>
<tr>
<th></th>
<th>Unexpected</th>
<th>Deliberate</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronous</td>
<td>fault</td>
<td>syscall</td>
</tr>
<tr>
<td>asynchronous</td>
<td>interrupt</td>
<td>signals</td>
</tr>
</tbody>
</table>
For next class ...

- Processes
- Continue to get familiar with the lab environment
- **Read xv6 book**
  - Chapter 3
  - Appendix A and B