Interrupt Handling

- **Two objectives**
  - handle the interrupt and remove the cause
  - restore what was running before the interrupt
    - saved state may have been modified on purpose

- **Two “actors” in handling the interrupt**
  - the hardware goes first
  - the kernel code takes control by running the interrupt handler
Review: stack (aka call stack)

int main(argc, argv){
    ... 
    f(3.14) 
    ... 
}

int f(x){
    ... 
    g();
    ... 
}

int g(y){
    ...
}

PC/IP

arguments (3.14)
return address
saved FP (main)
local variables
saved registers
scratch space

user stack

stack frame for main()
stack frame for f()
stack frame for g()
A Tale of Two Stack Pointers

- Interrupt handler is a program: it needs a stack!
  so, each process has two stacks pointers:
  one when running in user mode
  a second one when running in kernel mode

- Why not using the user-level stack pointer?
  user SP cannot be trusted to be valid or usable
  user stack may not be large enough, and may spill to overwrite important data
  security:
    e.g., kernel could leave sensitive data on stack
Handling Interrupts: HW

On interrupt, hardware:
- sets supervisor mode (if not set already)
- disable (masks) interrupts (partially privileged)
- pushes PC, SP, and PSW of user program on interrupt stack
- sets PC to point to the first instruction of the appropriate interrupt handler
  - depends on interrupt type
  - interrupt handler specified in interrupt vector loaded at boot time

Interrupt Vector
- I/O interrupt handler
- System Call handler
- Page fault handler
- ...
- Condition codes
Handling Interrupts: SW

- We are now running the interrupt handler!
  - IH first pushes the registers’ contents (needed to run the user process) on the interrupt stack
  - need registers to run the IH
  - only saves necessary registers (that’s why done in SW, not HW)
Typical Interrupt Handler Code

HandleInterruptX:

    PUSH %Rn
    ...  
    PUSH %R1
    CALL _handleX
    POP %R1
    ...  
    POP %Rn
    RETURN_FROM_INTERRUPT

only need to save registers not saved by the handler function

restore the registers saved above
Returning from an Interrupt

- Hardware pops PC, SP, PSW

- Depending on content of PSW
  switch to user mode
  enable interrupts

- From exception and system call, increment PC
  on return (we don’t want to execute again the same instruction)

  on exception, handler changes PC at the base of the stack

  on system call, increment is done by hw when saving user level state
Starting a new process: the recipe

1. Allocate & initialize PCB
2. Setup initial page table (to initialize a new address space)
3. Load program intro address space
4. Allocate user-level and kernel-level stacks.
5. Copy arguments (if any) to the base of the user-level stack
6. Simulate an interrupt
   a) push on kernel stack initial PC, user SP
   b) push PSW (supervisor mode off, interrupts enabled)
7. Clear all other registers
8. RETURN_FROM_INTERRUPT
Interrupt Handling
on x86

User-level Process

foo() {
    while(...) {
        x = x+1;
        y = y-2
    }
}

Stack

Registers

Kernel

Code

handler() {
    pusha
    ...
}

Interrupt Stack
Interrupt Handling on x86

User-level Process

```cpp
foo() {
    while(...) {
        x = x+1;
        y = y-2
    }
}
```

Stack

Registers

```
SS:ESP
CS:EIP
EFLAGS
Other
Registers:
EAX, EBX,
...
```

Hardware performs these steps

1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack

Kernel

```
handler() {
    pusha
    ...
}
```
Interrupt Handling on x86

User-level Process

```
foo() {
    while(...) {
        x = x+1;
        y = y-2
    }
}
```

Registers

- SS:ESP
- CS:EIP
- EFLAGS
- Other Registers: EAX, EBX, ...

Kernel

```
handler() {
    pusha
    ...
}
```

Hardware performs these steps:
1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack
Interrupt Handling on x86

User-level Process

Code
foo() {
    while(...) {
        x = x+1;
        y = y-2
    }
}

Stack

Registers

SS:ESP
CS:EIP
EFLAGS
Other
Registers: EAX, EBX, ...

Kernel

Code
handler() {
    pusha
    ...
}

Interrupt Stack

SS:ESP
CS:EIP
EFLAGS

Hardware performs these steps:
1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack
Interrupt Handling on x86

User-level Process

Code

foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}

Stack

Registers

SS:ESP
CS:EIP
EFLAGS
Other
Registers: EAX, EBX, ...

Kernel

Code

handler() {
  pusha
  ...
}

Interrupt Stack

SS:ESP
CS:EIP
EFLAGS

Hardware performs these steps

1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack
6. Save error code (optional)
Interrupt Handling on x86

User-level Process

Code
foo() {
  while(...) {
    x = x + 1;
    y = y - 2
  }
}

Stack

Registers

SS:ESP
CS:EIP
EFLAGS
Other Registers: EAX, EBX, ...

Kernel

Code
handler() {
  pusha
  ...
}

Interrupt Stack

Hardware performs these steps

1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack
6. Save error code (optional)
Interrupt Handling on x86

User-level Process

Code

foo() {
    while(...) {
        x = x+1;
        y = y-2
    }
}

Stack

Registers

SS:ESP
CS:EIP
EFLAGS
Other Registers: EAX, EBX, ...

Kernel

Code

handler() {
    pusha
    ...
}

Interrupt Stack

SS:ESP
CS:EIP
EFLAGS
Error

Hardware performs these steps

1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack
6. Save error code (optional)
7. Transfer control to interrupt handler

Software (handler) performs this step

8. Handler pushes select registers on stack
Interrupt Handling on x86

User-level Process

Code
foo() {
    while(...) {
        x = x+1;
        y = y-2
    }
}

Stack

Registers

SS:ESP
CS:EIP
EFLAGS
Other
Registors: EAX, EBX, ...

Hardware performs these steps
1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack
6. Save error code (optional)
7. Transfer control to interrupt handler

Software (handler) performs this step
8. Handler pushes select registers on stack

Kernel

Code
handler() {
    pusha
    ...
}

Interrupt Stack

SS:ESP
CS:EIP
EFLAGS
Error
Interrupt Handling on x86

User-level Process

Code

```c
foo() {
    while(...) {
        x = x+1;
        y = y-2
    }
}
```

Stack

Registers

- SS:ESP
- CS:EIP
- EFLAGS
- Other Registers: EAX, EBX, ...

Kernel Code

```c
handler() {
    pusha ...
}
```

Interrupt Stack

- SS:ESP
- CS:EIP
- EFLAGS
- Error
- Select Registers: SS, ESP, EAX, EBX, ...

Hardware performs these steps:

1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack
6. Save error code (optional)
7. Transfer control to interrupt handler

Software (handler) performs this step:

8. Handler pushes select registers on stack
Interrupt Safety

- Kernel should disable device interrupts as little as possible
  - Interrupts are best serviced quickly
- Thus, device interrupts are often disabled selectively
  - E.g., clock interrupts enabled during disk interrupt handling
- This leads to potential “race conditions”
  - System’s behavior depends on timing of uncontrollable events
Interrupt Race Example

- Disk interrupt handler enqueues a task to be executed after a particular time while clock interrupts are enabled.
- Clock interrupt handler checks queue for tasks to be executed and may remove tasks from the queue.
- Clock interrupt may happen during enqueue.

Concurrent access to a shared data structure (the queue!)
Making code interrupt-safe

- Make sure interrupts are disabled while accessing mutable data!
- But don’t we have locks?

Consider

```c
void function ()
{
    lock(mtx);
    /* code */
    unlock(mtx);
}
```

Is function thread-safe?
Operates correctly when accessed simultaneously by multiple threads
To make it so, grab a lock

Is function interrupt-safe?
Operates correctly when called again (re-entered) before it completes
To make it so, disable interrupts
Example of Interrupt-Safe Code

```c
void enqueue(struct task *task) {
    int level = interrupt_disable();
    /* update queue */
    interrupt_restore(level);
}
```

- Why not simply re-enable interrupts?

  Say we did. What if then we call `enqueue` from code that expects interrupts to be disabled?

    Oops...

    Instead, remember interrupt level at time of call; when done, restore that level
Many Standard C Functions are not Interrupt-Safe

- Pure system calls are interrupt-safe
e.g., read(), write(), etc.

- Functions that don’t use global data are interrupt-safe
e.g., strlen(), strcpy(), etc.

- malloc(), free(), and printf() are not interrupt-safe
  must disable interrupts before using them in an interrupt handler
  and you may not want to anyway (printf() is huge!)

But they are all thread-safe!
System calls

Programming interface to the services the OS provides:

- read input/write to screen
- create/read/write/delete files
- create new processes
- send/receive network packets
- get the time / set alarms
- terminate current process
...

The Skinny

- Simple and powerful interface allows separation of concern
  - Eases innovation in user space and HW
- “Narrow waist” makes it highly portable
  - Robust (small attack surface)
- Internet IP layer also offers skinny interface

- Much care spent in keeping interface secure
  - E.g., parameters first copied to kernel space, then checked to prevent user program from changing them after they are checked!

<table>
<thead>
<tr>
<th>Portable OS Kernel</th>
<th>Portable OS Library</th>
<th>Web Browsers</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>ARM</td>
<td>PowerPC</td>
<td>10Mbps/100Mbps/1Gbps Ethernet</td>
</tr>
<tr>
<td>1802.11 a/b/g/n</td>
<td>SCSI</td>
<td>Graphics accelerators</td>
<td>LCD Screens</td>
</tr>
<tr>
<td>Compilers</td>
<td>Databases</td>
<td>Word Processing</td>
<td>Web Servers</td>
</tr>
</tbody>
</table>
Executing a System Call

- **Process:**
  - Calls system call function in library
  - Places arguments in registers and/or pushes them onto user stack
  - Places syscall type in a dedicated register
  - Executes syscall machine instruction

- **Kernel**
  - Executes syscall interrupt handler
  - Places result in dedicated register
  - Executes RETURN_FROM_INTERRUPT

- **Process:**
  - Executes RETURN_FROM_FUNCTION
int main(argc, argv) {
    c = read(fd, buffer, nbytes)
    ...
}

UPC: user program counter
USP: user stack pointer
KPC: kernel program counter
KSP: kernel stack pointer

note: interrupt stack is empty while process running
int main(argc, argv) {
    ...
    c = read(fd, buffer, nbytes)
}

_executing_read_
_system_call_

_usp_

_ksp_

_user space_

_kernel space_

_UPC: user program counter_

_KPC: kernel program counter_

_USP: user stack pointer_

_KSP: kernel stack pointer_

_note: interrupt stack is empty while process running_
Executing read System Call

int main(argc, argv){
    ...
    c = read(fd, buffer, nbytes)
    ...
}

_read:
mov READ, %R0
syscall
return

user space

kernel space

UPC: user program counter
USP: user stack pointer
KPC: kernel program counter
KSP: kernel stack pointer
note: interrupt stack is empty while process running
Executing read System Call

```
int main(argc, argv){
    ...
    c = read(fd, buffer, nbytes)
    ...
}

_read:
    mov READ, %R0
    syscall
    return
```

```
HandleIntrSyscall:
  push %Rn
  ...
  push %R1
  call _handleSyscall
  pop %R1
  ...
  pop %Rn
  return_from_interrupt
```
int main(argc, argv){
    ...
    c = read(fd, buffer, nbytes)
    ...
}

_read:
    mov READ, %R0
    syscall
    return

HandleIntrSyscall:
    push %Rn
    ...
    push %R1
    call __handleSyscall
    pop %R1
    pop %Rn
    return_from_interrupt
### Executing read System Call

```c
int main(argc, argv){
    ...
    c = read(fd, buffer, nbytes)
    ...
}
```

HandleIntrSyscall:
- push %Rn
- ...
- push %R1
- call _handleSyscall
- pop %R1
- pop %Rn
- return_from_interrupt

**Diagram:***
- **User Space (USP)**
  - **User Stack**
  - **Stack Frame for main()**
    - **Address**
    - **Return**
- **Kernel Space (KSP)**
  - **Interrupt Stack**
  - **USP, UPC, PSW**
  - **Saved Registers**
- **HandleIntrSyscall**
  - push %Rn
  - ...
  - push %R1
  - call _handleSyscall
  - pop %R1
  - pop %Rn
  - return_from_interrupt

**Stack Frames:**
- **Stack Frame for main()**
- **Stack Frame for _read**
Executing read System Call

```c
int main(argc, argv) {
    ... 
    c = read(fd, buffer, nbytes) 
    ... 
}

_read: 
    mov READ, %R0
    syscall 
    return
```

```
HandleIntrSyscall: 
    push %Rn 
    ... 
    push %R1 
    call __handleSyscall 
    pop %R1 
    pop %Rn 
    return_from_interrupt
```

```
int handleSyscall(int type) {
    switch (type) {
        case READ: ... 
    }
}
```
Executing read System Call

```c
int main(argc, argv){
    ...
    c = read(fd, buffer, nbytes)
    ...
}

_read:
    mov READ, %R0
syscall
return

// stack frame for main()

stack frame for _read

user stack

HandleIntrSyscall:
push %Rn
...
push %R1
call _handleSyscall return address
pop %R1
...
pop %Rn
return_from_interrupt

// stack frame for handleSyscall()

USP, UPC, PSW
saved registers

interrupt stack

int handleSyscall(int type){
    switch (type) {
    case READ: ...
    }
}

USP, UPC, PSW
saved registers

interrupt stack

KPC

int handleSyscall(int type){
    switch (type) {
    case READ: ...
    }
}
```

user space

kernel space
What if read needs to block?

- read may need to block if
  - It reads from a terminal
  - It reads from disk, and block is not in cache
  - It reads from a remote file server

  We should run another process!
How to run multiple processes
The Problem

- Say (for simplicity) we have a single core CPU
- A process physically runs on the CPU
- Yet each process somehow has its own
  Registers
  Memory
  I/O Resources
  “thread of control”
- Need to multiplex/schedule to create virtual CPUs for each process
Process Control Block

- A per-process data structure held by OS, with
  location in memory (page table)
  location of executable on disk
  id of user executing this process (uid)
  process identifier (pid)
  process status (running, waiting, etc.)
  scheduling info
  interrupt stack
  saved kernel SP (when process is not running)
    points into interrupt stack
    interrupt stack contains saved registers and kernel call stack for this process
  ...and more