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)

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

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

int g(y)
{
    ...
}
```

- **PC/IP**
- **SP**
- **FP**
- **user stack**
- **arguments (3.14)**
- **return address**
- **saved FP (main)**
- **local variables**
- **saved registers**
- **scratch space**
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
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

{ only need to save registers not saved by the handler function

{ restore the registers saved above

RETURN_FROM_INTERRUPT
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

Code

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

Stack

Kernel

Code

handler() {
  pusha
  ...
}

Interrupt Stack

Registers

Stack segment

Code segment

EFLAGS

Other Registers:
EAX, EBX, ...

SS:ESP

CS:EIP

Program counter

Stack pointer

Offset

Flags

Interrupt 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, ...

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

Code
handler() {
  pusha
  ...
}

Interrupt 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**

#### 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

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

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)
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)
7. Transfer control to interrupt handler
8. Handler pushes select registers on stack

Software (handler) performs this step
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

Other Registers: EAX, EBX, ...

Kernel

Code
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
  - 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!

Portable OS Kernel

System call interface

Portable OS Library

Compilers

Databases

Web Browsers

Email

Web Servers

Word Processing

SCSI

Graphics accelerators

1802.11 a/b/g/n

10Mbps/100Mbps/1Gbps Ethernet

x86 ARM PowerPC

SCSI

Linux

x86 ARM PowerPC

1802.11 a/b/g/n

10Mbps/100Mbps/1Gbps Ethernet

Graphics accelerators

LCD Screens
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)
    ...  
}

Note: interrupt stack is empty while process running

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

UPC → \[ \rightarrow \] KSP
USP → \[ \rightarrow \] interrupt stack

user space
kernel space

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

_executing:

_user space

_kernel space

_USERP: user program counter
_KPC: kernel program counter
_USERSP: user stack pointer
_KSP: kernel stack pointer

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

_read:
    mov R:EAD, %R0
    syscall
    return

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

__read:
    mov RREAD, %R0
    syscall
    return

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

user space

kernel space
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
```

```
user space

kernel space
```

```
USP, UPC, PSW
```

```
USP, UPC
```
Executing read System Call

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

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

user space

kernel space

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

USP, UPC, PSW

saved registers

interrupt stack

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

_read:
    mov R:READ, %R0
    syscall
    return

int handleSyscall(int type) {
    switch (type) {
    case READ: ...
    }
}
int main(argc, argv){
    ...
    c = read(fd, buffer, nbytes)
    ...
}

_read:
    mov R:READ, %R0
    syscall
    return

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

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