Interrupts and System Calls

Seyyed Ahmad Javadi

sajavadi@aut.ac.ir

(Based on slides by Don Porter at UNC)

https://www.cs.unc.edu/~porter/

Questions from last session

```
section .data                                   ;Data segment
   userMsg db 'Please enter a number: '         ;Ask the user to enter a number
   lenUserMsg equ $-userMsg                     ;The length of the message
   dispMsg db 'You have entered: '
   lenDispMsg equ $-dispMsg

section .bss              ;Uninitialized data
   num resb 5

section .text             ;Code Segment
   global _start

_start:                   ;User prompt
   mov eax, 4
   mov ebx, 1
   mov ecx, userMsg
   mov edx, lenUserMsg
   int 80h
```

db: data byte

equ: equivalent

resb: reserve byte

https://www.tutorialspoint.com/assembly_programming/assembly_basic_syntax.htm

Questions from last session

- Compiler vs interpreter

- int instruction
  - The instruction generates a software call to an interrupt handler.
  - Example
    - Trap to debugger: int $3
    - Trap to interrupt 0xff: int $0xff

https://docs.oracle.com/cd/E19455-01/806-3773/instructionset-74/index.html

Background: Control Flow

```
x = 2, y = true
if (y) {
    2 /= x;
    printf(x);
} //...
```

```
void printf(va_args)
{
    //...
}
```

Background: Control Flow

pc

```
x = 2, y = true
if (y) {
    2 /= x;
    printf(x);
} //...
```

```
void printf(va_args)
{
    //...
}
```

Background: Control Flow

pc

```
x = 2, y = true
if (y) {
    2 /= x;
    printf(x);
} //...
```

```
void printf(va_args)
{
    //...
}
```

Background: Control Flow

```
x = 2, y = true
if (y) {
    2 /= x;
    printf(x);
} //...
```

pc

```
void printf(va_args)
{
    //...
}
```

Background: Control Flow

```
x = 2, y = tr
if (y) {
    2 /= x;
    printf(x);
} //...
```

pc

```
void printf(va_args)
{
    //...
}
```

Background: Control Flow

```
x = 2, y = true
if (y) {
    2 /= x;
    printf(x);
} //...
```

```
void printf(va_args)
{
    //...
}
```

pc

Background: Control Flow

```
x = 2, y = true
if (y) {
    2 /= x;
    printf(x);
} //...
```

pc

```
void printf(va_args)
{
    //...
}
```

Regular control flow: branches and calls (logically follows source code)

Background: Control Flow

```
x = 0, y = true
if (y) {
    2 /= x;
    printf(x);
} //...
```

Background: Control Flow

pc

```
x = 0, y = true
if (y) {
    2 /= x;
    printf(x);
} //...
```

Background: Control Flow

pc

```
x = 0, y = true
if (y) {
    2 /= x;
    printf(x);
} //...
```

Background: Control Flow

```
x = 0, y = true
if (y) {
    2 /= x;
    printf(x);
} //...
```

pc

Divide by zero!
Program can't make progress!

Background: Control Flow

```
x = 0, y = true
if (y) {
    2 /= x;
    printf(x);
} //...
```

pc

```
void
handle_divzero(){
    x = 2;
}
```

Divide by zero!
Program can't make progress!

Background: Control Flow

```
x = 0, y = true
if (y) {
    2 /= x;
    printf(x);
} //...
```

```
void
handle_divzero(){
    x = 2;
}
```

pc

Divide by zero!
Program can't make progress!

Background: Control Flow

```
x = 0, y = true
if (y) {
    2 /= x;
    printf(x);
} //...
```

pc

```
void
handle_divzero(){
    x = 2;
}
```

Divide by zero!
Program can't make progress!

Background: Control Flow

```
x = 0, y = true
if (y) {
    2 /= x;
    printf(x);
} //...
```

pc

```
void
handle_divzero(){
    x = 2;
}
```

Divide by zero!
Program can't make progress!

Irregular control flow: exceptions, system calls, etc.

Lecture goal

- Understand the hardware tools available for irregular control flow.
  - I.e., things other than a branch in a running program
- Building blocks for context switching, device management, etc.

Two types of interrupts

- **Synchronous**: will happen every time an instruction executes (with a given program state)
  - Divide by zero
  - System call
  - Bad pointer dereference

- **Asynchronous**: caused by an external event
  - Usually device I/O
  - Timer ticks (well, clocks can be considered a device)

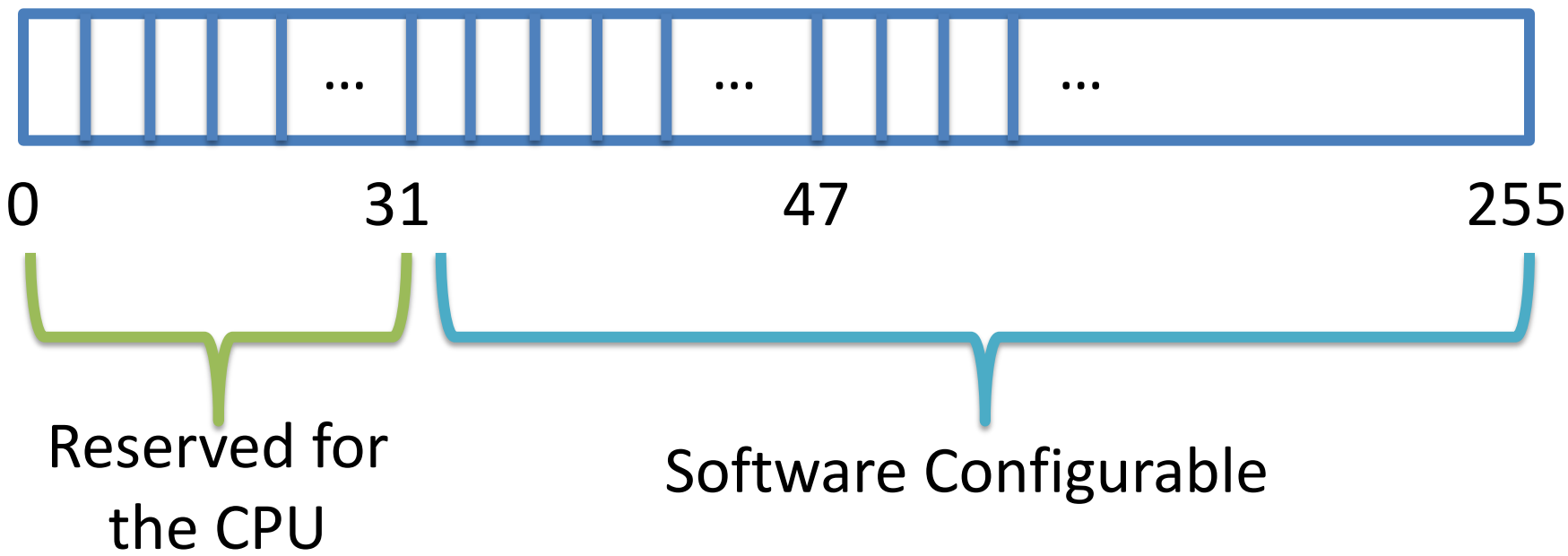
Intel nomenclature

- **Interrupt** – only refers to asynchronous interrupts
- **Exception** – synchronous control transfer
- Note: from the programmer's perspective, these are handled with the same abstractions

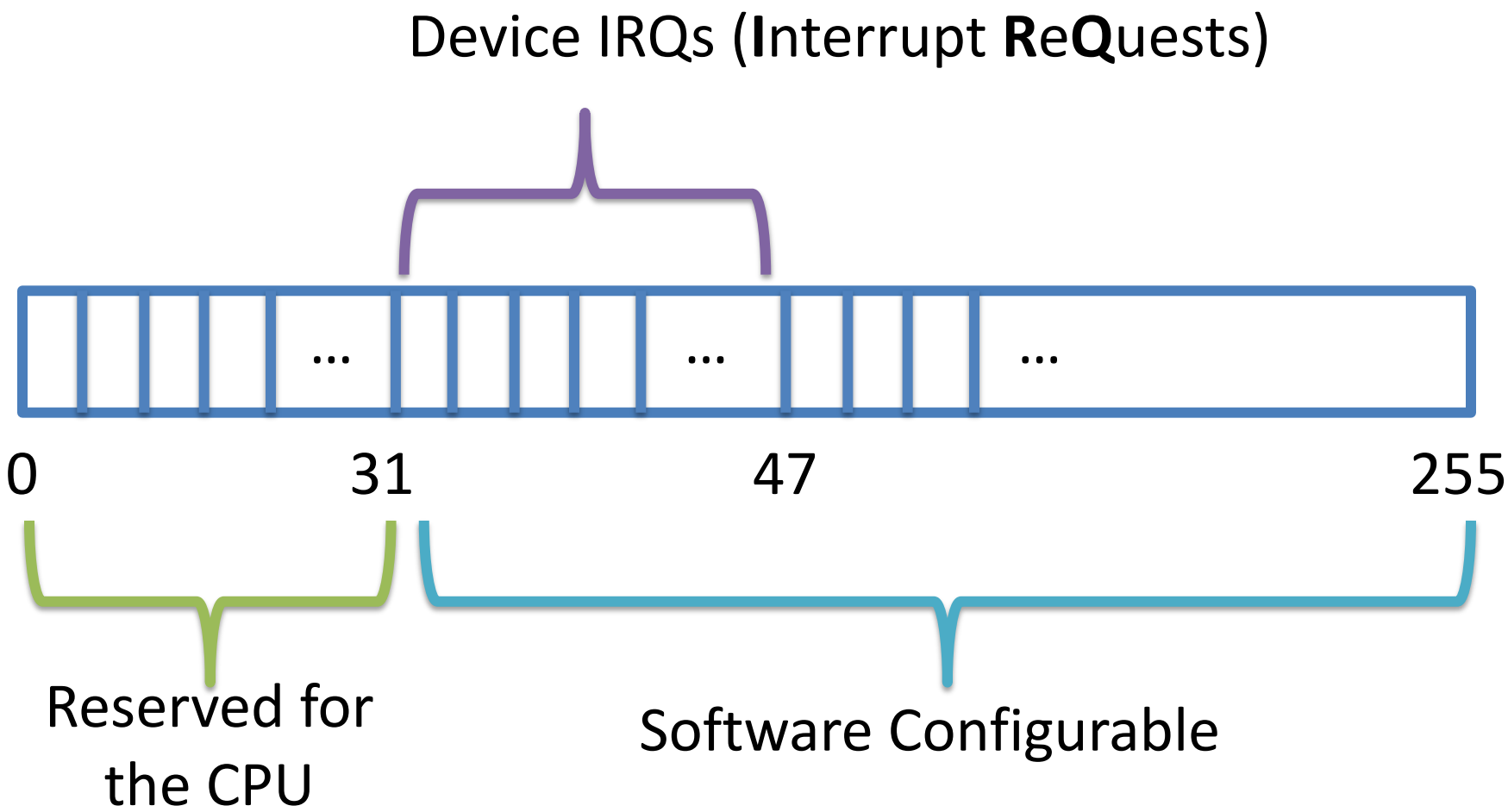
Interrupt overview

- Each interrupt or exception includes a number indicating its type.
- E.g., 14 is a page fault, 3 is a debug breakpoint.
- This number is the index into an interrupt table.

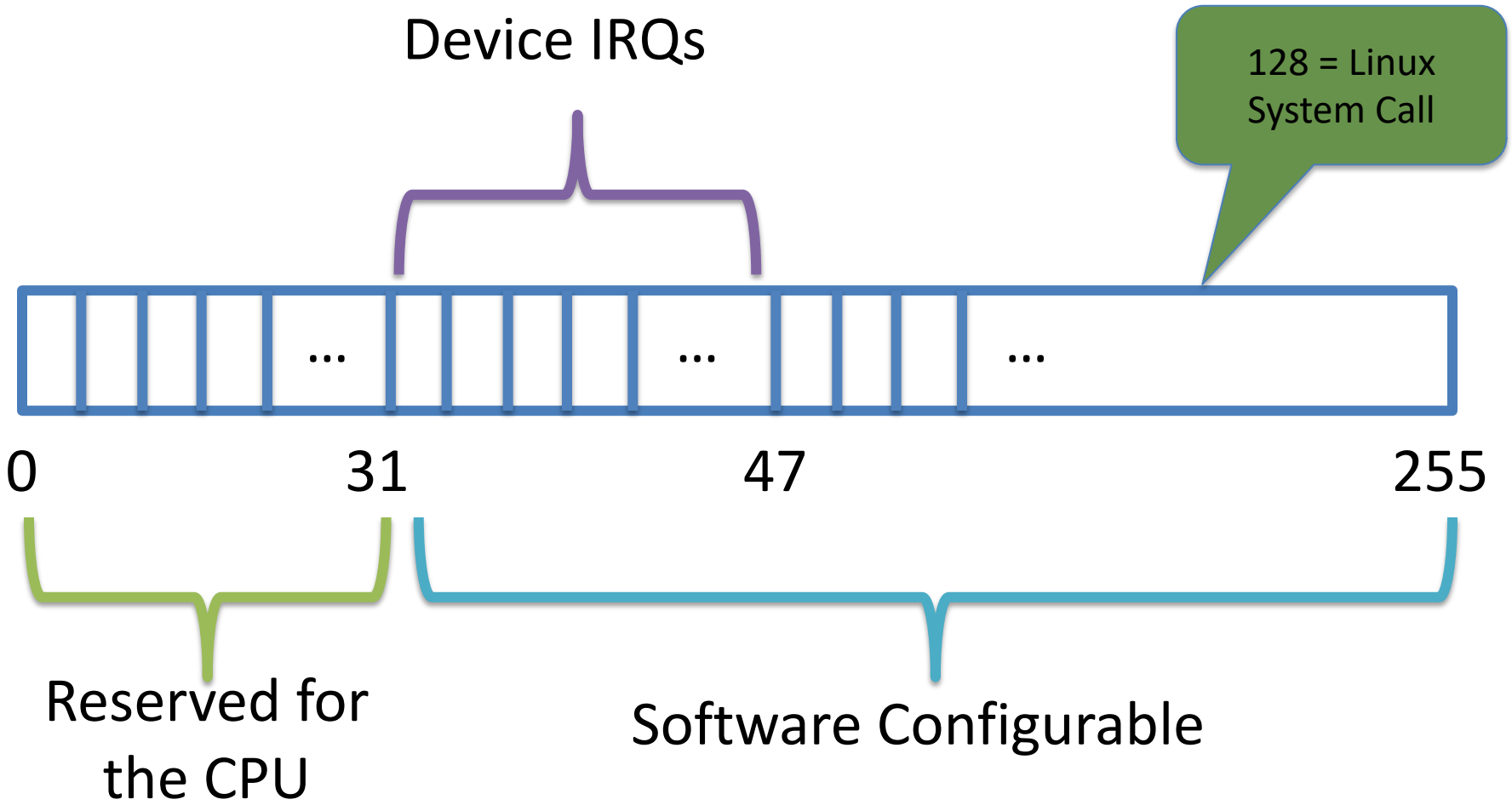
x86 interrupt table



x86 interrupt table



x86 interrupt table



x86 interrupt overview

- Each type of interrupt is assigned an index from 0—255.
- 0—31 are for processor interrupts; generally fixed by Intel
  - E.g., 14 is always for page faults
- 32—255 are software configured
  - 0x80 issues system call in Linux (more on this later)

Software interrupts

- The `int <num>` instruction allows software to raise an interrupt
  - 0x80 is just a Linux convention.
- There are a lot of spare indices
  - You could have multiple system call tables for different purposes or types of processes!
    - Windows does: one for the kernel and one for win32k

What happens (generally):

- Control jumps to the kernel
  - At a prescribed address (the interrupt handler)
- The register state of the program is dumped on the kernel's stack
- Kernel code runs and handles the interrupt
- When handler completes, resume program (see `iret` instr.)

System call “interrupt”

- Originally, system calls issued using `int` instruction
- Dispatch routine was just an interrupt handler
- Like interrupts, system calls are arranged in a table
  - See arch/x86/kernel/syscall_table*.S in Linux source
- Program selects the one it wants by placing index in `eax` register
  - Arguments go in the other registers by calling convention
  - Return value goes in `eax`