Running Programs on a System

Exceptional Control Flow

Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU's control flow (or flow of control)

Physical control flow

Time

<startup>
inst_1
inst_2
inst_3
…
inst_n
<shutdown>

Altering the Control Flow

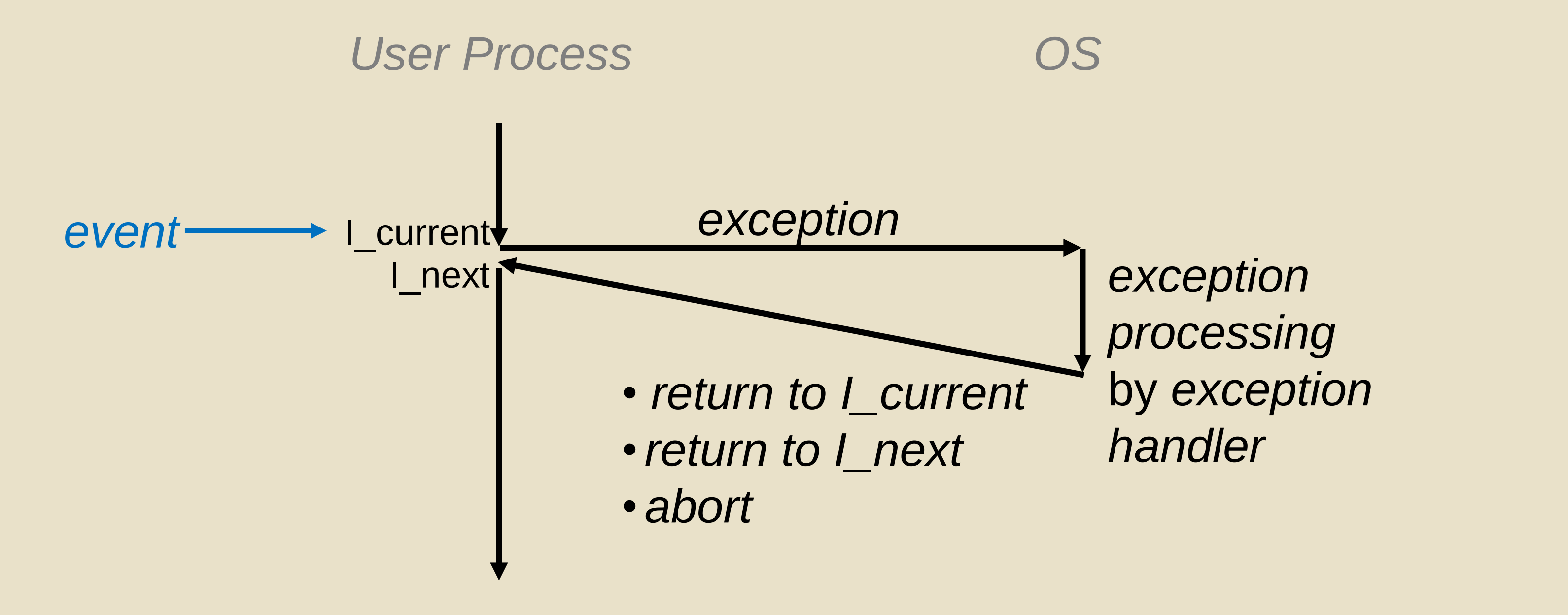
- Up to now: few mechanisms for changing control flow:
  - Jumps and branches, call and return
    - react to changes in program state
  - Signals
    - sent by the kernel
- But how exactly do we react to changes in system state?
  - data arrives from a disk or a network adapter
  - instruction divides by zero
  - user hits Ctrl-C at the keyboard
  - system timer expires
- System needs mechanisms for "exceptional control flow"

Exceptional Control Flow

- Exists at all levels of a computer system
- Low level mechanisms
  - Exceptions
    - change in control flow in response to a system event (i.e., change in system state)
  - Combination of hardware and OS software
- Higher level mechanisms
  - Process context switch
  - Signals
  - Nonlocal jumps: setjmp()/longjmp()
  - Implemented by either:
    - OS software (context switch and signals)
    - C language runtime library (nonlocal jumps)

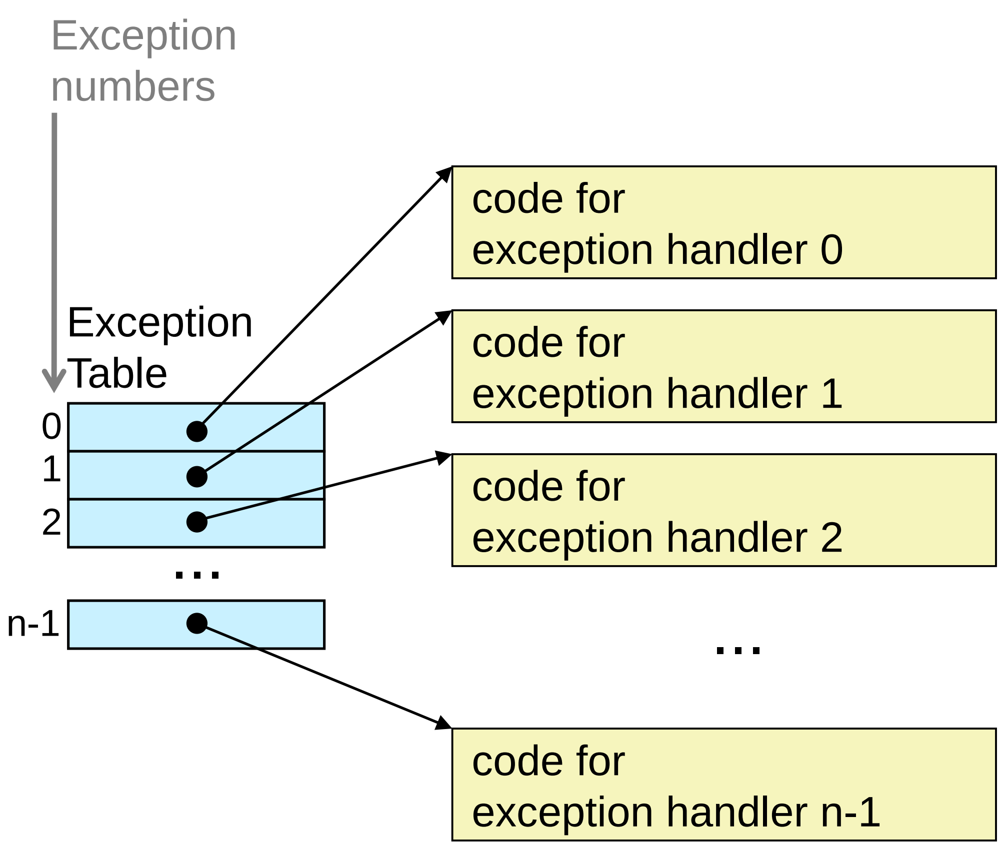
Exceptions

- Low-level exceptions provide the “glue” between hardware events and the higher-level mechanisms
  - transfer of control to the OS in response to some event (i.e., change in proces sor state)



- Examples:
  div by 0, arithmetic overflow, page fault, I/O request completes, Ctrl-C

Interrupt Vectors



- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs

Intel x86 Exceptions and Interrupts

Table 6-1. Exceptions and Interrupts

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

source:
Intel 64 and IA-32 Architectures Software Developer's Manual
http://download.intel.com/design/processor/manuals/253665.pdf

Exceptions and Processor Mode

- User programs run in **user mode**
  - restricted access to memory, devices, etc
  - to serve system calls and talk to the hardware, the process may need to run in kernel mode
  - Problem: how can a user process invoke a kernel function?

→ Exceptions cause a switch to kernel mode

CSE 컴퓨터공학부 Department of Computer Science & Engineering

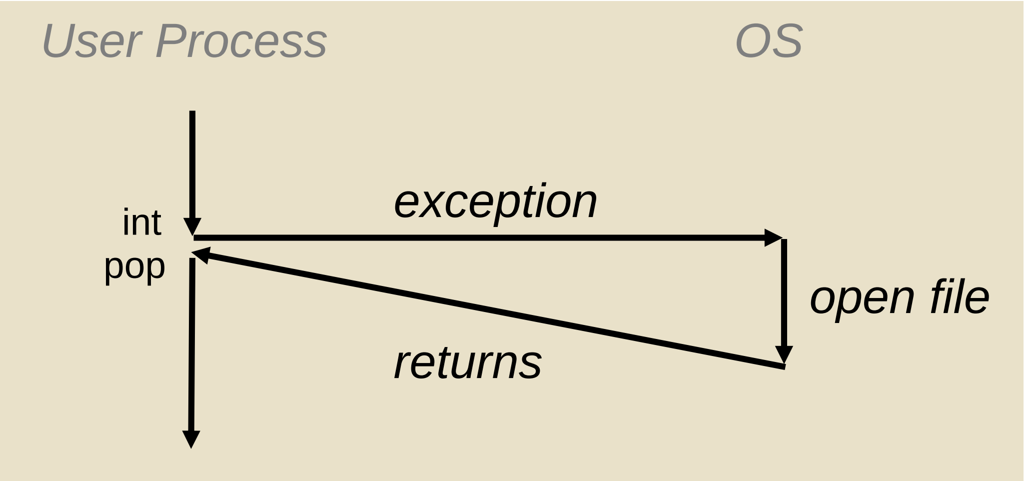
Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - Traps
    - Intentional
    - Examples: system calls, breakpoint traps, special instructions
    - Returns control to “next” instruction
  - Faults
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting (“current”) instruction or aborts
  - Aborts
    - unintentional and unrecoverable
    - Examples: parity error, machine check
    - Aborts current program

Trap Example: Opening File

- User calls: open(filename, options)
- Function open executes system call instruction int

```
0804d070 <__libc_open>:
 . . .
 804d082:       cd 80                    int     $0x80
 804d084:       5b                       pop     %ebx
 . . .
```



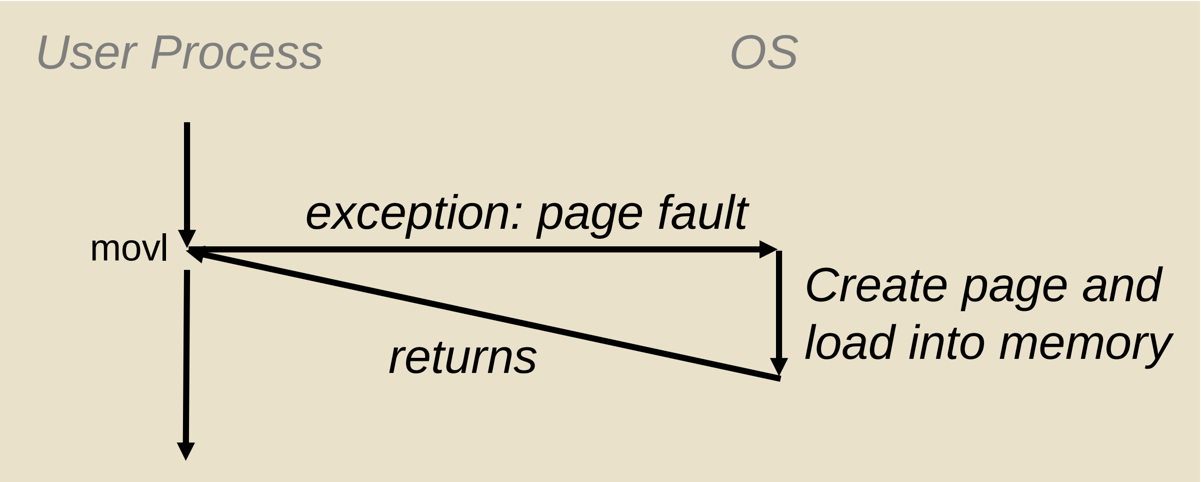
- OS must find or create file, get it ready for reading or writing
- Returns integer file descriptor

Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
80483b7:      c7 05 10 9d 04 08 0d    movl    $0xd,0x8049d10
```

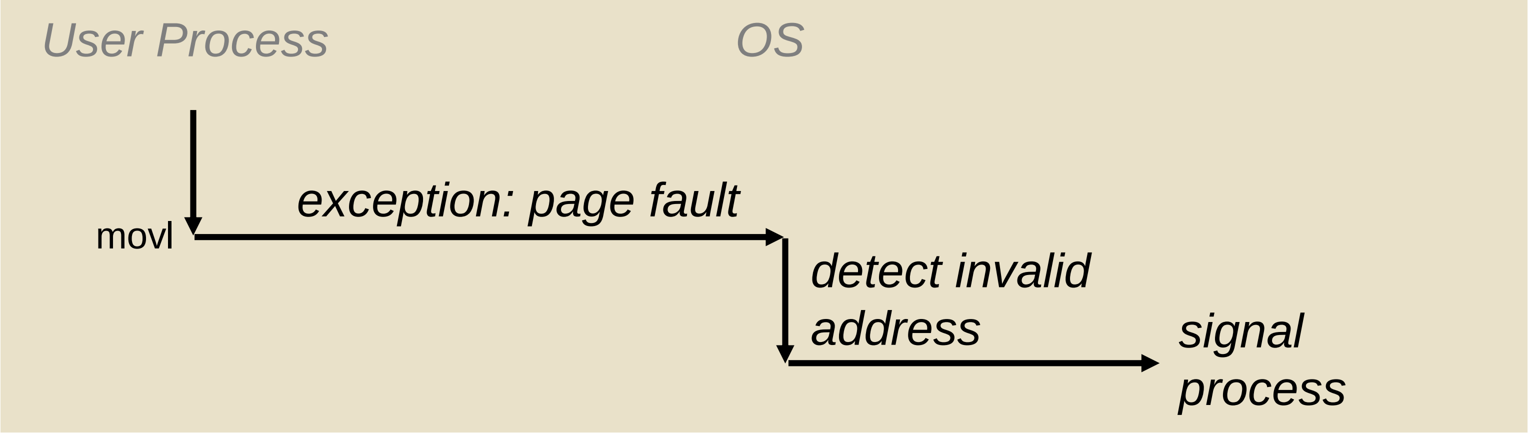


- Page handler must load page into physical memory
- Returns to faulting instruction
- Successful on second try

Fault Example: Invalid Memory Reference

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7:        c7 05 60 e3 04 08 0d    movl    $0xd,0x804e360
```



- Page handler detects invalid address
- Sends SIGSEGV signal to user process
- User process exits with “segmentation fault”

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor's interrupt pin
  - Handler returns to "next" instruction

- Examples:
  - I/O interrupts
    - hitting Ctrl-C at the keyboard
    - arrival of a packet from a network
    - arrival of data from a disk
  - Hard reset interrupt
    - hitting the reset button
  - Soft reset interrupt
    - hitting Ctrl-Alt-Delete on a PC

CSE 컴퓨터공학부 Department of Computer Science & Engineering

I/O Interrupts

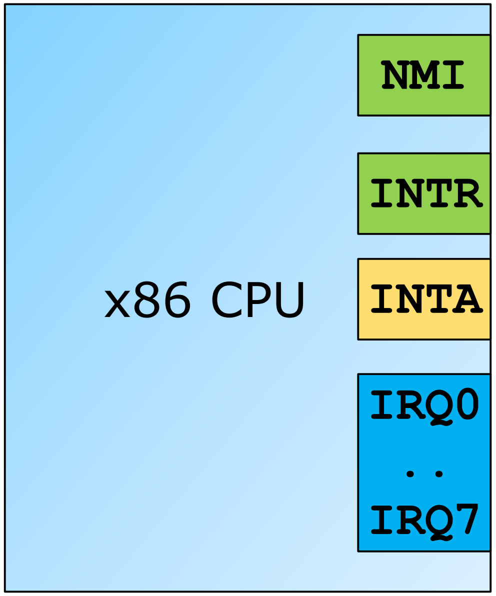
- CPU interrupt-request line triggered by I/O device
- Interrupt handler receives interrupt
- AND with *interrupt mask register* to ignore or delay some interrupts
- Interrupt vector routine determines the source of the interrupt and dispatches it to the correct handler
  - priority based
  - some interrupts are non-maskable (NMI)
- The same mechanism is used for exceptions

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Basic x86 Interrupts

- PIC: Intel 8259(A)
  - introduced in 1976, survived for 30 years, now being phased out in favor if the Intel APIC Architecture.
  - 8259A interface still provided by the southbridge chipset on x86 motherboards

- x86 CPUs

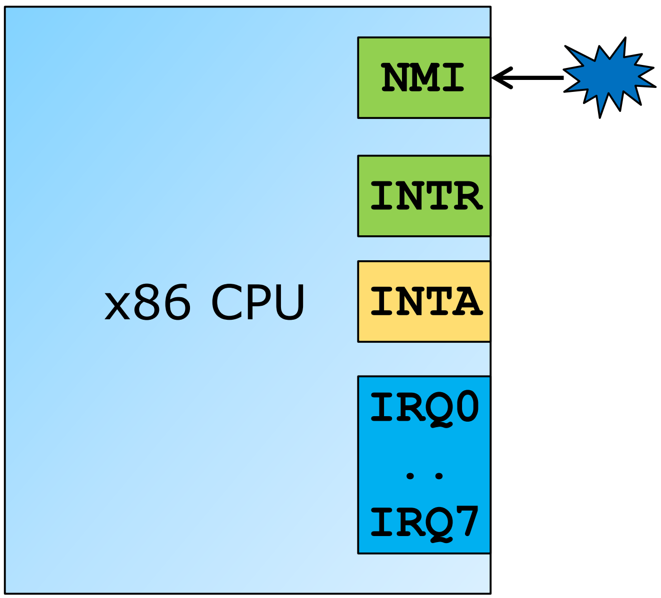


- INTR: interrupt request
- INTA: interrupt acknowledge
- IRQ0~7: interrupt request lines

- cascading possible

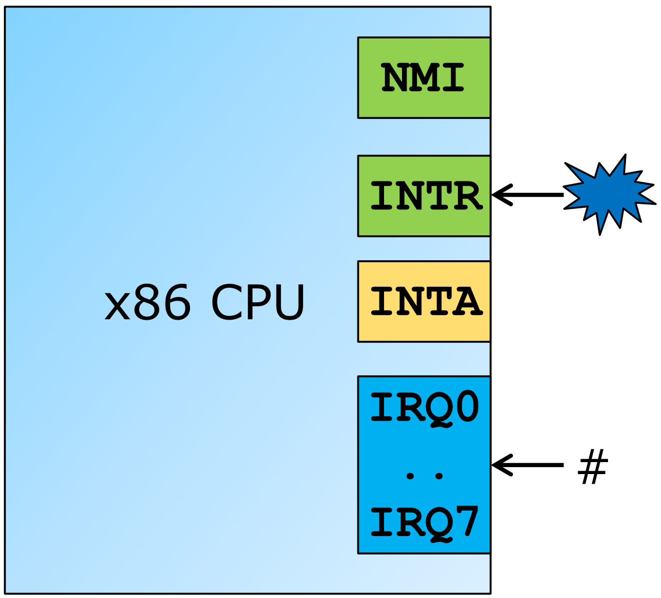
- NMI: non-maskable interrupt

x86 Interrupts: NMI



- NMI asserted
  - CPU completes current instruction
  - issues exception #2
- Cannot be disabled
- Reserved for
  - major hardware faults
    - memory parity error
    - watchdog timers
- Multiple sources: who caused the NMI?
  - OS handler must poll potential sources
  - for normal use: too slow, second NMI might occur while first is still being handled

x86 Interrupts: IRQ



- Interrupt request
  - can be disabled with the `CLI/STI` instruction (clears/sets IE status flag)
  - if enabled, complete current instruction, then
- CPU acknowledges interrupt using INTA pin
- Interrupt vector is then supplied on the data bus
- CPU issues exception # from the vector

Implementation of Linux/IA32 System Calls

- Application programs use system calls to request a service from the kernel
- System calls are invoked via the `int 0x80` instruction
  - parameters passed through registers (not the stack)
    - `eax`: system call number
    - `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`: (up to) 6 arbitrary arguments (interpretation depends on the invoked system call)
    - `esp` cannot be used (overwritten when the CPU enters kernel mode)
- libc provides convenient wrappers for most system calls

Implementation of Linux/IA32 System Calls

- Directly invoking Linux/IA32 System calls

```
#include <stdlib.h>

int main(void)
{
  write(1, "hello, world\n", 13);
  exit(0);
}
```

```
$ gcc -o hello hello.c
$ ./hello
hello, world
$
```

```
int main()
{
  my_write(1, "hello, world\n", 13);
  my_exit(0);
}
```

myhello.c

```
.section .text
.globl my_write
.globl my_exit

# void my_write(int fd, char *string,
                unsigned int length);
my_write:

  push   %ebp
  mov    %esp, %ebp
  push   %ebx

  mov    $4, %eax            # eax: 4 (write syscall)
  mov    8(%ebp), %ebx       # ebx: fd (ebp+8)
  mov    12(%ebp), %ecx      # ecx: string (ebp+12)
  mov    16(%ebp), %edx      # edx: length (ebp+16)
  shr    $1, %edx            # ecx: string (ebp+12)

  int    $0x80               # invoke system call

  pop    %ebx
  pop    %ebp

  ret

…
```

mylib.c

Linux/IA32 System Calls

- Directly invoking Linux/IA32 System calls

```
int main()
{
  my_write(1, "hello, world\n", 13);
  my_exit(0);
}                                myhello.c
```

```
…

# void my_exit(int nr)
my_exit:

  push  %ebp
  mov   %esp, %ebp
  push  %ebx

  mov   $1, %eax            # eax: 1 (exit syscall)
  mov   8(%ebp), %ebx       # ebx: nr (ebp+8)

  int   $0x80               # invoke system call

  pop   %ebx
  pop   %ebp

  ret                                       mylib.c
```

```
$ gcc -o my_hello myhello.c mylib.s
$ ./my_hello
hello, world
$
```


CSE 컴퓨터공학부 Department of Computer Science & Engineering