CS 201

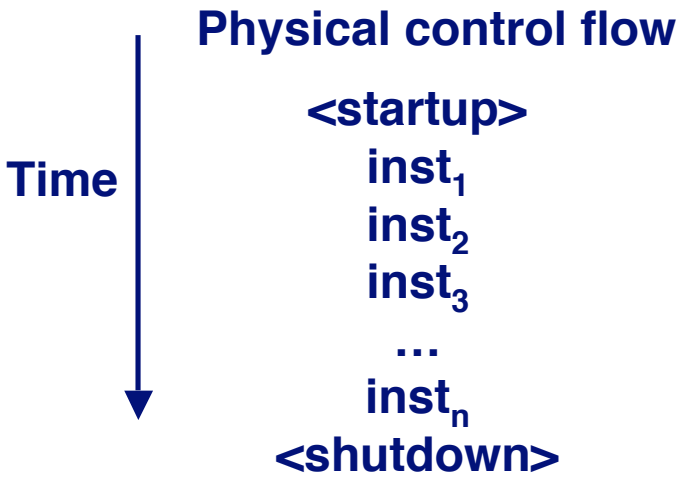
Exceptions and Processes

Gerson Robboy
Portland State University

Control Flow

Computers Do One Thing

- From startup to shutdown, a CPU reads and executes (interprets) a sequence of instructions, one at a time.



Altering the Control Flow

Up to Now: we know the flow of control is not always sequential

- Jumps and branches
- Call and return

The O. S. also needs to react to events.

Certain events change the state of the CPU:

- data arrives from a disk or a network adapter.
- An instruction divides by zero
- Instruction accesses a memory address that doesn't exist
- User hits control-c at the keyboard
- System timer expires

System needs mechanisms for "exceptional control flow"

Exceptional Control Flow

Some events cause abrupt changes to flow of control.

Exceptions are kind of like procedure calls into the kernel

- Change in control flow in response to an event
- Combination of hardware and OS software

Example

User code dereferences a NULL pointer

- **Not a valid address in user space.**

What happens next?

What does the CPU do?

Where does flow of control go?

What does the operating system do?

Two kinds of exceptions

Synchronous and asynchronous

Synchronous exceptions are traps and faults

- Initiated by software
- "Synchronous" means in the flow
  - As in, "It's synchronicity, man."
- What's an example?

Asynchronous exceptions are interrupts

- Initiated by hardware
- "Asynchronous" means out of the blue
- What's an example?

Asynchronous Exceptions (Interrupts)

Events external to the processor

Suppose you have a CPU and many devices.

At any moment each device may have data available

- The user may have hit a key on the keyboard
- There may be an incoming packet on the network
- The CPU may have requested data from the disk, and that sector may be rolling around under the read head now

One method: Poll all devices for data

How often?

- What happens if you poll too often?
- Not often enough?
- What if a subroutine runs a long time and has no polling code in it?

Asynchronous Exceptions (Interrupts)

Examples:

- I/O interrupts
  - hitting a key at the keyboard
  - arrival of a packet from a network
  - arrival of a data sector from a disk
- Hard reset interrupt
  - hitting the reset button
- Timer interrupt

A procedure in the kernel ("handler") handles in the event.

- Flow of control returns to the interrupted code
- Like a call/return.

More about Interrupts

An interrupt controller "raises" interrupts

- **We're talking about hardware here**
- **The controller may be integrated in the CPU or a separate component**
- **Different events correspond to different interrupt levels**

The CPU checks its interrupt pin after each instruction.

The CPU responds to an interrupt by changing the flow of control *after* (not during) execution of a machine instruction.

Synchronous Exceptions

Events that occur as a result of executing an instruction:

- **Traps**
  - **Intentional**
  - **Examples: system calls, breakpoint traps**
- **Faults**
  - **Unintentional but possibly recoverable**
  - **Examples: page faults (recoverable), protection faults (unrecoverable).**
- **Aborts**
  - **unintentional and unrecoverable**
  - **Examples: parity error, machine check.**

Traps

A software interrupt.

- **A mechanism for making system calls into the kernel.**

On ia32, the *int* instruction invokes an interrupt.

The interrupt handler, in this case, serves a system call.

- **After it's done, then what?**
- **Where does control resume?**

Why do we need traps?

Why not just do system calls by calling to an address in the kernel?

Faults

A software error requiring action by the kernel.

- An instruction in user code accesses a not-present page, causing a page fault. The handler in the kernel gets the page, so now it is accessible. What next?
- An instruction in user code accesses a non-existent page, causing a page fault. The handler in the kernel finds that this is not a valid address for this process. What should the kernel do?
- User code generates a floating point overflow. The kernel has a handler that fixes up the result properly; e. g., sets the result to infinity. Then what?
- An instruction in user code divides by zero. How should the kernel handle this?

Aborts

A hardware error requiring action by the kernel.

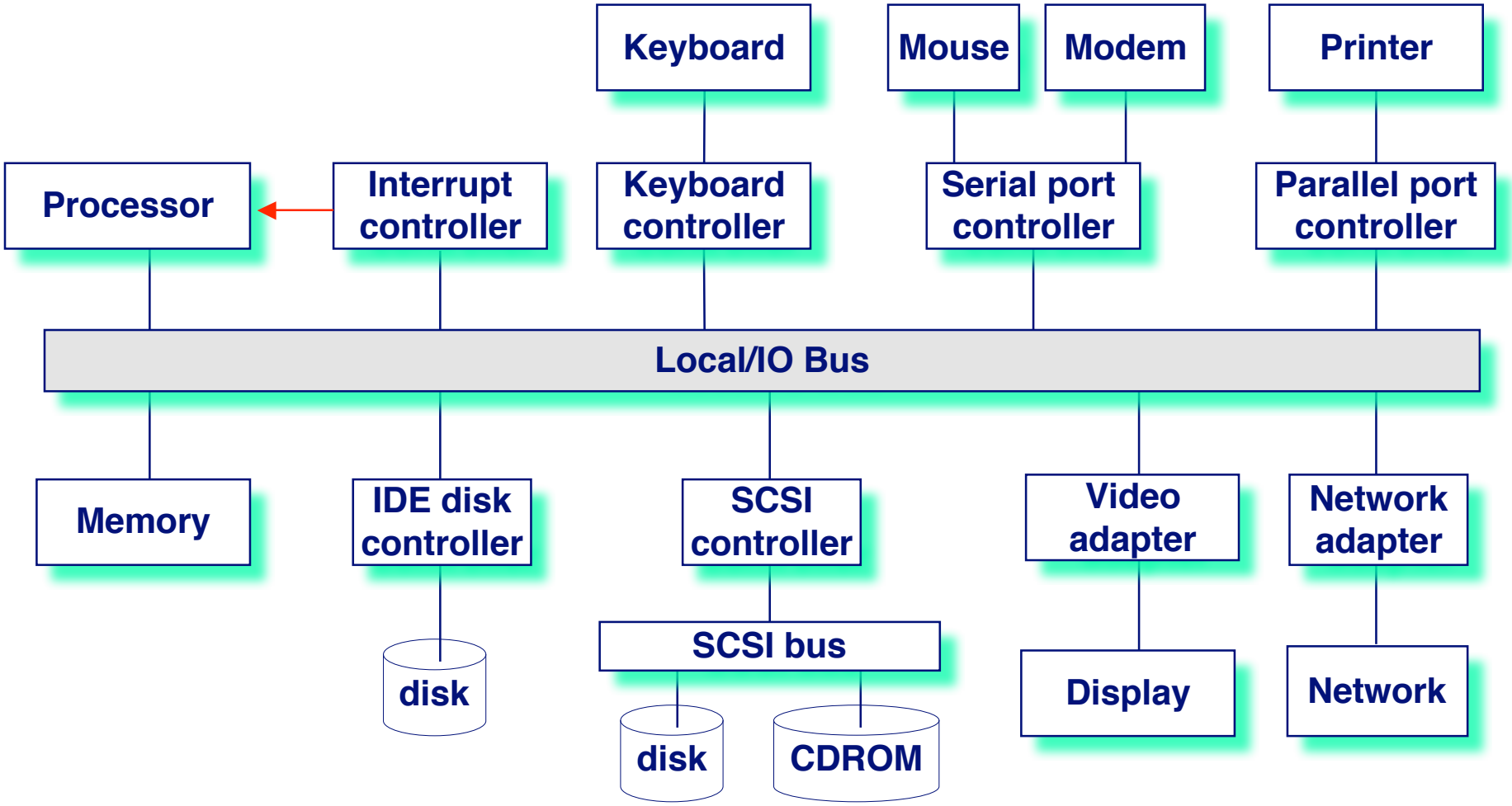
Example: The computer's memory is physically broken at a certain address. Acess to that address causes a "machine check." The kernel should isolate that part of memory and not use it again.

There is no recovery. The faulting process must be aborted.

- But the operating system itself may be able to keep running

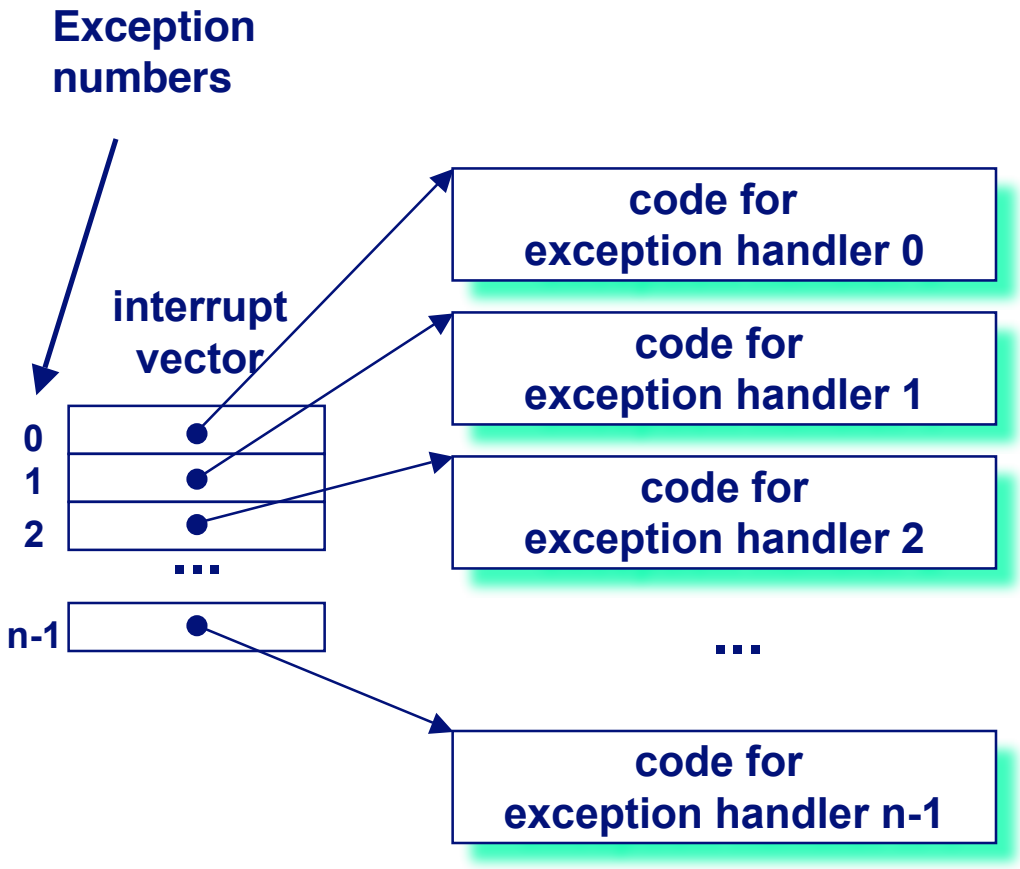
What if a machine check occurs at an address in the kernel?

System context for exceptions



How an exception gets to the handler

An interrupt vector is an array of pointers in memory, owned by the O. S. kernel.



- Each type of event has a unique exception number k
- Jump table entry k is a pointer to to a function (exception handler) in the kernel.
- The CPU hardware calls handler k each time exception k occurs.

More about interrupt vectors

An interrupt vector is an array in memory of pointers to procedures, nothing more.

- **Each entry is initialized with a pointer to the handler for that level of interrupt.**
- **How does the vector get initialized?**

When an exception occurs, the hardware causes a sort of indirect jump via that pointer to the handler.

- **The hardware also saves the information necessary to return to the interrupted code**

What prevents us, the users, from writing our own interrupt vector, and then handling the exceptions ourselves?

Does the CPU hardware inherently prevent this?

What problem would it cause if we could do it?

Process

Definition: A *process* is an instance of a running program.

- **One of the most fundamental concepts in computer science.**
- **Not the same as "program" or "processor"**

A *program* is a set of instructions and initialized data in a file, usually found on a disk.

A *process* is an instance of that program while it is running, along with the state of all the CPU registers and the values of data in memory.

Process

This is an operating system concept.

The operating system manages a set of processes.

At a given moment in time, one process is running on the CPU.

- Other processes are waiting for a chance to have CPU time.
- The Operating System schedules processes so they each get their share of CPU time.

So the O. S. kernel gives each process a *virtual machine*.

For any one process, the computer behaves as though that process owns its own computer, complete with CPU and dedicated memory.

To that process, what do other processes look like?

To that process, what do I/O devices look like?

Processes

The *process* concept provides each program with two key abstractions:

- Logical control flow
  - Each program seems to have exclusive use of the CPU.
- Private address space
  - Each program seems to have exclusive use of main memory.

How are these Illusions maintained?

- Process executions interleaved (multitasking)
- The O. S. preserves the CPU context for each process
- The O. S. restores CPU context when it schedules a process
- Address spaces managed by virtual memory system

So, suppose a process is running in *user space*.

An exception occurs. What happens?

What code runs?

By the way, what do we mean by user space?
How is it different from kernel space?

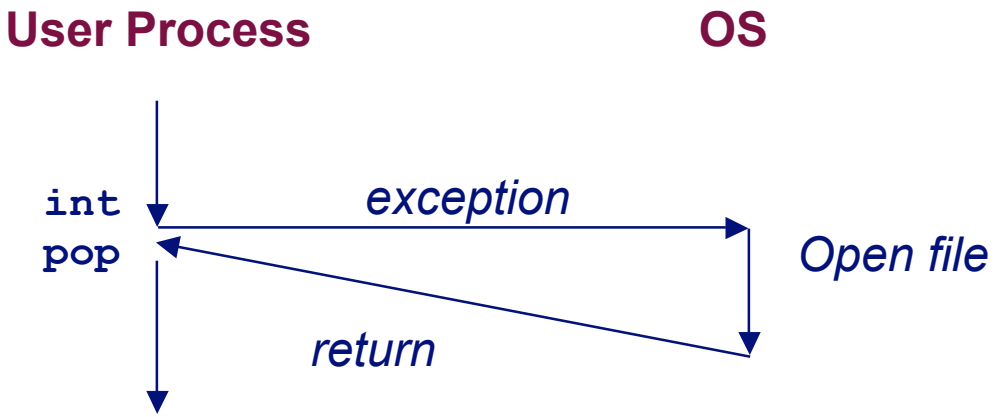
Trap Example

Opening a File

- User calls `open(filename, options)`

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

- The return from the interrupt is to the next instruction
- Returns integer file descriptor



Fault Example #1

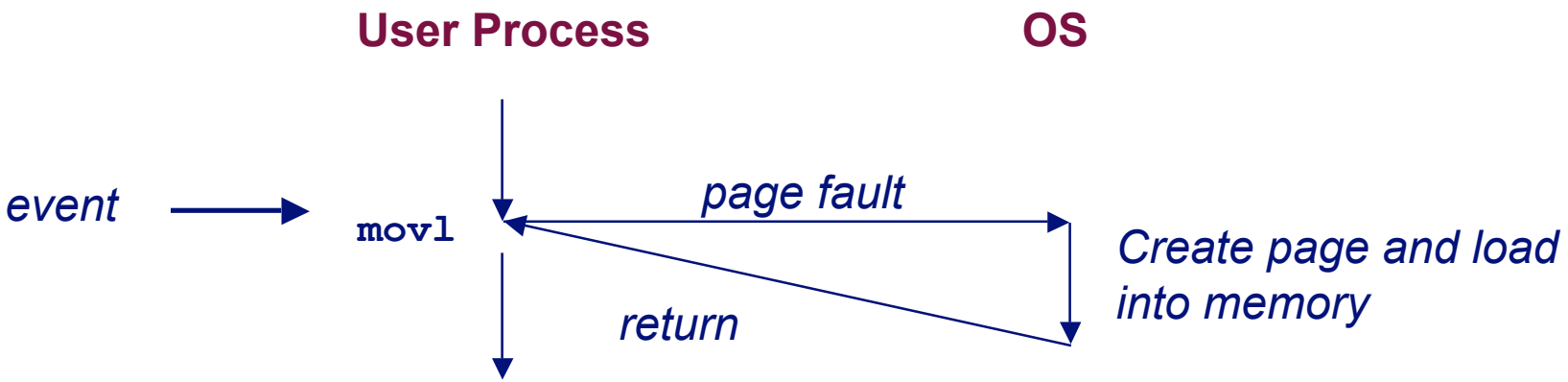
Memory Reference

- 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



Exercise

Write a short program that will access an invalid memory address, resulting in a fault. Just a *main* function that does nothing but cause a fault.

What will you see when you run this program on Linux?

Fault Example #2

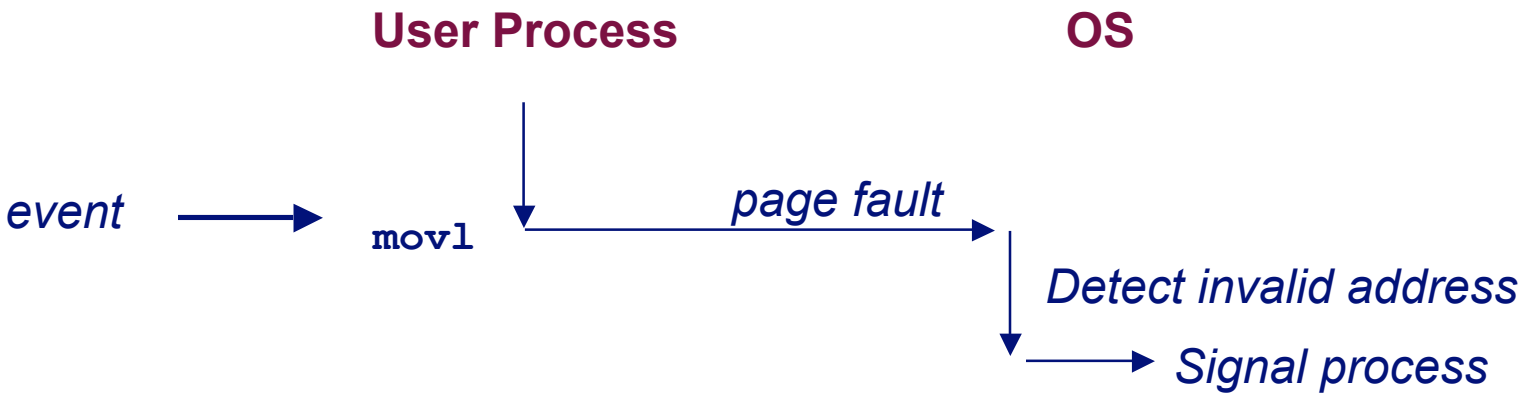
```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

Memory Reference

- User writes to memory location
- Address is not valid

```
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"



Summarizing

Exceptions

- Events that require nonstandard control flow

Generated externally

- interrupts
- asynchronous

Generated internally

- traps and faults
- synchronous