



# Exceptions and Processes

Much of the material for this lecture is drawn from  
*Computer Systems: A Programmer's Perspective* (Bryant & O'Hallaron) Chapter 8

# Goals of this Lecture



Help you learn about:

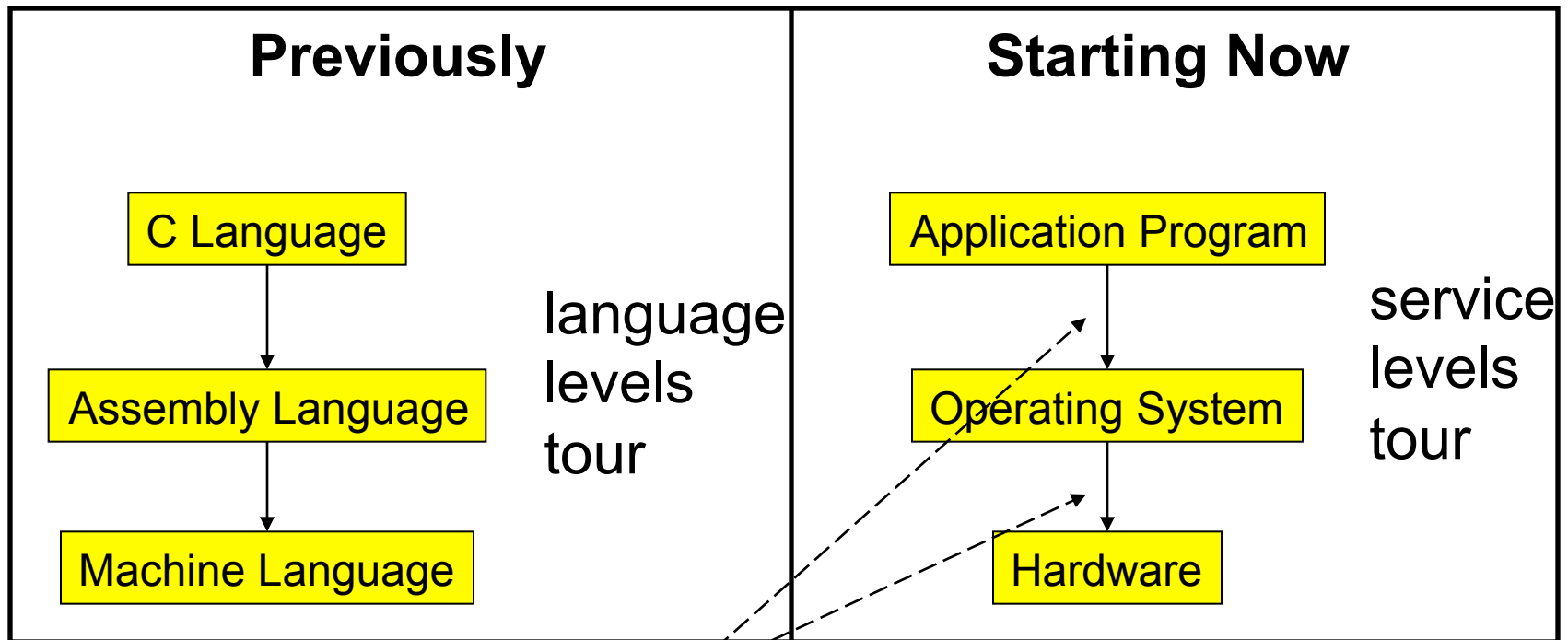
- **Exceptions**
- The **process** concept
- ... and thereby...
- How operating systems work
- How application programs interact with operating systems and hardware

The **process** concept is one of the most important concepts in system programming



# Context of this Lecture

Second half of the course



Application programs, OS, and hardware interact via **exceptions**



# Motivation

## Question:

- Executing program thinks it has exclusive use of all of memory
- But multiple executing programs must share one memory
- How is that illusion implemented?

## Question:

- Executing program thinks it has exclusive control of the CPU
- But multiple executing programs must share one CPU (or a few CPUs)
- How is that illusion implemented?

Answers: Exceptions...

# Agenda



**Exceptions**

Processes

Illusion: Private address space

Illusion: Private control flow

# Exceptions



## Exception

- An abrupt change in control flow in response to a change in processor state

# Synchronous Exceptions



## Some exceptions are **synchronous**

- Occur as result of actions of executing program
- Examples: Synchronous exception occurs when:
  - Application pgm requests I/O
  - Application pgm requests more heap memory
  - Application pgm attempts integer division by 0
  - Application pgm attempts to access privileged memory
  - Application pgm accesses variable that is not in physical memory
    - See later in this lecture
    - See upcoming *Virtual Memory* lecture

# Asynchronous Exceptions



## Some exceptions are **asynchronous**

- Do not occur (directly) as result of actions of executing program
- Examples: Asynchronous exception occurs when:
  - User presses key on keyboard
  - Disk controller finishes reading data
  - Hardware timer expires



# Exceptions Note



Note:

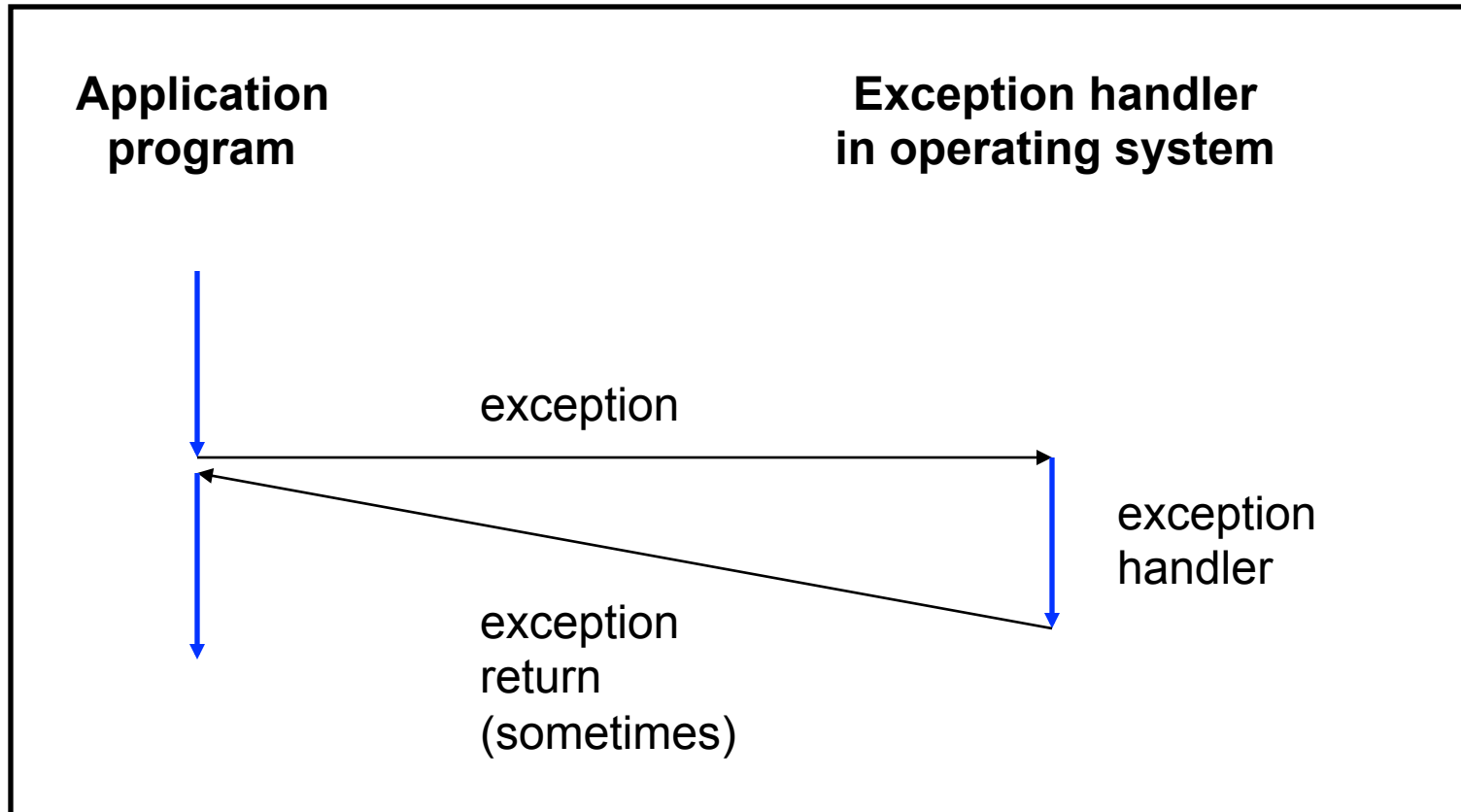
Exceptions in OS  $\neq$  exceptions in Java



Implemented using  
**try/catch** and  
**throw** statements



# Exceptional Control Flow





# Exceptions vs. Function Calls

Handling an exception is **similar to** calling a function

- CPU pushes arguments onto stack
- Control transfers from original code to other code
- Other code executes
- Control returns to some instruction in original code

Handling an exception is **different from** calling a function

- CPU pushes **additional data** onto stack
  - E.g. values of all registers
- CPU pushes data onto **OS' s stack**, not application pgm' s stack
- Handler runs in **kernel/privileged mode**, not in **user mode**
  - Handler can execute all instructions and access all memory
- Control **might return** to some instruction in original code
  - Sometimes control returns to **next** instruction
  - Sometimes control returns to **current** instruction
  - Sometimes control does not return at all!

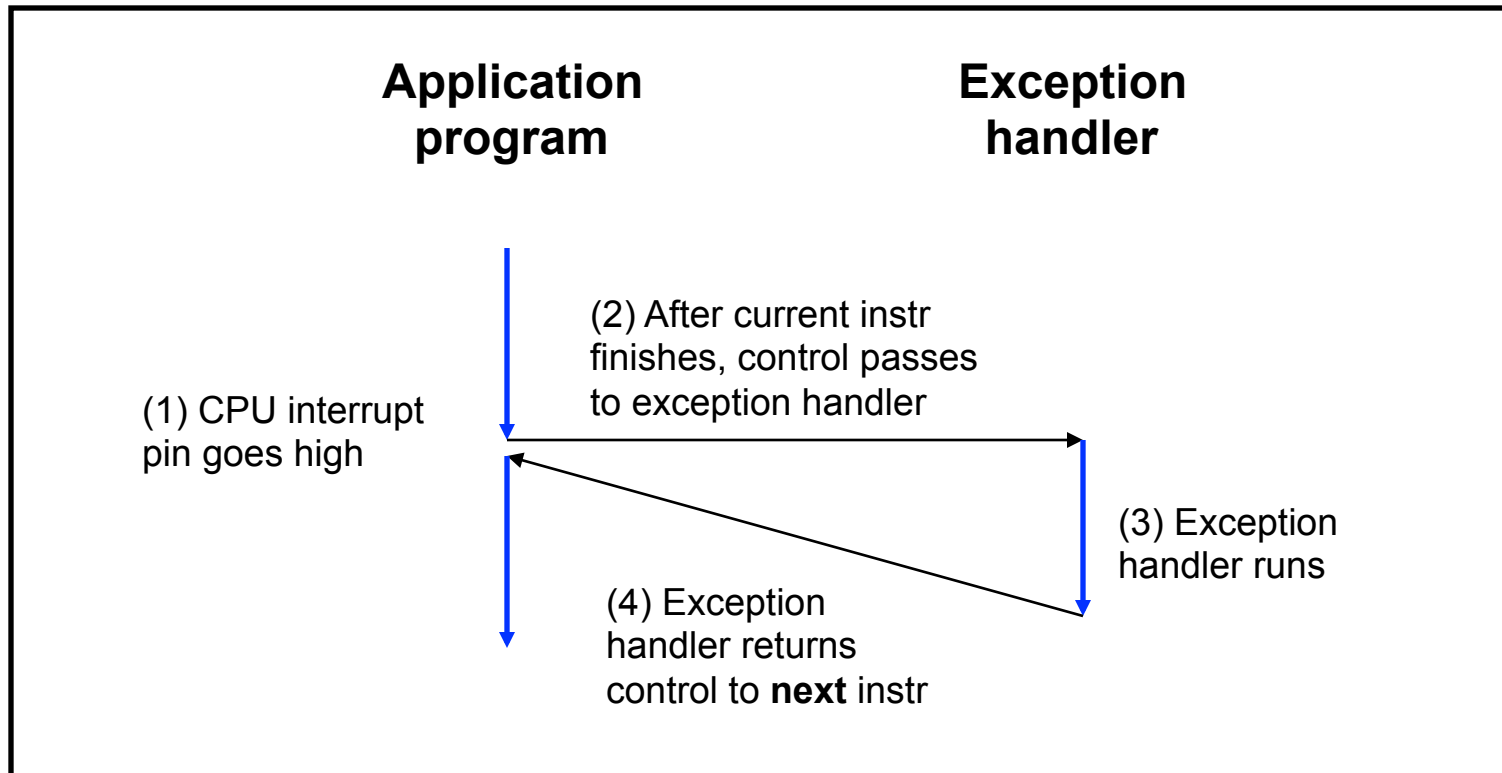
# Classes of Exceptions



There are 4 classes of exceptions...



# (1) Interrupts



**Occurs when:** External (off-CPU) device requests attention

**Examples:**

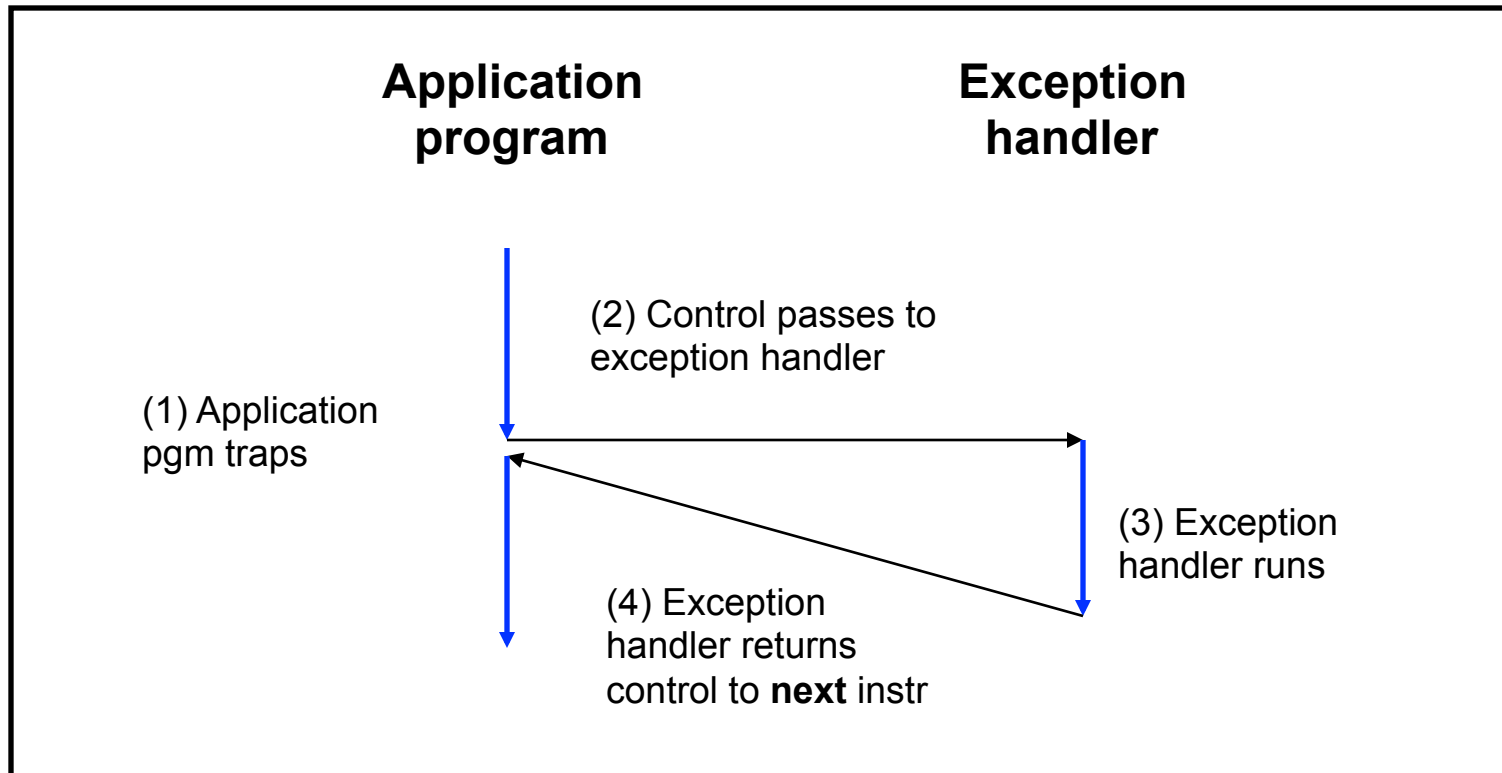
User presses key

Disk controller finishes reading/writing data

Hardware timer expires



## (2) Traps



**Occurs when:** Application pgm requests OS service

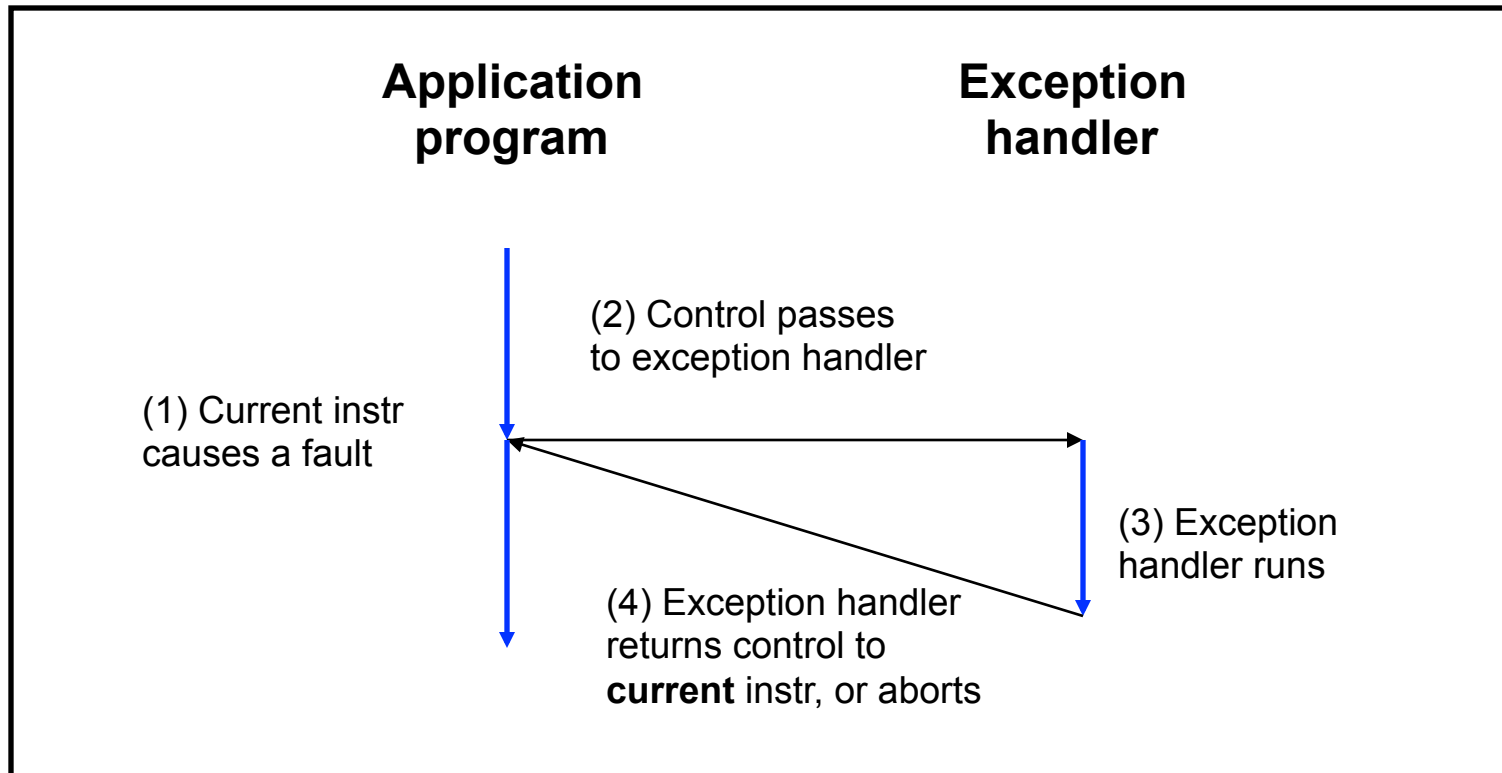
**Examples:**

Application pgm requests I/O

Application pgm requests more heap memory

Traps provide a function-call-like interface between application pgm and OS

# (3) Faults



**Occurs when:** Application pgm causes a (possibly recoverable) error

**Examples:**

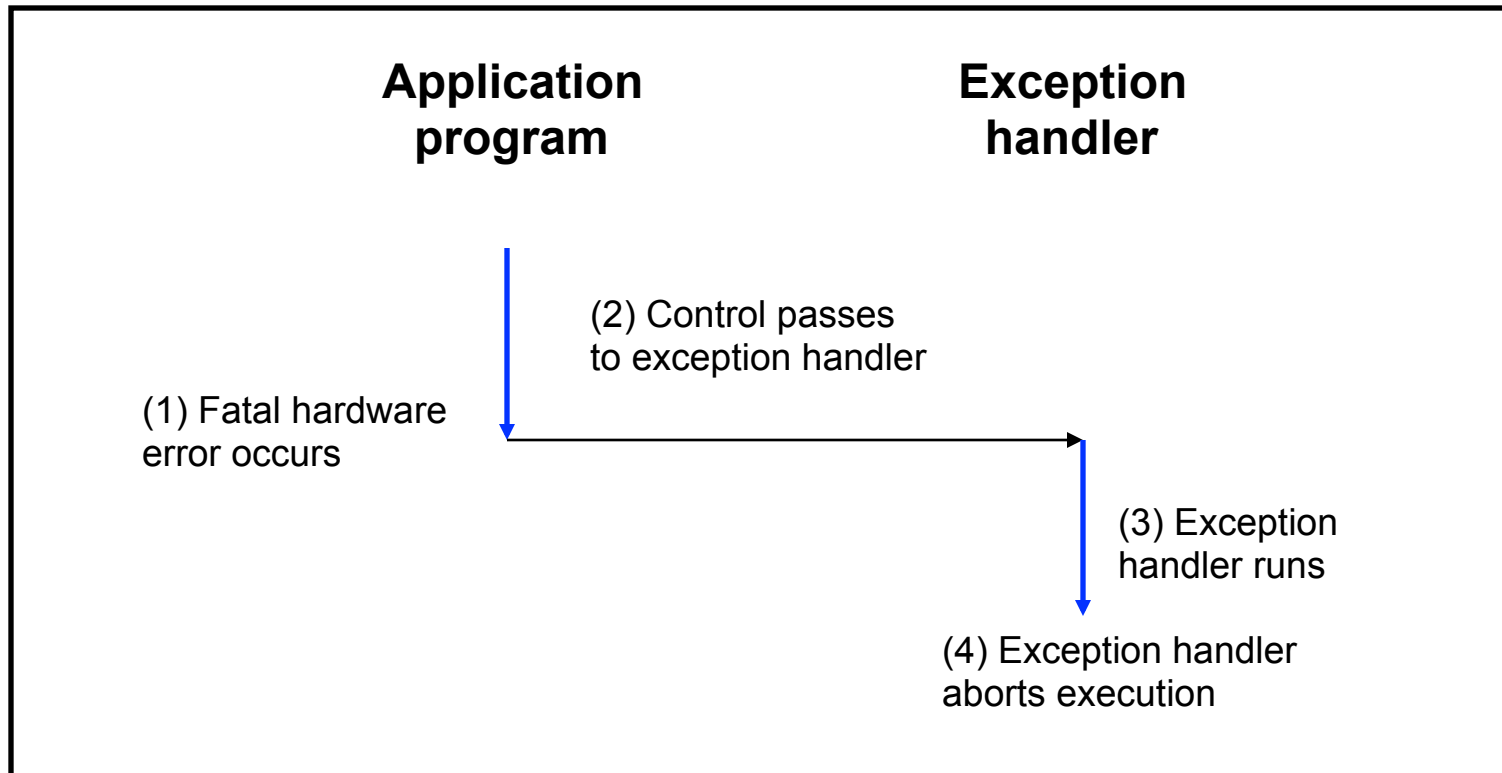
Application pgm divides by 0

Application pgm accesses privileged memory (seg fault)

Application pgm accesses data that is not in physical memory (page fault)



# (4) Aborts



**Occurs when:** HW detects a non-recoverable error

**Example:**

Parity check indicates corruption of memory bit (overheating, cosmic ray!, etc.)



# Summary of Exception Classes



Class	Occurs when	Asynch /Synch	Return Behavior
<b>Interrupt</b>	External device requests attention	Asynch	Return to next instr
<b>Trap</b>	Application pgm requests OS service	Sync	Return to next instr
<b>Fault</b>	Application pgm causes (maybe recoverable) error	Sync	Return to current instr (maybe)
<b>Abort</b>	HW detects non-recoverable error	Sync	Do not return

# Aside: Exceptions in IA-32 Processors



Each exception has a number  
Some exceptions in IA-32 processors:

Exception #	Exception
0	Fault: Divide error
13	Fault: Segmentation fault
14	Fault: Page fault (see <i>Virtual Memory</i> lecture)
18	Abort: Machine check
32-127	Interrupt or trap (OS-defined)
<b>128</b>	<b>Trap</b>
129-255	Interrupt or trap (OS-defined)

# Aside: Traps in IA-32 Processors



To execute a trap, application program should:

- Place number in EAX register indicating desired OS service
- Place arguments in EBX, ECX, EDX, ESI, EDI, EBP registers
- Execute assembly language instruction `int $128`

Example: To request change in size of heap section of memory (see *Dynamic Memory Management* lecture)...

```
movl $45, %eax  
movl $newAddr, %ebx  
int $128
```

Place 45 (change size of heap section) in EAX

Place new address of end of heap in EBX

Execute trap



# Aside: System-Level Functions

Traps are wrapped in **system-level functions**

Example: To change size of heap section of memory...

```
/* unistd.h */  
int brk(void *addr);  
...
```

**brk ()** is a  
system-level  
function

```
/* unistd.s */  
Defines brk() in assembly lang  
Executes int instruction  
...
```

```
/* client.c */  
...  
brk(newAddr);  
...
```

A call of a system-level function,  
that is, a **system call**

See Appendix for some Linux system-level functions

# Agenda



Exceptions

**Processes**

Illusion: Private address space

Illusion: Private control flow

# Processes



## Program

- Executable code
- A static entity

## Process

- An instance of a program in execution
- A dynamic entity: has a time dimension
- Each process runs one program
  - E.g. process 12345 might be running emacs
- One program can run in multiple processes
  - E.g. Process 12345 might be running emacs, and process 54321 might also be running emacs – for the same user or for different users

# Processes Significance



Process abstraction provides application pgms with two key illusions:

- Private address space
- Private control flow

**Process is a profound abstraction in computer science**

# Agenda



Exceptions

Processes

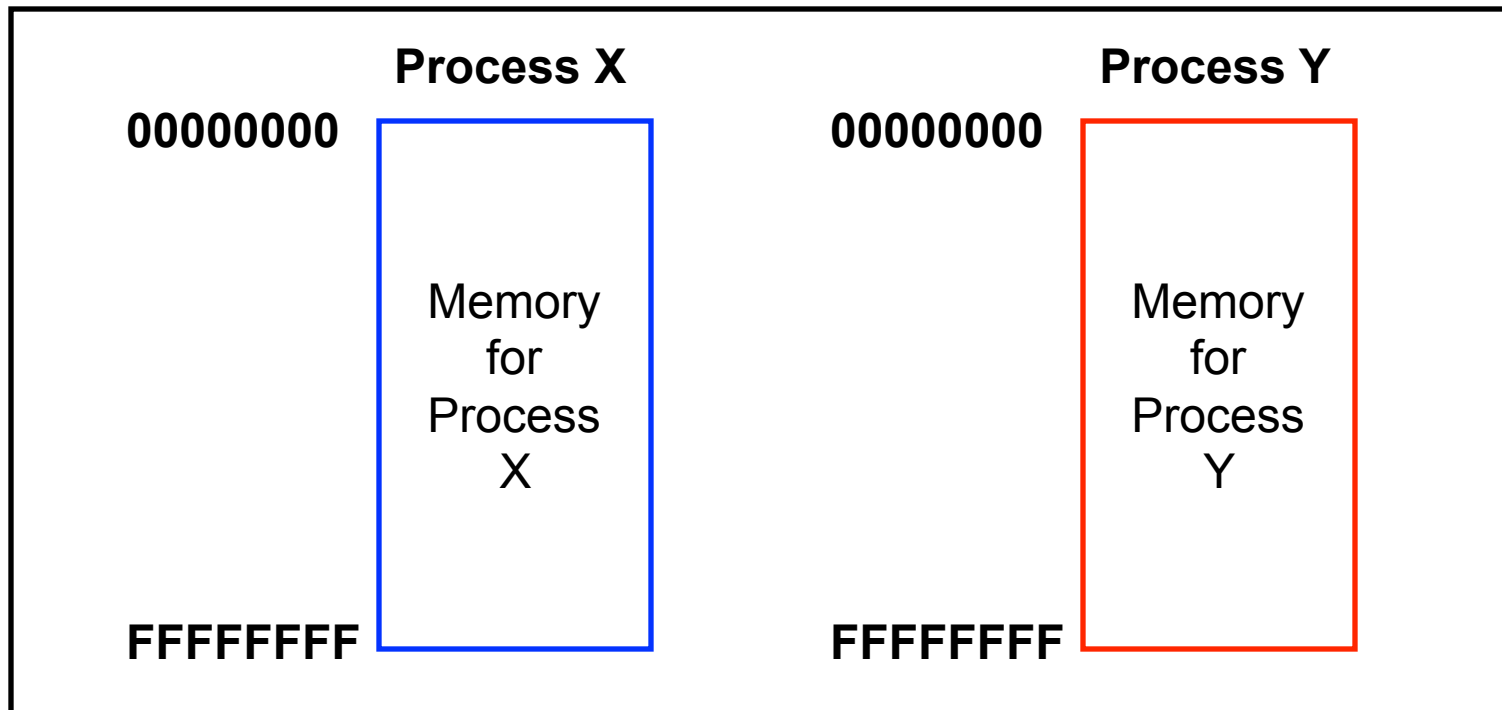
**Illusion: Private address space**

Illusion: Private control flow





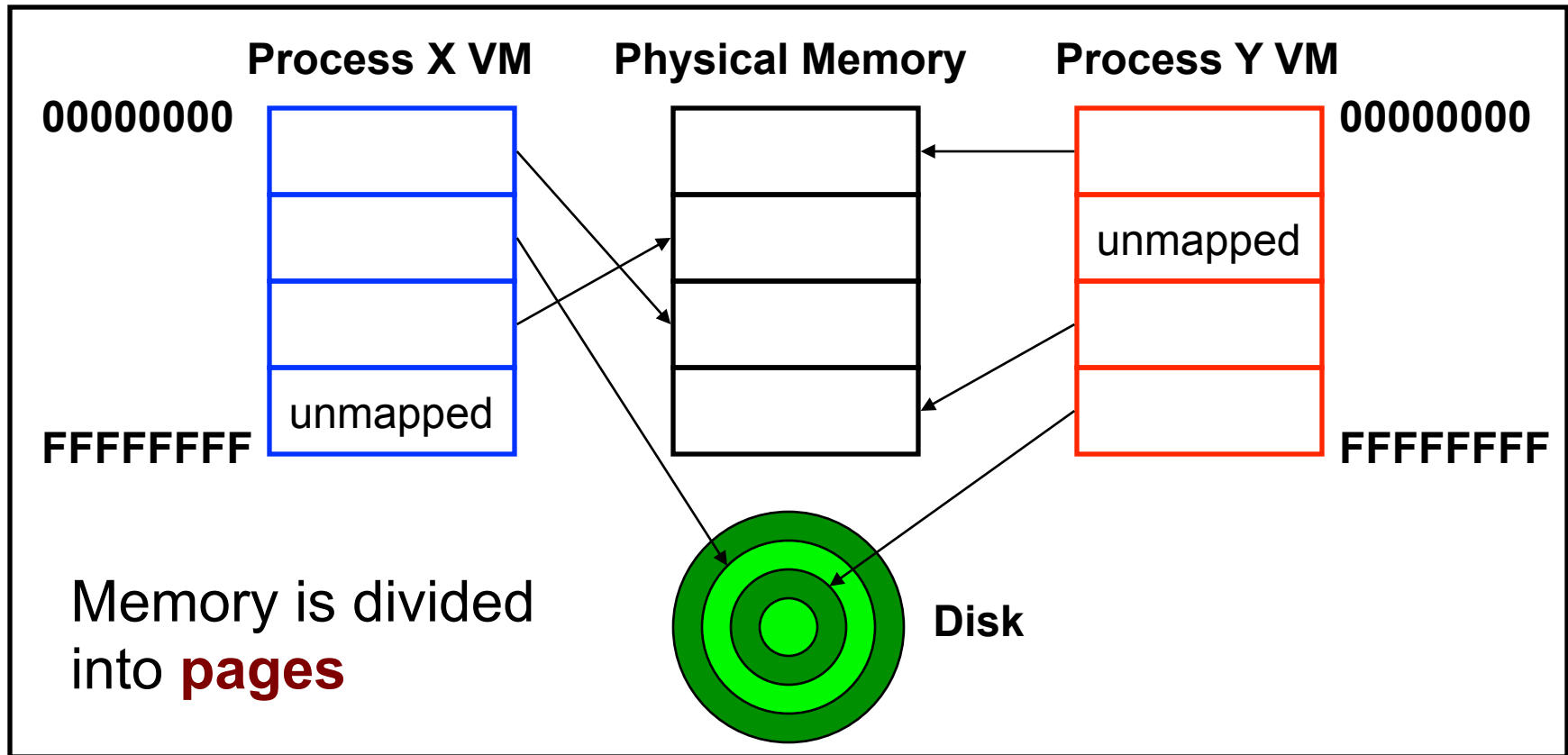
# Private Address Space: Illusion



Hardware and OS give each application process the illusion that it is the only process using memory



# Private Address Space: Reality



All processes use the same physical memory  
Hardware and OS provide application pgms with  
a **virtual** view of memory, i.e. **virtual memory (VM)**

# Private Address Space: Implementation



## Question:

- How do the CPU and OS implement the illusion of private address space?
- That is, how do the CPU and OS implement virtual memory?

## Answer:

- Exceptions!
- Specifically, **page faults**
- Overview now, details next lecture...

# Private Address Space Example 1



## Private Address Space Example 1

- Process executes instruction that references virtual memory
- CPU determines virtual page
- CPU checks if required virtual page is in physical memory: yes
- CPU does load/store from/to physical memory



# Private Address Space Example 2

## Private Address Space Example 2

- Process executes instruction that references virtual memory
- CPU determines virtual page
- CPU checks if required virtual page is in physical memory: no!
  - CPU generates **page fault**
  - OS gains control of CPU
  - OS evicts some page from physical memory to disk, loads required page from disk to physical memory
  - OS returns control of CPU to process - to same instruction
- Process executes instruction that references virtual memory
- CPU checks if required virtual page is in physical memory: yes
- CPU does load/store from/to physical memory

Exceptions (specifically, **page faults**) enable the illusion of private address spaces

# Agenda



Exceptions

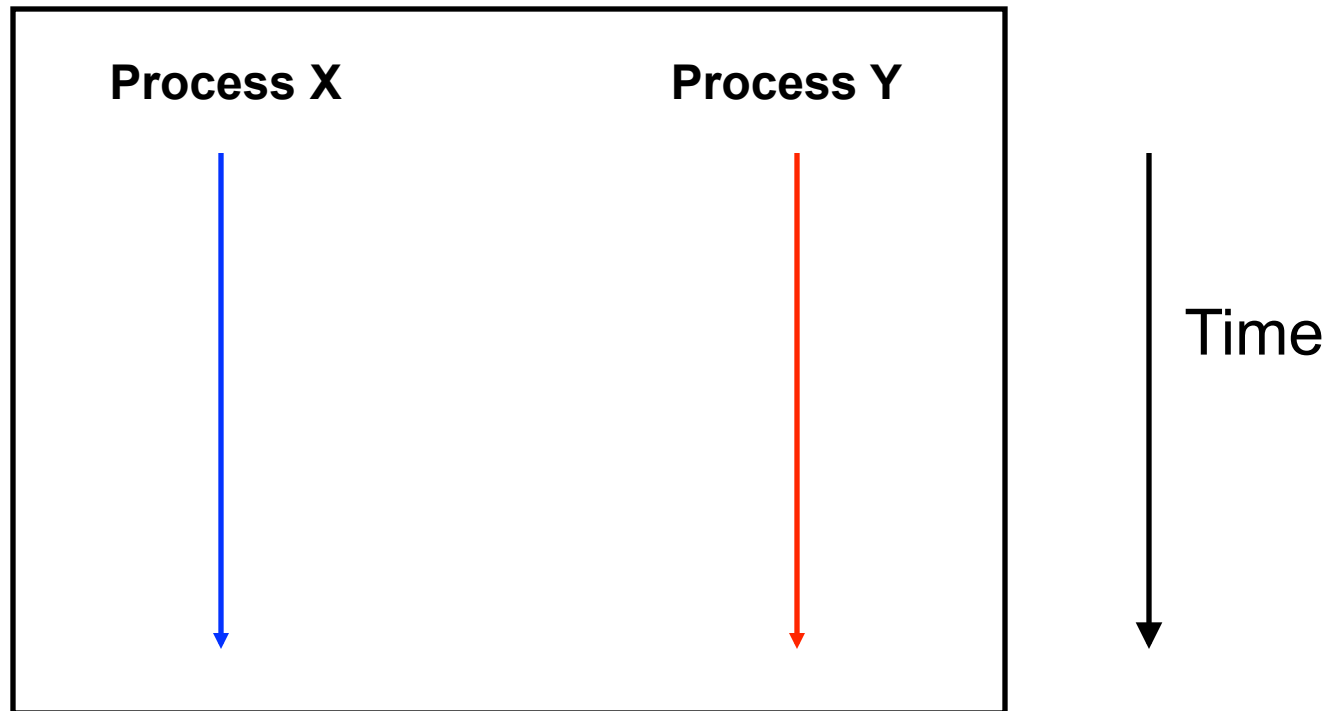
Processes

Illusion: Private address space

**Illusion: Private control flow**



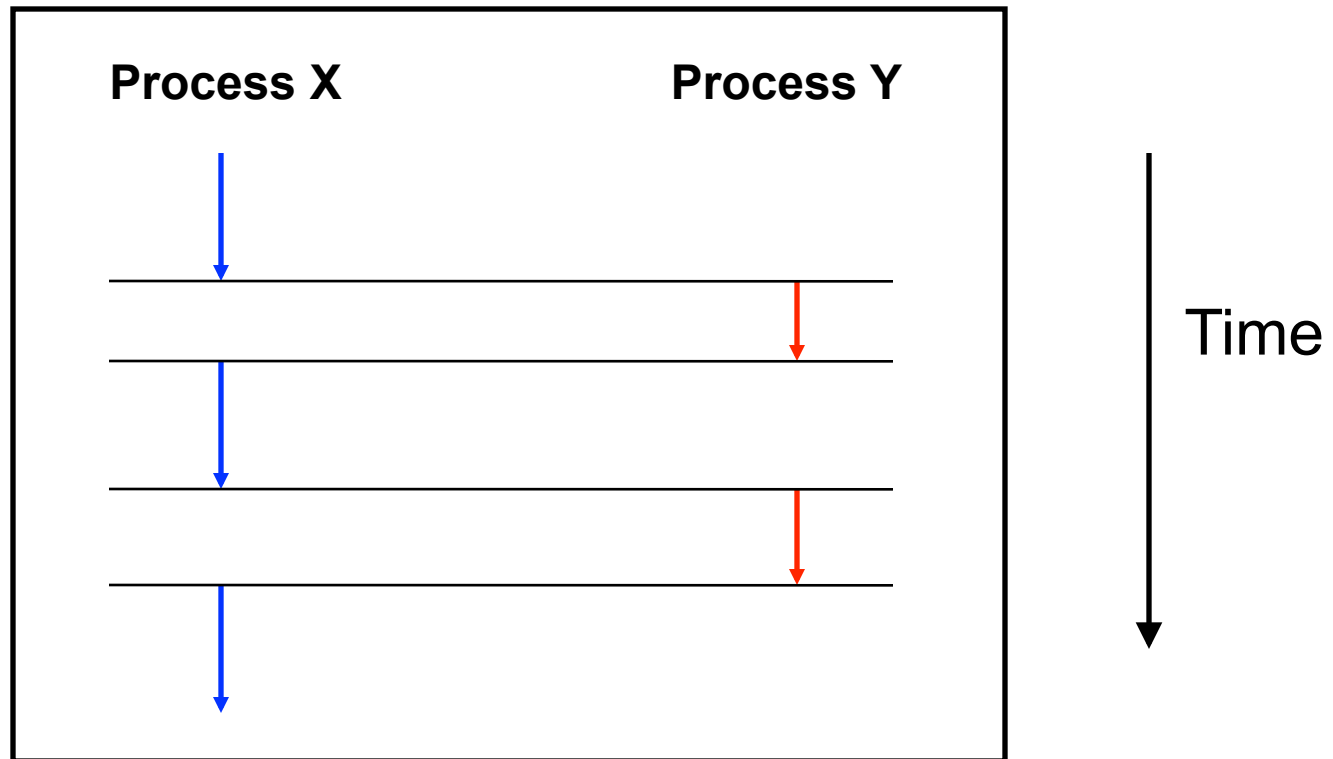
# Private Control Flow: Illusion



Simplifying assumption: only one CPU

Hardware and OS give each application process the illusion that it is the only process running on the CPU

# Private Control Flow: Reality



Multiple processes share the CPU  
Multiple processes run **concurrently**  
OS occasionally **preempts** running process



# Process Status



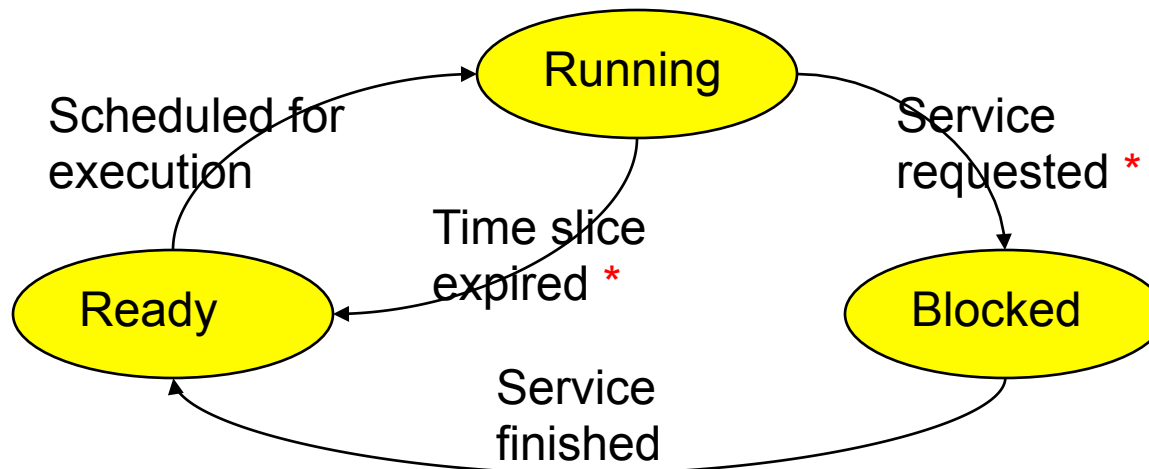
More specifically...

At any time a process has **status**:

- **Running**: CPU is executing process' s instructions
- **Ready**: Process is ready for OS to assign it to the CPU
- **Blocked**: Process is waiting for some requested service (typically I/O) to finish



# Process Status Transitions



\* Preempting transition

**Service requested:** OS moves running process to blocked set because it requested a (time consuming) system service (often I/O)

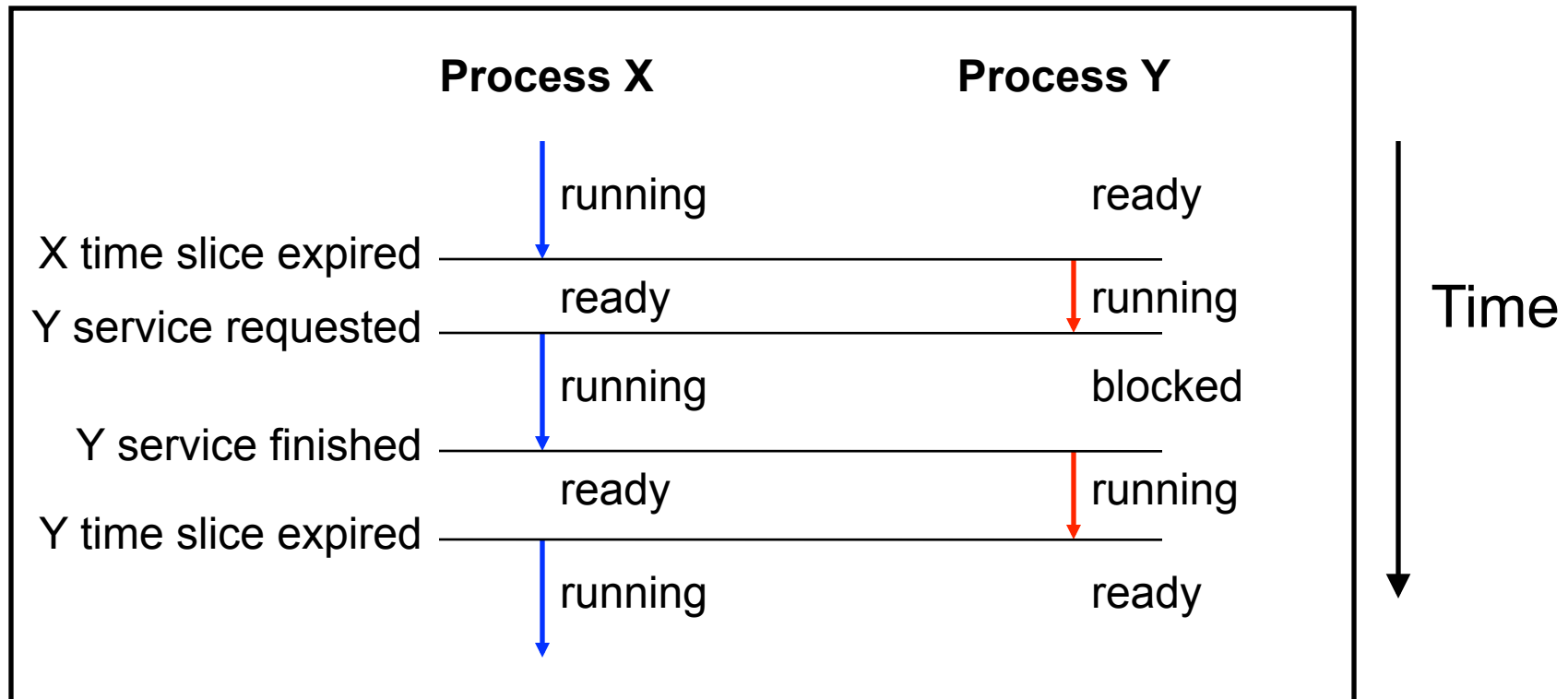
**Service finished:** OS moves blocked process to ready set because the requested service finished

**Time slice expired:** OS moves running process to ready set because process consumed its fair share of CPU time

**Scheduled for execution:** OS selects some process from ready set and assigns CPU to it



# Process Status Transitions Over Time



Throughout its lifetime a process' s status switches between running, ready, and blocked



# Private Control Flow: Implementation (1)

## Question:

- How do CPU and OS implement the illusion of private control flow?
- That is, how do CPU and OS implement process status transitions?

## Answer (Part 1):

- Contexts and context switches...

# Process Contexts



## Each process has a **context**

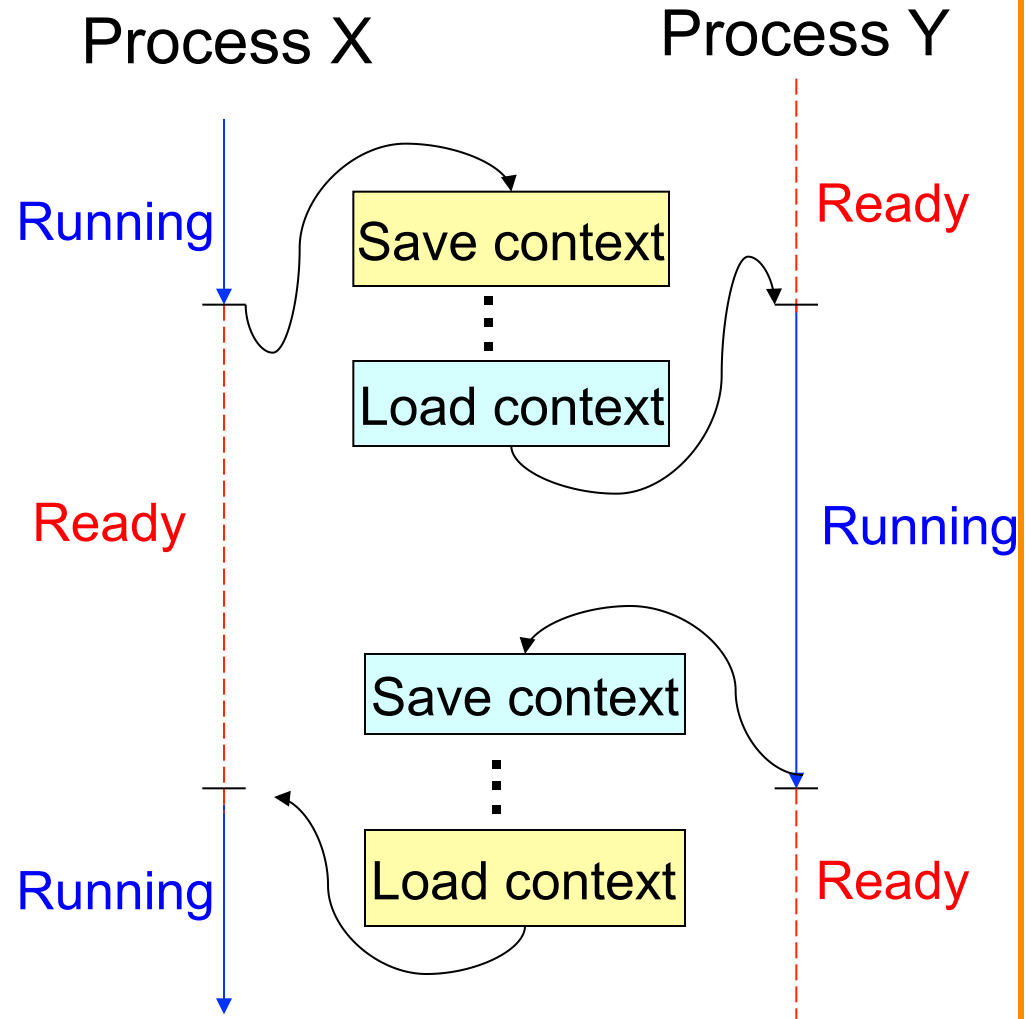
- The process's state, that is...
- Register contents
  - EIP, EFLAGS, EAX, EBX, etc. registers
- Memory contents
  - TEXT, RODATA, DATA, BSS, HEAP, and STACK

# Context Switch



## Context switch:

- OS saves context of running process
- OS loads context of some ready process
- OS passes control to newly restored process



# Aside: Process Control Blocks



## Question:

- Where does OS save a process' s context?

## Answer:

- In its **process control block (PCB)**

## Process control block (PCB)

- A data structure
- Contains all data that OS needs to manage the process

# Aside: Process Control Block Details



## Process control block (PCB):

Field	Description
ID	Unique integer assigned by OS when process is created
Status	Running, ready, or waiting
Hierarchy	ID of parent process ID of child processes (if any) (See <b>Process Management</b> Lecture)
Priority	High, medium, low
Time consumed	Time consumed within current time slice
<b>Context</b>	<b>When process is not running...</b> <b>Contents of all registers</b> <b>(In principle) contents of all of memory</b>
Etc.	



# Context Switch Efficiency



## Observation:

- During context switch, OS must:
  - Save context (register and memory contents) of running process to its PCB
  - Restore context (register and memory contents) of some ready process from its PCB

## Question:

- Isn't that **very** expensive (in terms of time and space)?

# Context Switch Efficiency



## Answer:

- Not really!
- During context switch, OS **does** save/load **register** contents
  - But there are few registers
- During context switch, OS **does not** save/load **memory** contents
  - Each process has a **page table** that maps virtual memory pages to physical memory pages
  - During context switch, need only deactivate process X page table and activate process Y page table
  - See *Virtual Memory* lecture



# Private Control Flow: Implementation (2)

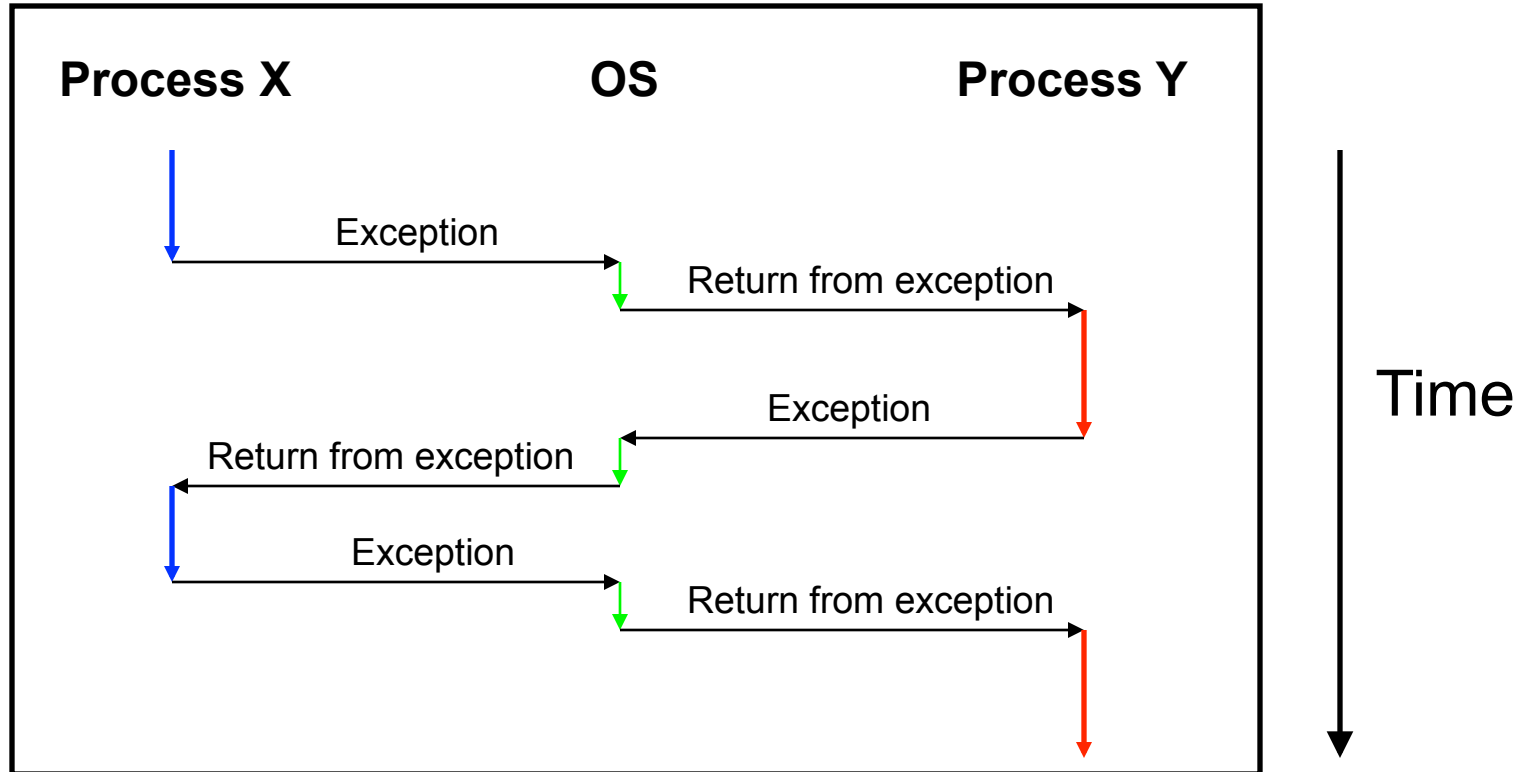
## Question:

- How do CPU and OS implement the illusion of private control flow?
- That is, how do CPU and OS implement process status transitions?
- That is, how do CPU and OS implement context switches?

## Answer (Part 2):

- Exceptions!
- Context switches occur while the OS handles exceptions...

# Exceptions and Context Switches



Context switches occur  
while OS is handling exceptions

# Exceptions and Context Switches



## Exceptions occur frequently

- Process explicitly requests OS service (trap)
- Service request fulfilled (interrupt)
- Process accesses VM page that is not in physical memory (fault)
- Etc.
- ... And if none of them occur for a while ...
- Expiration of hardware timer (interrupt)

Whenever OS gains control of CPU via exception...

It has the option of performing context switch

# Private Control Flow Example 1



## Private Control Flow Example 1

- Process X is running
- Hardware clock generates **interrupt**
- OS gains control of CPU
- OS examines “time consumed” field of process X’s PCB
- OS decides to do context switch
  - OS saves process X’s context in its PCB
  - OS sets “status” field in process X’s PCB to *ready*
  - OS adds process X’s PCB to the ready set
  - OS removes process Y’s PCB from the ready set
  - OS sets “status” field in process Y’s PCB to running
  - OS loads process Y’s context from its PCB
- Process Y is running

# Private Control Flow Example 2



## Private Control Flow Example 2

- Process Y is running
- Process Y executes **trap** to request read from disk
- OS gains control of CPU
- OS decides to do context switch
  - OS saves process Y's context in its PCB
  - OS sets "status" field in process Y's PCB to blocked
  - OS adds process Y's PCB to the blocked set
  - OS removes process X's PCB from the ready set
  - OS sets "status" field in process X's PCB to running
  - OS loads process X's context from its PCB
- Process X is running

# Private Control Flow Example 3



## Private Control Flow Example 3

- Process X is running
- Read operation requested by process Y completes => disk controller generates **interrupt**
- OS gains control of CPU
- OS sets “status” field in process Y’s PCB to ready
- OS moves process Y’s PCB from the blocked list to the ready list
- OS examines “time consumed within slice” field of process X’s PCB
- OS decides not to do context switch
- Process X is running



# Private Control Flow Example 4



## Private Control Flow Example 4

- Process X is running
- Process X accesses memory, generates **page fault**
- OS gains control of CPU
- OS evicts page from memory to disk, loads referenced page from disk to memory
- OS examines “time consumed” field of process X’s PCB
- OS decides not to do context switch
- Process X is running

Exceptions enable the illusion of private control flow

# Summary



## Exception: an abrupt change in control flow

- **Interrupt:** asynchronous; e.g. I/O completion, hardware timer
- **Trap:** synchronous; e.g. app pgm requests more heap memory, I/O
- **Fault:** synchronous; e.g. seg fault, page fault
- **Abort:** synchronous; e.g. failed parity check

## Process: An instance of a program in execution

- CPU and OS give each process the illusion of:
  - Private address space
    - Reality: **virtual memory**
  - Private control flow
    - Reality: **Concurrency, preemption, and context switches**
- Both illusions are implemented using exceptions

# Appendix: System-Level Functions



## Linux system-level functions for **I/O management**

Number	Function	Description
3	read()	Read data from file descriptor; called by getchar(), scanf(), etc.
4	write()	Write data to file descriptor; called by putchar(), printf(), etc.
5	open()	Open file or device; called by fopen()
6	close()	Close file descriptor; called by fclose()
8	creat()	Open file or device for writing; called by fopen(..., "w")
19	lseek()	Position file offset; called by fseek()

Described in *I/O Management* lecture

# Appendix: System-Level Functions



## Linux system-level functions for **process management**

Number	Function	Description
1	exit()	Terminate the current process
2	fork()	Create a child process
7	wait()	Wait for child process termination
11	execvp()	Execute a program in the current process
20	getpid()	Return the process id of the current process

Described in *Process Management* lecture

# Appendix: System-Level Functions



## Linux system-level functions for I/O redirection and inter-process communication

Number	Function	Description
41	dup()	Duplicate an open file descriptor
42	pipe()	Create a channel of communication between processes

Described in *Process Management* lecture

# Appendix: System-Level Functions



## Linux system-level functions for **dynamic memory management**

Number	Function	Description
45	brk()	Move the program break, thus changing the amount of memory allocated to the HEAP
45	sbrk()	(Variant of previous)
90	mmap()	Map a virtual memory page
91	munmap()	Unmap a virtual memory page

Described in *Dynamic Memory Management* lecture

# Appendix: System-Level Functions



## Linux system-level functions for **signal handling**

Number	Function	Description
27	alarm()	Deliver a signal to a process after a specified amount of wall-clock time
37	kill()	Send signal to a process
67	sigaction()	Install a signal handler
104	setitimer()	Deliver a signal to a process after a specified amount of CPU time
126	sigprocmask()	Block/unblock signals

Described in *Signals* lecture