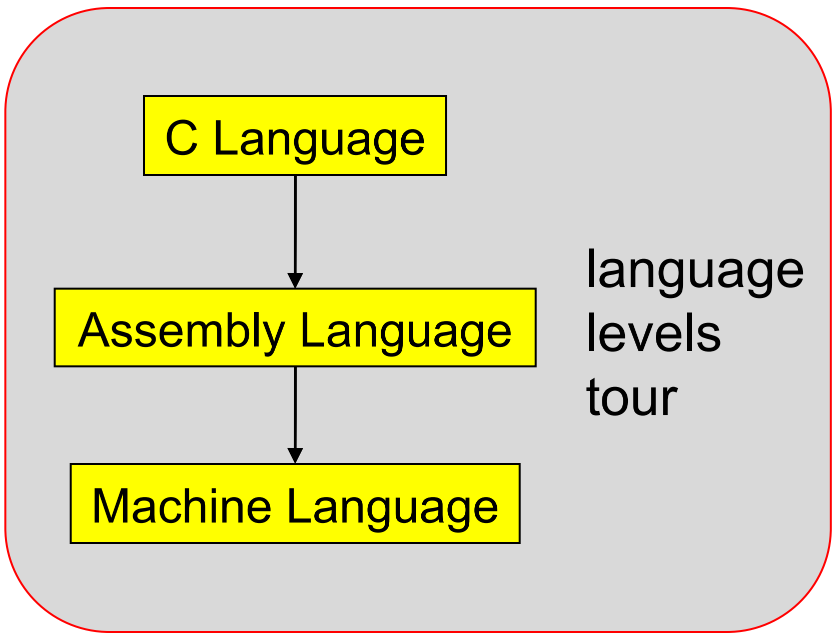
Exceptions and Processes

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

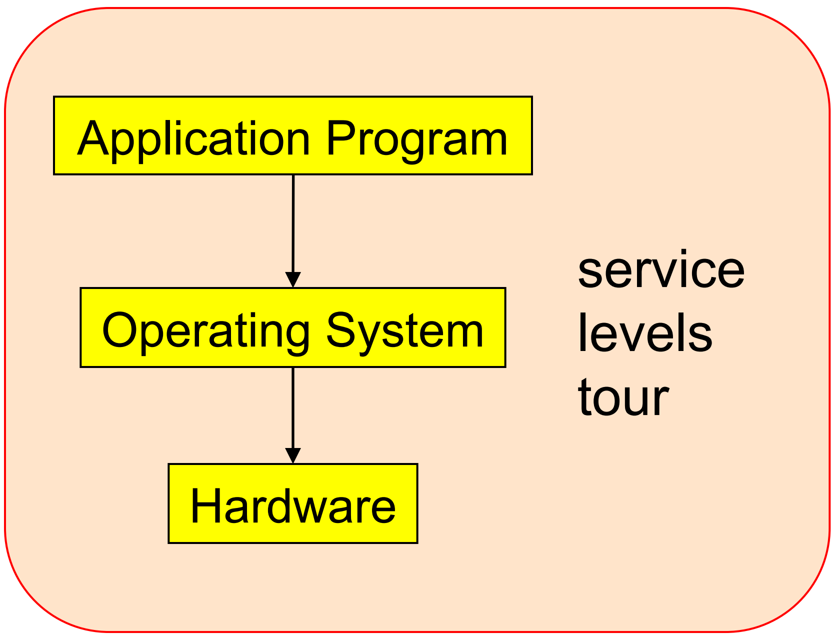
Context of this Lecture

"Under the hood"

Previously



Now



Goals of this Lecture

Help you learn about:

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

Agenda

Processes

Illusion: Private address space

Illusion: Private control flow

Exceptions

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. the process with Process ID 12345 might be running emacs
- One program can run in multiple processes
  - E.g. PID 12345 might be running emacs, and
    PID 23456 might also be running emacs –
    for the same user or for different users

Processes Significance

Process abstraction provides two key illusions:

- Processes believe they have a *private address space*
- Processes believe they have *private control flow*

Process is a profound abstraction in computer science

Agenda

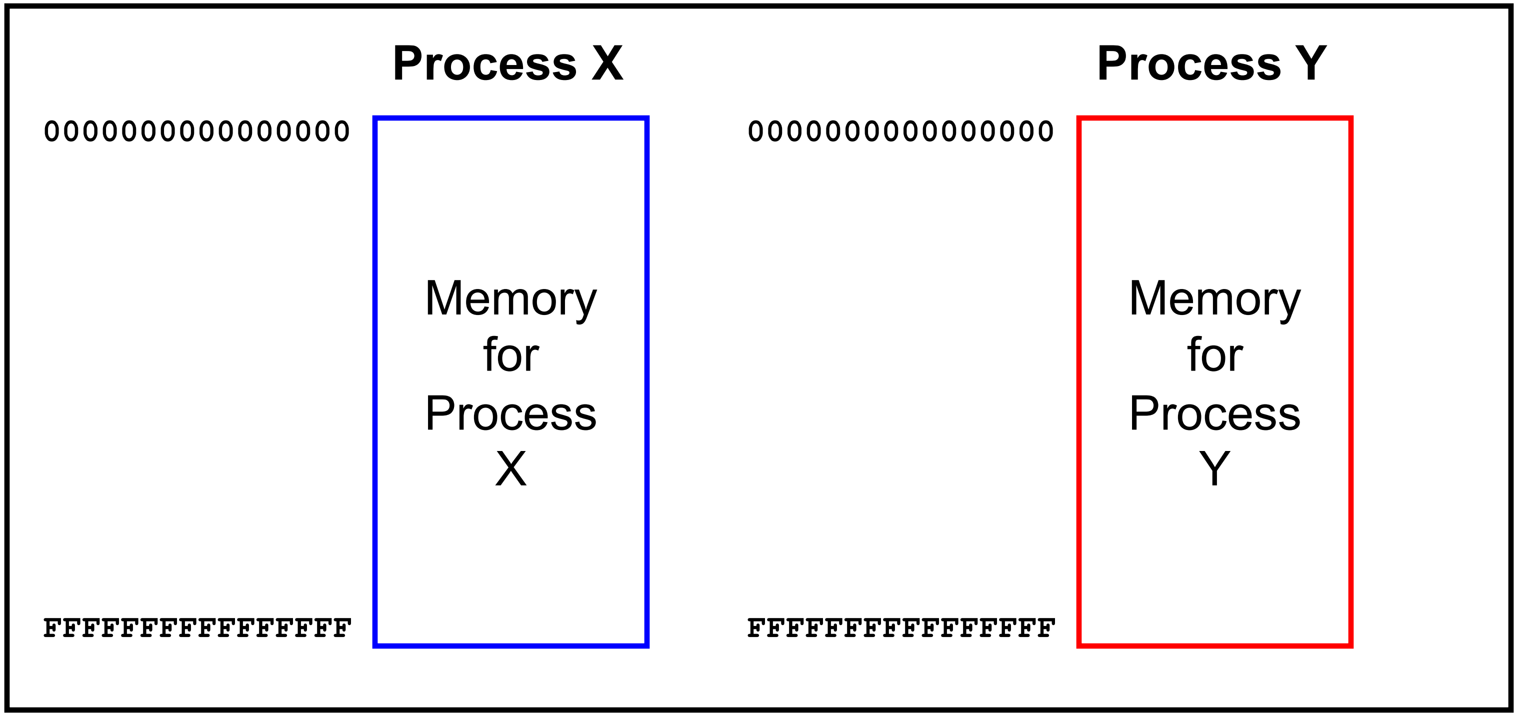
Processes

Illusion: Private address space

Illusion: Private control flow

Exceptions

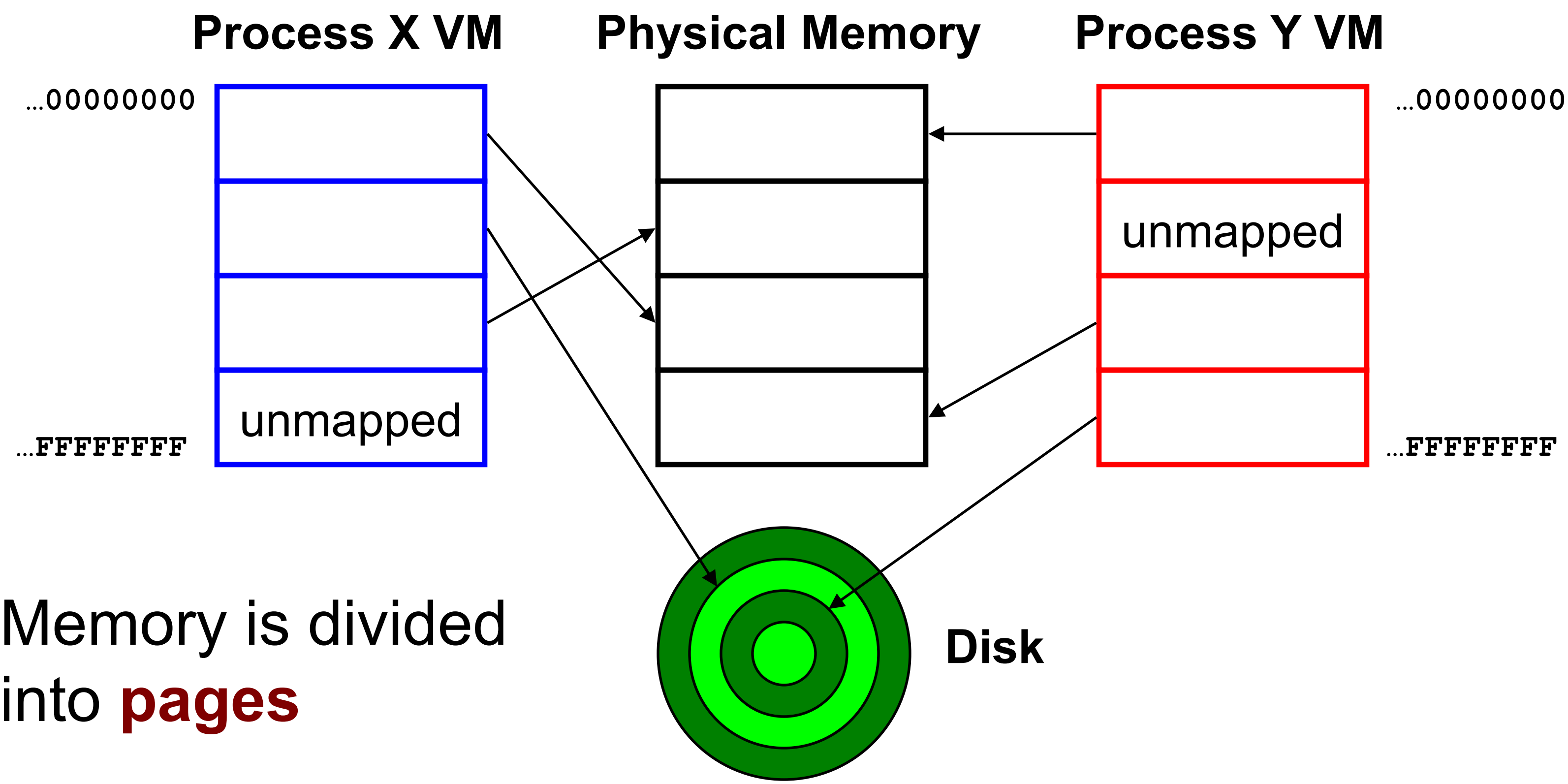
Private Address Space: Illusion



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

- Enables multiple simultaneous instances of one program!

Private Address Space: Reality



Memory is divided into **pages**

All processes use the same physical memory.
Hardware and OS provide programs 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:

- Page tables: “directory” mapping virtual to physical addresses
- **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`

iClicker Question coming up . . .

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 (potentially) 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

Virtual memory enables the illusion of private address spaces

iClicker Question

Q: What effect does virtual memory have on the speed and security of processes?

	Speed	Security
A.	↑	↑
B.	↓	↑
C.	↑	no change
D.	↑	↓
E.	↓	↓

Agenda

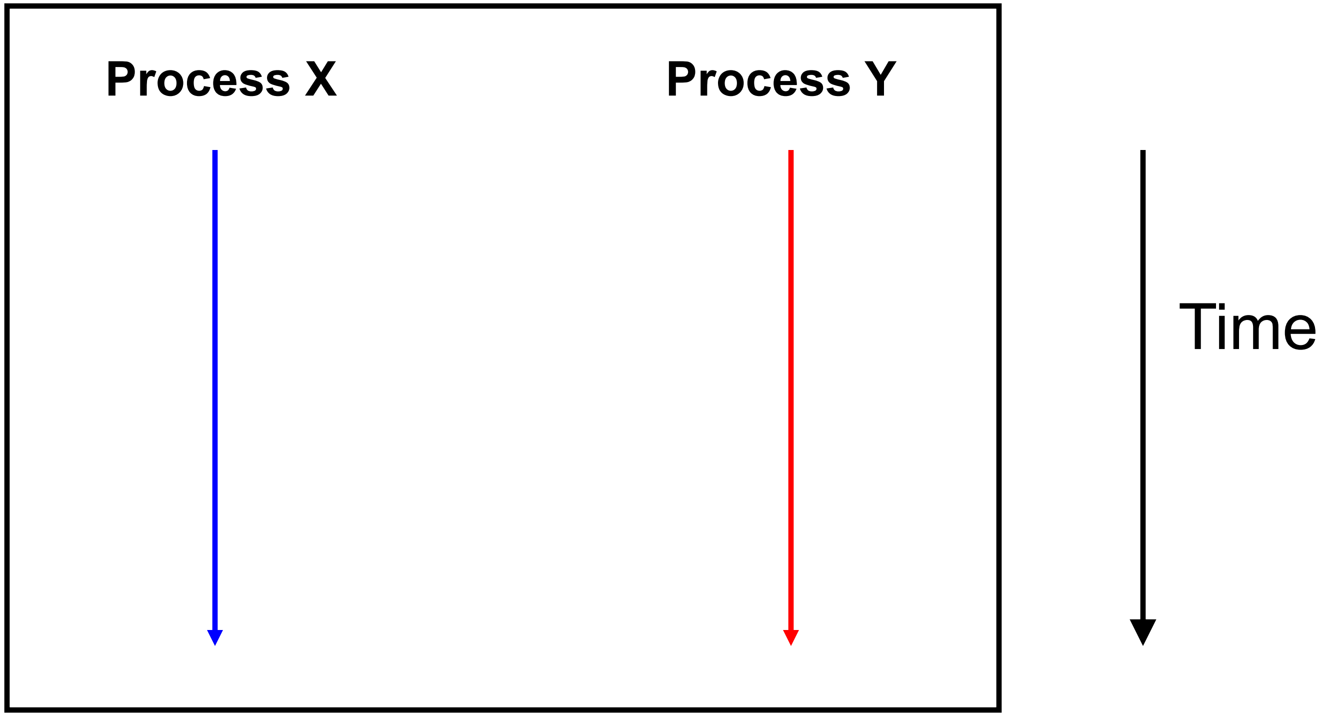
Processes

Illusion: Private address space

Illusion: Private control flow

Exceptions

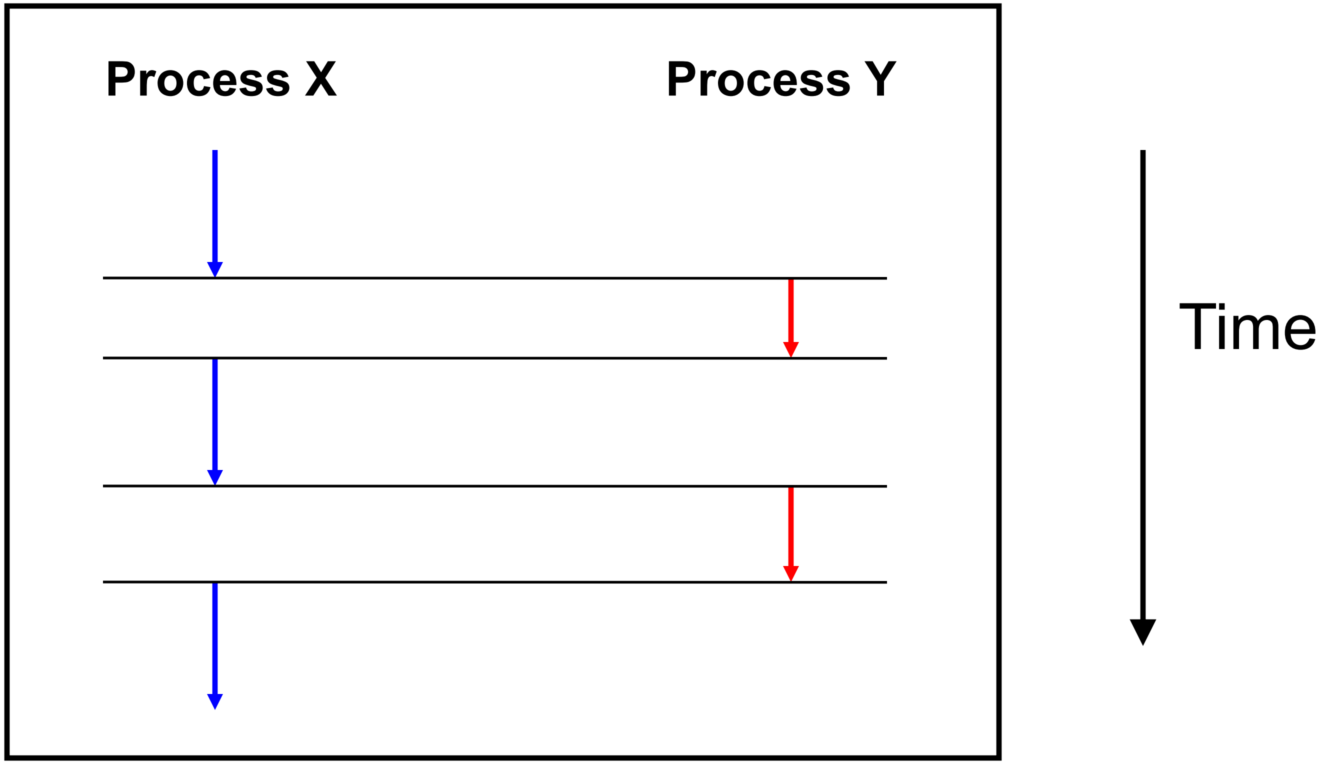
Private Control Flow: Illusion



Simplifying assumption: only one CPU / core

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 are *time-sliced* to run **concurrently**

OS occasionally **preempts** running process to give other processes their fair share of CPU time

Process Status

More specifically…

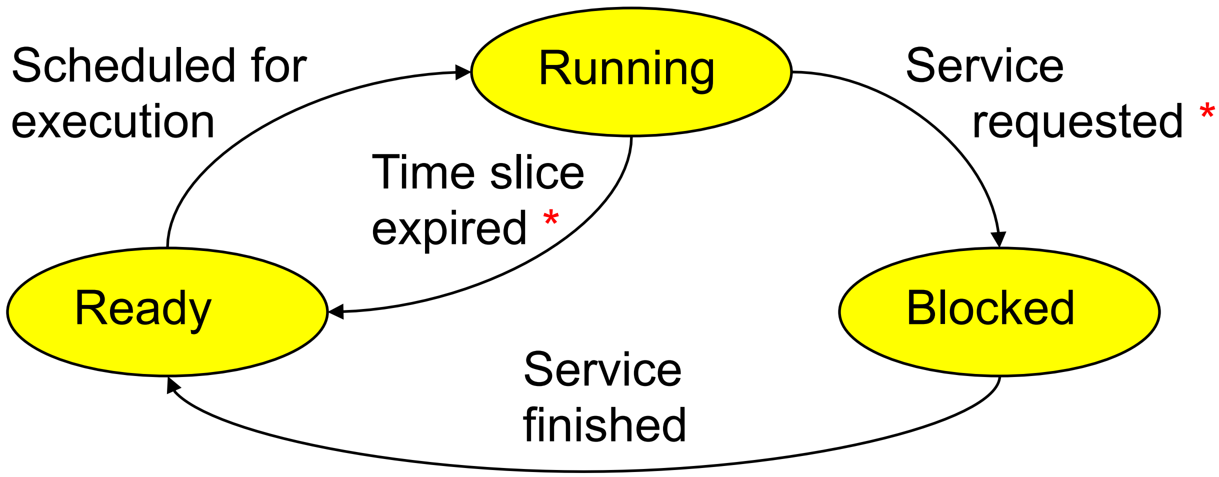
At any time a process has **status**:

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

Modern machines may have multiple CPUs or “cores”, but the same principles apply if #processes > #cores

- For simplicity, we will speak of “the” CPU

Process Status Transitions



* Preempting transition

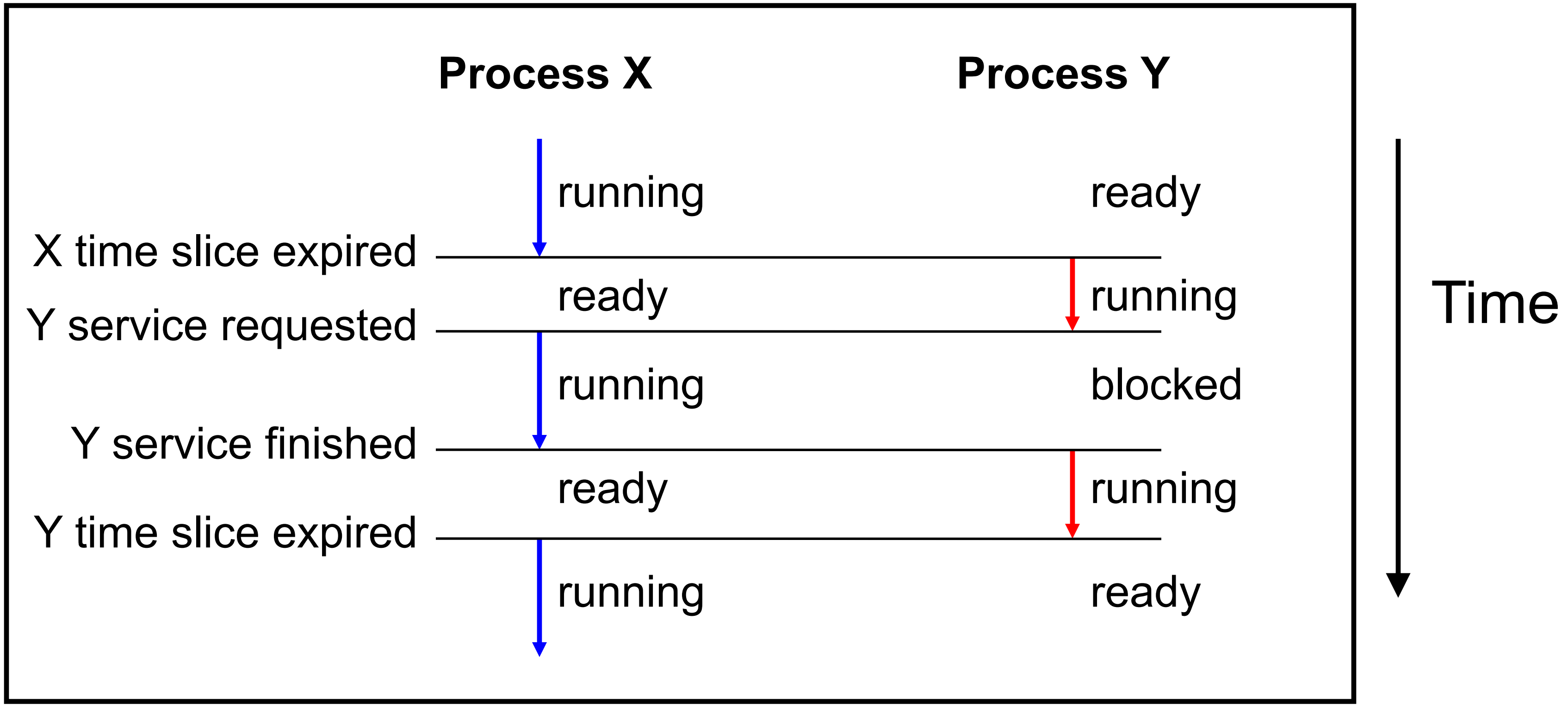
Scheduled for execution: OS selects some process from ready set and assigns CPU to it

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

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

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

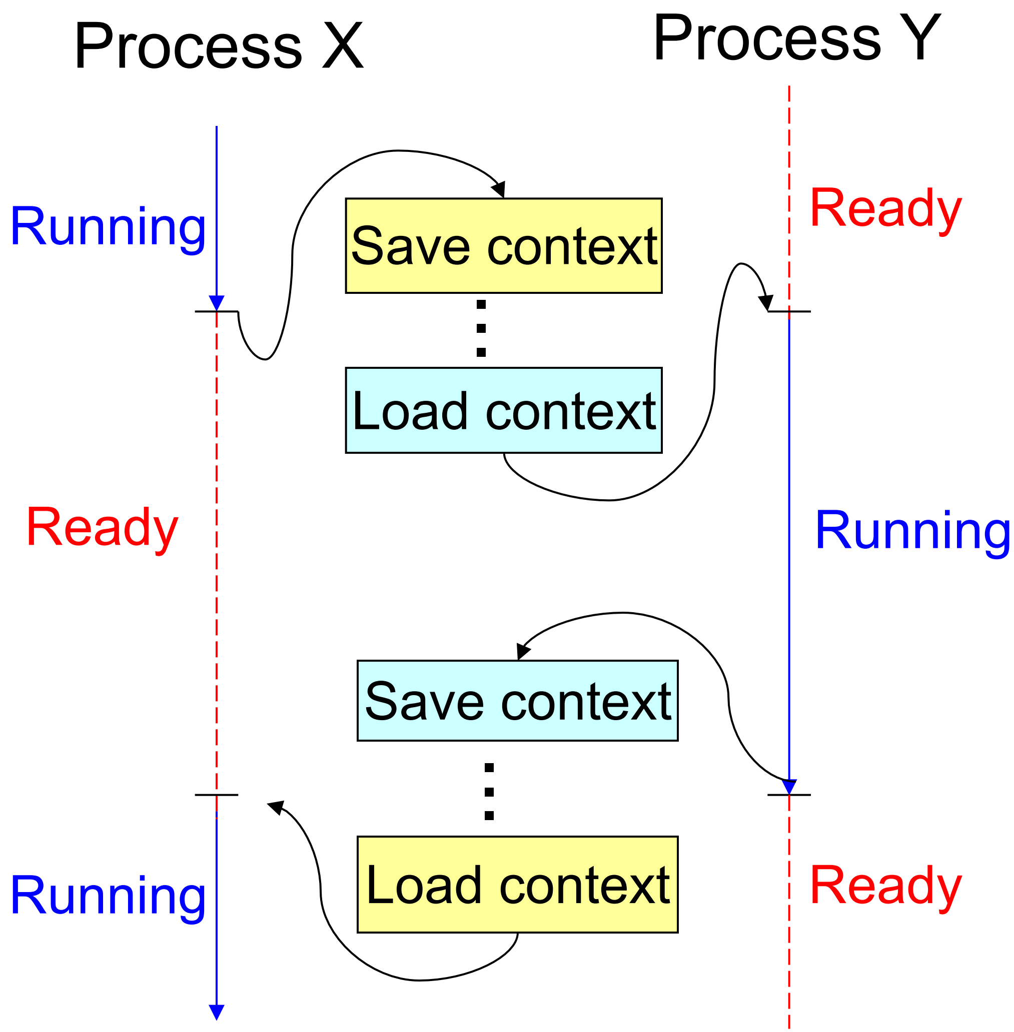
Each process has a **context**

- The process's state, that is…
- Register contents
  - X0..X30, SP, PSTATE, 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 ***Process Management*** Lecture)
Priority	High, medium, low
Time consumed	Time consumed within current time slice
Context	**When process is not running…** **Contents of all registers** **(In principle) contents of all of memory**
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, OS tells hardware to start using a different process's page tables
  - 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):

- Context switches occur while the OS handles **exceptions**…

Agenda

Processes

Illusion: Private address space

Illusion: Private control flow

Exceptions

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:
  - **System call:** Application requests I/O
  - **System call:** Application 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

Asynchronous Exceptions

Some exceptions are **asynchronous**

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

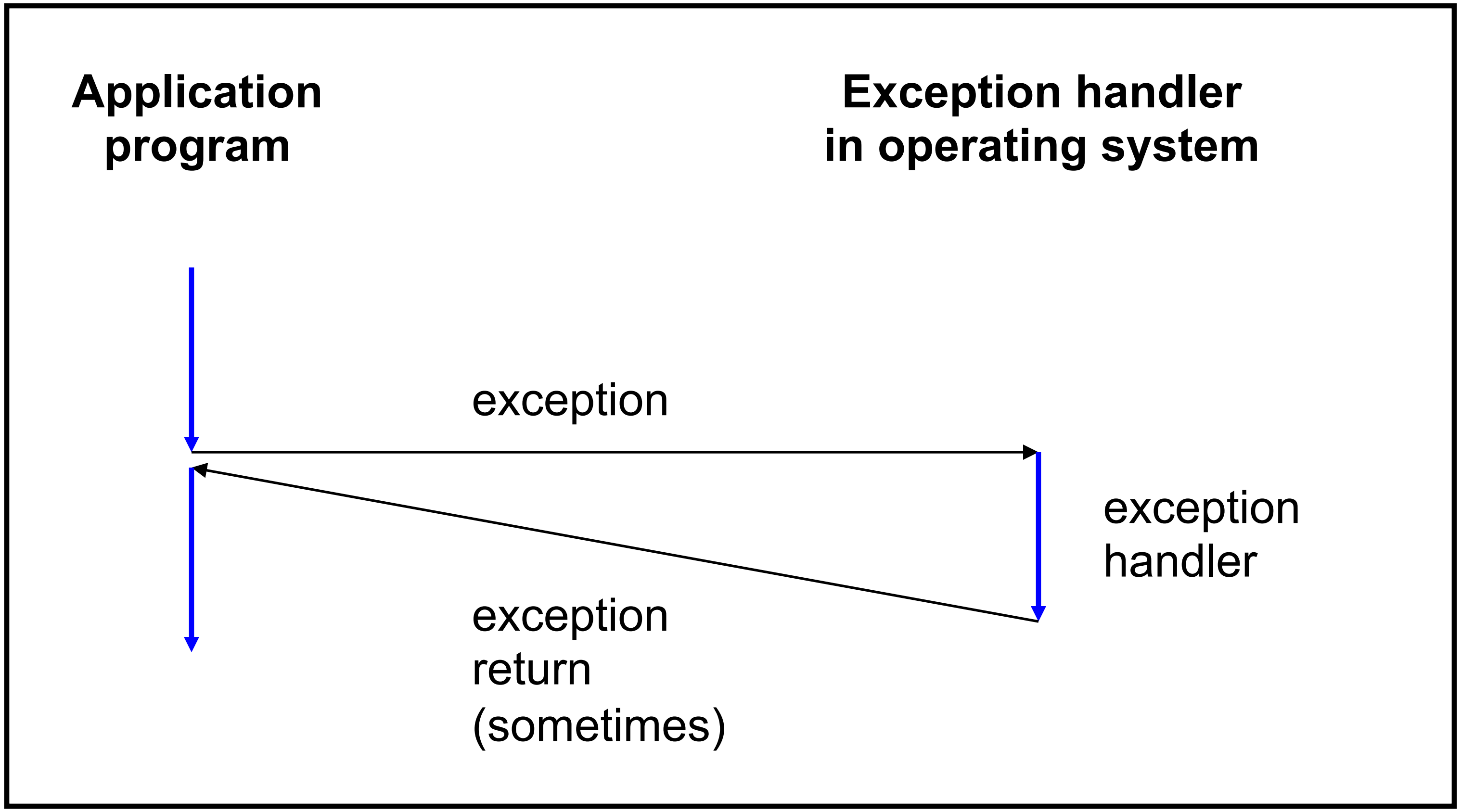
Exceptions Note

Note:

Exceptions in OS ≠ 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

- 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

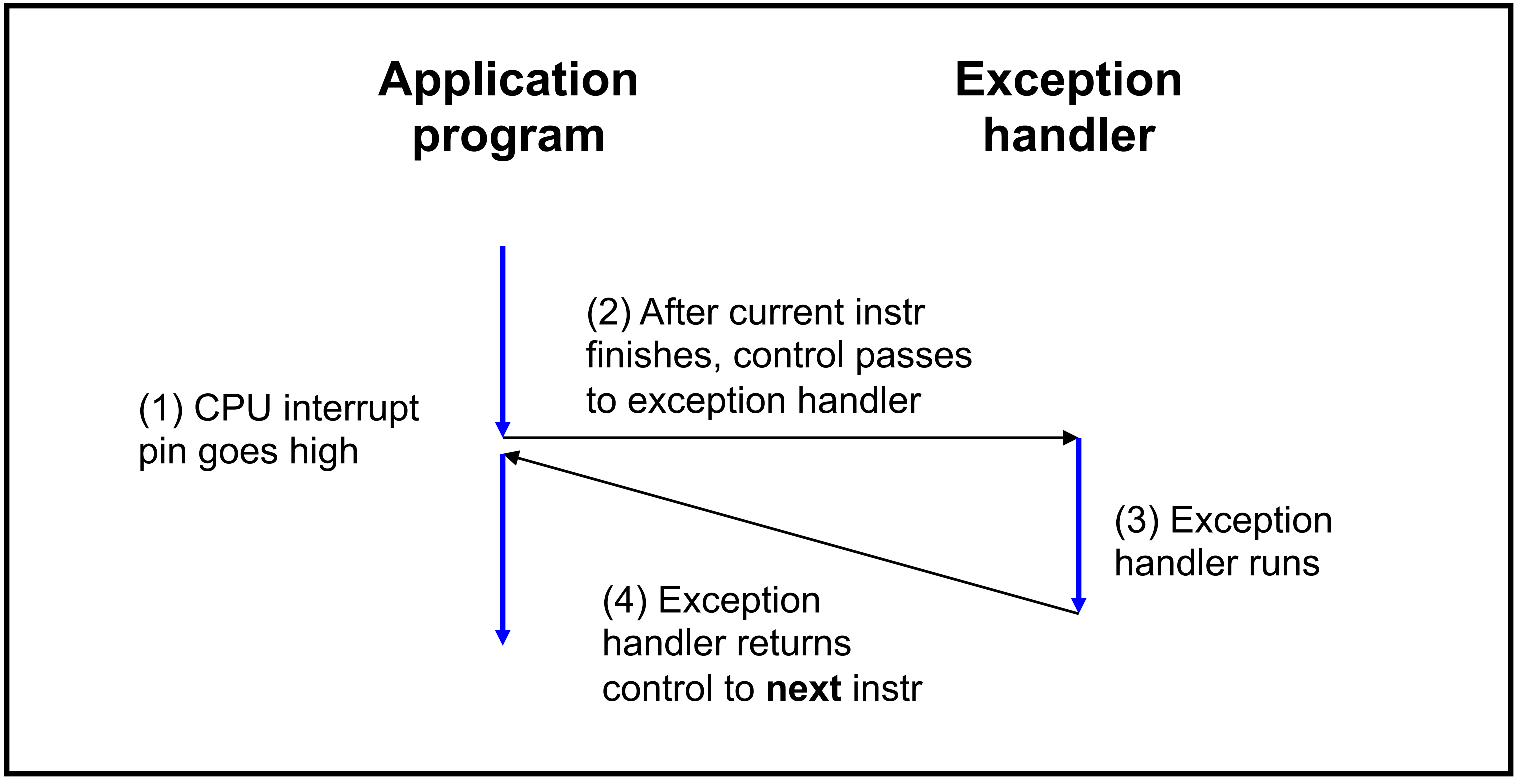
- CPU saves **additional data**
  - E.g. values of all registers
- CPU pushes data onto **OS's stack**, not application pgm's stack
- Handler runs in **kernel/privileged mode**, not in **user mode**
  - Handler can execute all instructions and access all memory
- Control **might return** to some instruction 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



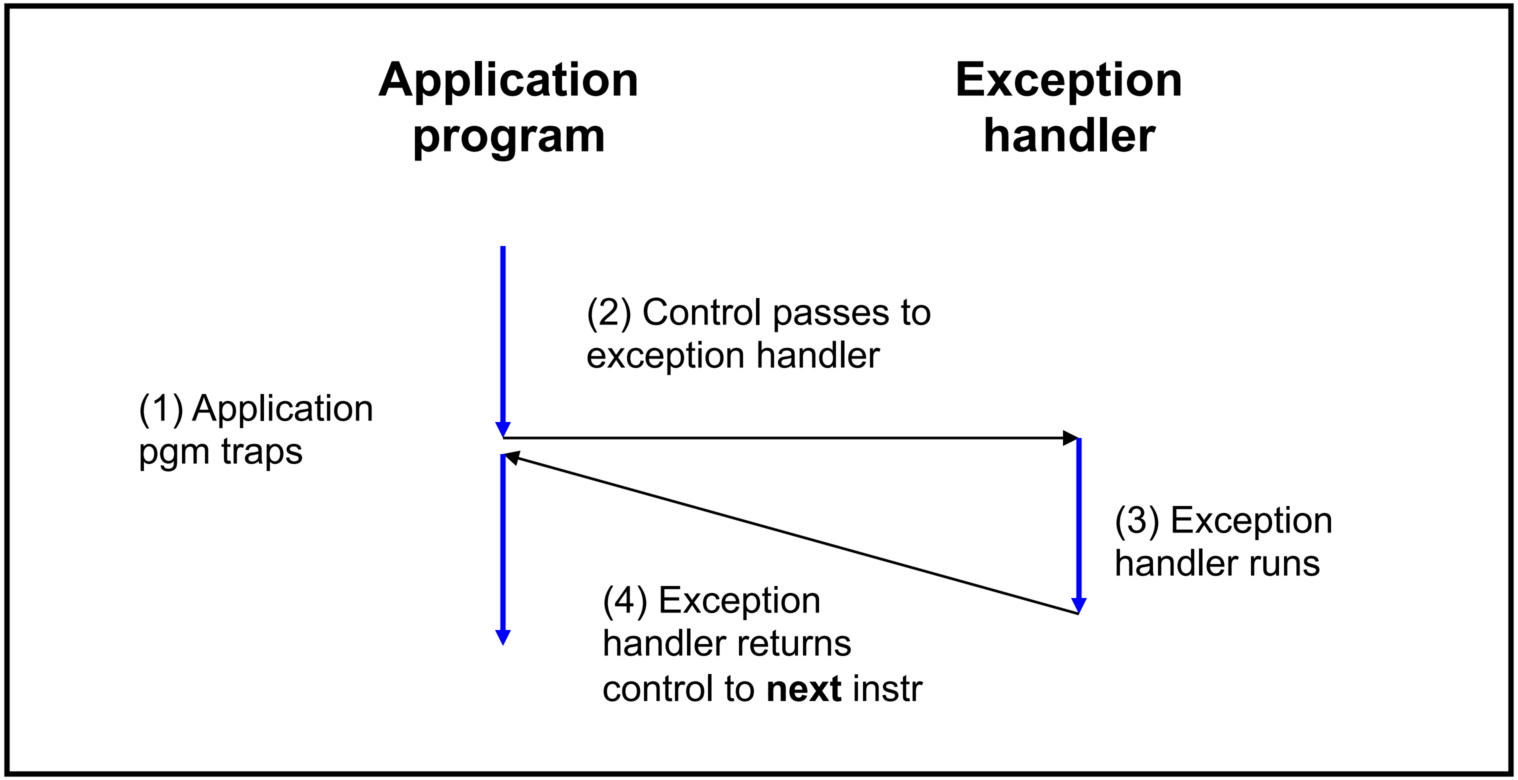


Occurs when: External (off-CPU) device requests attention

Examples:

- User presses key
- Disk controller finishes reading/writing data
- Network packet arrives

(2) Traps



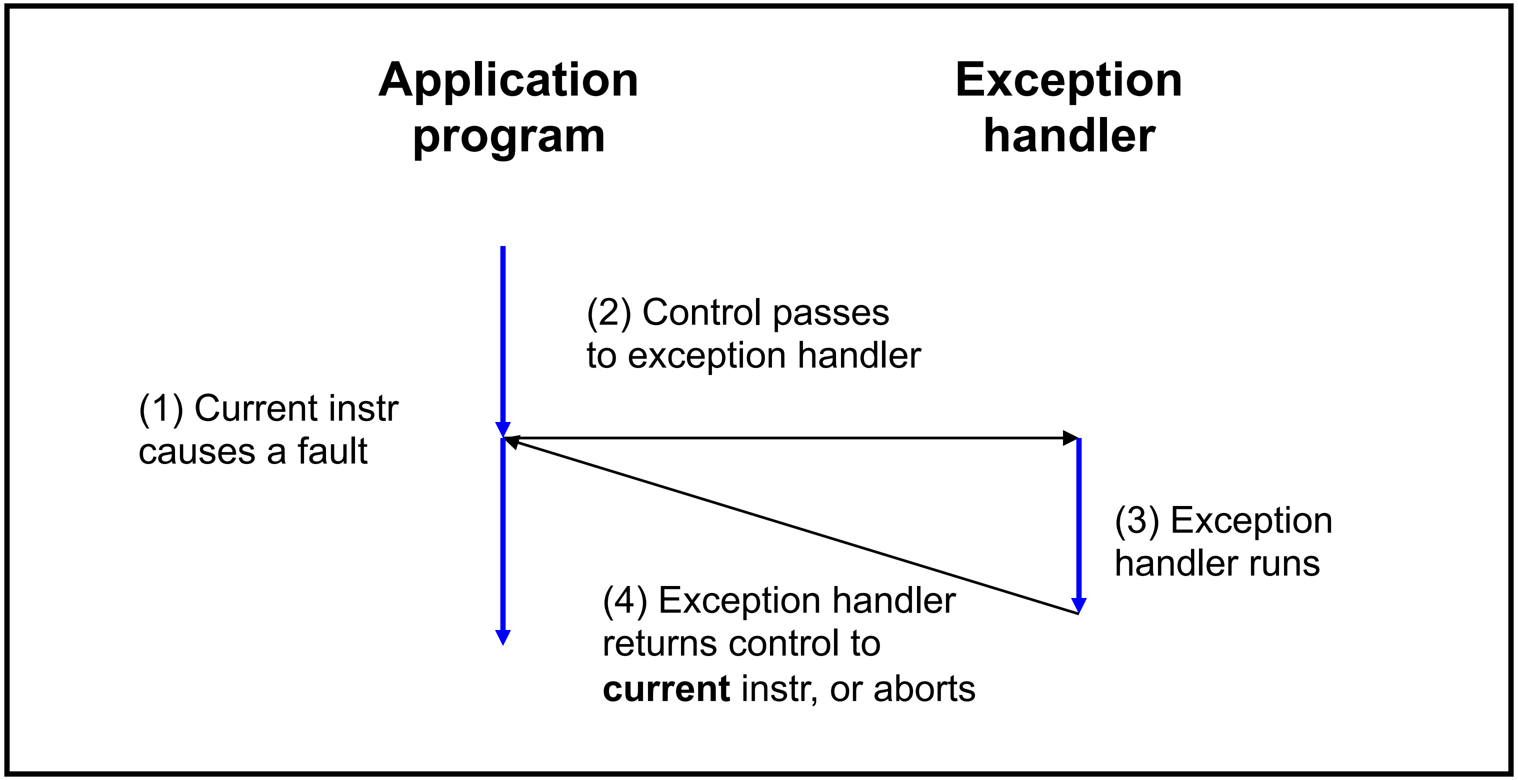
Occurs when: Application pgm requests OS service

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

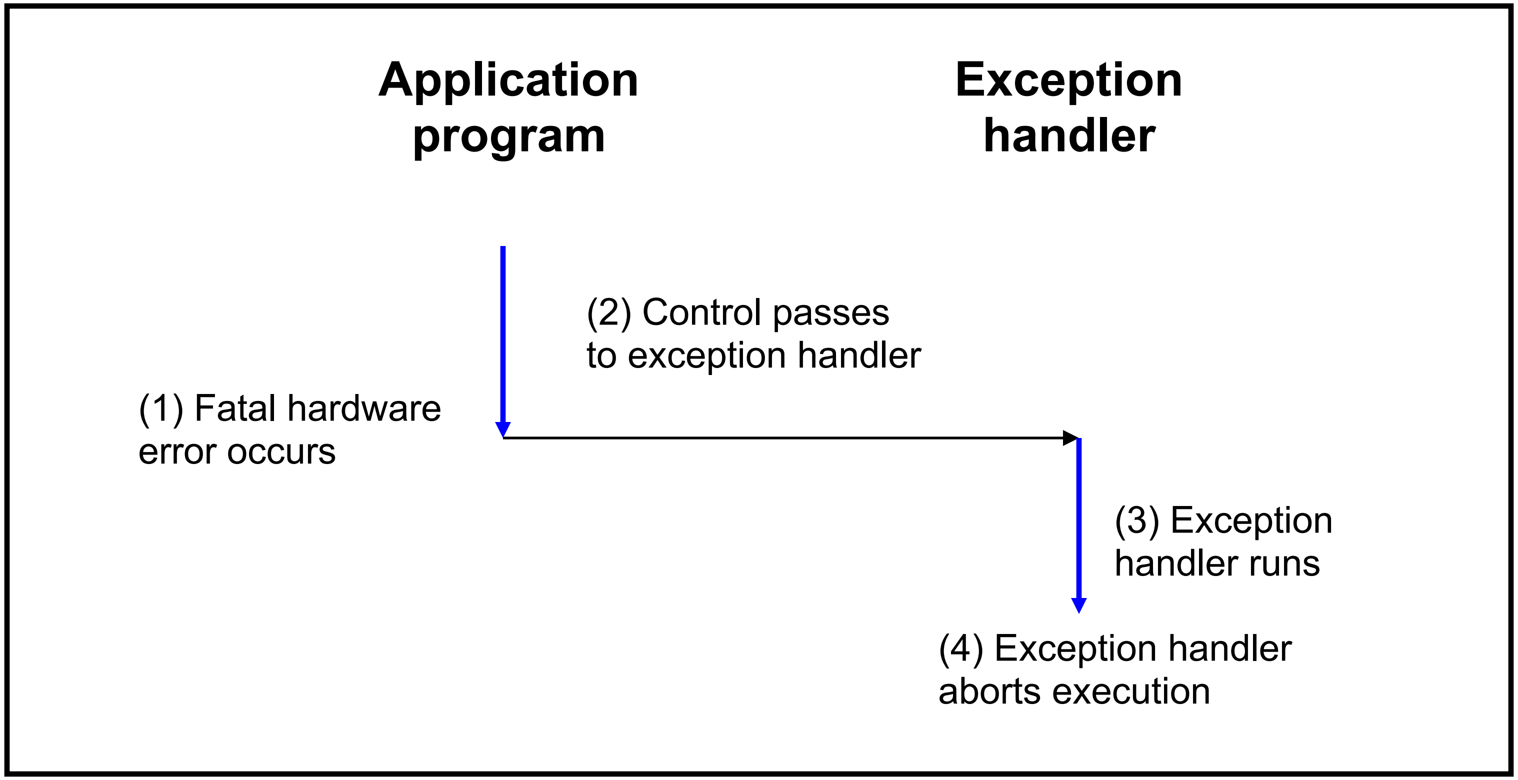


Occurs when: Application pgm causes a (possibly recoverable) error

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
Interrupt	External device requests attention	Asynch	Return to next instr
Trap	Application pgm requests OS service	Sync	Return to next instr
Fault	Application pgm causes (maybe recoverable) error	Sync	Return to current instr (maybe)
Abort	HW detects non-recoverable error	Sync	Do not return

Aside: Traps in Linux / AArch64

To execute a trap, application program should:

- Place number in X8 register indicating desired OS service
- Place arguments in X0..X7 registers
- Execute assembly language "supervisor call" instruction: `svc 0`

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

```
mov x8, 214
adr x0, newAddr
svc 0
```

Place 214 (change size of heap section) in X8

Place new address of end of heap in X0

Execute trap

Aside: System-Level Functions

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

- Part of C library, but not portable to other OS-es

Example: To change size of heap section of memory…

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

`brk()` is a system-level function

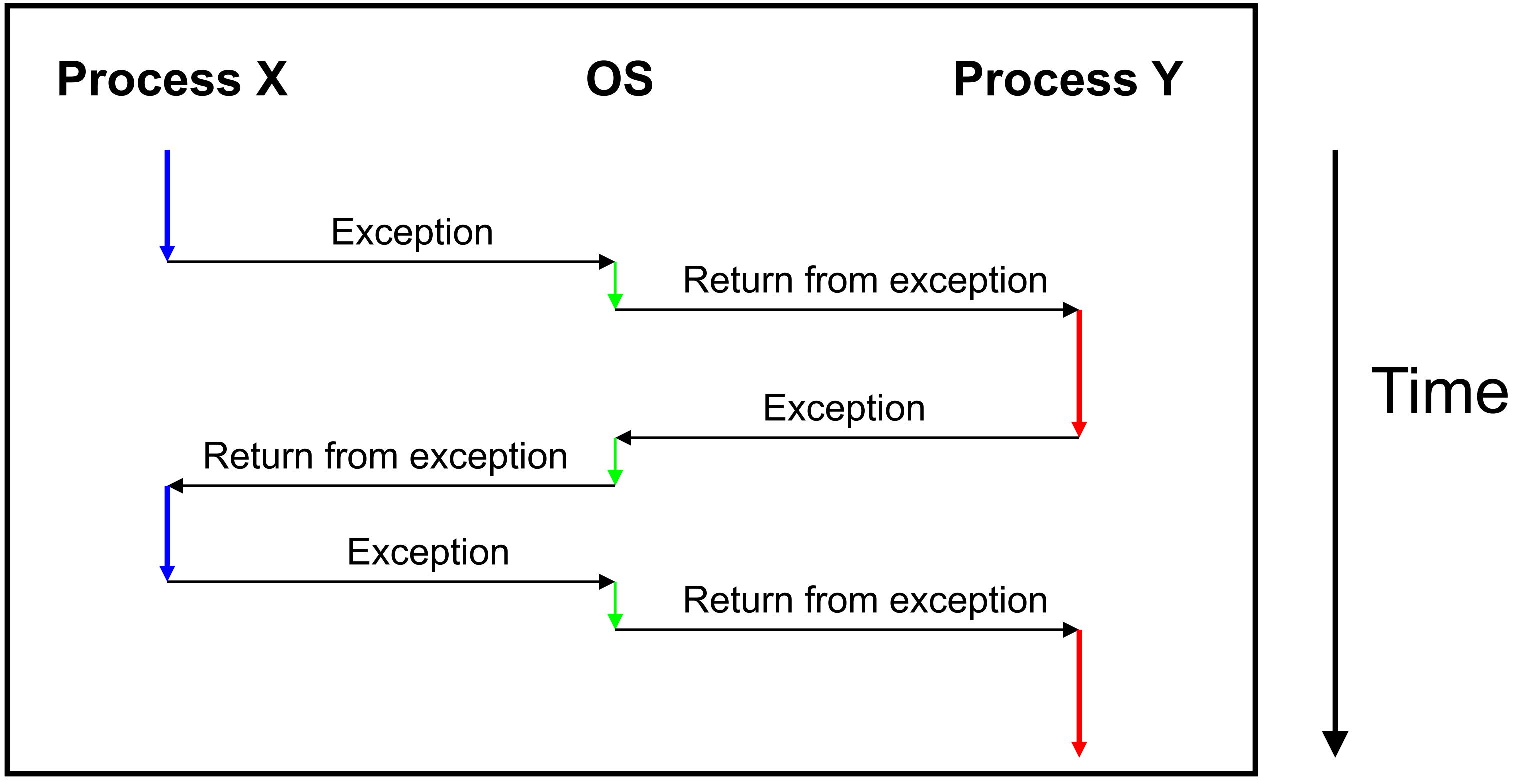
```
/* unistd.s */
brk:    mov x8, 214
        adr x0, newAddr
        svc 0
        ret
```

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

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

See Appendix for some Linux system-level functions

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

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

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

Appendix: System-Level Functions

The following tables present system-level functions that implement the “traditional Unix” API

- Implemented under the traditional names in the Linux C library for compatibility
- But, do not necessarily correspond 1:1 to system traps in Linux – for example, Linux/AArch64 has one openat() trap that accomplishes the effects of open() and creat()

Appendix: System-Level Functions

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

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

Described in ***I/O Management*** lecture

Appendix: System-Level Functions

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

Function	Description
exit()	Terminate the current process
fork()	Create a child process
wait()	Wait for child process termination
execvp()	Execute a program in the current process
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**

Function	Description
dup()	Duplicate an open file descriptor
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**

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

Described in ***Dynamic Memory Management*** lecture

Appendix: System-Level Functions

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

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

Described in ***Signals*** lecture