Exceptions and Processes

The material for this lecture is drawn from
Computer Systems: A Programmer’s Perspective (Bryant & O’Hallaron) Chapter 8

Goals of this Lecture

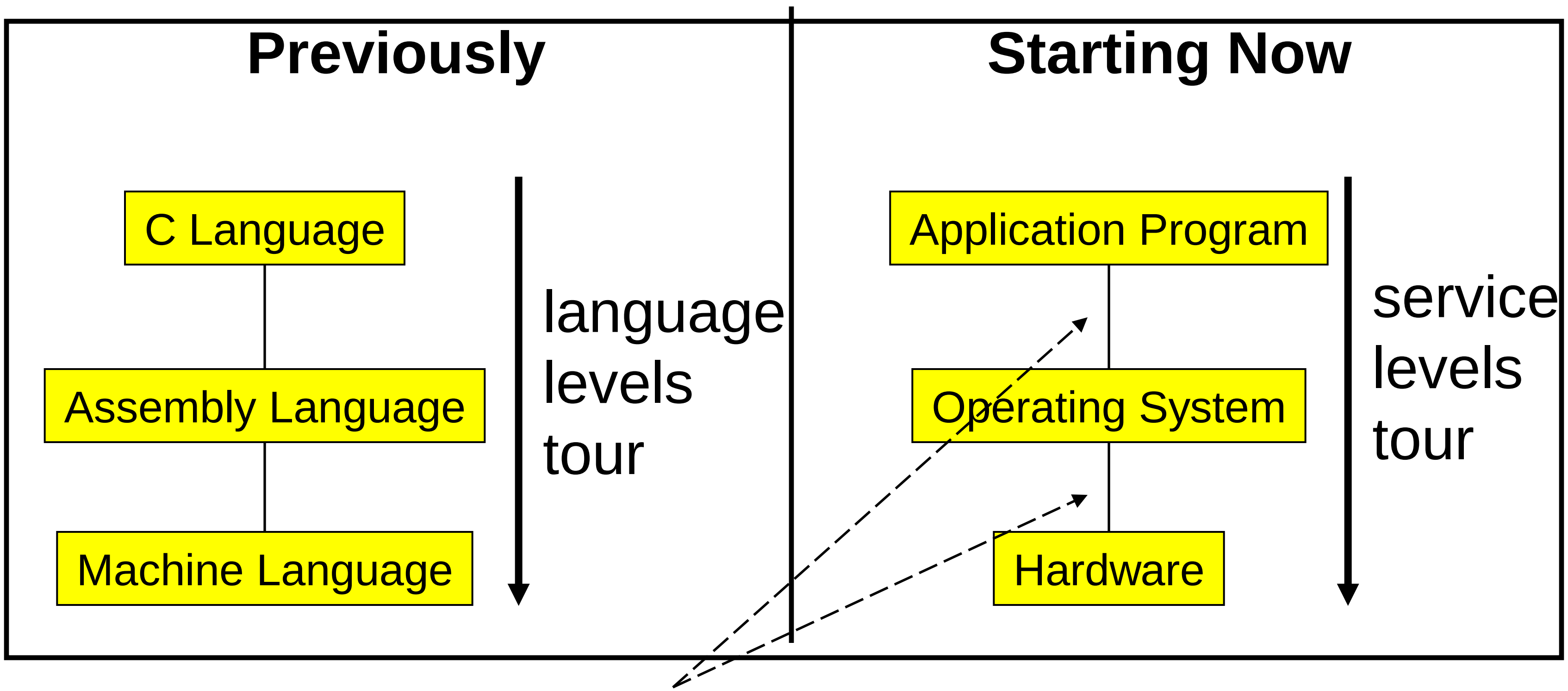
- Help you learn about:
  - **Exceptions**
  - The **process** concept

  … 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 systems 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 control of the CPU
- But multiple executing programs must share one CPU (or a few CPUs)
- How is that illusion implemented?

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?

Answers: Exceptions…

Exceptions

- **Exception**
  - An abrupt change in control flow in response to a change in processor state
- Examples:
  - Application program:
    - Requests I/O
    - Requests more heap memory
    - Attempts integer division by 0
    - Attempts to access privileged memory
    - Accesses variable that is not in real memory (see upcoming "Virtual Memory" lecture)
  - User presses key on keyboard
  - Disk controller finishes reading data

} Synchronous (application program examples)

} Asynchronous (key press, disk controller)

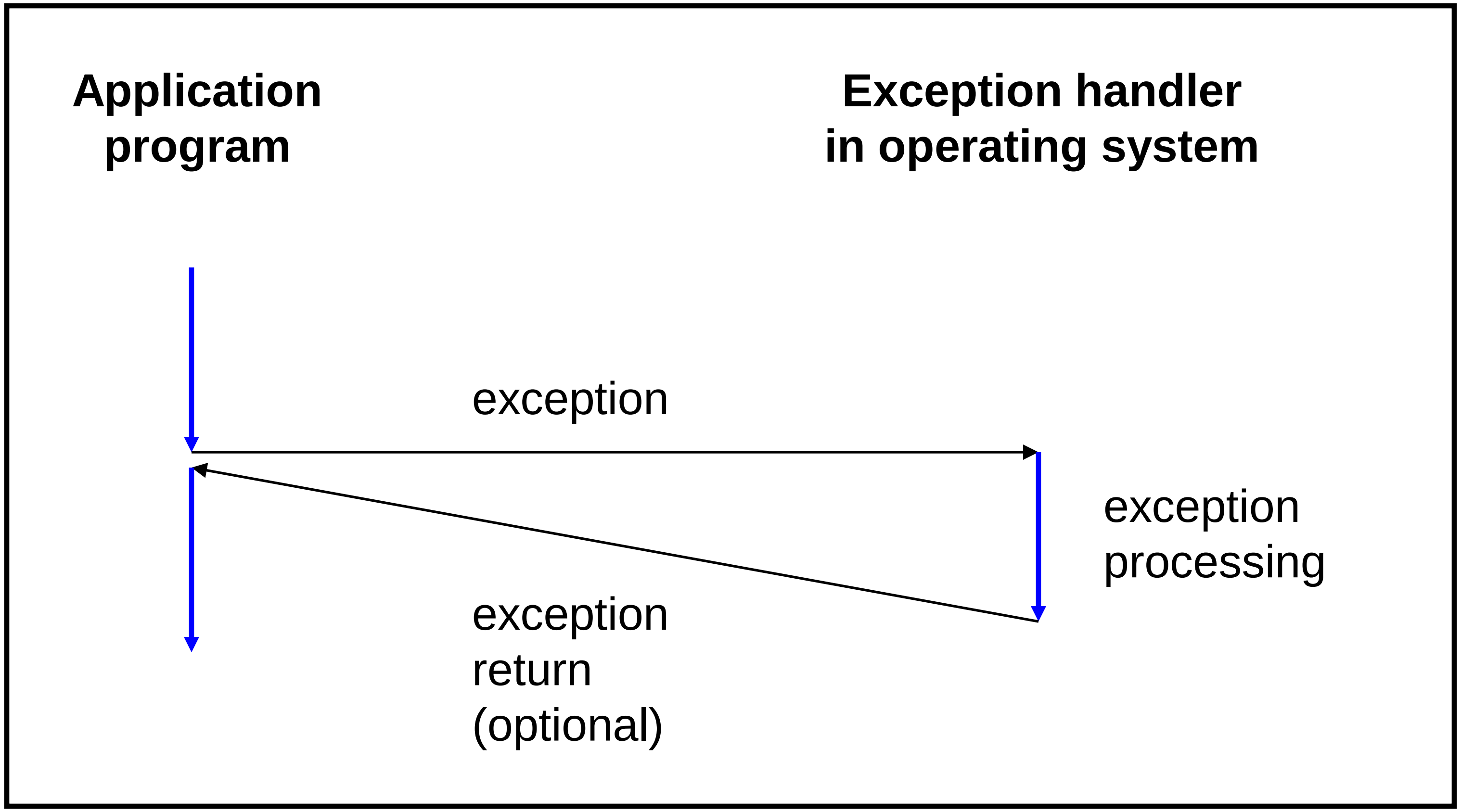
Exceptions Note

- Note:

  Exceptions in OS ≠ exceptions in Java

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

Exceptional Control Flow



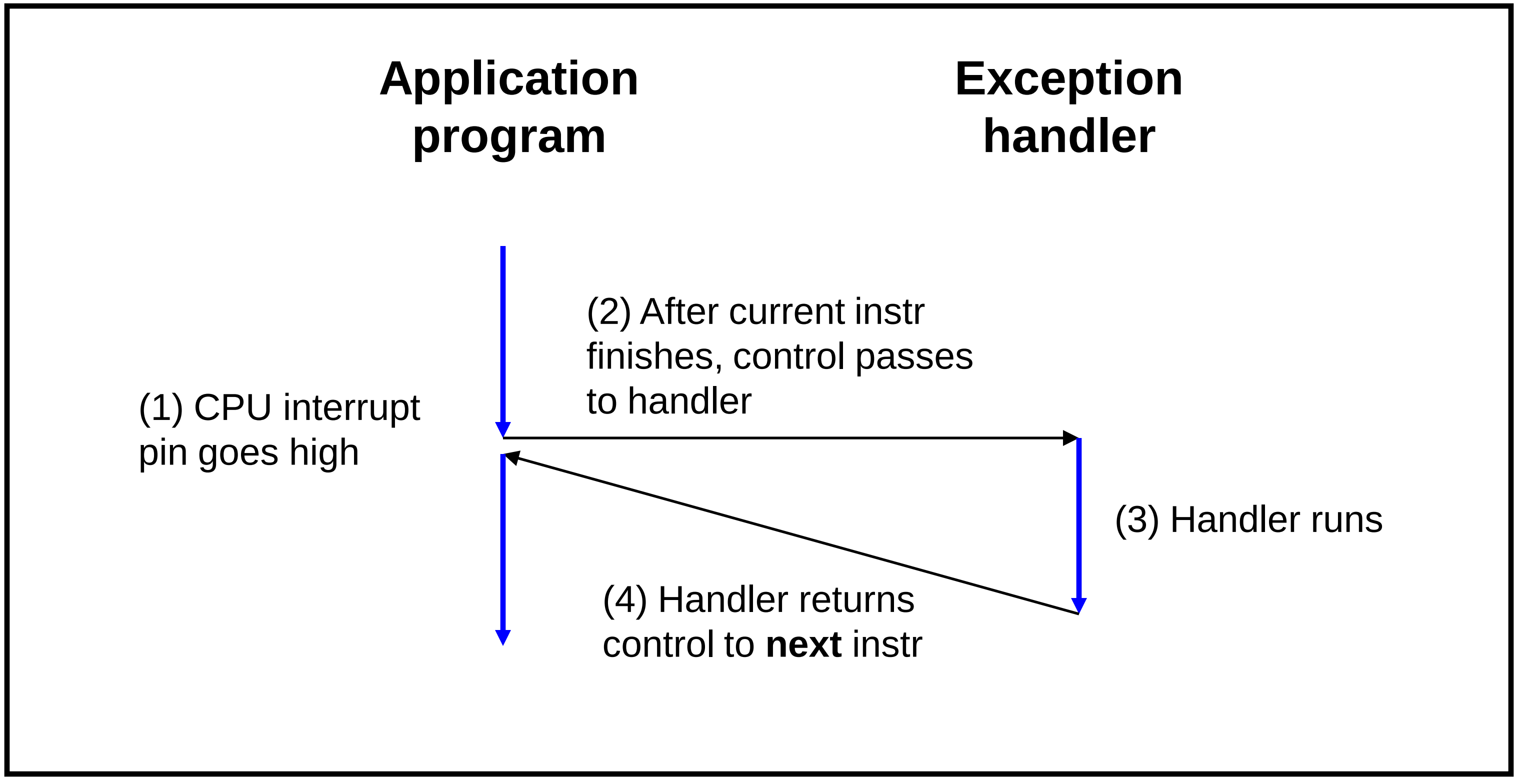
Exceptions vs. Function Calls

- Exceptions are **similar to** function calls
  - Control transfers from original code to other code
  - Other code executes
  - Control returns to original code
- Exceptions are **different from** function calls
  - Processor pushes **additional state** onto stack
    - E.g. values of all registers
  - Processor pushes data onto **OS's stack**, not application pgm's stack
  - Handler runs in **privileged mode**, not in **user mode**
    - Handler can execute all instructions and access all memory
  - Control **might return** to next instruction
    - Control sometimes returns to **current** instruction
    - Control sometimes does not return at all!

Classes of Exceptions

- There are 4 classes of exceptions…

(1) Interrupts



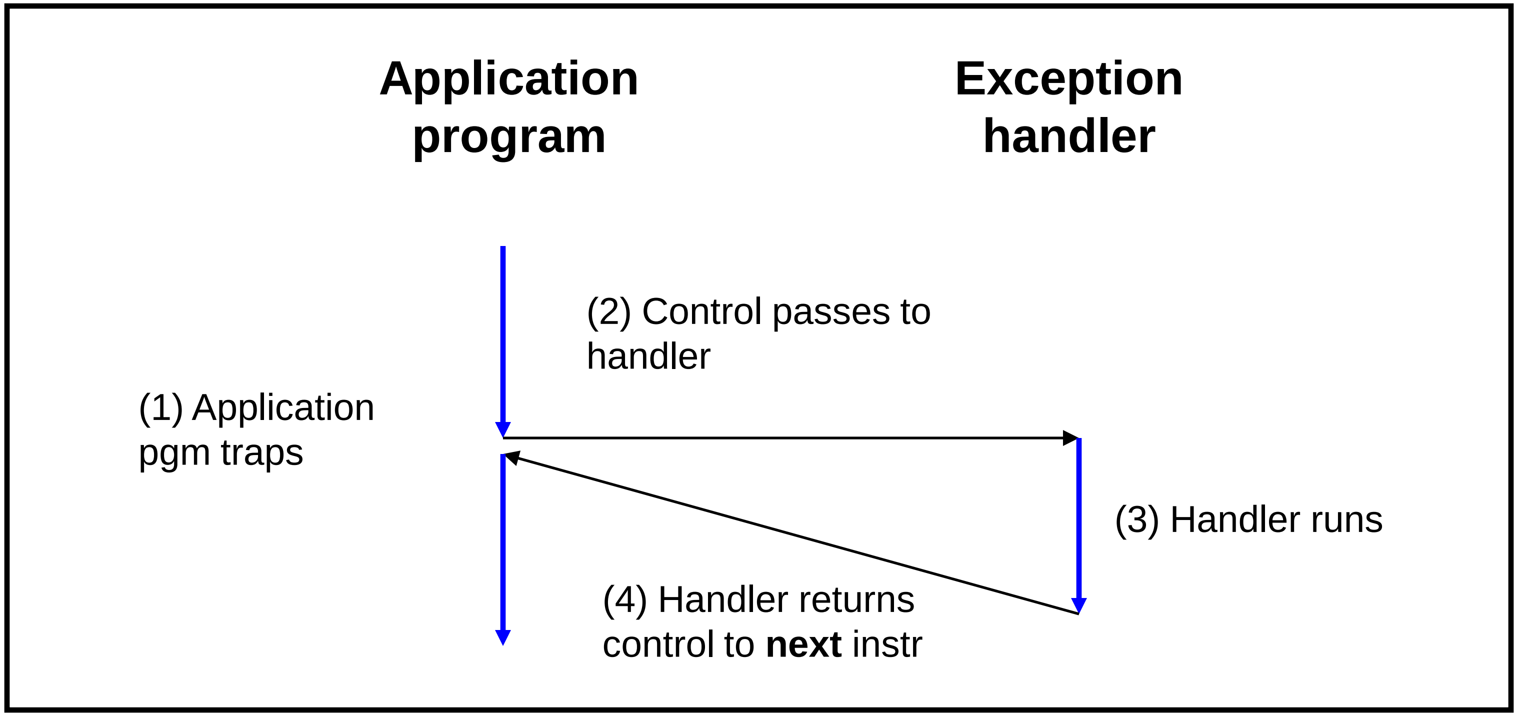
Cause: Signal from I/O device

Examples:

User presses key

Disk controller finishes reading/writing data

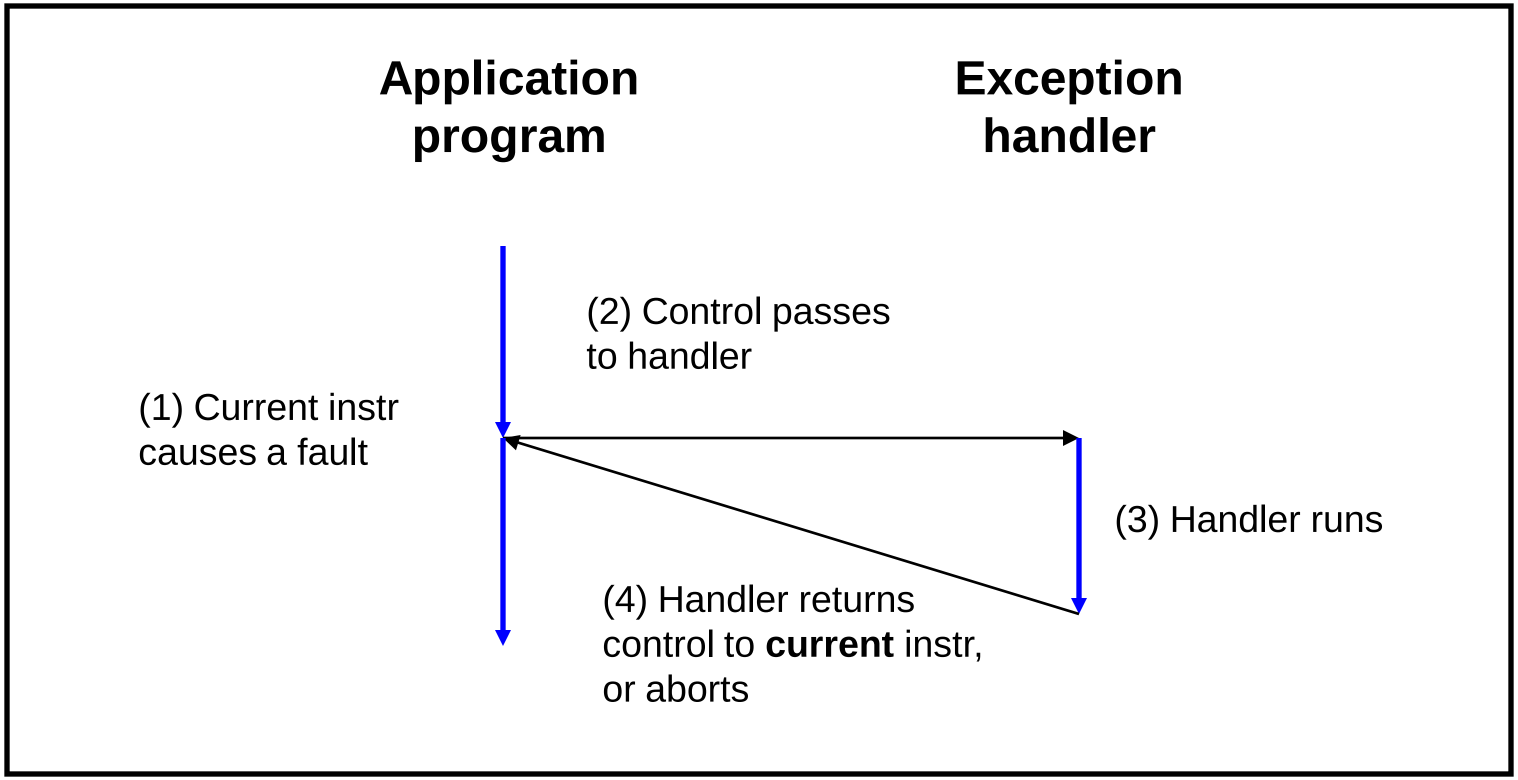
(2) Traps



Cause: Intentional (application pgm requests OS service)
Examples:
Application pgm requests more heap memory
Application pgm requests I/O

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

(3) Faults



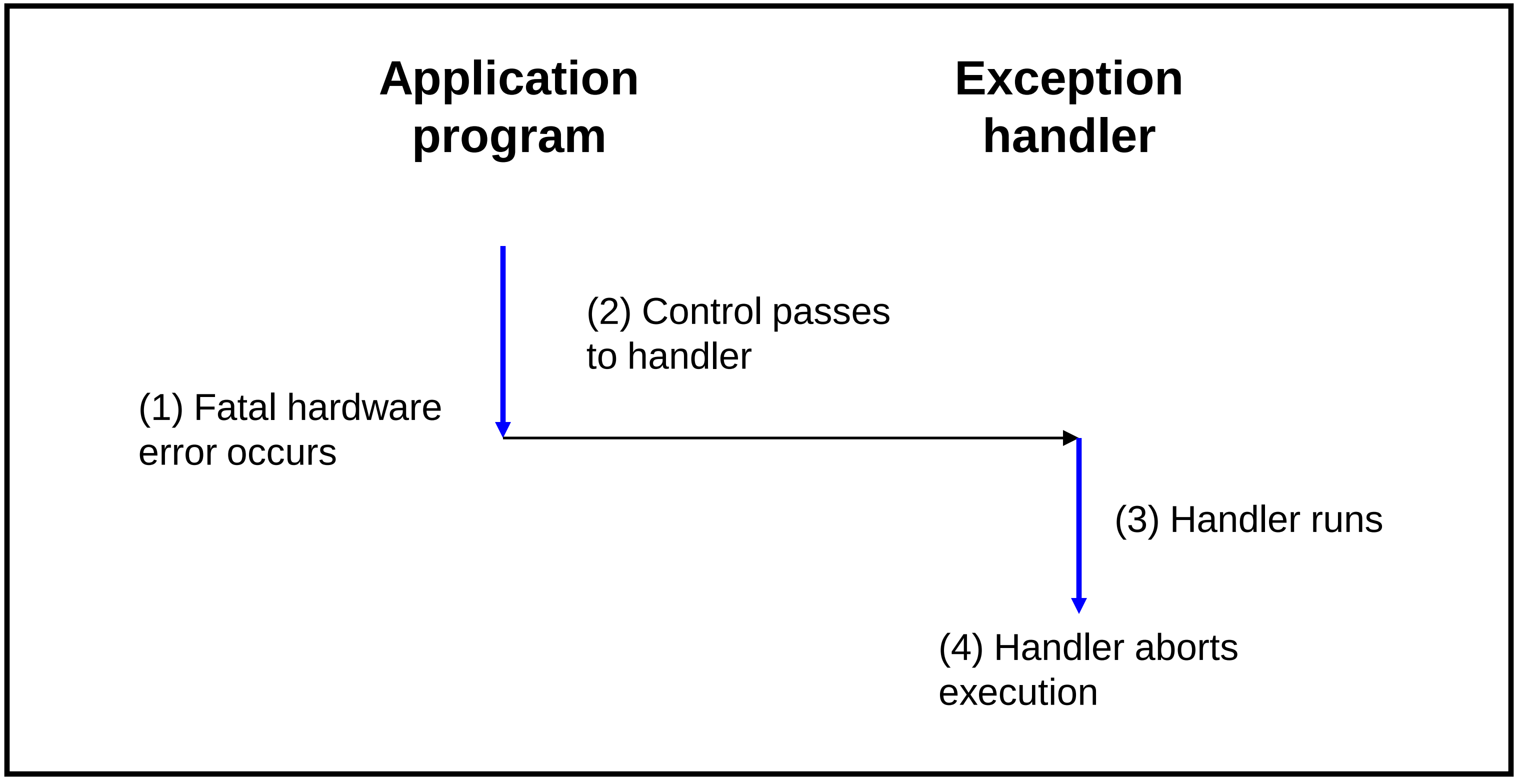
Cause: Application pgm causes (possibly) recoverable error

Examples:

Application pgm accesses privileged memory (seg fault)

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

(4) Aborts



Cause: Non-recoverable error

Example:

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

Summary of Exception Classes

Class	Cause	Asynch/Synch	Return Behavior
Interrupt	Signal from I/O device	Asynch	Return to next instr
Trap	Intentional	Sync	Return to next instr
Fault	(Maybe) recoverable error	Sync	(Maybe) return to current instr
Abort	Non-recoverable error	Sync	Do not return

Exceptions in Intel Processors

Each exception has a number
Some exceptions in Intel processors:

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

Traps in Intel Processors

- To execute a trap, application program should:
  - Place number in EAX register indicating desired functionality
  - Place parameters in EBX, ECX, EDX registers
  - Execute assembly language instruction "int 128"
- Example: To request more heap memory…

In Linux, 45 indicates request for more heap memory

```
movl    $45, %eax
movl   $1024, %ebx
int     $128
```

Causes trap

Request is for 1024 bytes

System-Level Functions

- For convenience, traps are wrapped in **system-level functions**
- Example: To request more heap memory…

```
/* unistd.h */
void *sbrk(intptr_t increment);
…
```

`sbrk()` is a system-level function

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

```
/* client.c */
…
sbrk(1024);
…
```

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

See Appendix for list of some Linux system-level functions

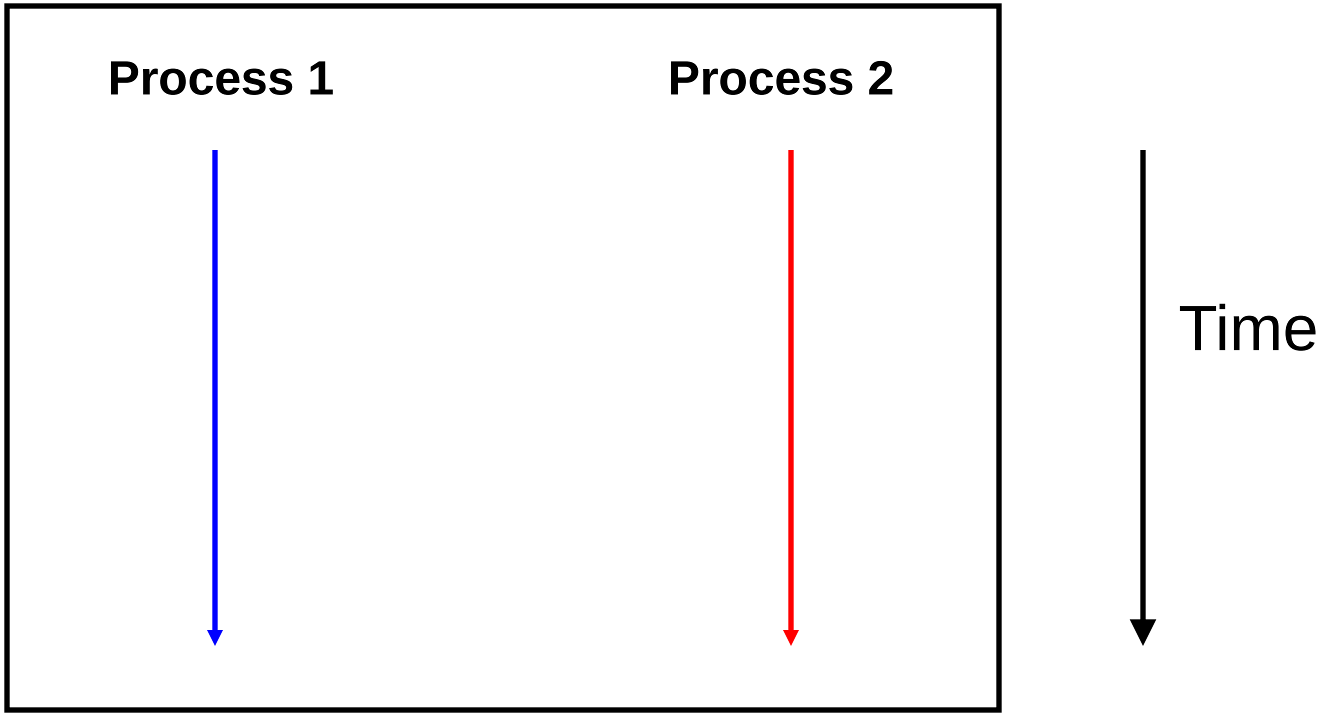
Processes

- **Program**
  - Executable code
- **Process**
  - An instance of a program in execution
- Each program runs in the **context** of some process
- **Context** consists of:
  - Process ID
  - Address space
    - TEXT, RODATA, DATA, BSS, HEAP, and STACK
  - Processor state
    - EIP, EFLAGS, EAX, EBX, etc. registers
  - Etc.

Significance of Processes

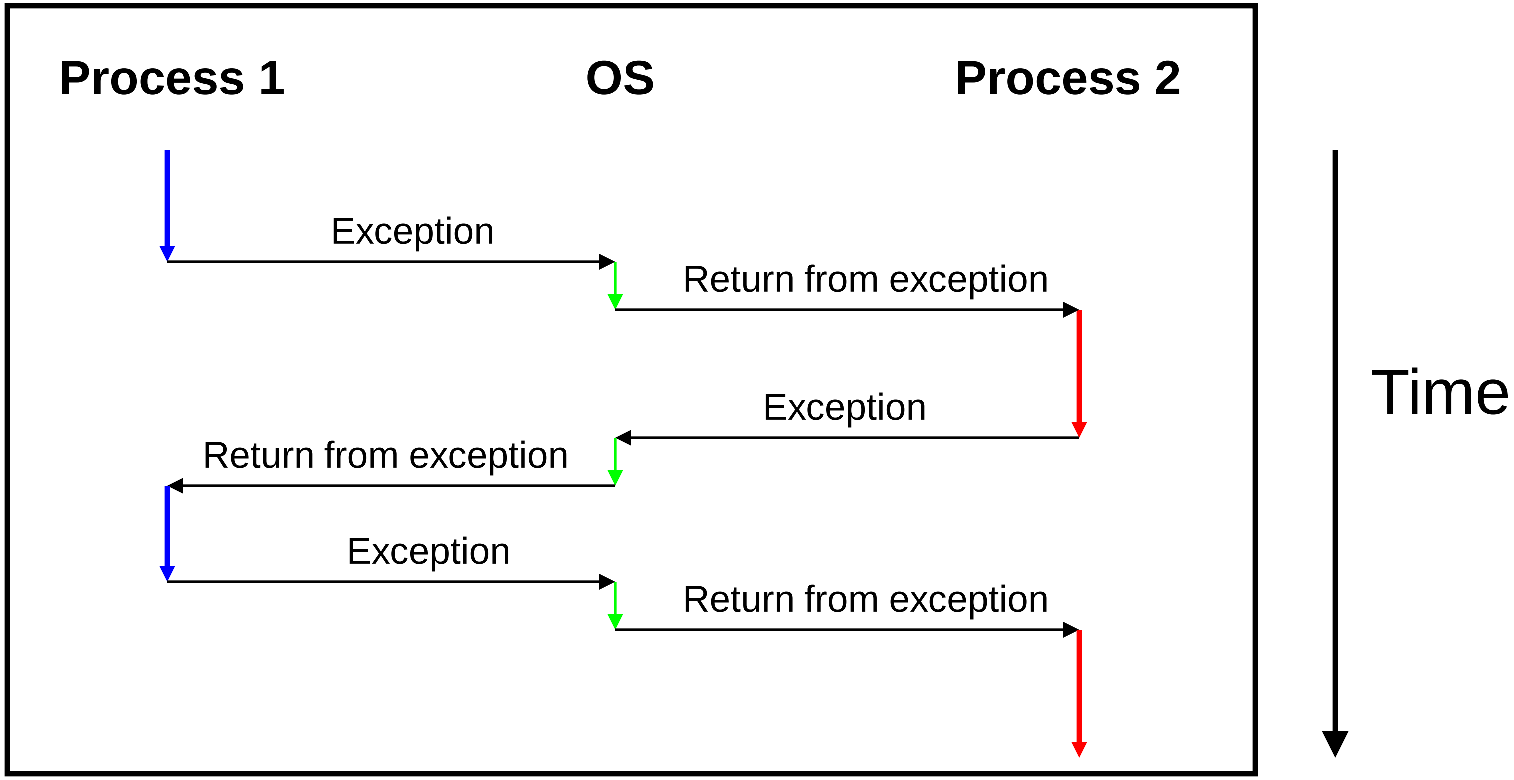
- **Process** is a profound abstraction
- The process abstraction provides application pgms with two key illusions:
  - Private control flow
  - Private address space

Private Control Flow: Illusion



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

Private Control Flow: Reality



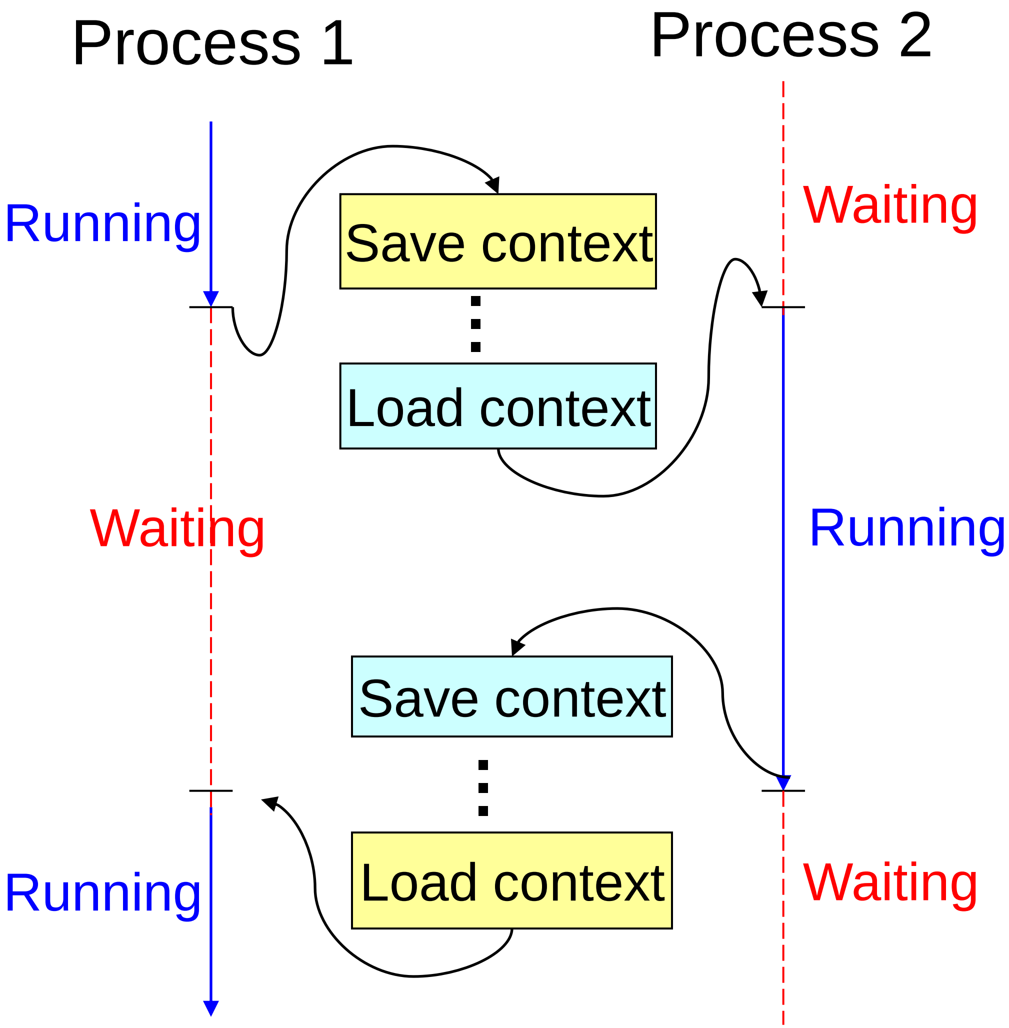
All application processes -- and the OS process -- share the same CPU(s)

Context Switches

- **Context switch**
  - The activity whereby the OS assigns the CPU to a different process
  - Occurs during exception handling, at discretion of OS
- Exceptions can be caused:
  - Synchronously, by application pgm (trap, fault, abort)
  - Asynchronously, by external event (interrupt)
  - **Asynchronously, by hardware timer**
    - So no process can dominate the CPUs
- Exceptions are the mechanism that enables the illusion of private control flow

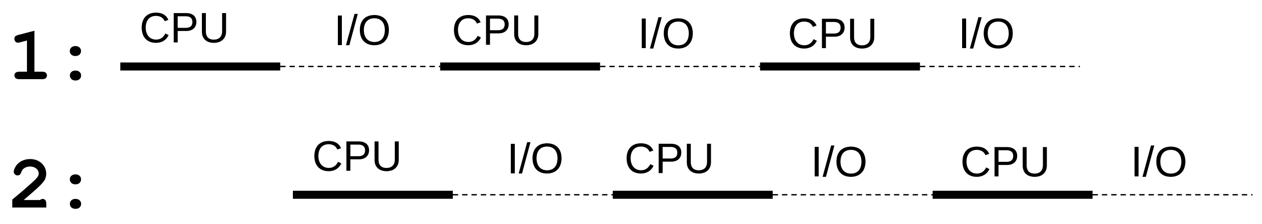
Context Switch Details

- Context
  - State the OS needs to restart a preempted process
- Context switch
  - Save the context of current process
  - Restore the saved context of some previously preempted process
  - Pass control to this newly restored process



When Should OS Do Context Switch?

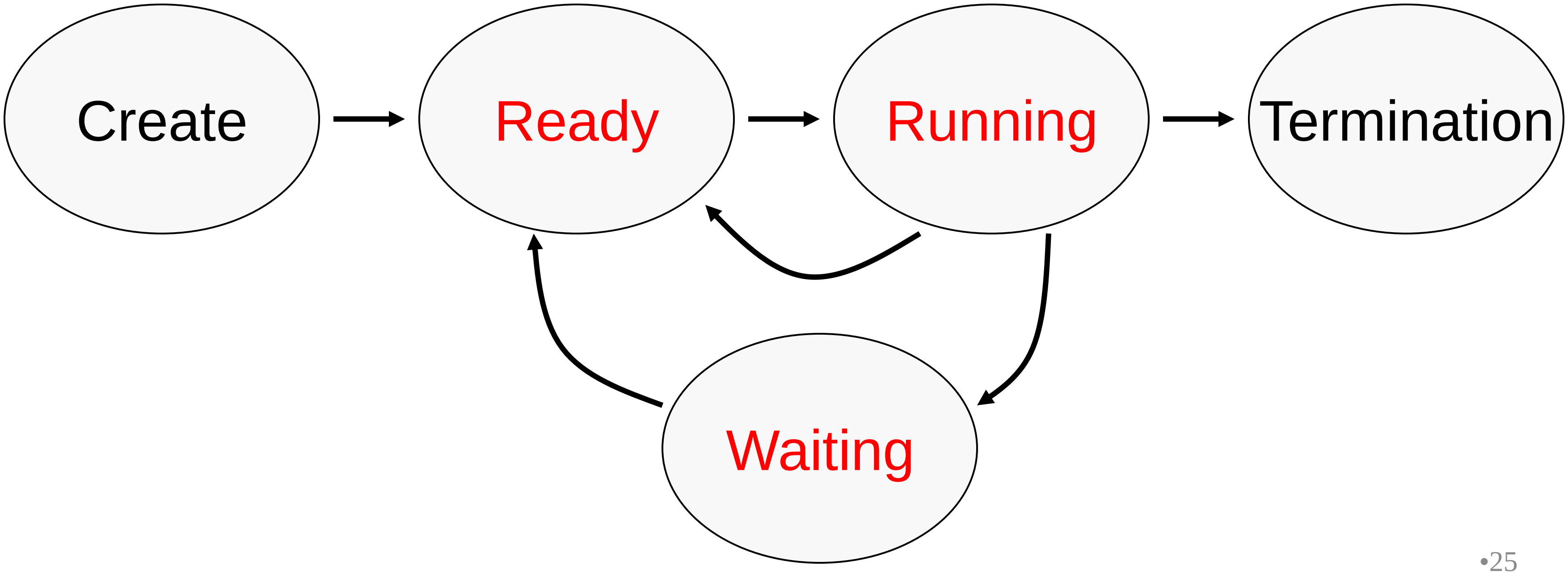
- When a process is stalled waiting for I/O
  - Better utilize the CPU, e.g., while waiting for disk access



- When a process has been running for a while
  - Sharing on a fine time scale to give each process the illusion of running on its own machine
  - Trade-off efficiency for a finer granularity of fairness

Life Cycle of a Process

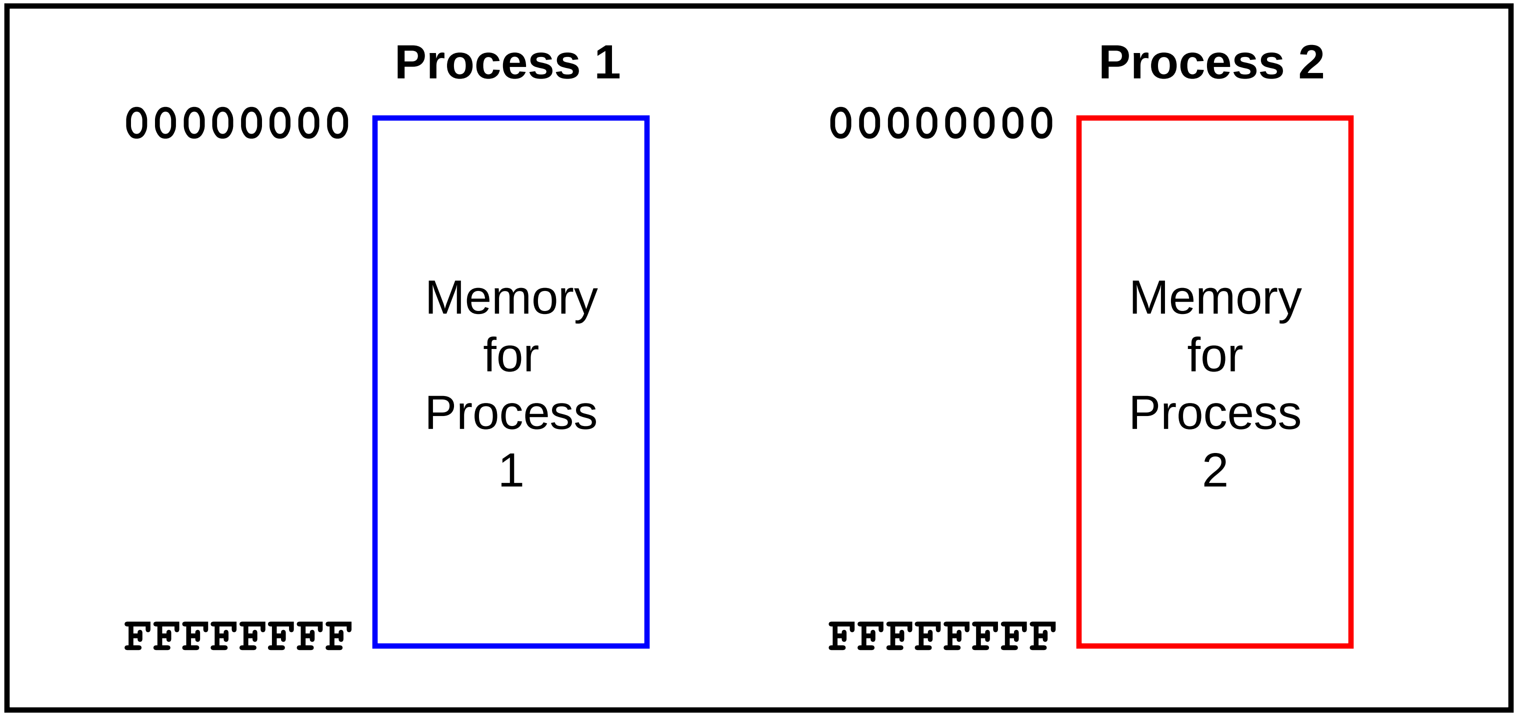
- **Running:** instructions are being executed
- **Waiting:** waiting for some event (e.g., I/O finish)
- **Ready:** ready to be assigned to a processor



Context Details

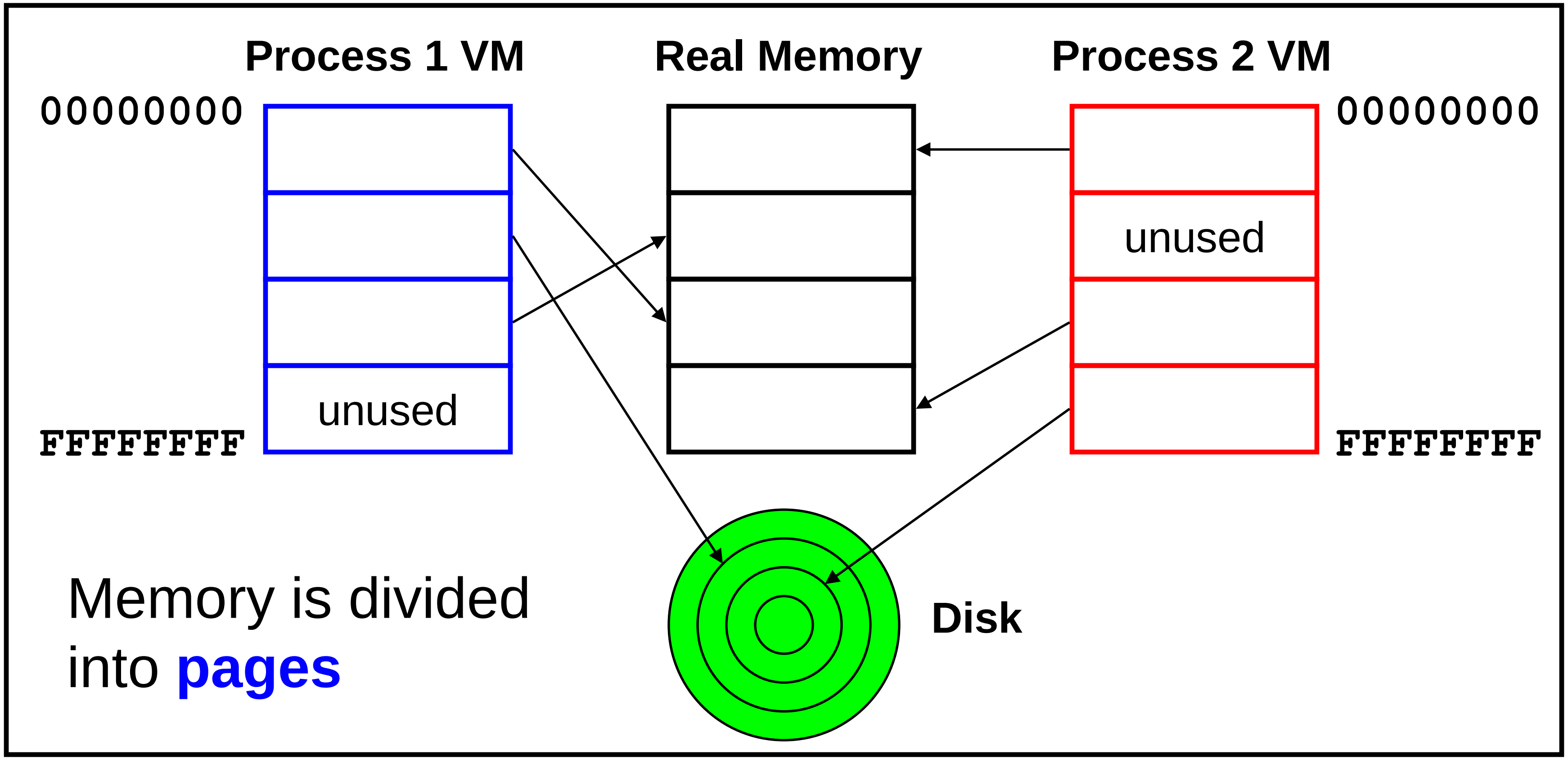
- What does the OS need to save/restore during a context switch?
  - Process state
    - New, ready, waiting, terminated
  - CPU registers
    - EIP, EFLAGS, EAX, EBX, …
  - I/O status information
    - Open files, I/O requests, …
  - Memory management information
    - Page tables (see "Virtual Memory" lecture)
  - Accounting information
    - Time limits, group ID, ...
  - CPU scheduling information
    - Priority, queues

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 real memory
Hardware and OS provide application pgms with a **virtual** view of memory, i.e. **virtual memory (VM)**

Private Address Space Details

- Exceptions (specifically, page faults) are the mechanism that enables the illusion of private address spaces

- See the **Virtual Memory** lecture for details

Summary

- **Exception**: an abrupt change in control flow
  - **Interrupts**: asynchronous; e.g. I/O completion, hardware timer
  - **Traps**: synchronous; e.g. app pgm requests more heap memory, I/O
  - **Faults**: synchronous; e.g. seg fault
  - **Aborts**: synchronous; e.g. parity error
- **Process**: An instance of a program in execution
  - Hardware and OS use exceptions to give each process the illusion of:
    - Private control flow (reality: **context switches**)
    - Private address space (reality: **virtual memory**)

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")`**

Described in **I/O Management** lecture

Appendix: System-Level Functions

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

Number	Function	Description
1	**`exit()`**	Terminate the process
2	**`fork()`**	Create a child process
7	**`waitpid()`**	Wait for process termination
7	**`wait()`**	(Variant of previous)
11	**`exec()`**	Execute a program in current process
20	**`getpid()`**	Get process id

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
63	**`dup2()`**	Close an open file descriptor, and duplicate an open file descriptor

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** lectures

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