Reading Reference: Textbook Chapter 1, 2

W2– OPERATING SYSTEMS ARCHITECTURAL INTERFACE, EXCEPTION CONTROL FLOW

CSCE 313 Spring 2017

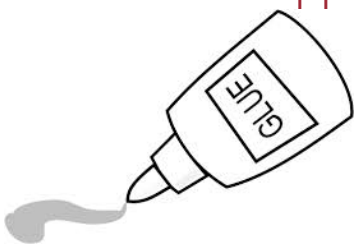
Content for this week

- OS **roles** and its key **challenges** (Text: Chap. 1)
- **Control Flow** in a modern computer system (Text: Chap. 2)
  - Normal flow of commands and data versus anything that happens “out of the ordinary” .. how do we handle that?
- **Architectural Interface** to the OS (Text: Chap. 2)
  - features we design in HW to facilitate the OS to meet some key challenges

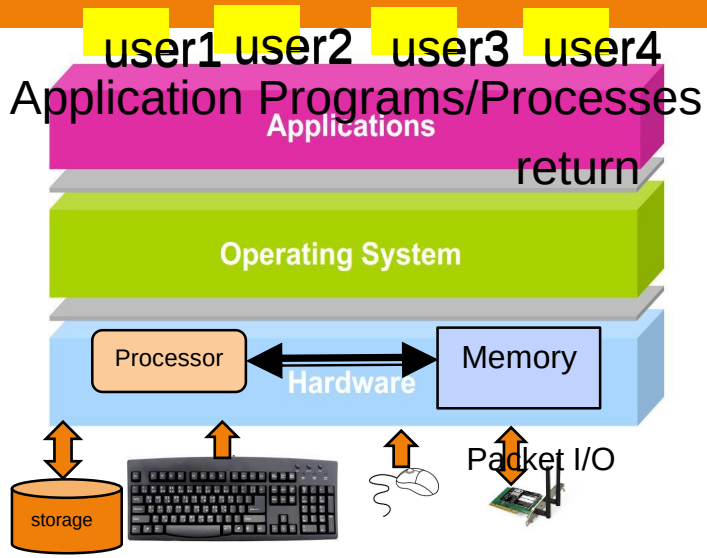
Operating System Roles

- Referee
  - Manage sharing of resources, Protection, Isolation
    - Resource allocation, isolation, communication

- Illusionist
  - Provide clean, easy to use abstractions of physical resources
    - Infinite memory, dedicated machine
    - Masking limitations, virtualization



- Glue
  - Common services
    - Storage, Window system, Networking
    - Sharing, Authorization
    - Look and feel



What, then, **is** an Operating System?

- The OS controls and coordinates the use of system resources.
- Primary goal: Provide a convenient environment for a user to access the available resources (CPU, memory, I/O)
  - Provide appropriate abstractions (files, processes, ...)
  - "virtual machine"
- Secondary goal: Efficient operation of the computer system.
- Key facets of Resource Management
  - Transforming: Create virtual substitutes that are easier to use.
  - Multiplexing: Create the illusion of multiple resources from a single resource
  - Scheduling: "Who gets the resource when?"

What an operating system is **not**

- OS is not a language or a compiler
- OS is not a command interpreter / window system
- OS is not a library of commands
- OS is not a set of utilities

Key OS Challenges

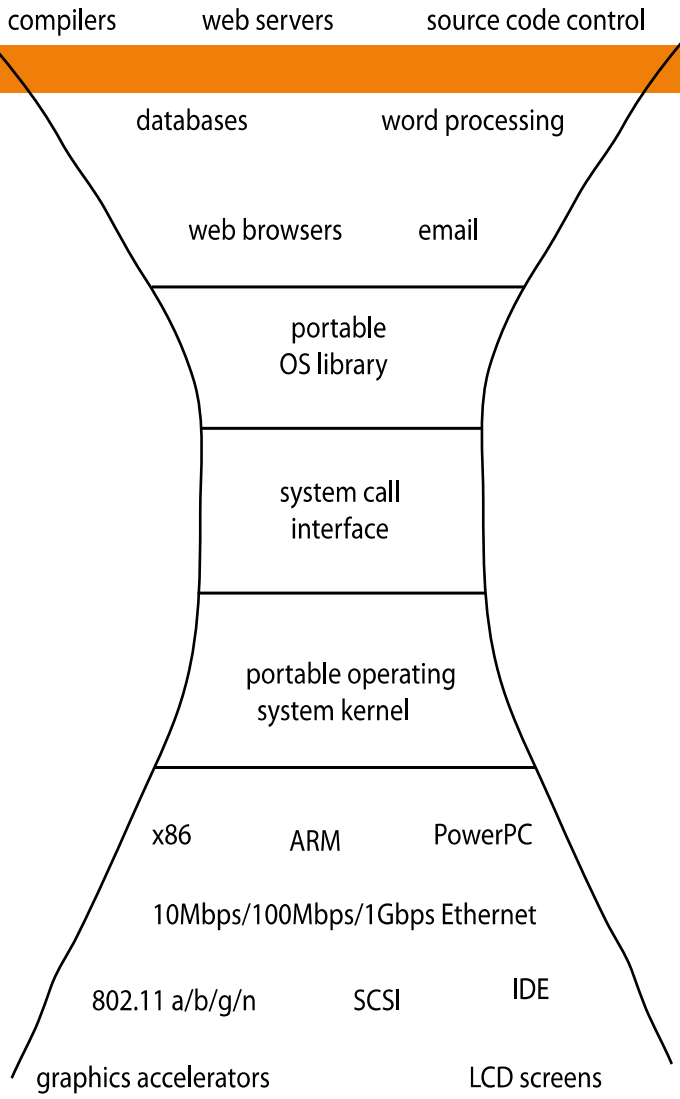
- **Reliability**
  - Does the system do what it was designed to do?
- **Availability**
  - What portion of the time is the system working?
  - Mean Time To Failure (MTTF), Mean Time to Repair

Key OS Challenges

- **Servicability**
  - Simplicity and Ease of system repair and maintenance
- **Security**
  - An OS needs both a security policy (what is permitted) and an enforcement mechanism (only allow permitted actions)
  - Can the system be compromised by an attacker?
  - Privacy: Data is accessible only to authorized users

Key OS Challenges

- **Portability**
  - For programs:
    - Application programming interface (API)
    - Abstract machine interface
  - For the operating system
    - Hardware abstraction layer



Key OS Challenges

- **Performance**
  - Latency/response time
    - How long does an operation take to complete?
  - Throughput
    - How many operations can be done per unit of time?
  - Overhead
    - How much extra work is done by the OS?
  - Fairness
    - How equal is the performance received by different users?
  - Predictability
    - How consistent is the performance over time?

Challenges in Modern OSs

- Smart Phones
  - Responsiveness, security
- Embedded Systems
  - Reliable
- Web Servers
  - Supporting billions of requests/sec efficiently
- Virtual Machines
  - Low overhead and also proper h/w virtualization
- Server Clusters
  - Hide the clustering details from application programs

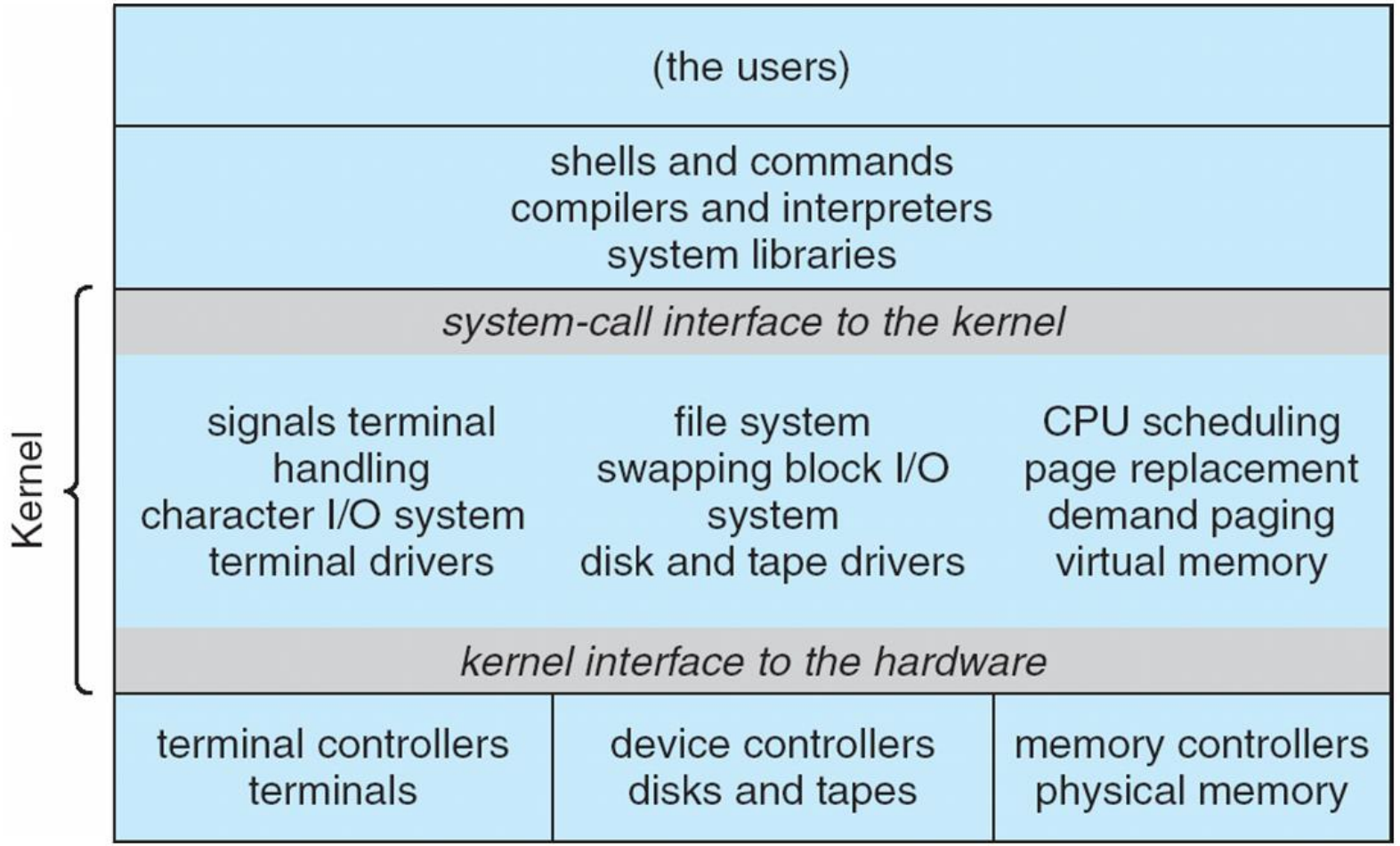
Challenges in Tomorrow's OSs

- Existing challenges would be more critical
  - OSs controlling future self driving cars, or traffic lights need to be absolutely *reliable, secure, and efficient*
- The future of OSs is intertwined with that of emerging computing hardware
  - Giant-scale data centers
  - Increasing numbers of processors per computer
  - Newer portable devices
  - Very large scale storage

Content for this week

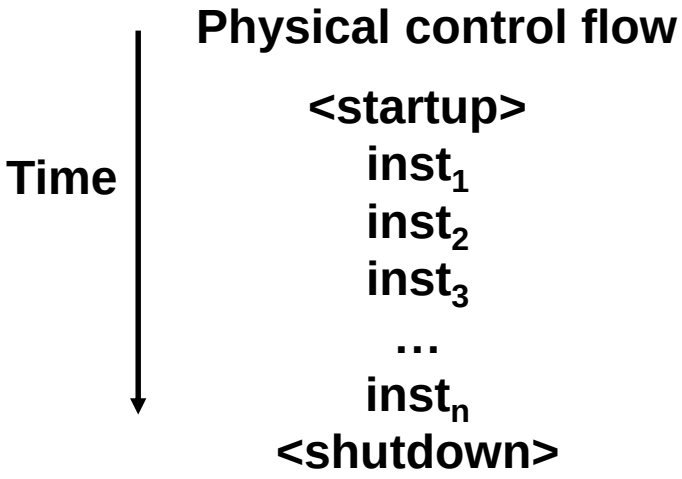
- OS **roles** and its key **challenges** (Text: Chap. 1)
- **Control Flow** in a modern computer system (Text: Chap. 2)
  - Normal flow of commands and data versus anything that happens “out of the ordinary” .. how do we handle that?
- **Architectural Interface** to the OS (Text: Chap. 2)
  - features we design in HW to facilitate the OS to meet some key challenges

Traditional UNIX System Structure



Control Flow

- Computers 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 system's physical *control flow* (or *flow of control*)

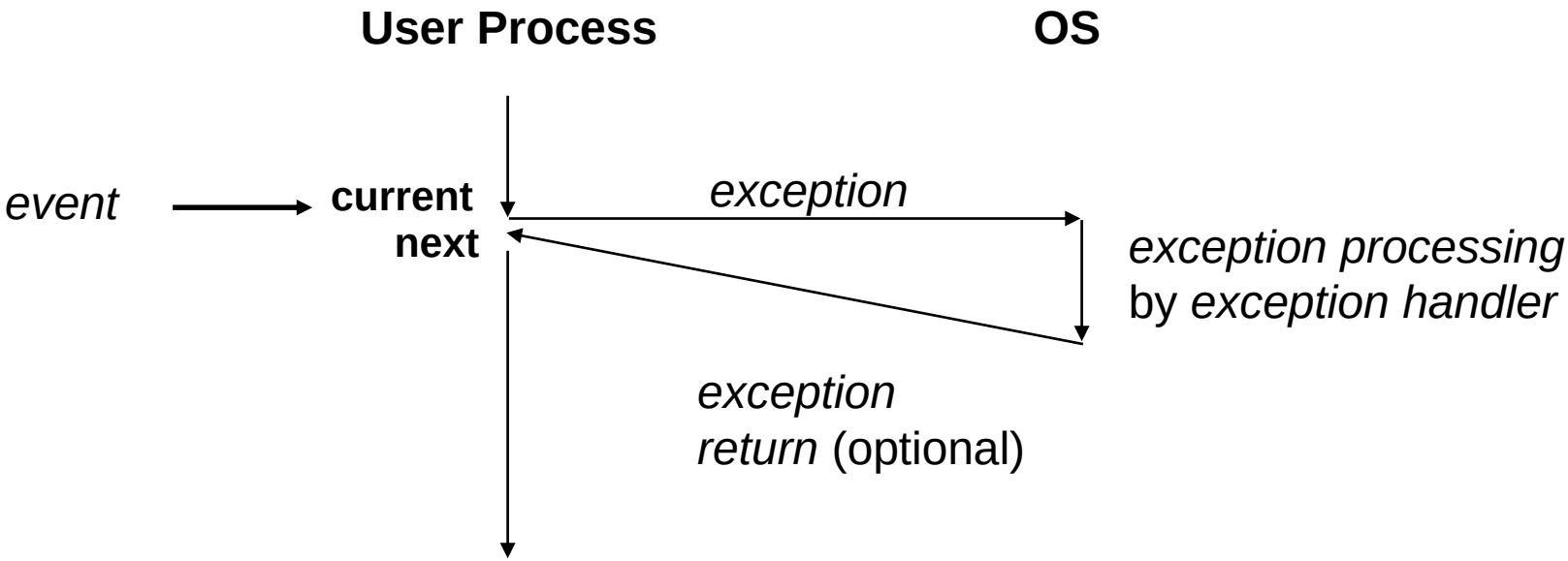


What alters the Control Flow?

- Program-assisted mechanisms for changing control flow:
  - Jumps and branches—react to changes in program state
  - Call and return using stack discipline—react to program state
- Insufficient for a useful system
  - Difficult for the CPU to react to other changes in system state
    - Data arrives from a disk or a network adapter
    - Instruction divides by zero
    - User hits control-C at the keyboard
- System needs mechanisms for "exception control flow"

Exception Control Flow

- An *exception* is a transfer of control to the OS in response to some *event* (i.e., change in processor state)



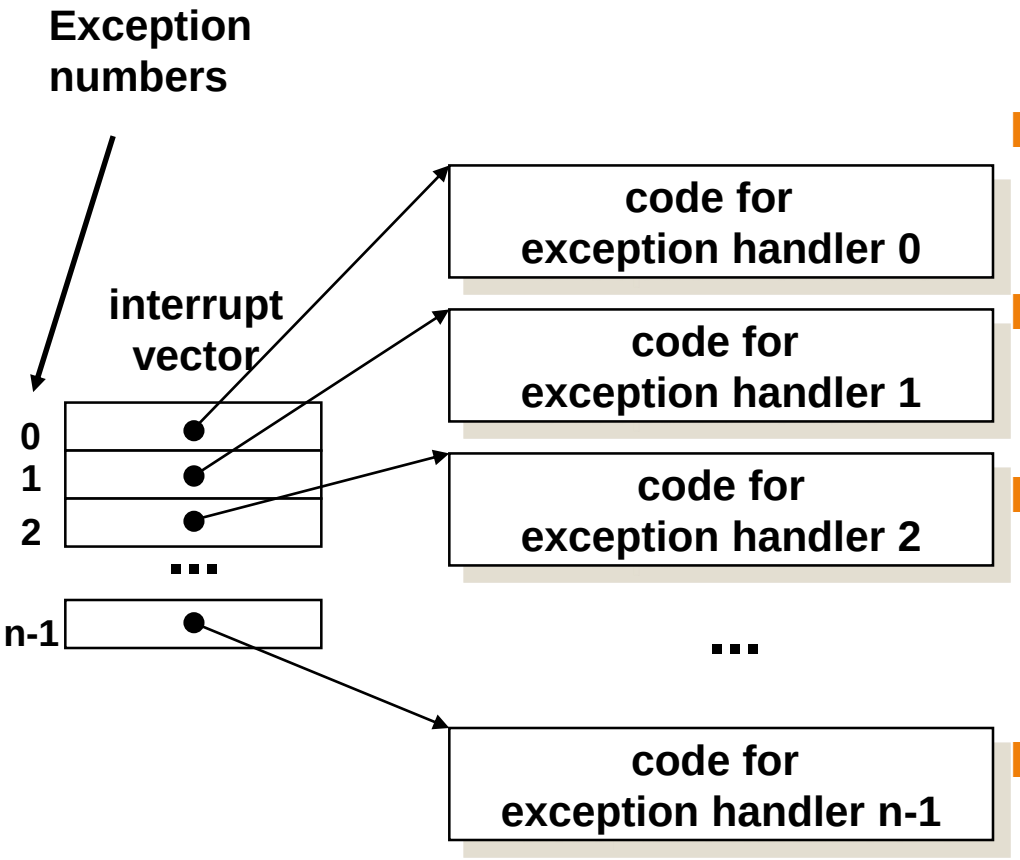
Types of Exceptions

- Synchronous (i.e. aligned to an event or time)
- Asynchronous (can happen without notice)

Asynchronous Exceptions (Interrupts)

- Caused by events external to processor
  - Indicated by setting the processor's interrupt pin(s)
  - Handler returns to "next" instruction.
- **Examples:**
  - I/O interrupts
    - Key pressed on the keyboard
    - Arrival of packet from network
  - Hard-reset interrupt
    - Hitting reset button
  - Soft-reset interrupt
    - Hitting control-alt-delete to initiate restart on a PC

Interrupt Vectors



- Each type of event has a unique exception number *k*
- Index into jump table (a.k.a., interrupt vector)
- Jump table entry *k* points to a function (exception handler).
- Handler *k* is called each time exception *k* occurs.

Synchronous Exceptions (Traps, Faults, Aborts)

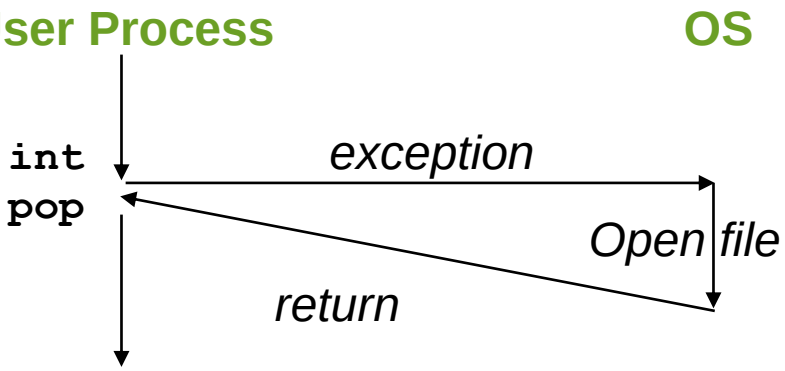
- Caused by events that occur as result of executing an instruction:
  - **Traps**
    - Intentional
    - Examples: system calls, breakpoint traps, special instructions
    - Returns control to “next” instruction
  - **Faults**
    - Unintentional but possibly recoverable
    - Examples: Page Faults
    - Either re-executes faulting (“current”) instruction or aborts

Synchronous Exceptions (Traps, Faults, Aborts)

- Caused by events that occur as result of executing an instruction:
  - **Aborts**
    - Unintentional and unrecoverable
    - Examples: parity error, machine check
    - Aborts current program or entire OS

Trap Example

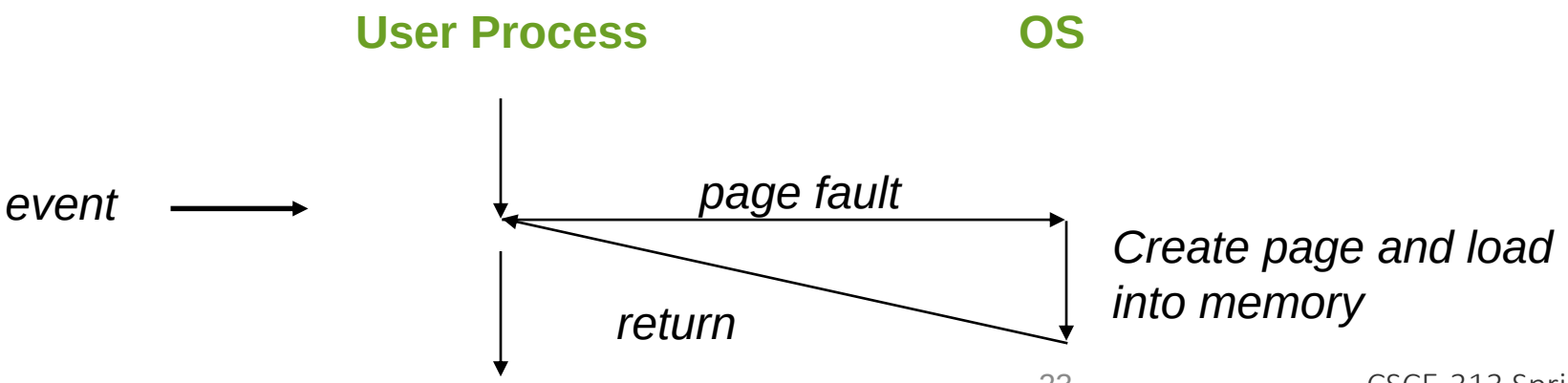
- **Opening a File**
  - User calls `open(filename, options)`
    - Function `open` executes system-call instruction: `int $0x80`
  - OS must find or create file, get it ready for reading or writing
  - Returns integer file descriptor



Fault Example #1

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

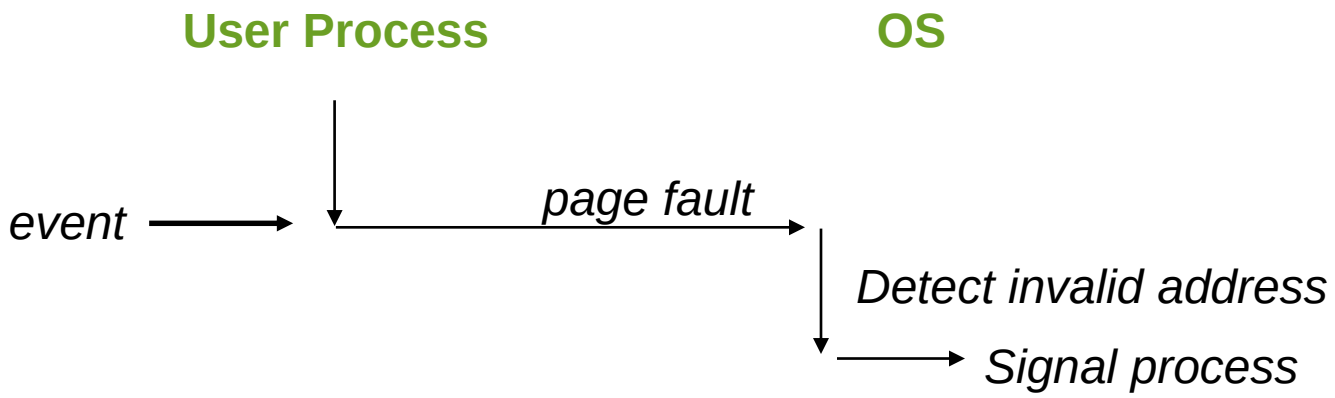
- Memory Reference
  - User writes to memory location
  - That portion (page) of user's memory is currently on disk
  - Page handler must load page into physical memory
  - Returns to faulting instruction
  - Successful on second try



Fault Example #2

- **Illegal Memory Reference**
  - User writes to memory location
  - Address is not valid
  - Page handler detects invalid address
  - Sends `SIGSEGV` signal to user process
  - User process exits with "segmentation fault"

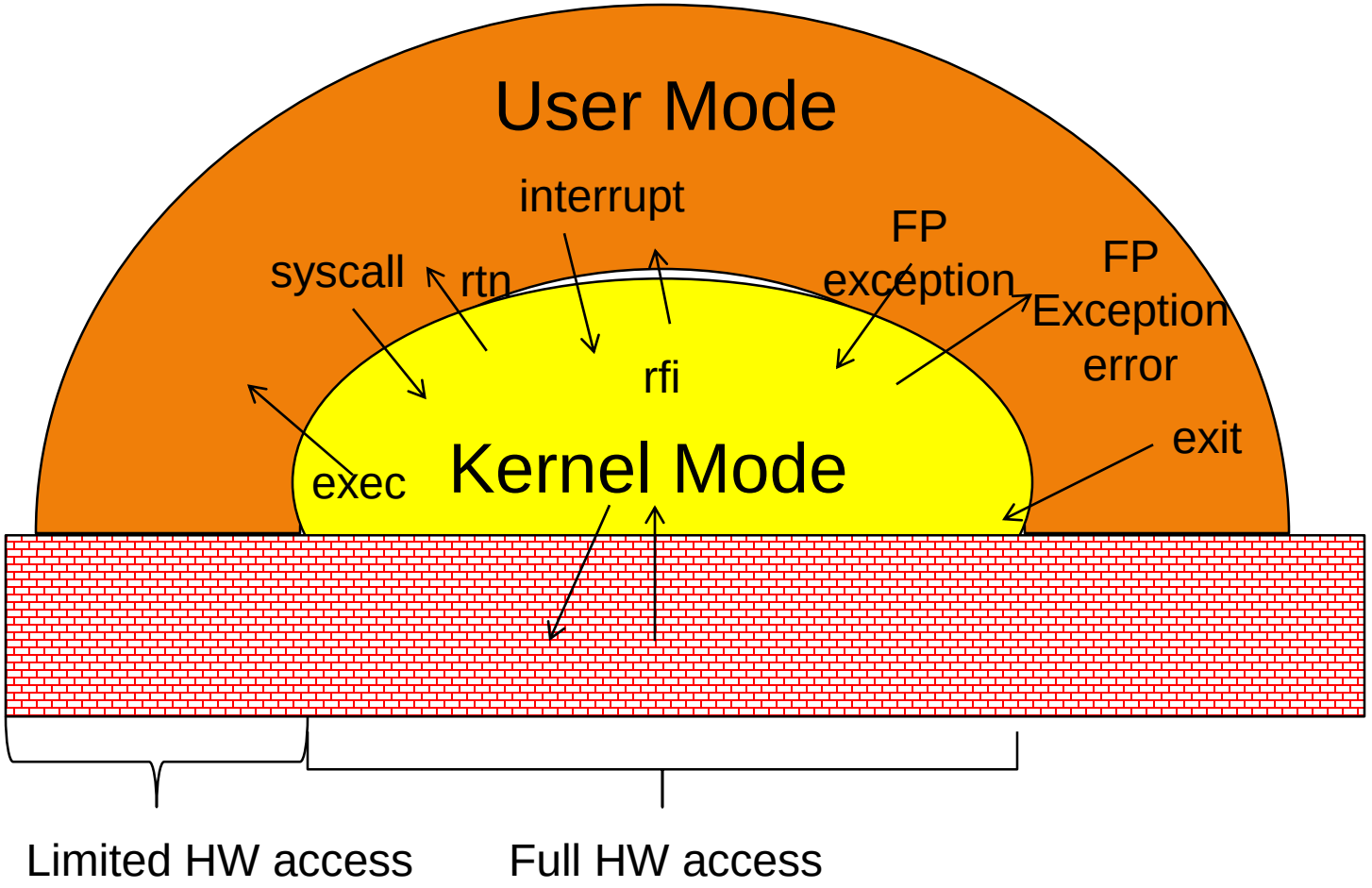
```
int a[1000];
main ()
{
    a[5000] = 13;
}
```



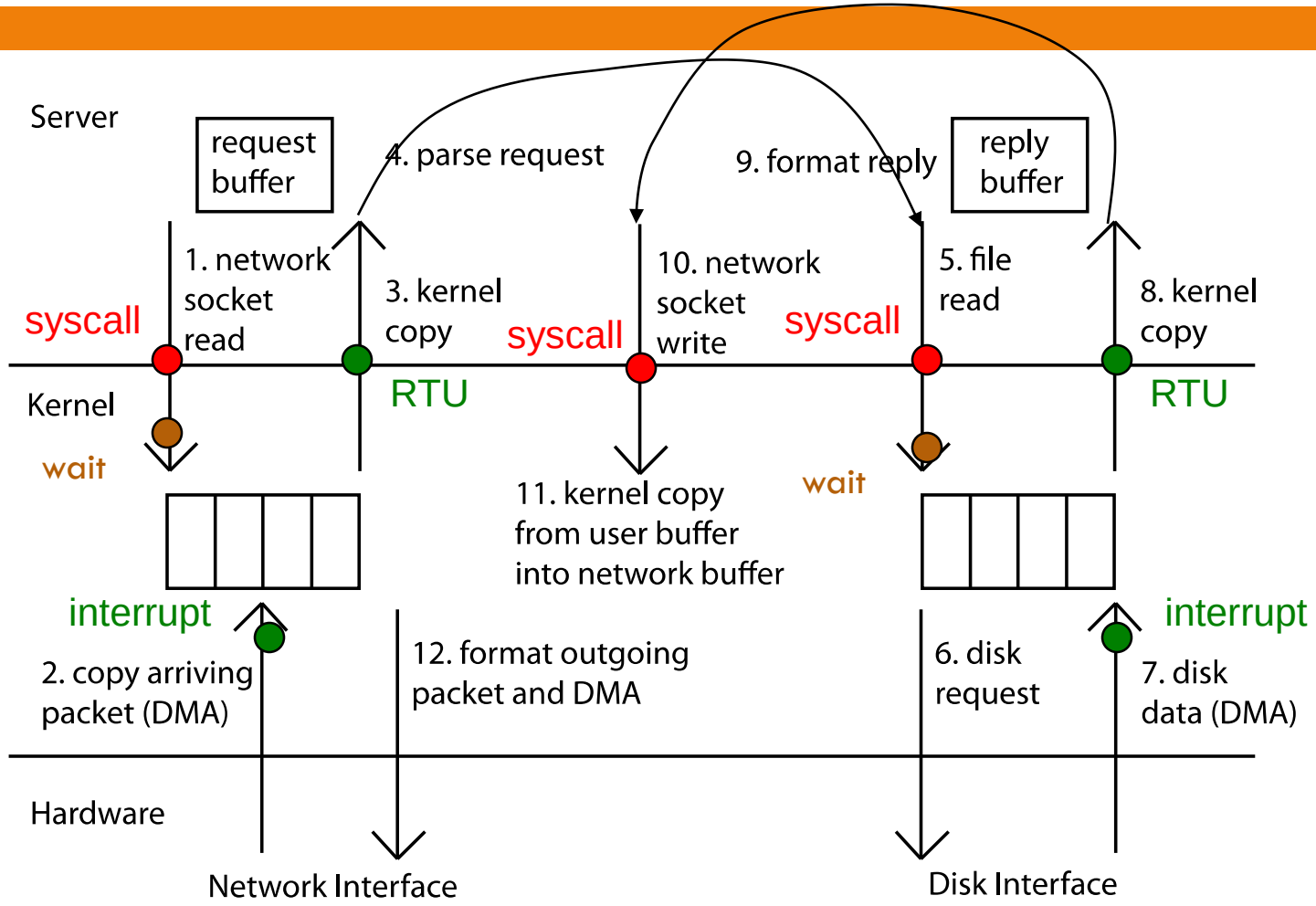
Summarizing Control Flow Exceptions

- Events that require nonstandard control flow
- Are Synchronous (Traps, Faults, Aborts) OR Asynchronous (I/O Interrupts, Hard or Soft Reset etc.)
- Generated Externally (interrupts) or Internally (traps and faults)
- OS decides how to handle

Preview: User/Kernel (Privileged) Mode



Example: Web Server

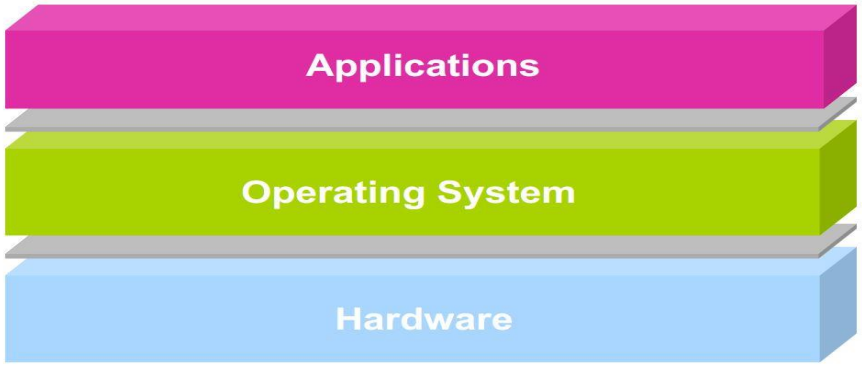


Content for this week

- OS **roles** and its key **challenges** (Text: Chap. 1)
- **Control Flow** in a modern computer system (Text: Chap. 2)
  - Normal flow of commands and data versus anything that happens "out of the ordinary" .. how do we handle that?
- **Architectural Interface** to the OS (Text: Chap. 2)
  - features we design in HW to facilitate the OS to meet some key challenges

Architectural Support for OS

- Operating systems mediate between applications and the physical hardware of the computer
  - Key goals of an OS are to enforce **protection** and **resource sharing**
    - If done well, applications can be oblivious to HW details



Challenge: Protection

- Why do we execute code with restricted privileges?
  - Either because the code is buggy or if it might be malicious
- Some examples:
  - A script running in a web browser
  - A program you just downloaded off the Internet
  - A program you just wrote that you haven't tested yet

Challenge: Resource Sharing

- How do we ensure that resources are fairly (and efficiently) shared amongst (and utilized by) user programs?
- Some examples:
  - Many students running code on a department machine
  - Amazon.com servicing concurrent users
  - Playing a movie on a computer while typing a project report and printing a document

Architectural Features

i.e. features we design in HW to facilitate the OS to meet some key challenges

- Privileged instructions
- Protection modes (user/kernel)
- Memory protection mechanisms
- Interrupts and exceptions
- System calls
- Timer (clock)
- I/O control and operation
- Synchronization primitives (e.g., atomic instructions)

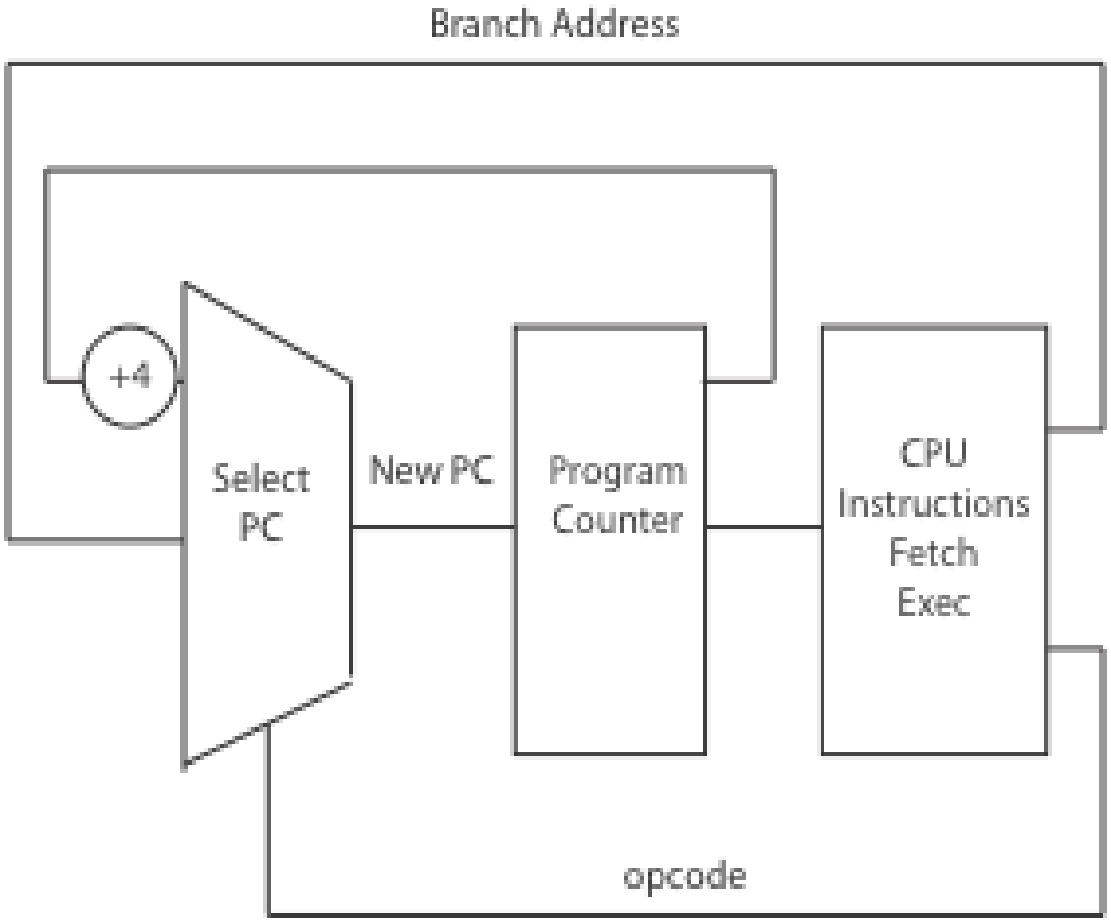
Main Points

- Dual-mode operation: user vs. kernel
  - Kernel-mode: execute with complete privileges
  - User-mode: execute with fewer privileges
- Safe control transfer
  - How do we switch from one mode to the other?

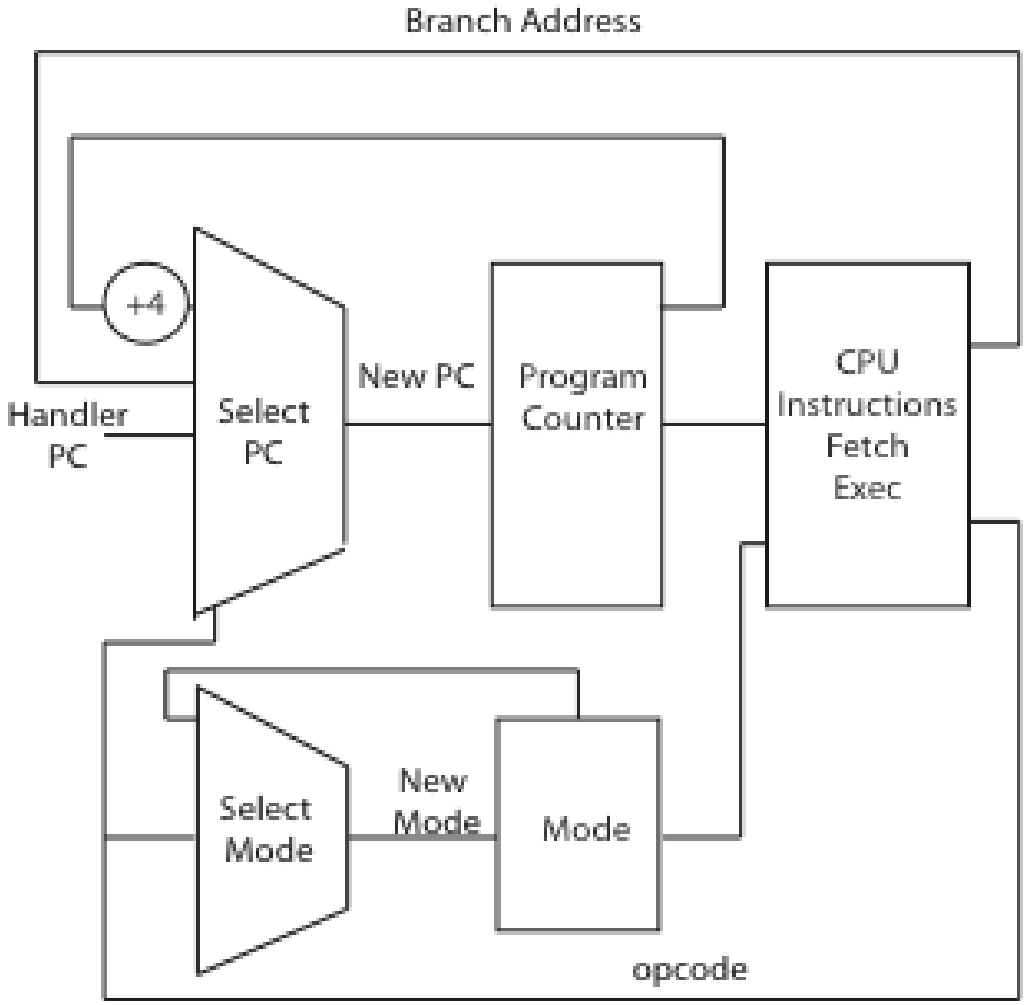
Hardware Support: Dual-Mode Operation

- Kernel mode
  - Execution with the full privileges of the hardware
    - E.g. Read/write to any memory, access any I/O device, read/write any disk sector, send/receive any packet
- User mode
  - Limited privileges
  - Only those granted by the operating system kernel
- On the x86, mode stored in EFLAGS register

A Model of a CPU



A CPU with Dual-Mode Operation



Hardware Support: Dual-Mode Operation

- Privileged instructions
  - Available to kernel
  - Not available to user code
- Limits on memory accesses
  - To prevent user code from overwriting the kernel or each other
- Timer
  - To regain control from a user program in a loop

Privileged Instructions

- A select few CPU instructions available only to the OS
  - Allows access to protected state
  - Perform global operations

Privileged Instructions - Examples

- Only the OS should be able to
  - Directly access I/O devices (disks, printers..)
    - Allows OS to enforce security and fairness
  - Manipulate memory management state
    - E.g., page tables, protection bits, TLB entries, etc.
  - Adjust protected control registers
    - User ← → Kernel modes or Raise/Lower interrupt level
  - Execute the halt instruction

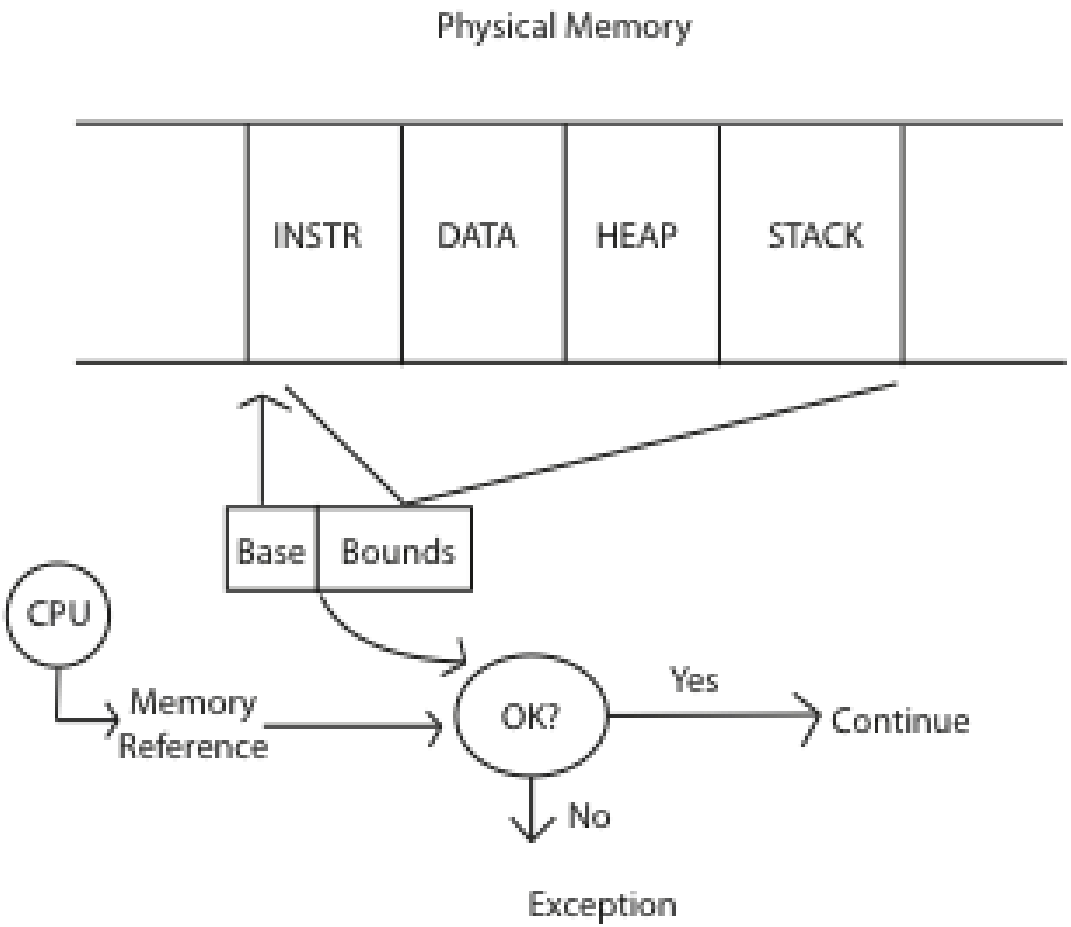
Question

- What should happen if a user program attempts to execute a privileged instruction?

Memory Protection

- Memory management hardware provides protection. Examples:
  - Base and limit registers
  - Page table pointers, Page Protection, Translation Lookaside Buffer (TLB)
- Manipulating memory management hardware uses protected (privileged) instructions

Memory Protection - Example



Hardware Timer

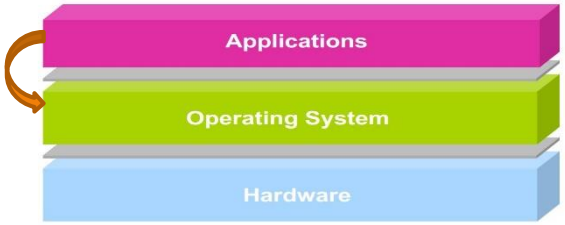
- Operating system timer is a critical building block
  - Many resources are time-shared; e.g., CPU
  - Allows OS to prevent infinite loops
- Fallback mechanism by which OS regains control
  - When timer expires, generates an interrupt
  - Handled by kernel, which controls resumption context
    - Basis for OS scheduler; more later…
  - Setting (and clearing) a timer is a privileged instruction

Question

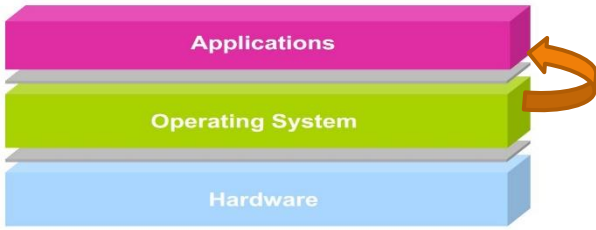
- For a “Hello world” program, the kernel must copy the string from the user program memory into the screen memory. Why must the screen’s buffer memory be protected?

User → Kernel Mode Switch

- From user-mode to kernel-mode
  - Interrupts
    - Triggered by timer and I/O devices
  - (Synchronous) Exceptions
    - Triggered by unexpected program behavior
    - Or malicious behavior!
  - System calls (traps) (aka protected procedure call)
    - Request by program for kernel to do some operation on its behalf
    - Only limited # of very carefully coded entry points



Kernel → User Mode Switch

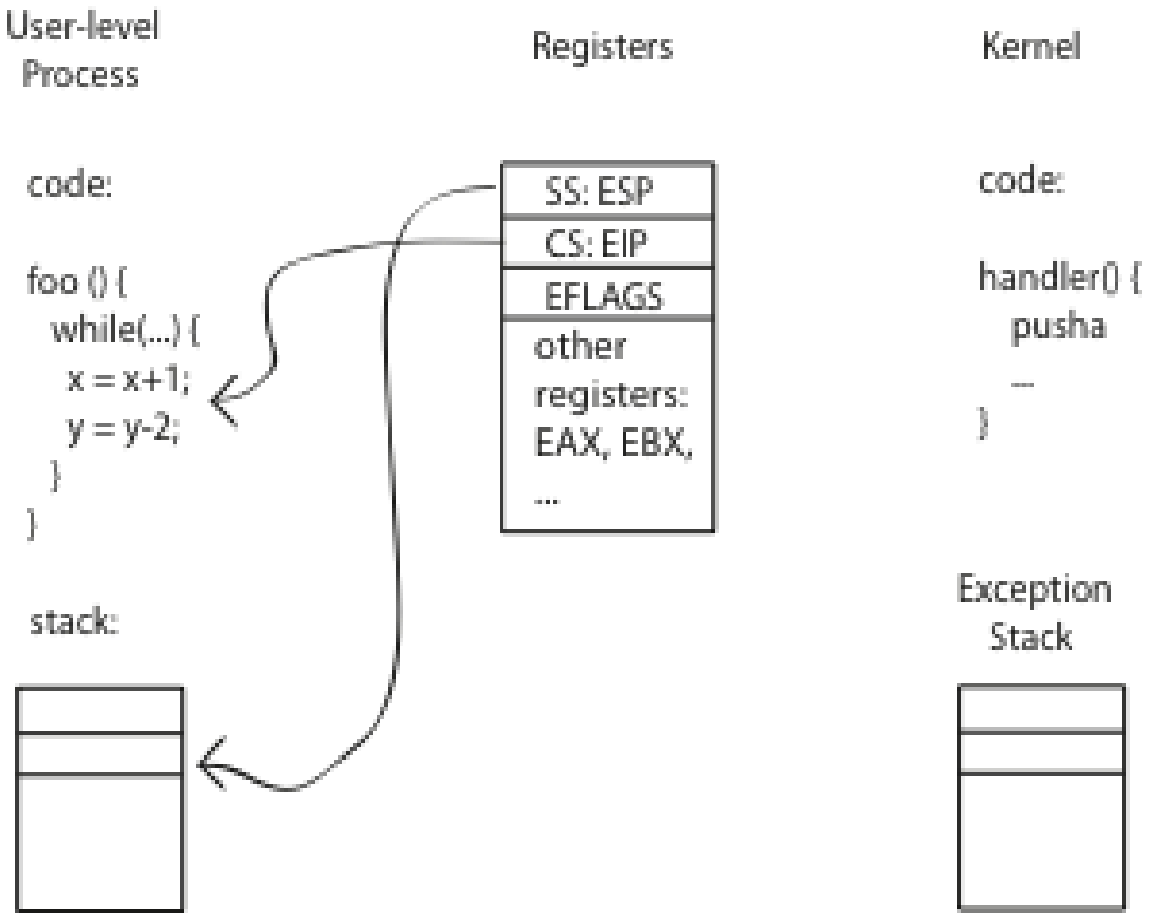


- From kernel-mode to user-mode
  - New process/new thread start
    - Jump to first instruction in program/thread
  - Return from interrupt, exception, system call
    - Resume suspended execution
  - Process/thread context switch
    - Resume some other process
  - User-level upcall
    - Asynchronous notification to user program by the kernel

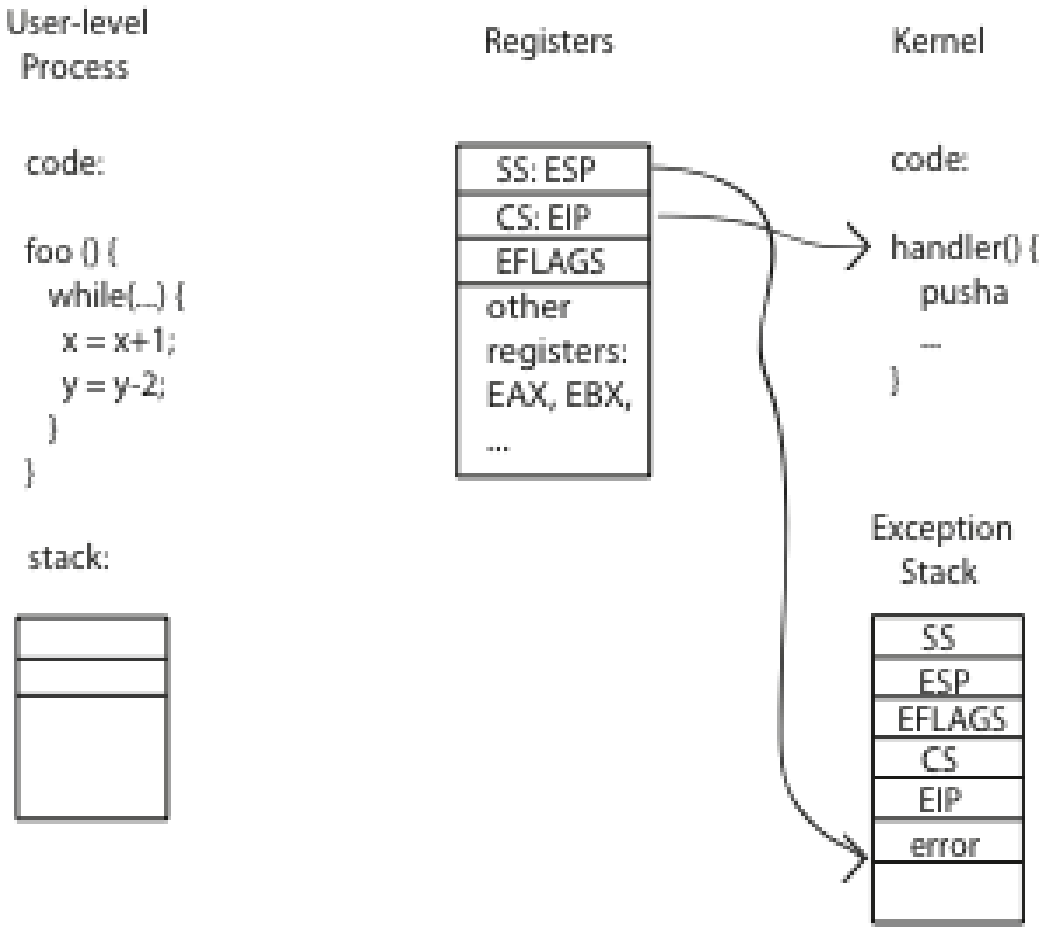
Transfer from User to Kernel Mode – Handling Interrupts

- On interrupt (x86)
  - Save current stack pointer
  - Save current program counter
  - Save current processor status word (condition codes)
  - Switch to kernel stack; put SP, PC, PSW on stack
  - Switch to kernel mode
  - Vector through interrupt table
  - Access the interrupt handler

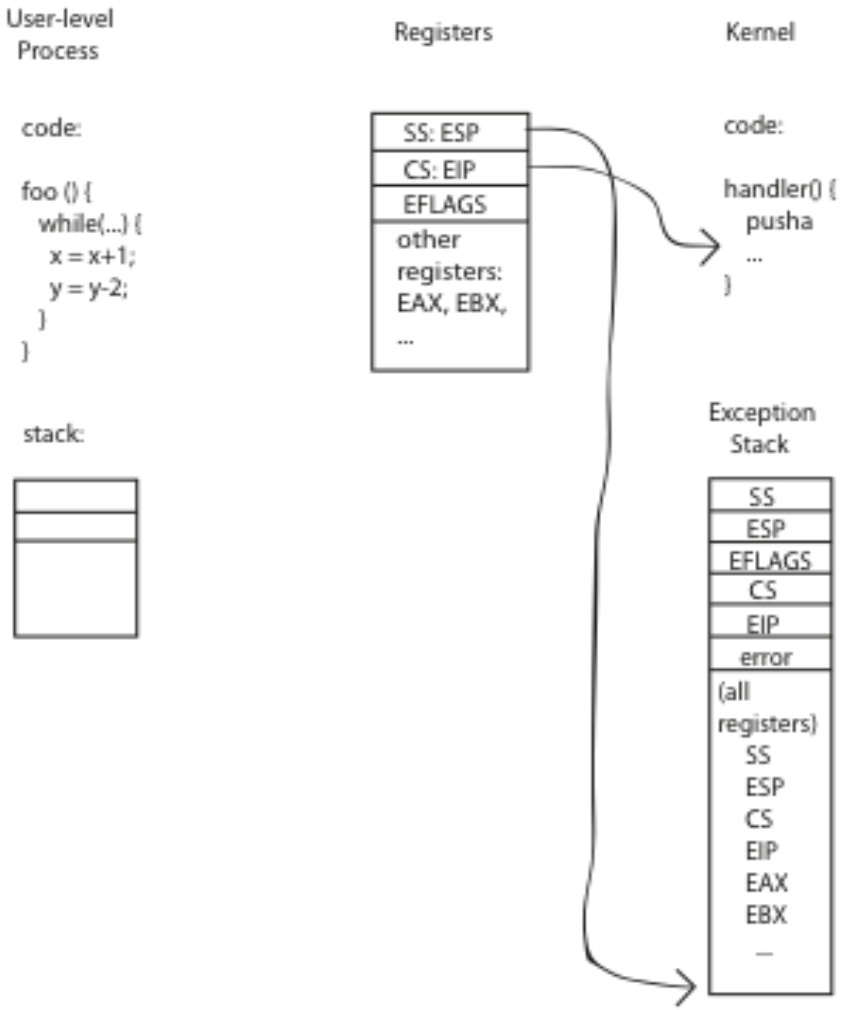
Before


User-level
Process
code:
foo () {
while(...) {
x = x+1;
y = y-2;
}
}
stack:
Registers
SS: ESP
CS: EIP
EFLAGS
other
registers:
EAX, EBX,
...
Kernel
code:
handler() {
pusha
...
}
Exception
Stack

During


User-level
Process
code:
foo () {
while(...) {
x = x+1;
y = y-2;
}
}
stack:
Registers
SS: ESP
CS: EIP
EFLAGS
other
registers:
EAX, EBX,
...
Kernel
code:
handler() {
pusha
...
}
Exception
Stack
SS
ESP
EFLAGS
CS
EIP
error

After


User-level Process
code:
foo () {
while(...) {
x = x+1;
y = y-2;
}
}
stack:
Registers
SS: ESP
CS: EIP
EFLAGS
other registers: EAX, EBX, ...
Kernel
code:
handler() {
pusha
...
}
Exception Stack
SS
ESP
EFLAGS
CS
EIP
error
(all registers)
SS
ESP
CS
EIP
EAX
EBX
...

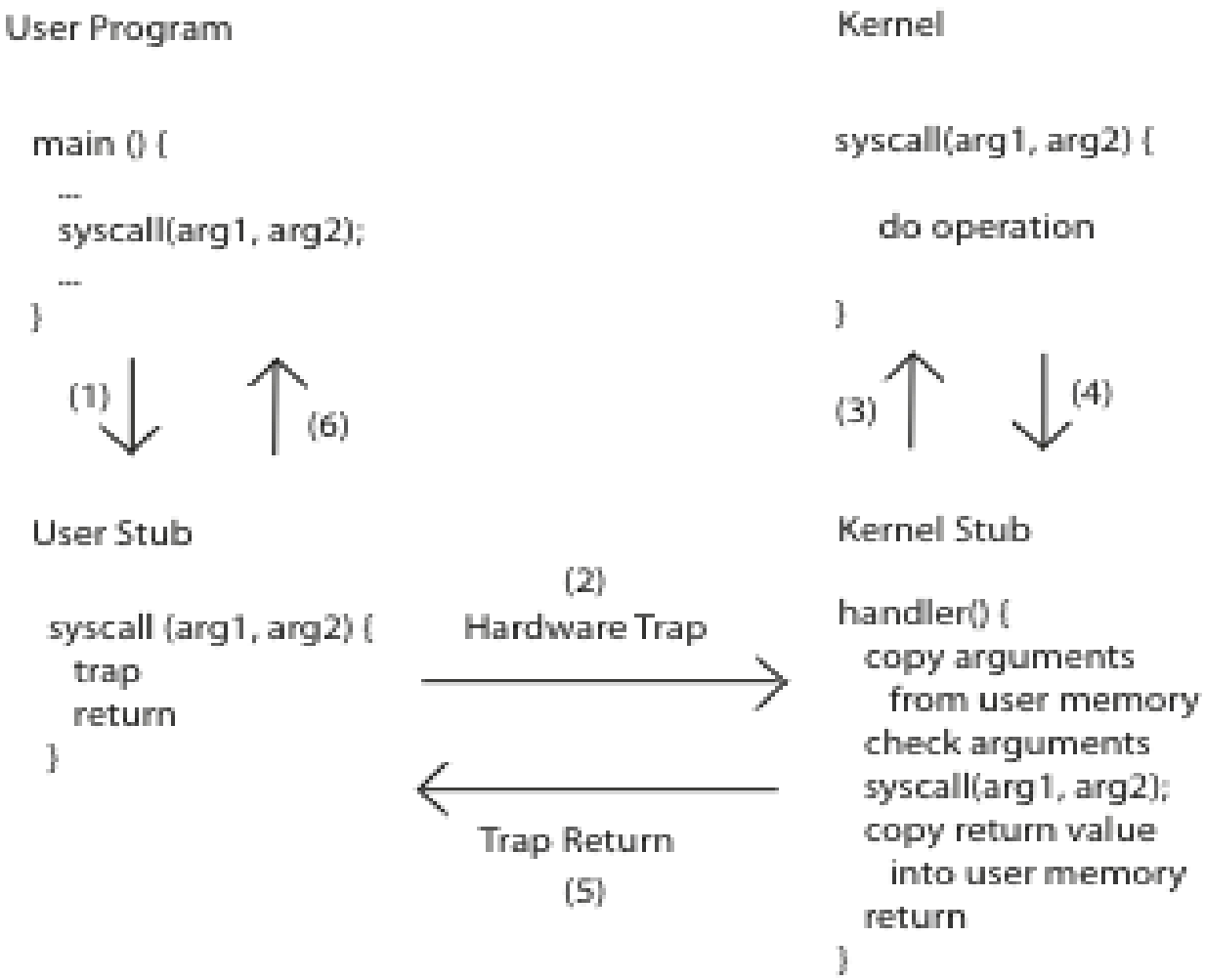
At the end of handler

- Handler restores saved registers
- Atomically return to interrupted process/thread
  - Restore program counter
  - Restore program stack
  - Restore processor status word/condition codes
  - Switch to user mode

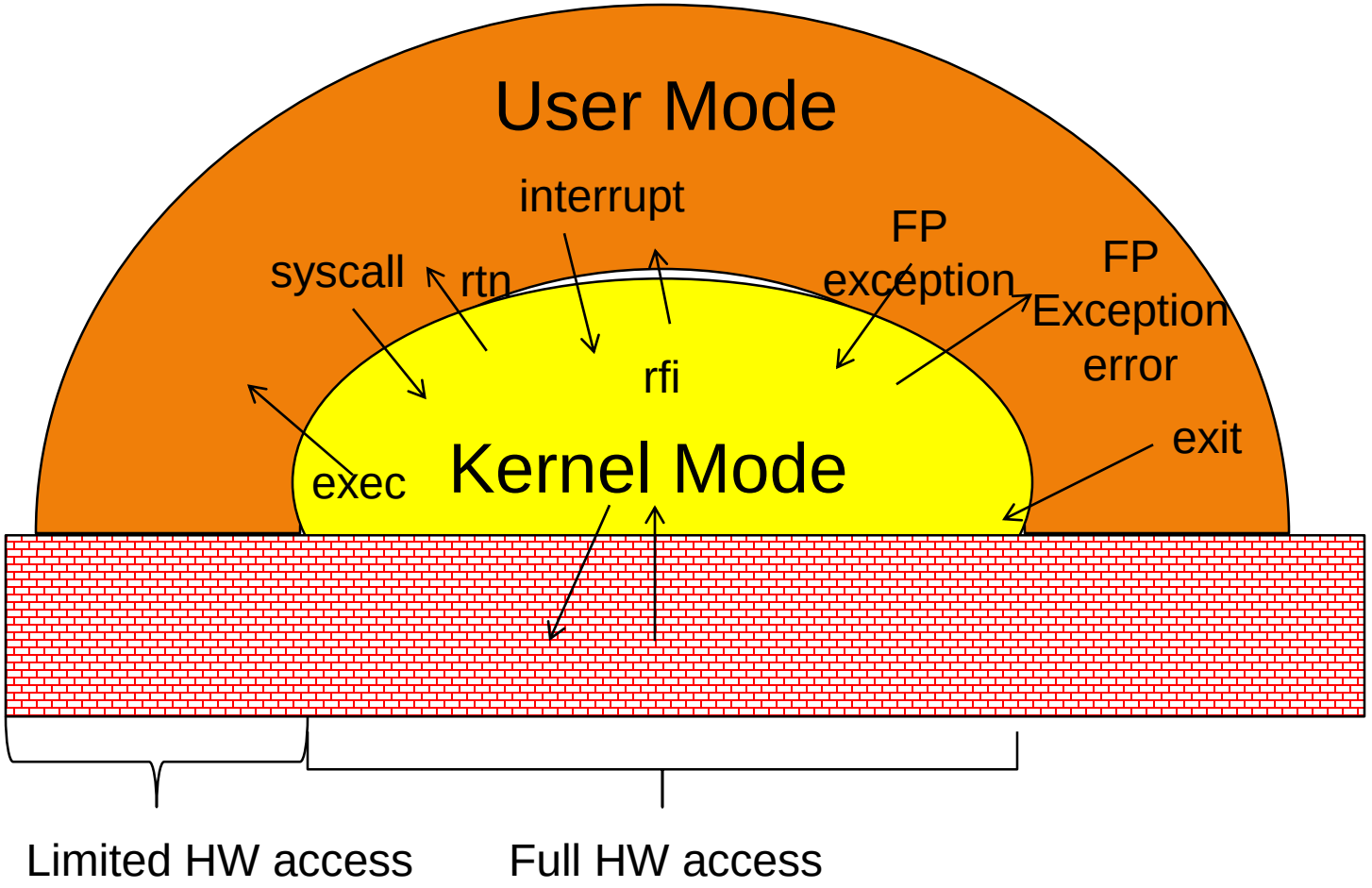
Kernel System Call Handler

- Locate arguments
  - In registers or on user stack
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - into user memory

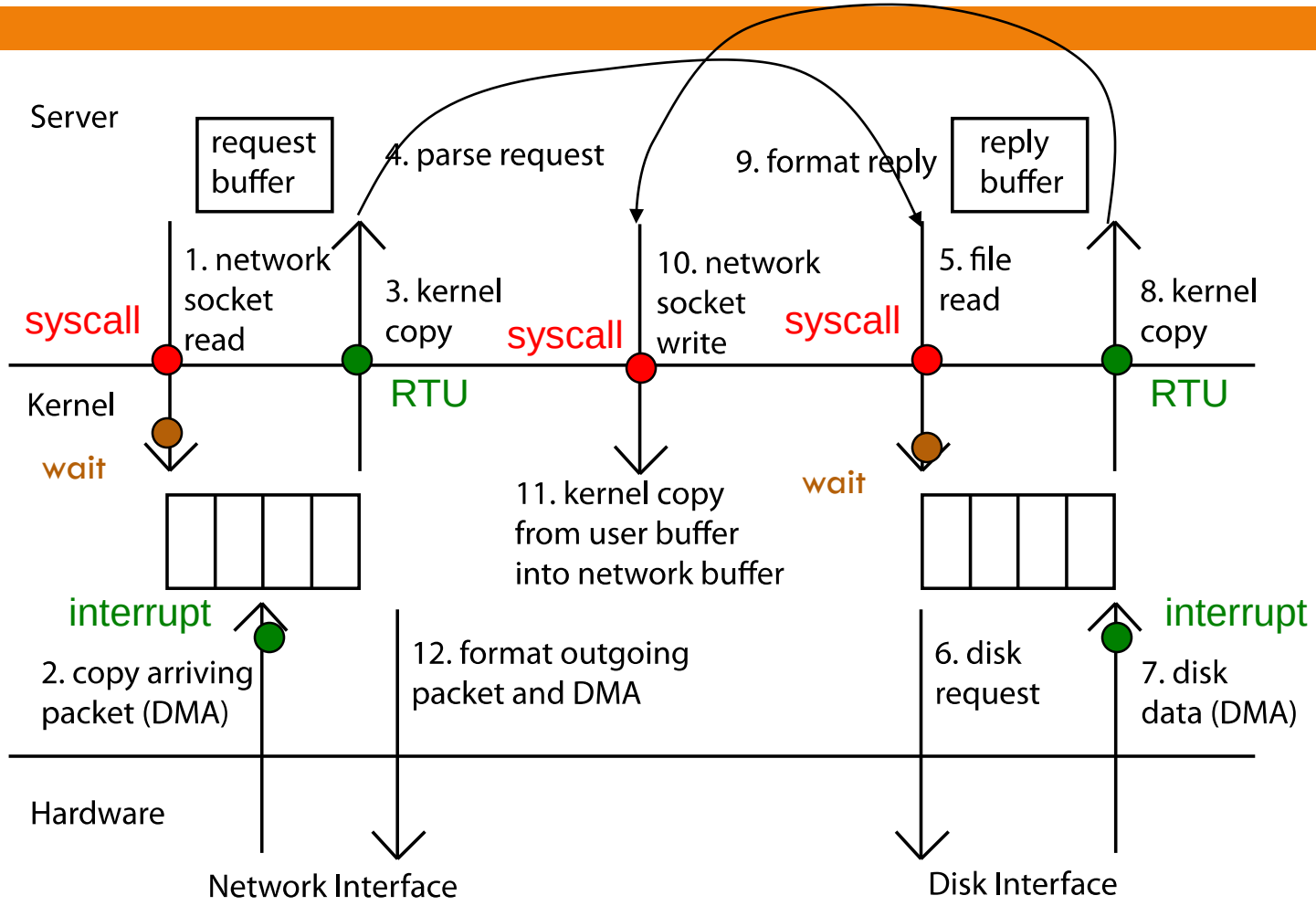
System Calls



Summary: User/Kernel (Privileged) Mode



Example: Web Server (Revisited)



Summary of Learnings

- OS **roles** and its key **challenges** (Text: Chap. 1)
- **Control Flow** in a modern computer system (Text: Chap. 2)
  - Normal flow of commands and data versus anything that happens "out of the ordinary" .. how do we handle that?
- **Architectural Interface** to the OS (Text: Chap. 2)
  - features we design in HW to facilitate the OS to meet some key challenges

A Real-Life Analogy (Approximate)

A Typical Coffee Shop	Computer System
Store	
Customer	
Barista/Cashier	
Coffee Machine	
Customer Order	
Order item not on Menu	
Telephone Call	
Fire Alarm	
>1 Customers being served	
Customer realizing at the counter that he needs to go to ATM to get money	

Next Week

- Process and Programming Interface