THE KERNEL ABSTRACTION

Slides adapted from Tom Anderson's "Operating Systems: Principles and Practice"

Challenge: Protection

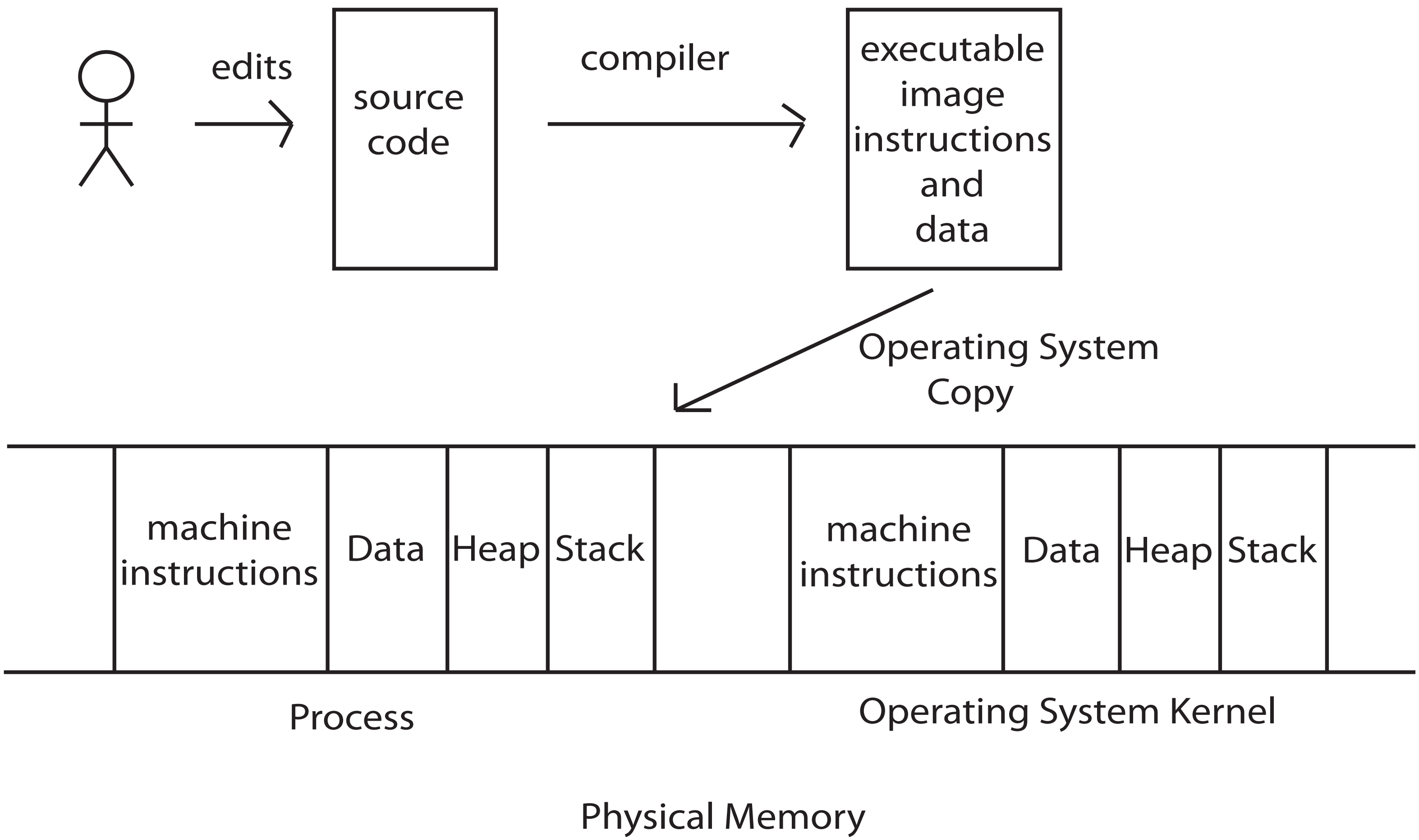
- How do we execute code with restricted privileges?
  - Either because the code is buggy or if it might be malicious

- 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

Main Points

- Process concept
  - A process is an OS abstraction for executing a program with limited privileges
- 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?

Process Concept



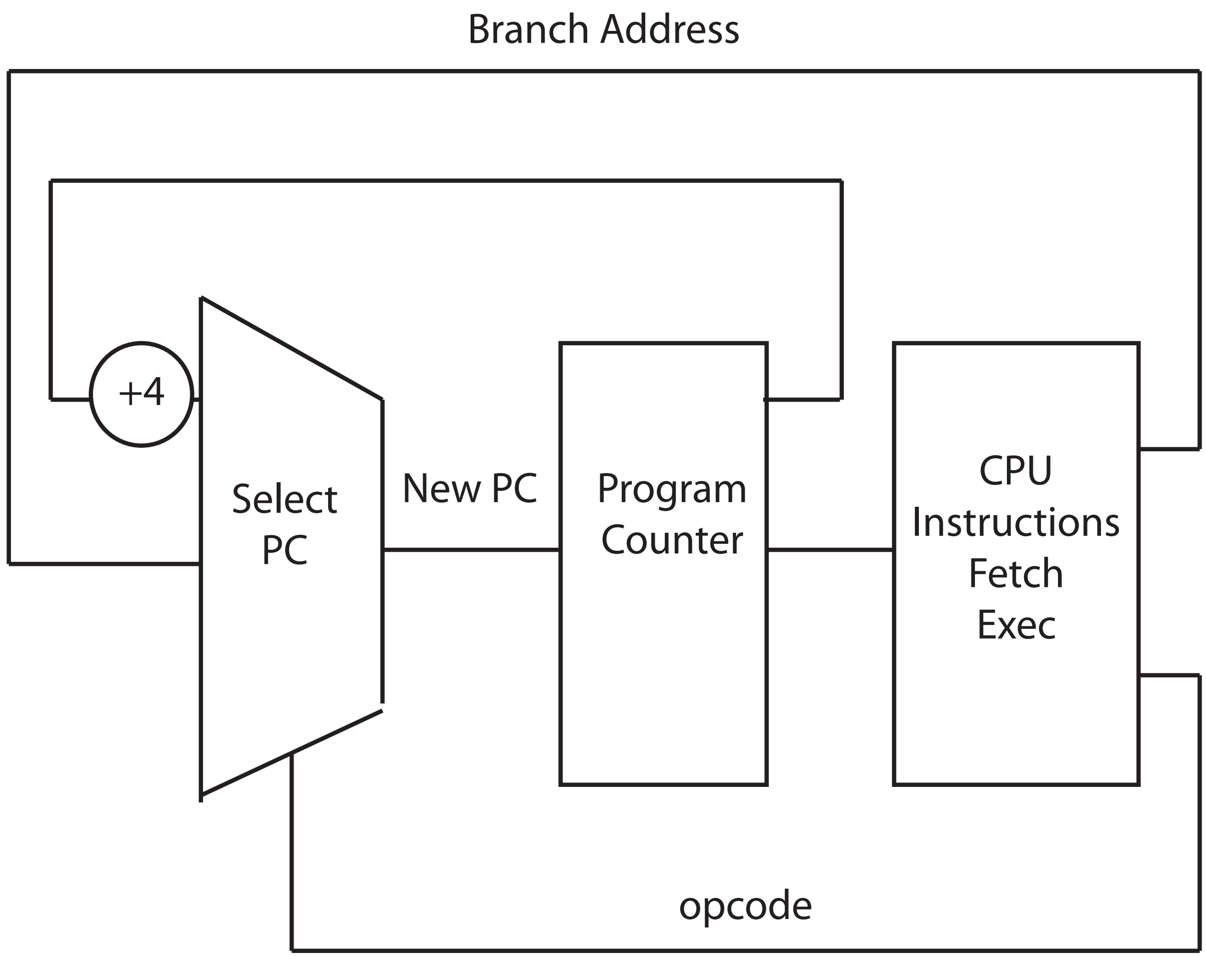
Process Concept

- Process: an instance of a program, running with limited rights
  - Process control block: the data structure the OS uses to keep track of a process
  - Two parts to a process:
    - Thread: a sequence of instructions within a process
      - Potentially many threads per process (for now 1:1)
      - Thread aka lightweight process
    - Address space: set of rights of a process
      - Memory that the process can access
      - Other permissions the process has (e.g., which procedure calls it can make, what files it can access)

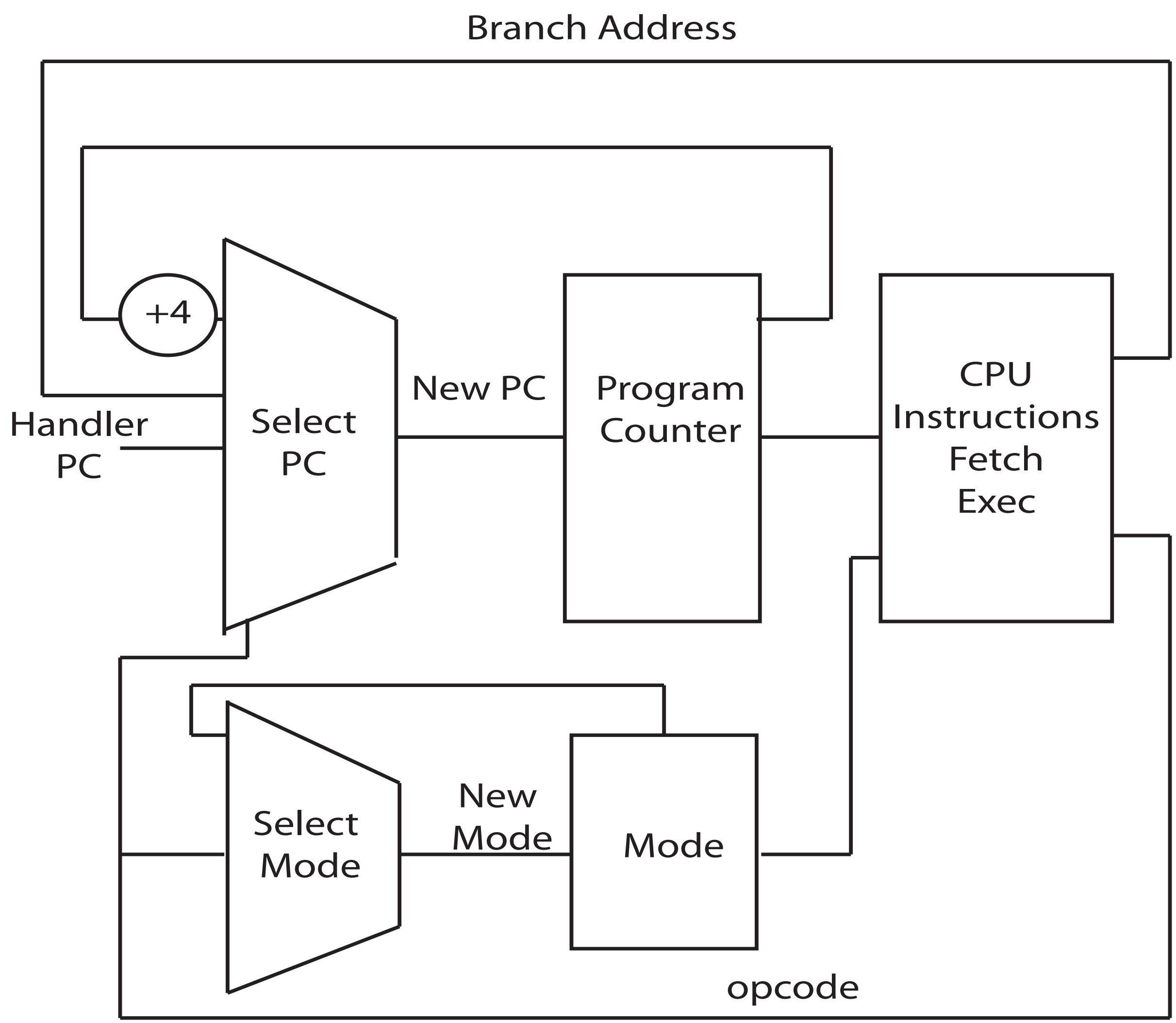
Hardware Support: Dual-Mode Operation

- Kernel mode
  - Execution with the full privileges of the hardware
  - Read/write to any memory, access any I/O device, read/write any disk sector, send/read 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



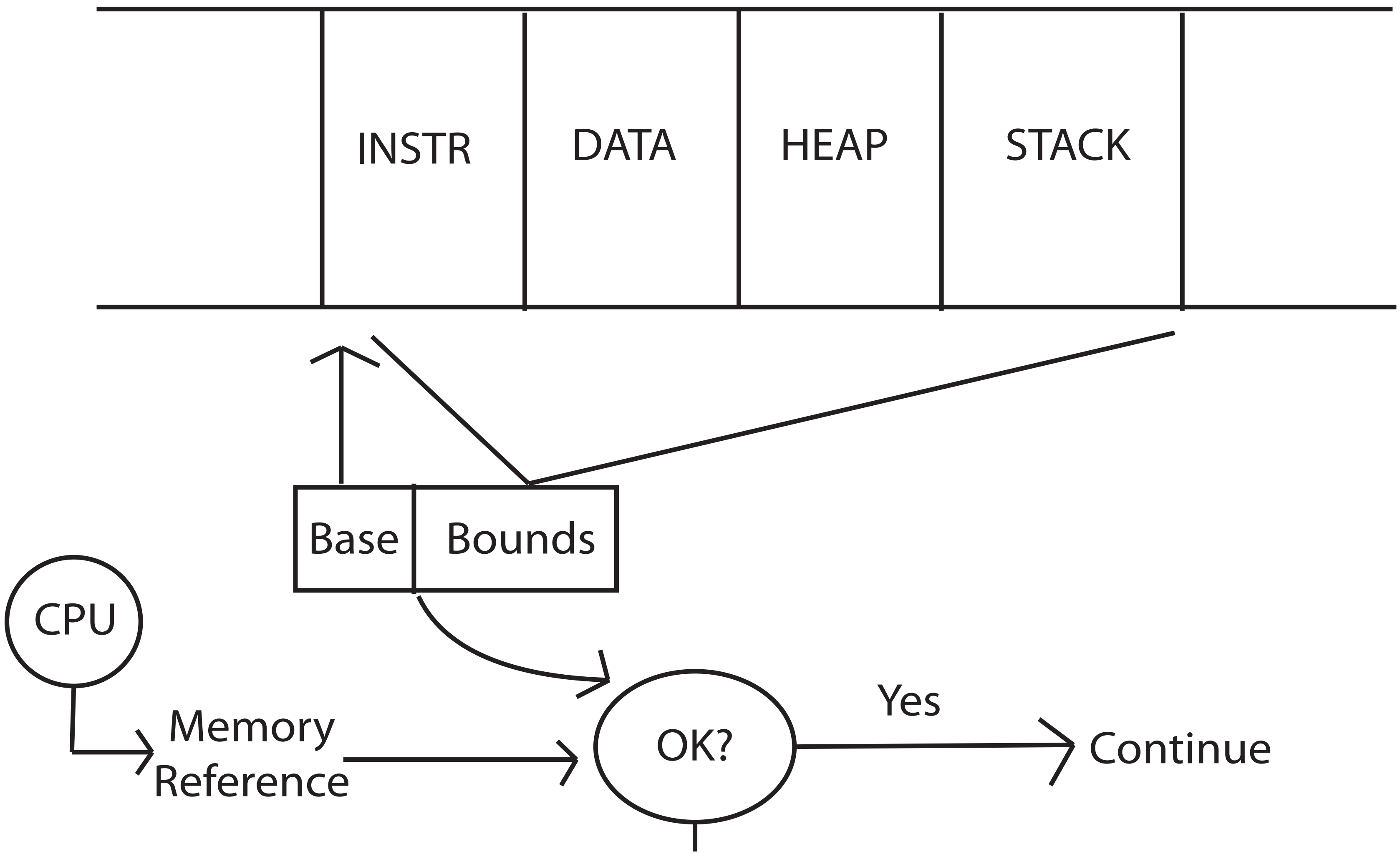
Hardware Support: Dual-Mode Operation

- Privileged instructions
  - Available to kernel
  - Not available to user code
- Limits on memory accesses
  - To prevent user code from overwriting the kernel
- Timer
  - To regain control from a user program in a loop
- Safe way to switch from user mode to kernel mode, and vice versa

Privileged instructions

- Examples?
  - Change the execution mode
  - Access memory positions it has no permission to
  - Input/Output operations
  - Jump into kernel code
  - Enable/disable interrupts
  - …
- What should happen if a user program attempts to execute a privileged instruction?
  - Processor exception

Memory Protection

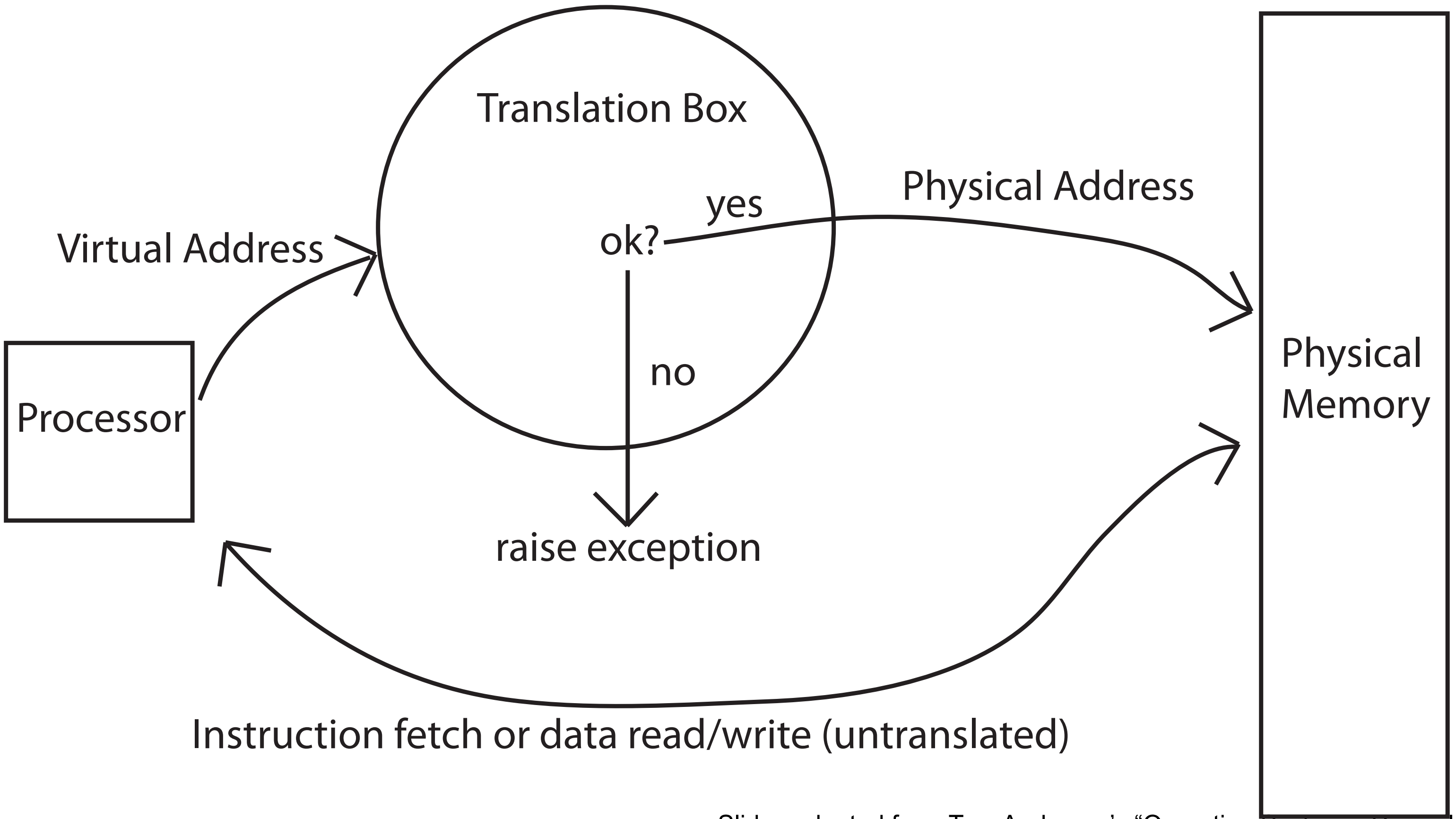


Towards Virtual Addresses

- Problems with base and bounds?
  - Expandable heap?
  - Expandable stack?
  - Memory sharing between processes?
  - Non-relative addresses – hard to move memory around
  - Memory fragmentation

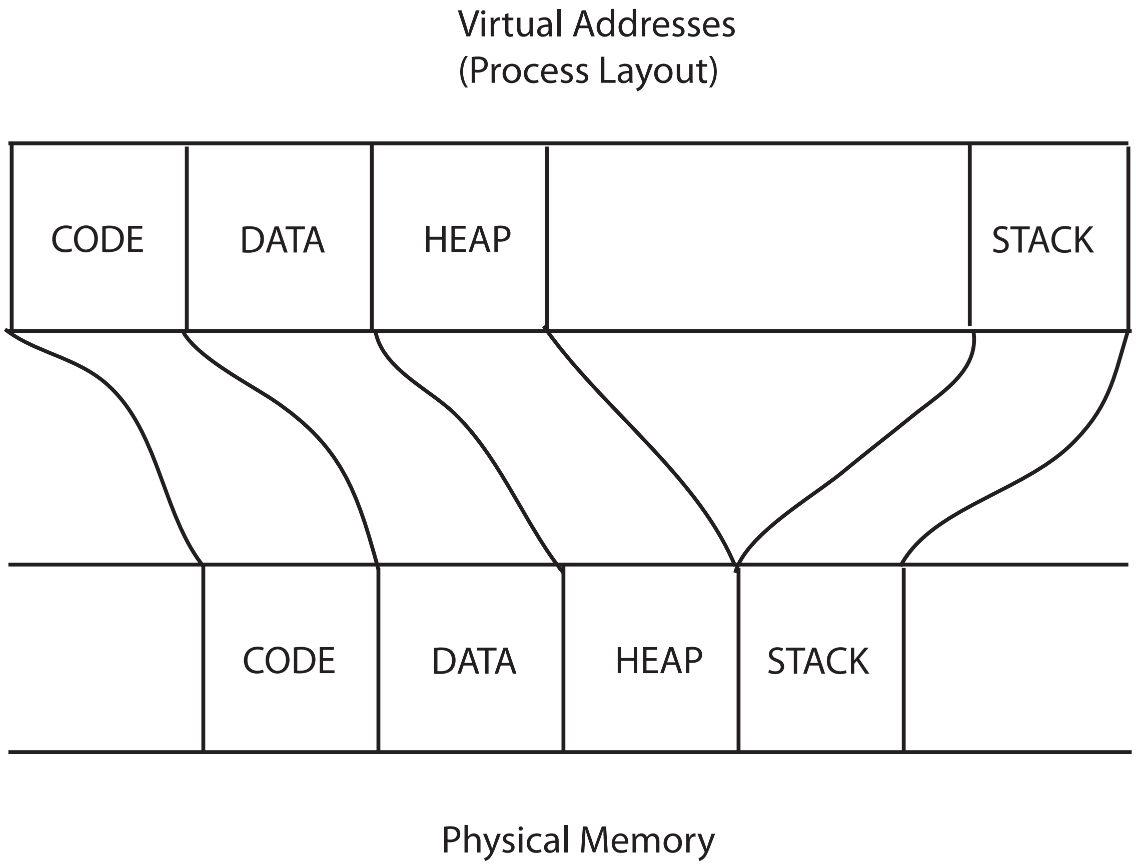
Virtual Addresses

- Translation done in hardware, using a table
- Table set up by operating system kernel



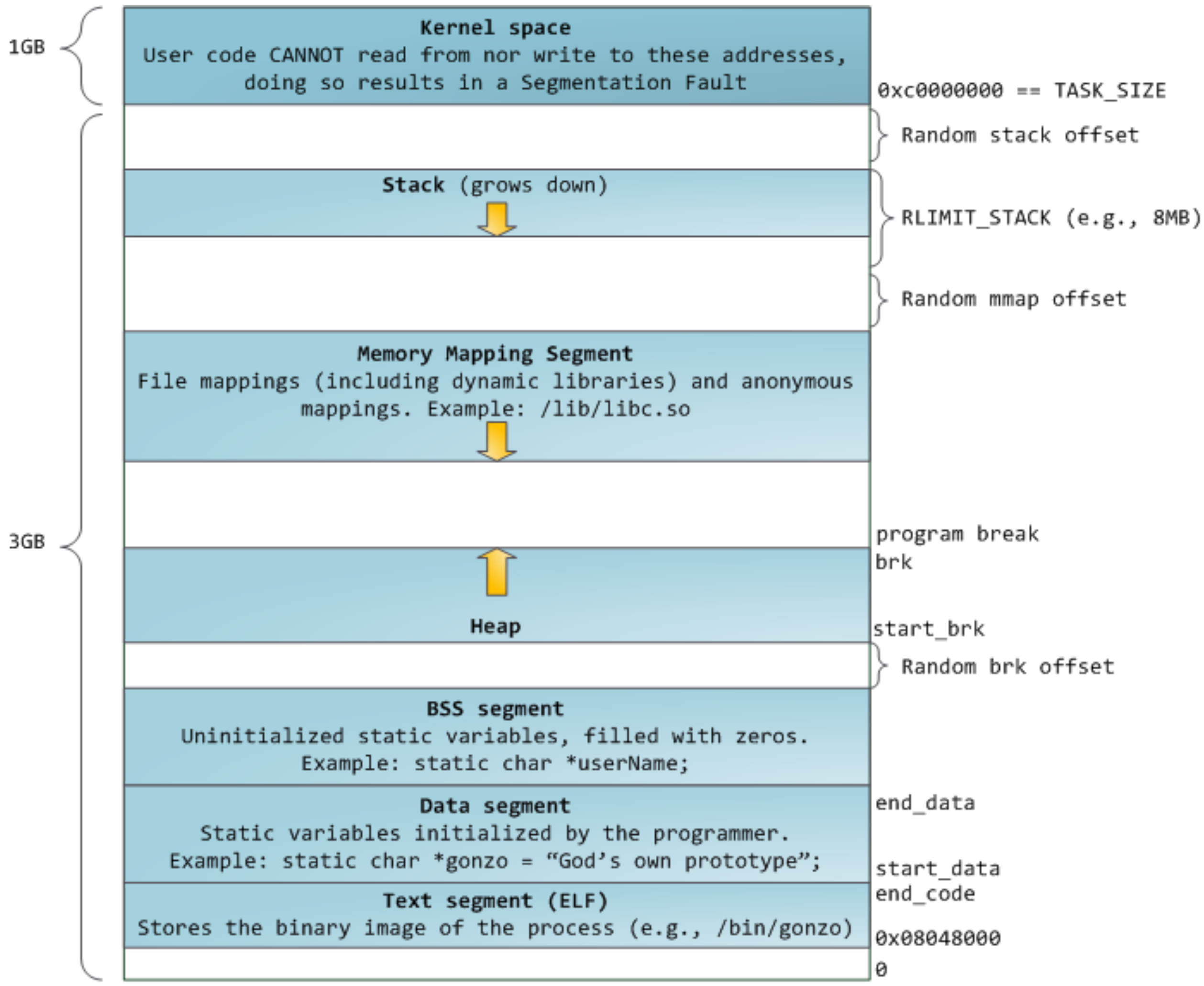
Virtual Address Layout

- Plus shared code segments, dynamically linked libraries, memory mapped files, …



Process Memory Map (Linux)

taken from http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/



Example: What Does this Do?

```
int staticVar = 0;      // a static variable
int main() {
    int localVar = 0;   // a procedure local variable

    staticVar += 1; localVar += 1;

    sleep(10);  // sleep causes the program to wait for x seconds
    printf ("static address: %x, value: %d\n", &staticVar, staticVar);
    printf ("procedure local address: %x, value: %d\n", &localVar,
  localVar);
}
```

Produces:

static address: 5328, value: 1
procedure local address: ffffffe2, value: 1

Hardware Timer

- Hardware device that periodically interrupts the processor
  - Returns control to the kernel timer interrupt handler

  - Interrupt frequency set by the kernel
    - Not by user code!

  - Interrupts can be temporarily deferred
    - Not by user code!
    - Crucial for implementing mutual exclusion

Mode Switch

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

Mode Switch

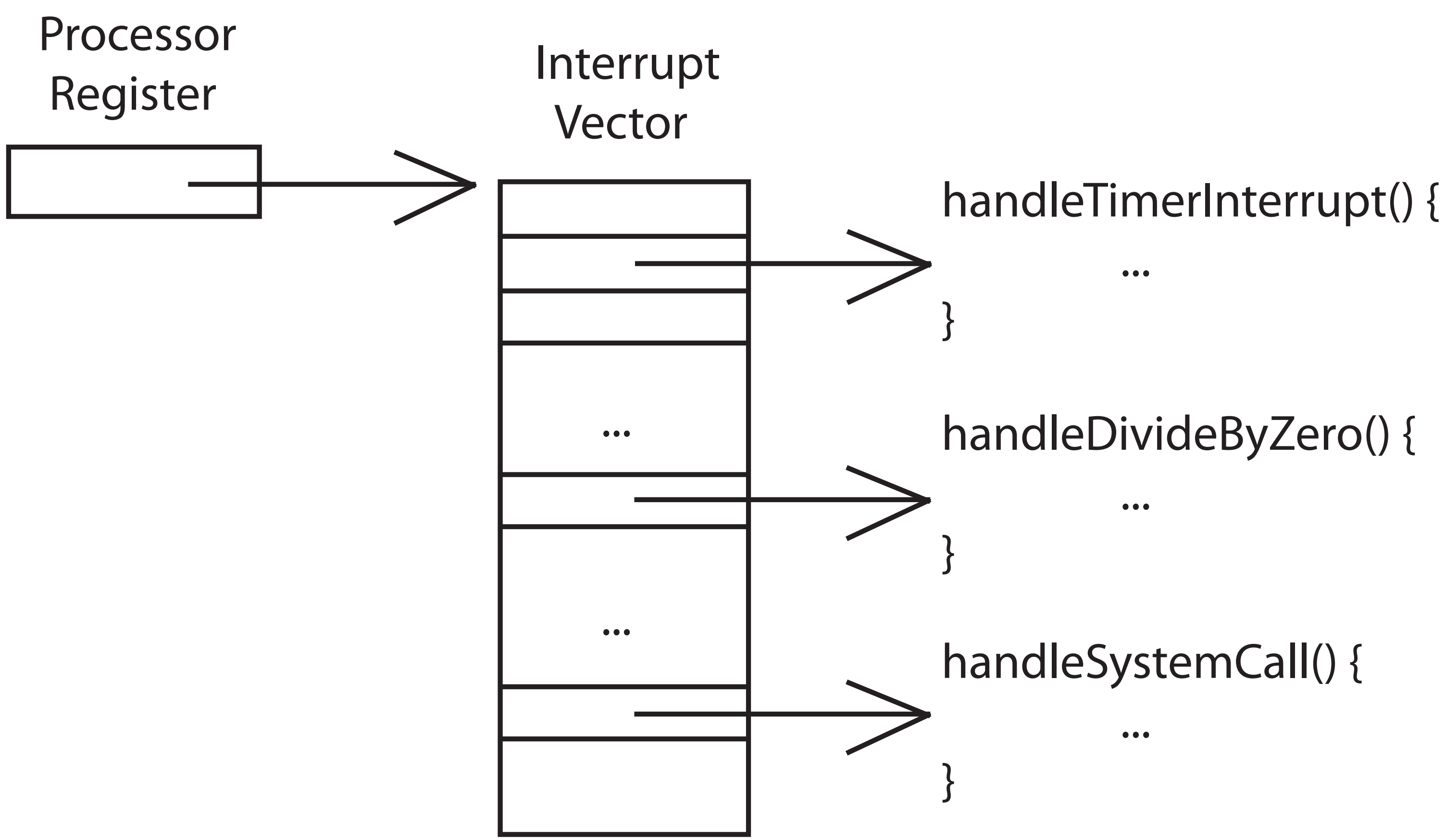
- From kernel-mode to user
  - New process/new thread start
    - Jump to first instruction in program/thread
  - Return from interrupt, exception, system call
    - Resume suspended execution
  - Process/thread context switch
    - Resume some other process
  - User-level upcall
    - Asynchronous notification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

How do we take interrupts safely?

- Limited number of entry points into kernel
  - Interrupt vector
- Handler works regardless of state of user code
  - Kernel interrupt stack
- Handler is non-blocking
  - Interrupt masking
- Atomic transfer of control
  - Single instruction to change:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode
- User program does not know interrupt occurred
  - Transparent restartable execution

Interrupt Vector

- Table set up by OS kernel; pointers to code to run on different events

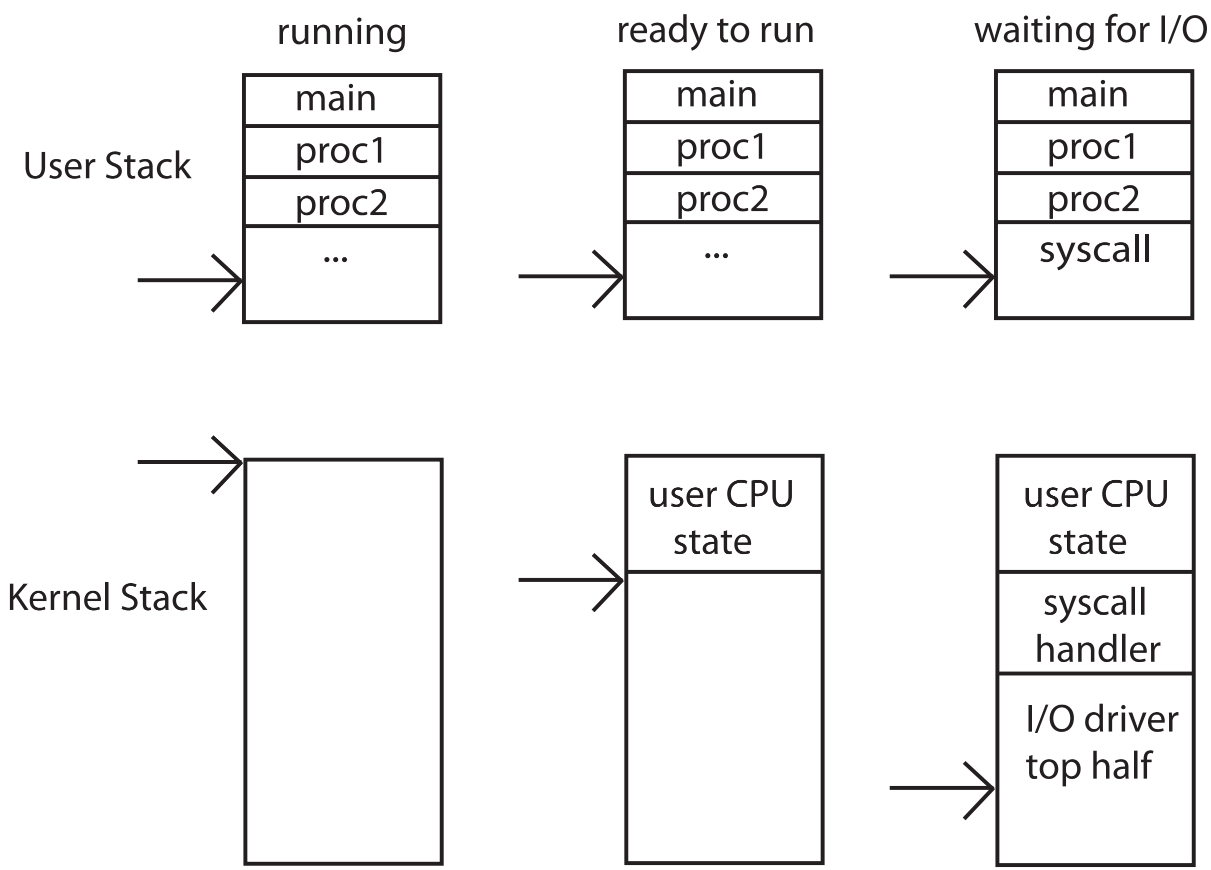


Interrupt Stack

- Per-processor, located in kernel (not user) memory
  - Usually a thread has both: kernel and user stack

- Why can't interrupt handler run on the stack of the interrupted user process?
  - Process' stack pointer may be corrupted
  - Prevent other threads to access/modify kernel internal information

Interrupt Stack



Interrupt Masking

- Interrupt handler runs with interrupts off
  - Reenabled when interrupt completes

- OS kernel can also turn interrupts off
  - Eg., when determining the next process/thread to run
  - If defer interrupts too long, can drop I/O events
  - On x86
    - CLI: disable interrrupts
    - STI: enable interrupts
    - Only applies to the current CPU

- Cf. implementing synchronization, chapter 5

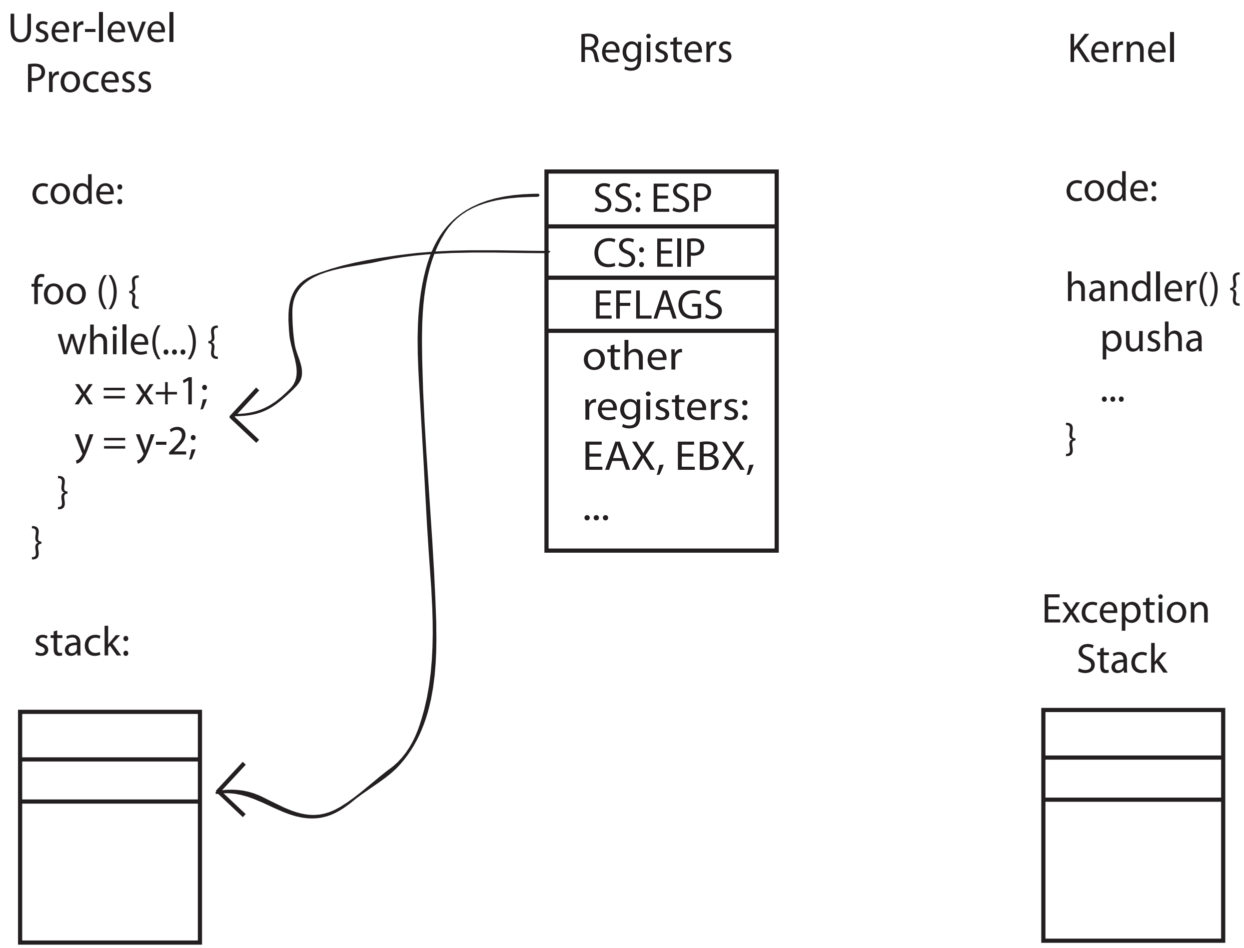
Interrupt Handlers

- Non-blocking, run to completion
  - Minimum necessary to allow device to take next interrupt
  - Any waiting must be limited duration
  - Wake up other threads to do any real work
- Rest of device driver runs as a kernel thread
  - Queues work for interrupt handler
  - (Sometimes) wait for interrupt to occur

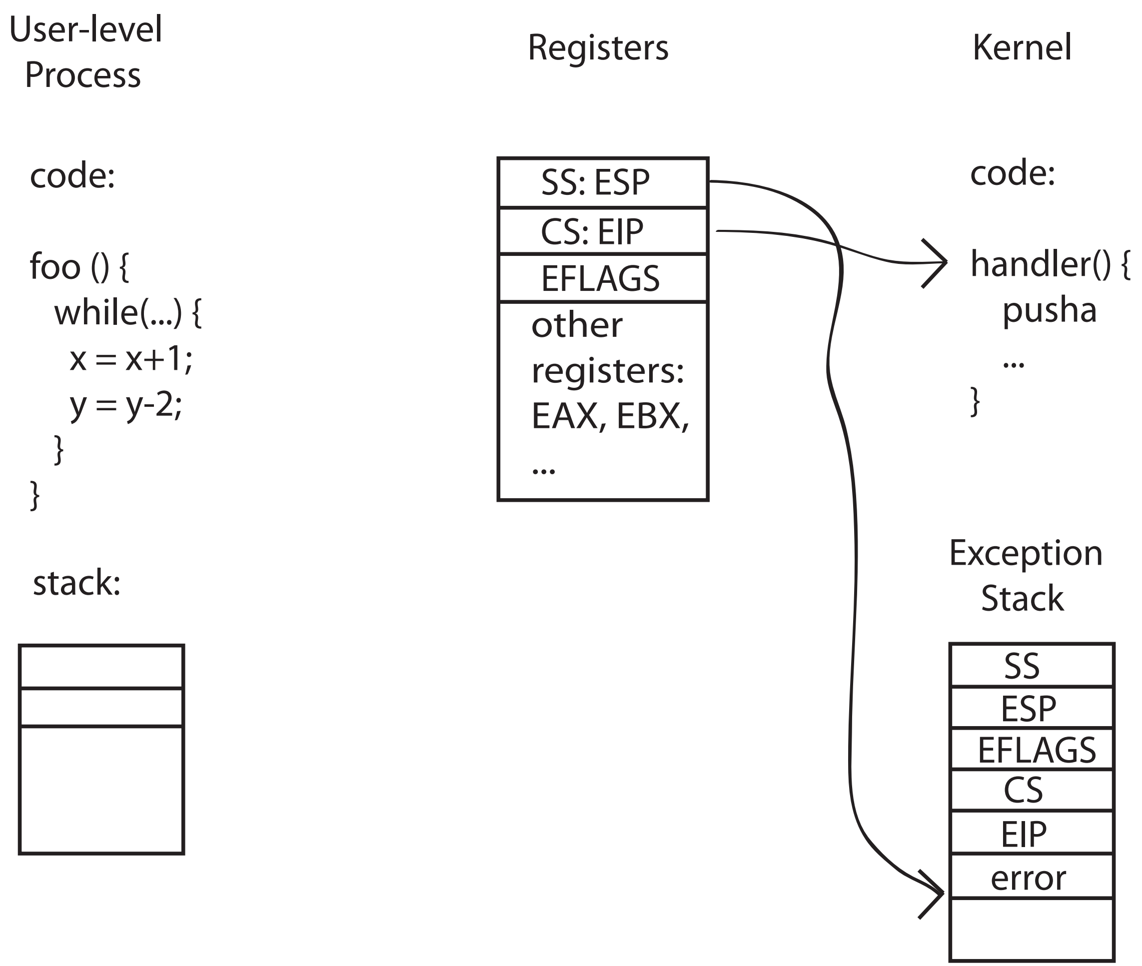
Atomic Mode Transfer

- 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
  - Interrupt handler saves registers it might clobber

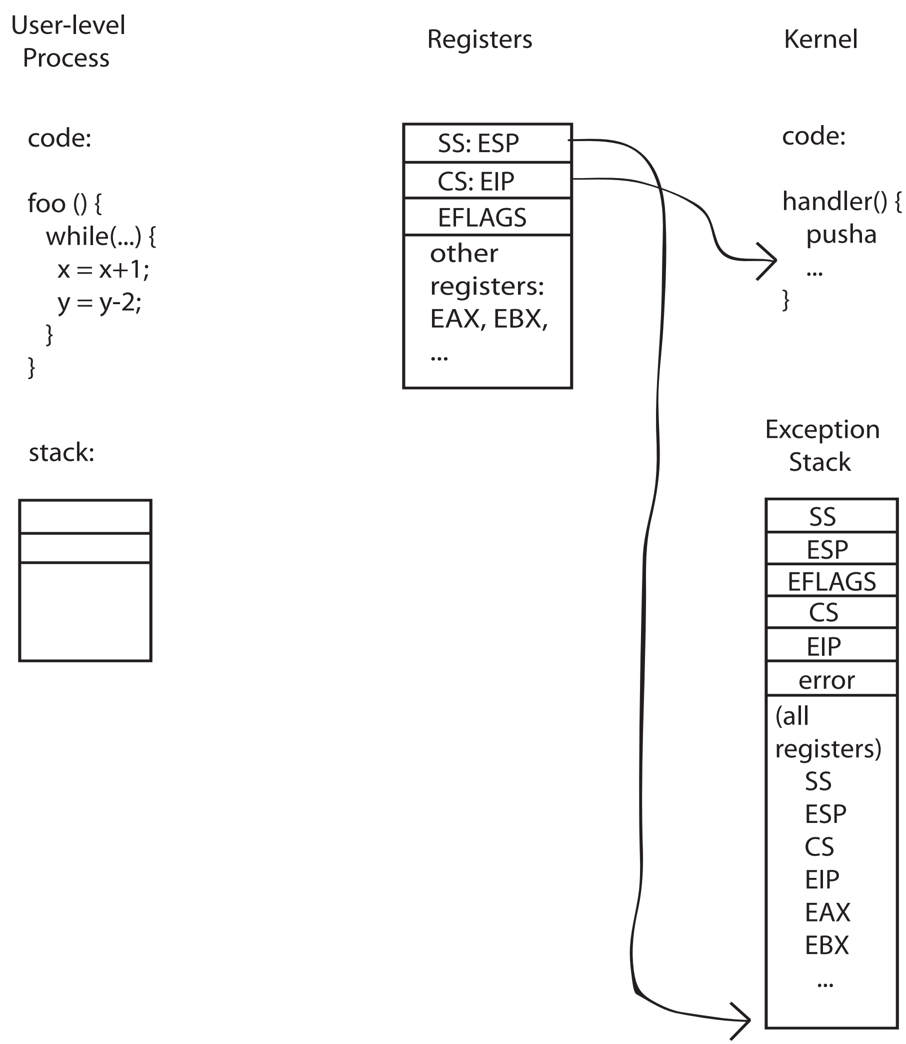
The x86 Example Before Interrupt



The x86 Example
Upon Interrupt Reception



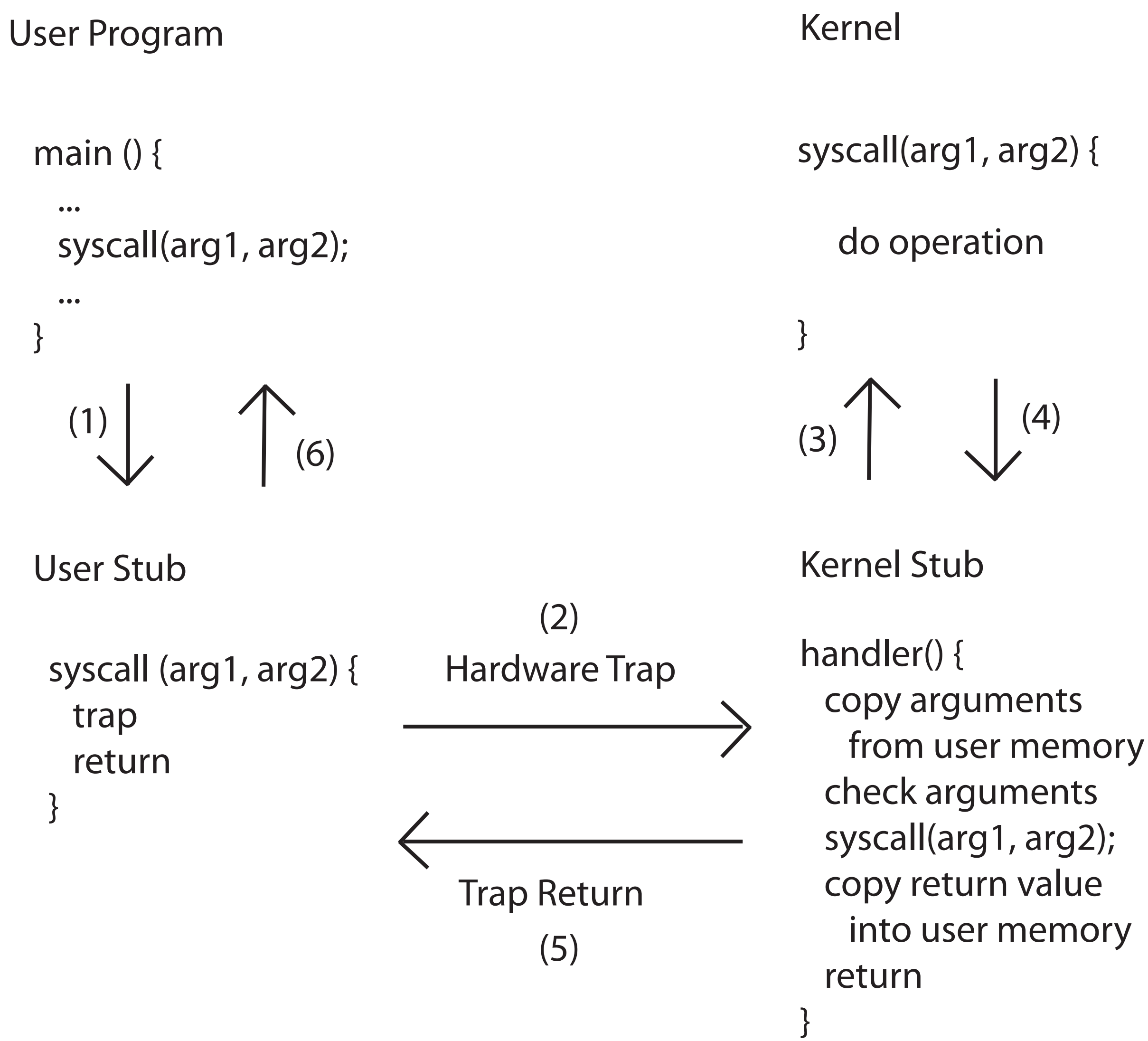
The x86 Example During the Handler's Execution



At 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

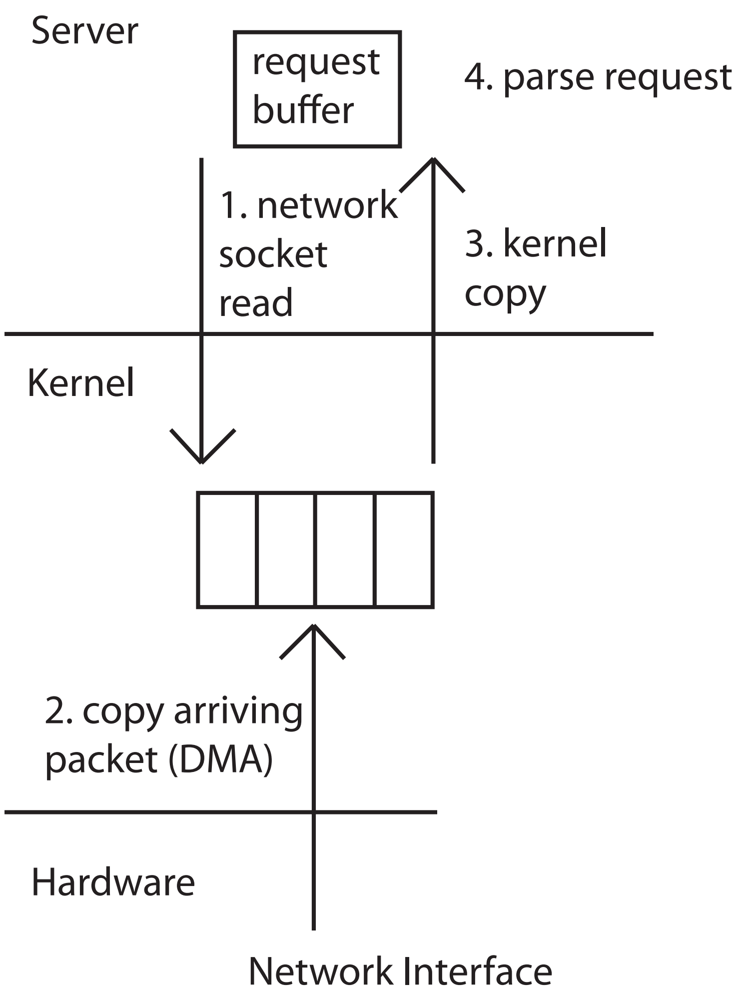
System Calls



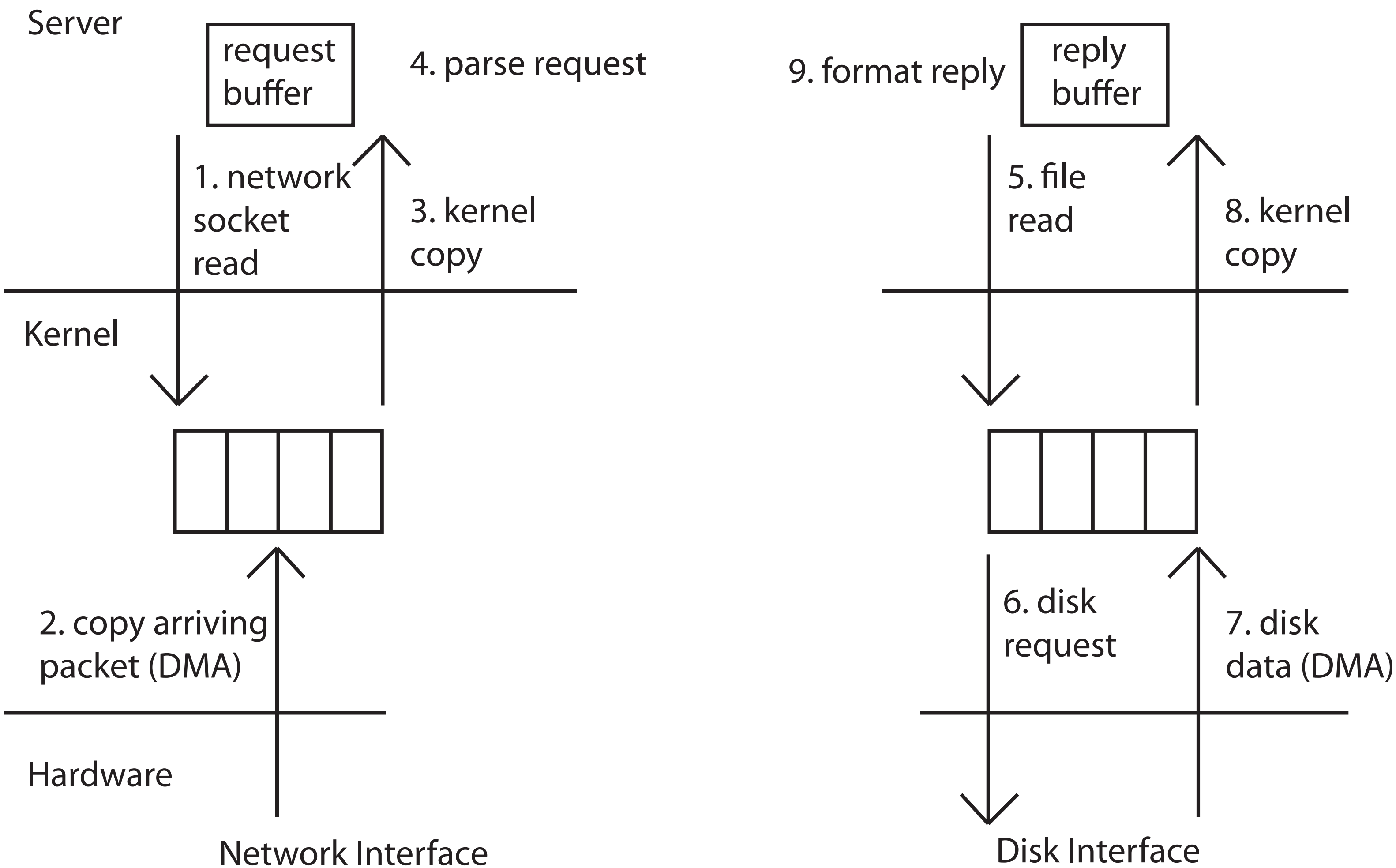
Kernel System Call Handler

- 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

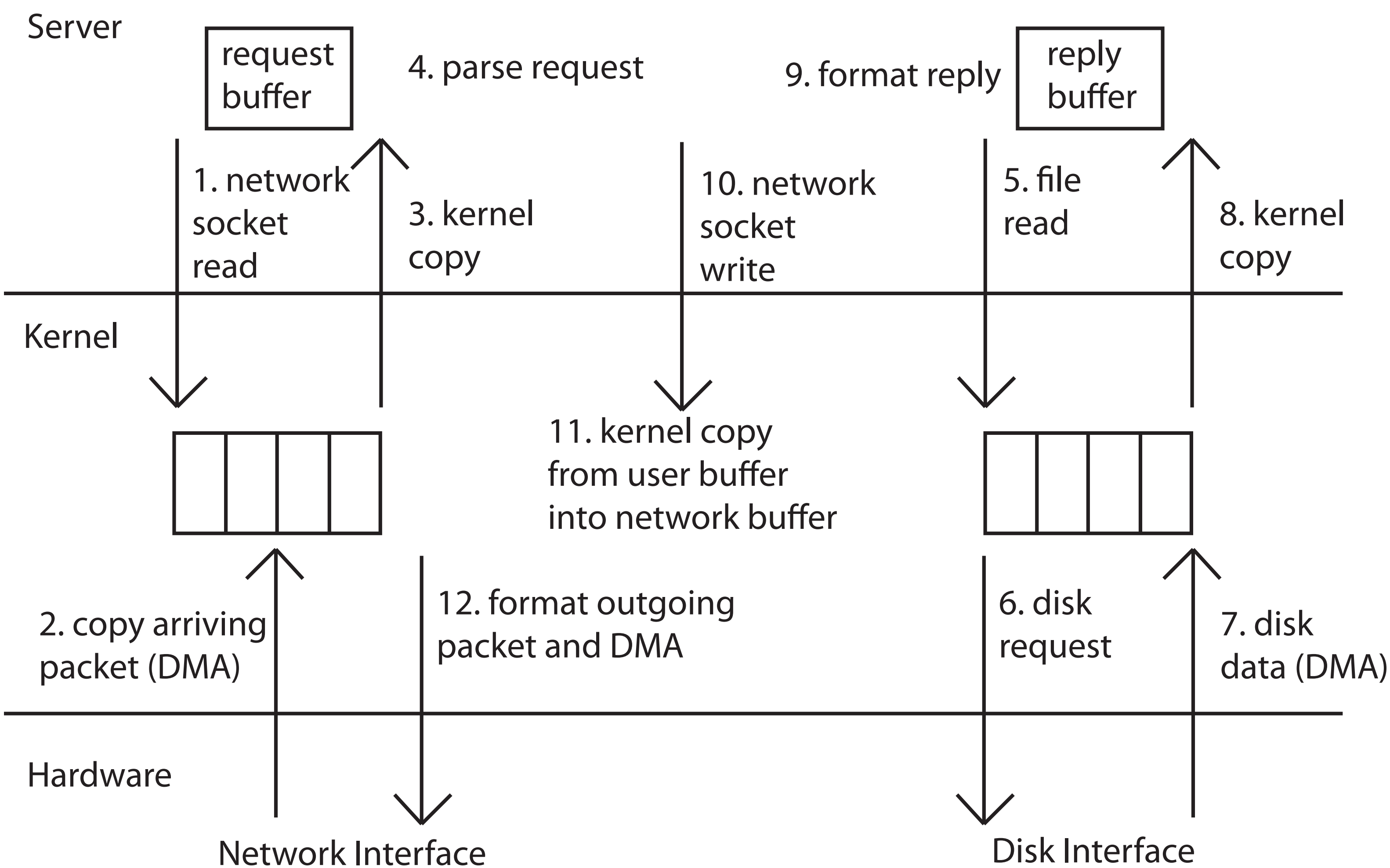
Web Server Example



Web Server Example



Web Server Example



New Process

- Create process
  1. Allocate and initialize the process control block (PCB)
  2. Allocate memory for the process
  3. Copy the program from disk into the newly allocated memory
  4. Allocate a user-level stack
  5. Allocate a kernel-level stack

- Run process
  1. Copy arguments into user memory
  2. Transfer control to user mode

Upcall: User-level interrupt

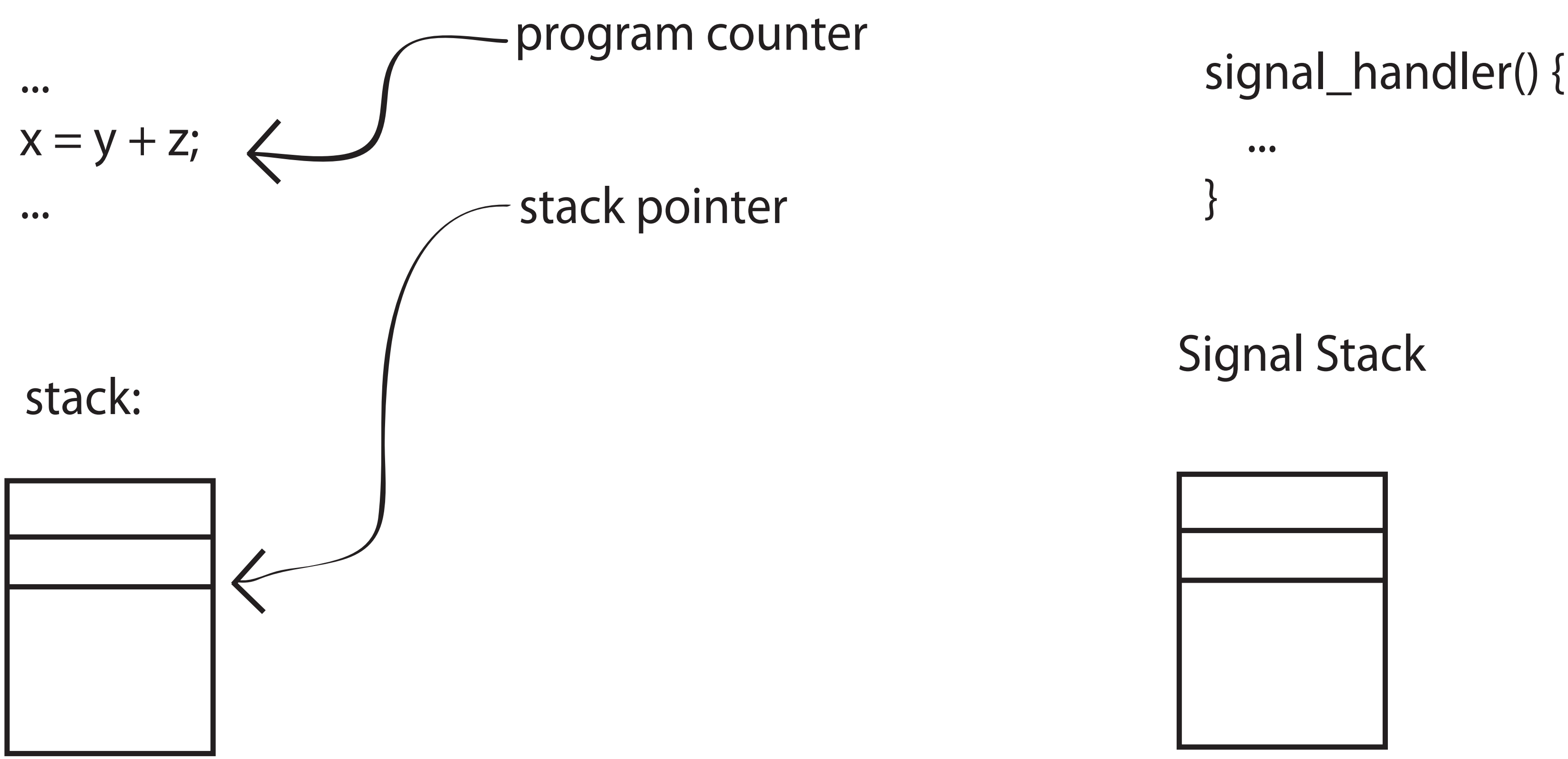
- AKA UNIX signal
  - Notify user process of event that needs to be handled right away

- Use-cases:
  - Preemptive user-level threads
  - Asynchronous I/O notification
  - Interprocess communication
  - User-level exception handling
  - User-level resource allocation

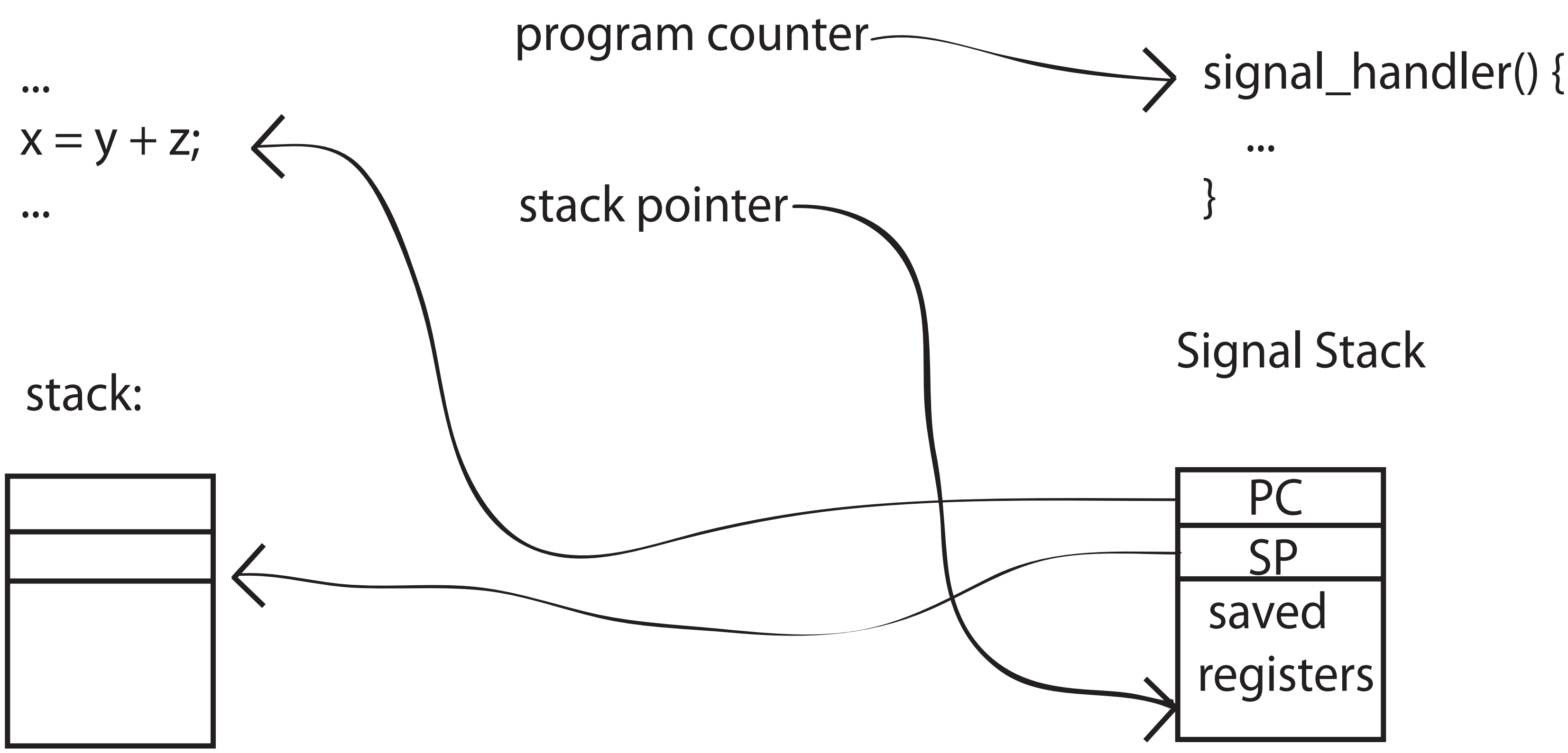
Upcall: User-level interrupt

- Direct analogue of kernel interrupts
  - Signal handlers – fixed entry points
  - Separate signal stack
  - Automatic save/restore registers – transparent resume
  - Signal masking: signals disabled while in signal handler

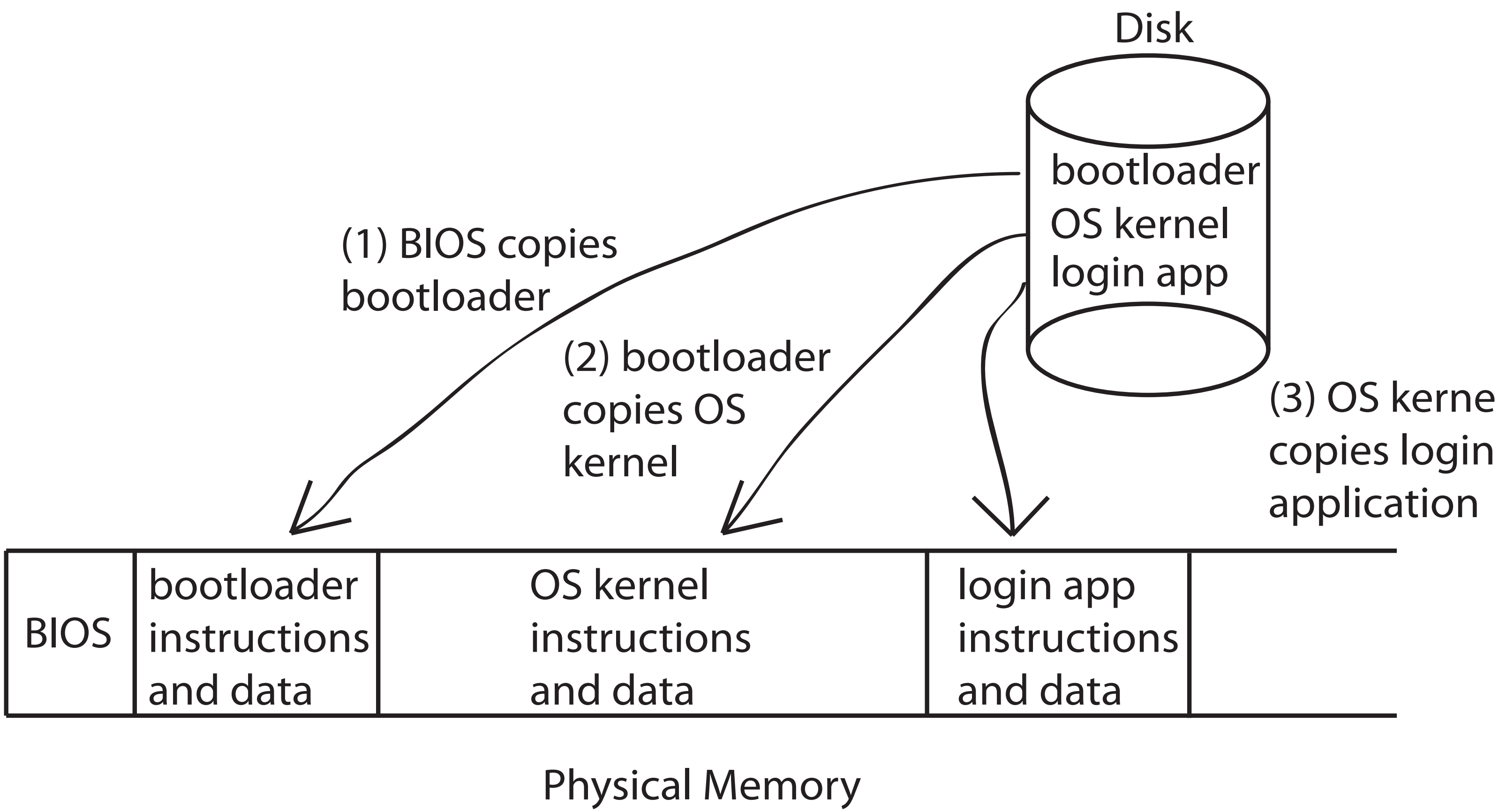
Upcall Example
Before a Unix signal



Upcall Example During a Unix signal handling



Booting


Disk
bootloader
OS kernel
login app
(1) BIOS copies bootloader
(2) bootloader copies OS kernel
(3) OS kernel copies login application
BIOS
bootloader instructions and data
OS kernel instructions and data
login app instructions and data
Physical Memory