The Kernel Abstraction

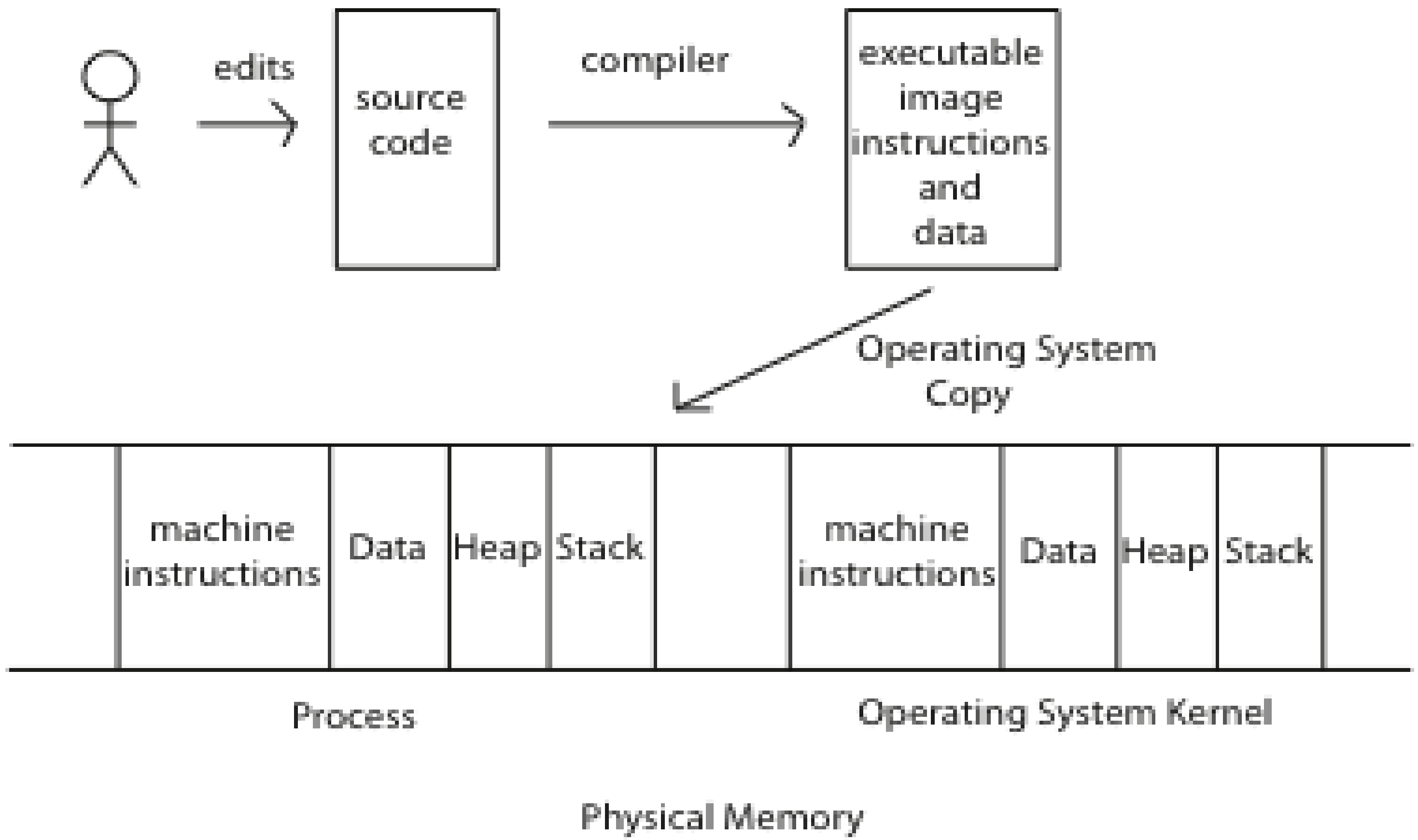
Challenge: Protection

- 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)

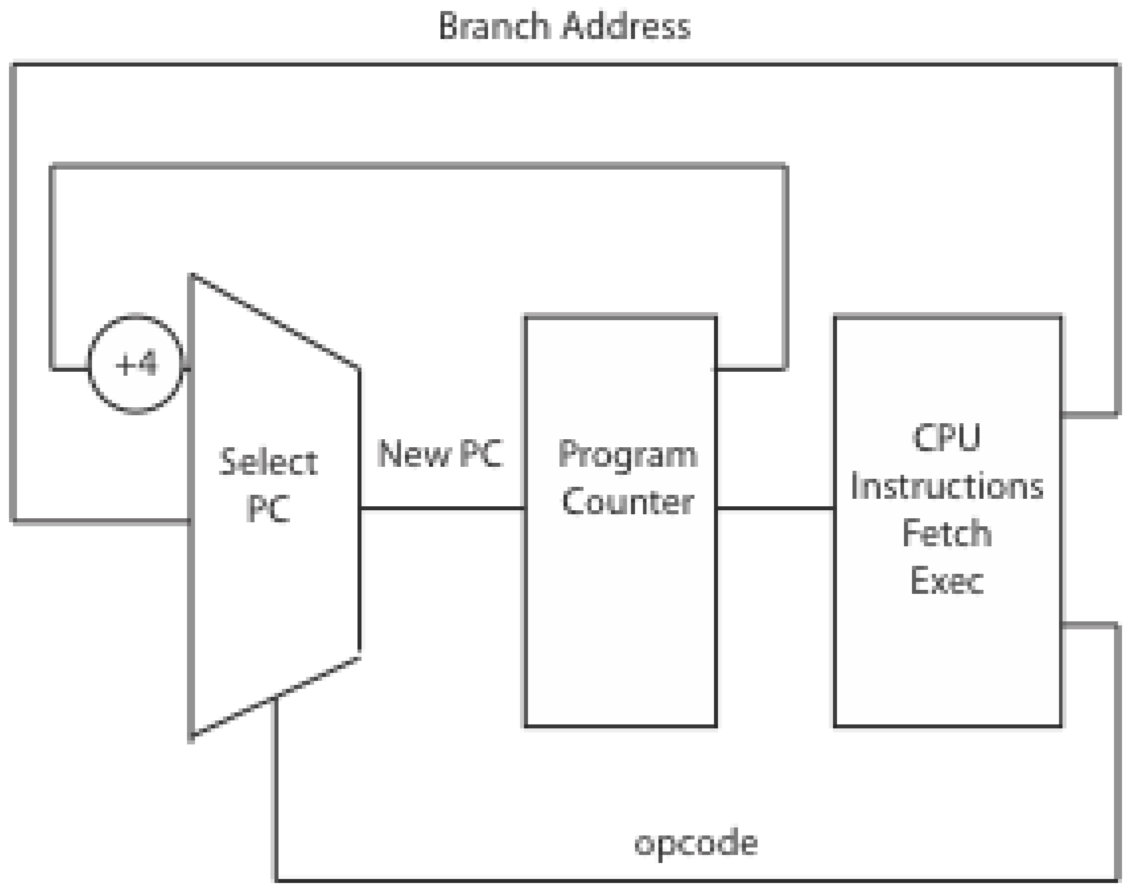
Thought Experiment

- How can we implement execution with limited privilege?
  - Execute each program instruction in a simulator
  - If the instruction is permitted, do the instruction
  - Otherwise, stop the process
  - Basic model in Javascript, …
- How do we go faster?
  - Run the unprivileged code directly on the CPU?

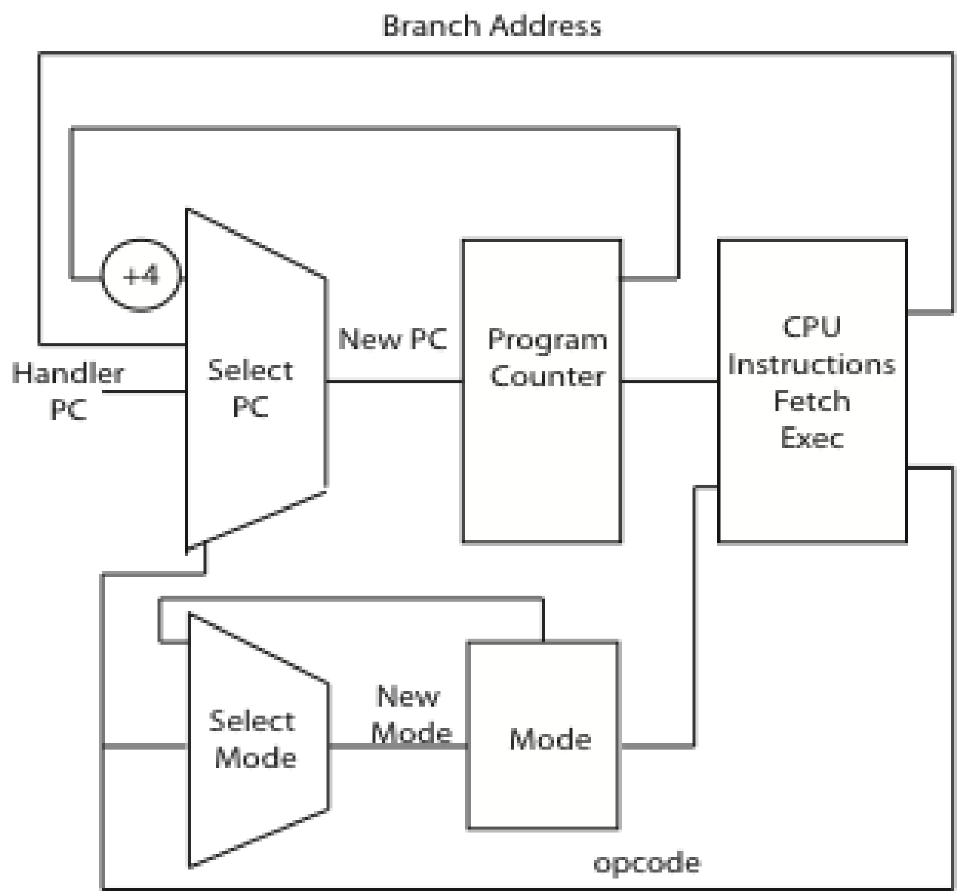
Hardware Support: Dual-Mode Operation

- 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

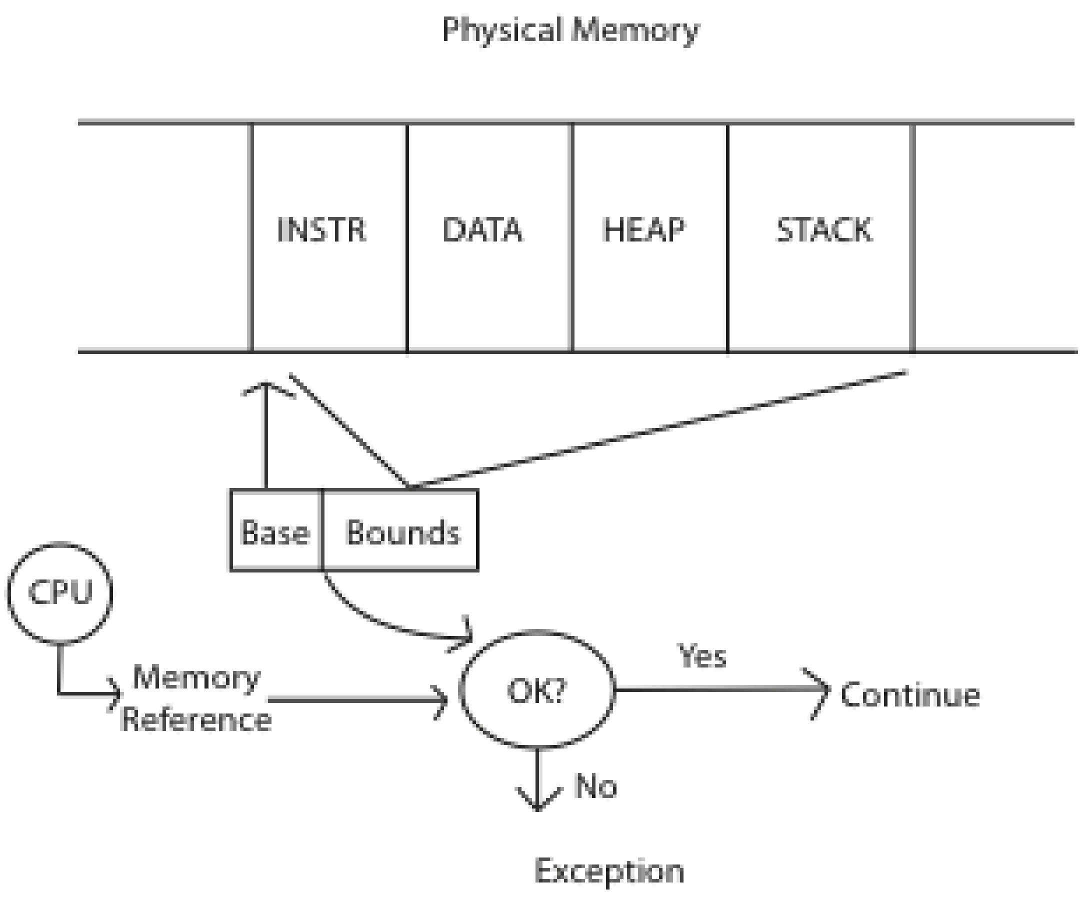
- Privileged instructions
  - Available to kernel
  - Not available to user code
- Limits on memory accesses
  - To prevent user code from overwriting the kernel
- Timer
  - To regain control from a user program in a loop
- Safe way to switch from user mode to kernel mode, and vice versa

Privileged instructions

- Examples?

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

Memory Protection

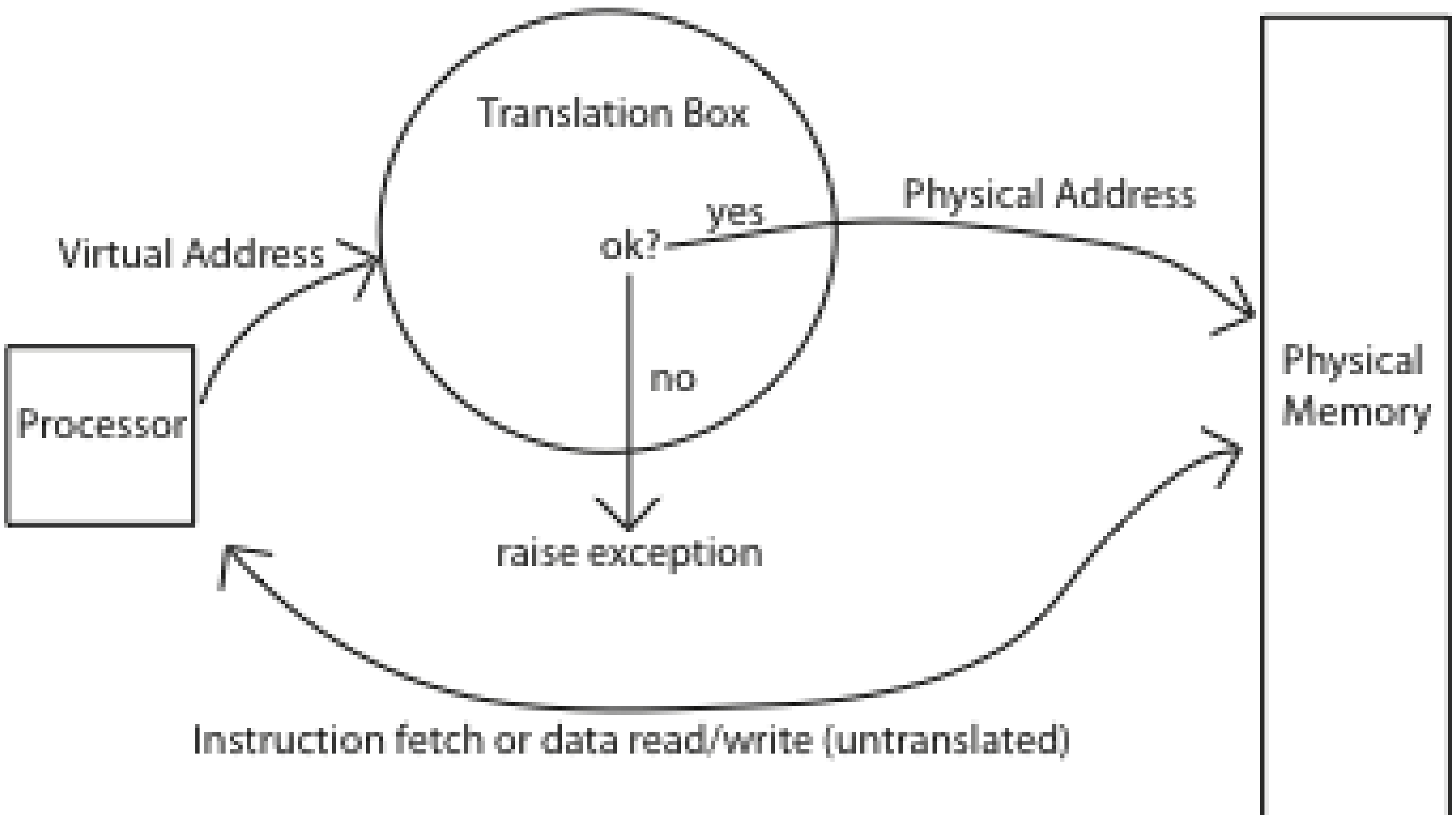


Towards Virtual Addresses

- Problems with base and bounds?

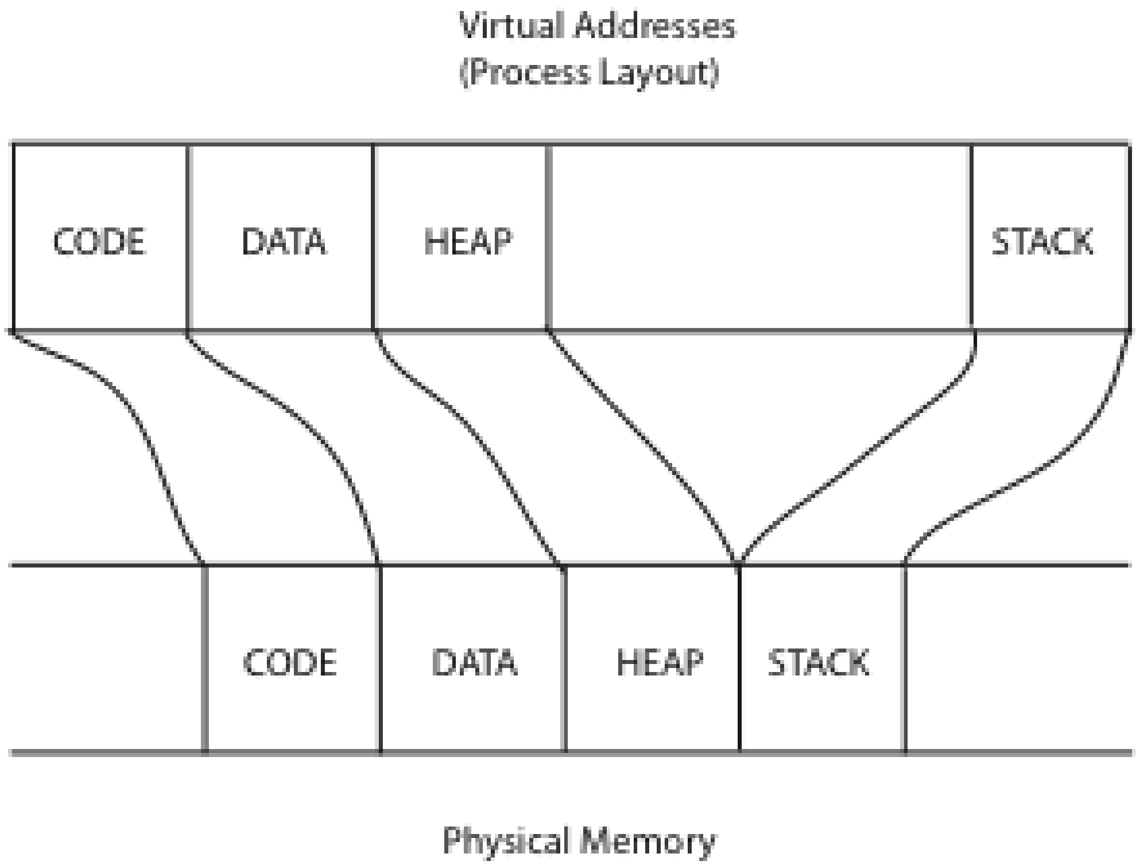
Virtual Addresses

- 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, …



Example: Corrected (What Does this Do?)

```
int staticVar = 0;      // a static variable
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

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?

Question

- Suppose we had a perfect object-oriented language and compiler, so that only an object's methods could access the internal data inside an object. If the operating system ran only programs written in that language, would it still need hardware memory address protection?

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 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
- Exceptions and system calls are *synchronous*, while interrupts are *asynchronous*

Mode Switch

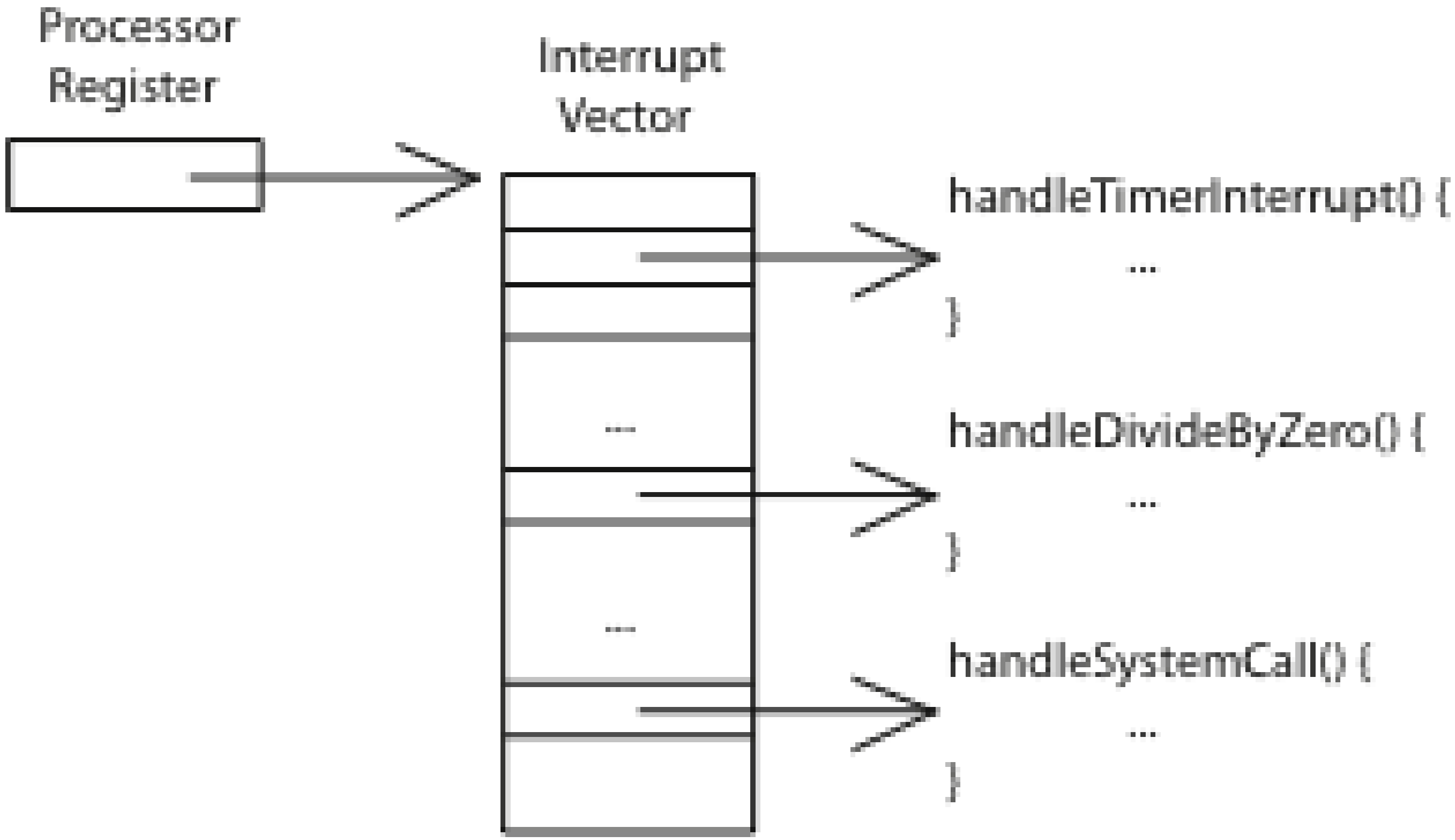
- From kernel-mode to user
  - New process/new thread start
    - Jump to first instruction in program/thread
  - Return from interrupt, exception, system call
    - Resume suspended execution
  - Process/thread context switch
    - Resume some other process
  - User-level upcall
    - Asynchronous notification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

How do we take interrupts safely?

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

Interrupt Vector

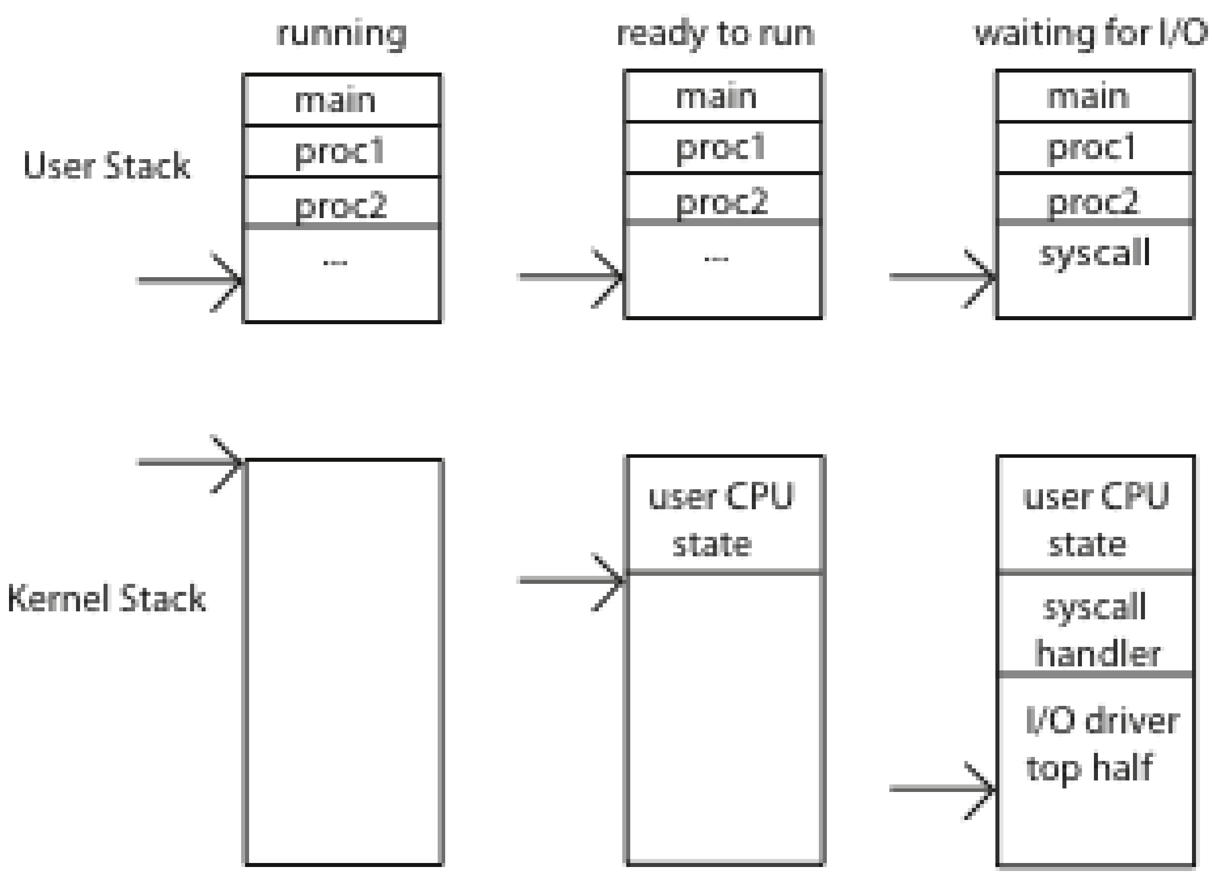
- Table set up by OS kernel; pointers to code to run on different events



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?

Interrupt Stack



Interrupt Masking

- Interrupt handler runs with interrupts off
  - Re-enabled when interrupt completes
- Kernel can also turn interrupts off
  - E.g., when determining the next process/thread to run
  - If defer interrupts too long, may drop I/O events
  - On x86
    - CLI: disable interrupts
    - STI: enable interrupts
    - Only applies to the current CPU
- Cf. implementing synchronization, chapter 5

Interrupt Handlers

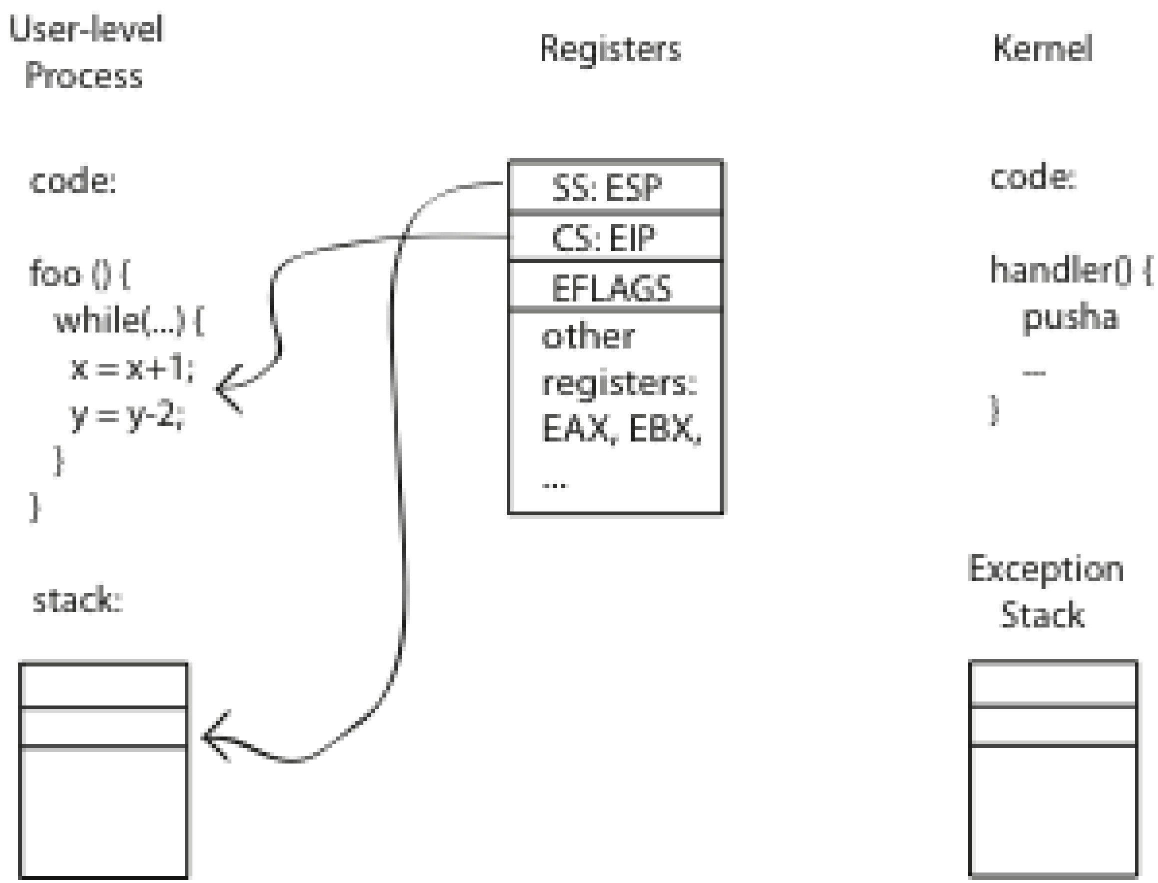
- Non-blocking, run to completion
  - Minimum necessary to allow device to take next interrupt
  - Any waiting must be limited duration
  - Wake up other threads to do any real work
- Rest of device driver runs as a kernel thread
  - Queues work for interrupt handler
  - (Sometimes) waits for interrupt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

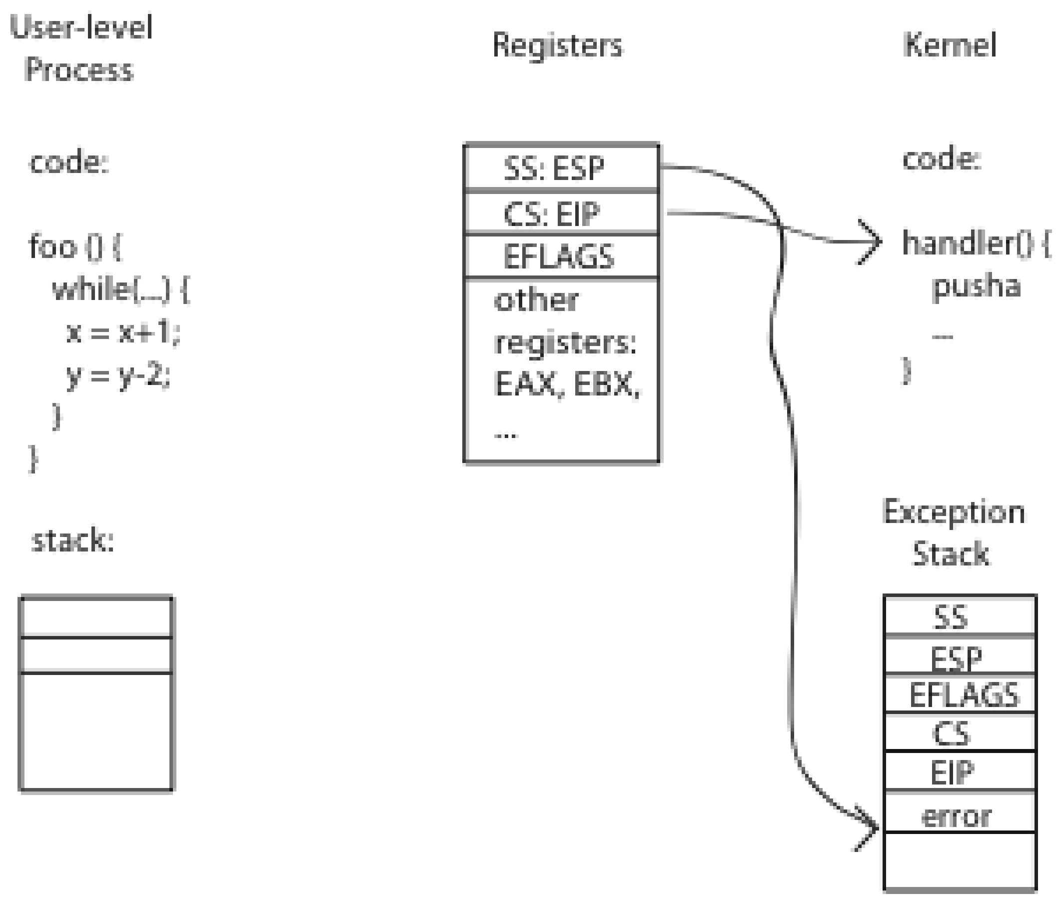
  - Then interrupt handler saves registers it might clobber

Before



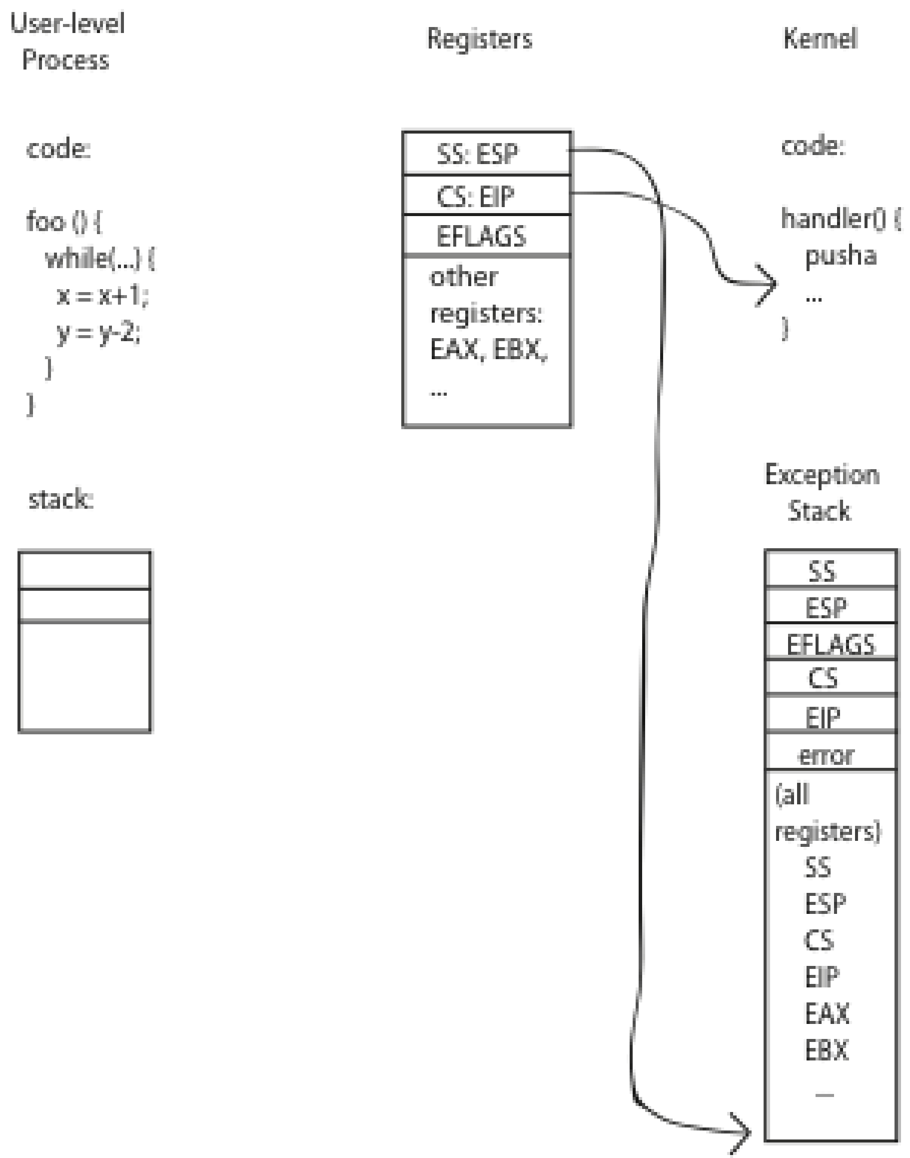
User-level
Process
code:
foo () {
while(...) {
x = x+1;
y = y-2;
}
}
stack:
Registers
SS: ESP
CS: EIP
EFLAGS
other
registers:
EAX, EBX,
...
Kernel
code:
handler() {
pusha
...
}
Exception
Stack

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

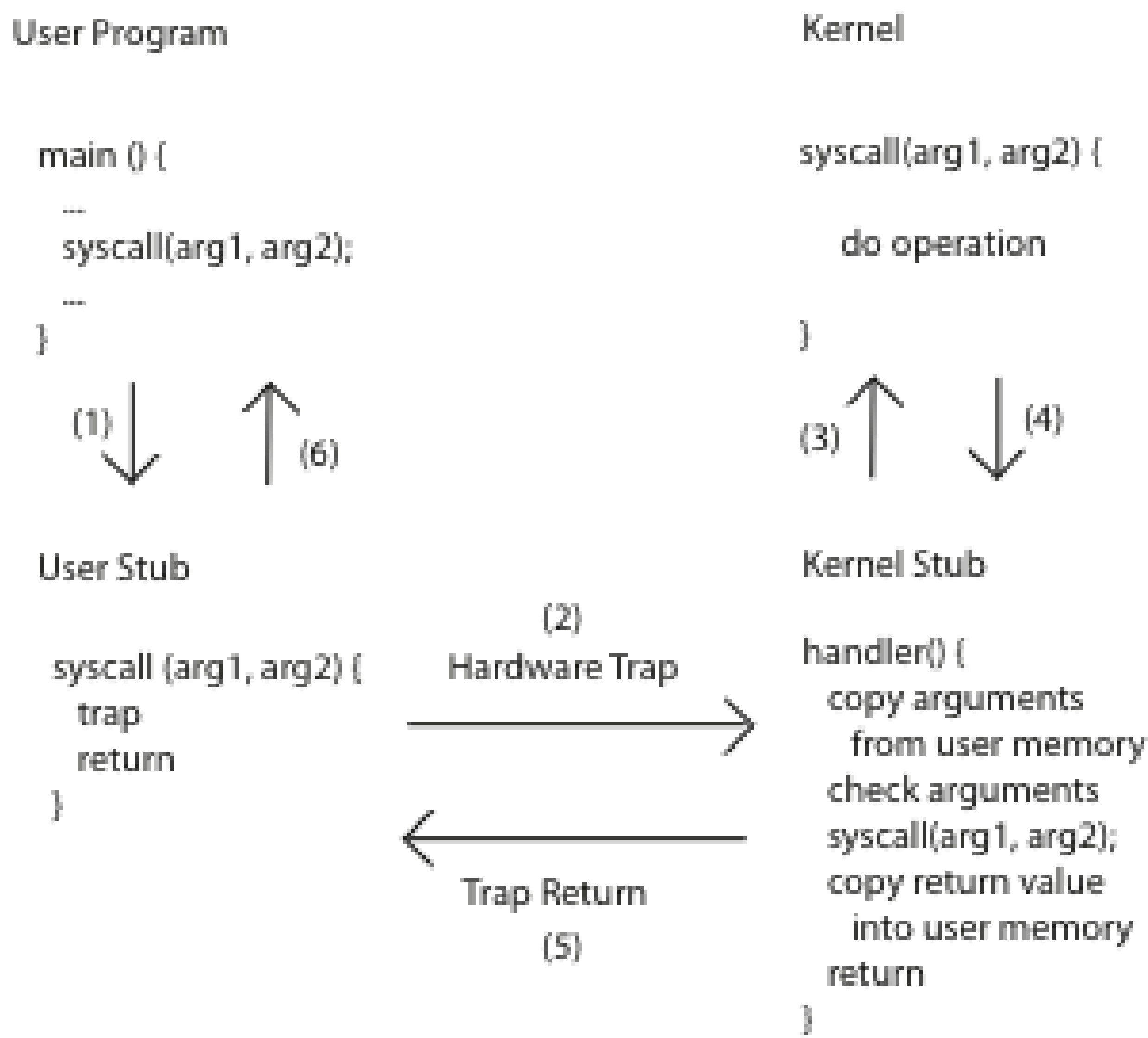


User-level Process
code:
foo () {
while(...) {
x = x+1;
y = y-2;
}
}
stack:
Registers
SS: ESP
CS: EIP
EFLAGS
other registers: EAX, EBX, ...
Kernel
code:
handler() {
pusha
...
}
Exception Stack
SS
ESP
EFLAGS
CS
EIP
error
(all registers)
SS
ESP
CS
EIP
EAX
EBX
...

At end of handler

- Resume process by doing the opposite:
  - Handler restores saved registers
  - Atomically return to interrupted process/thread
    - Restore program stack register
    - Restore processor status word/condition codes
    - Switch to user mode
    - Restore program counter

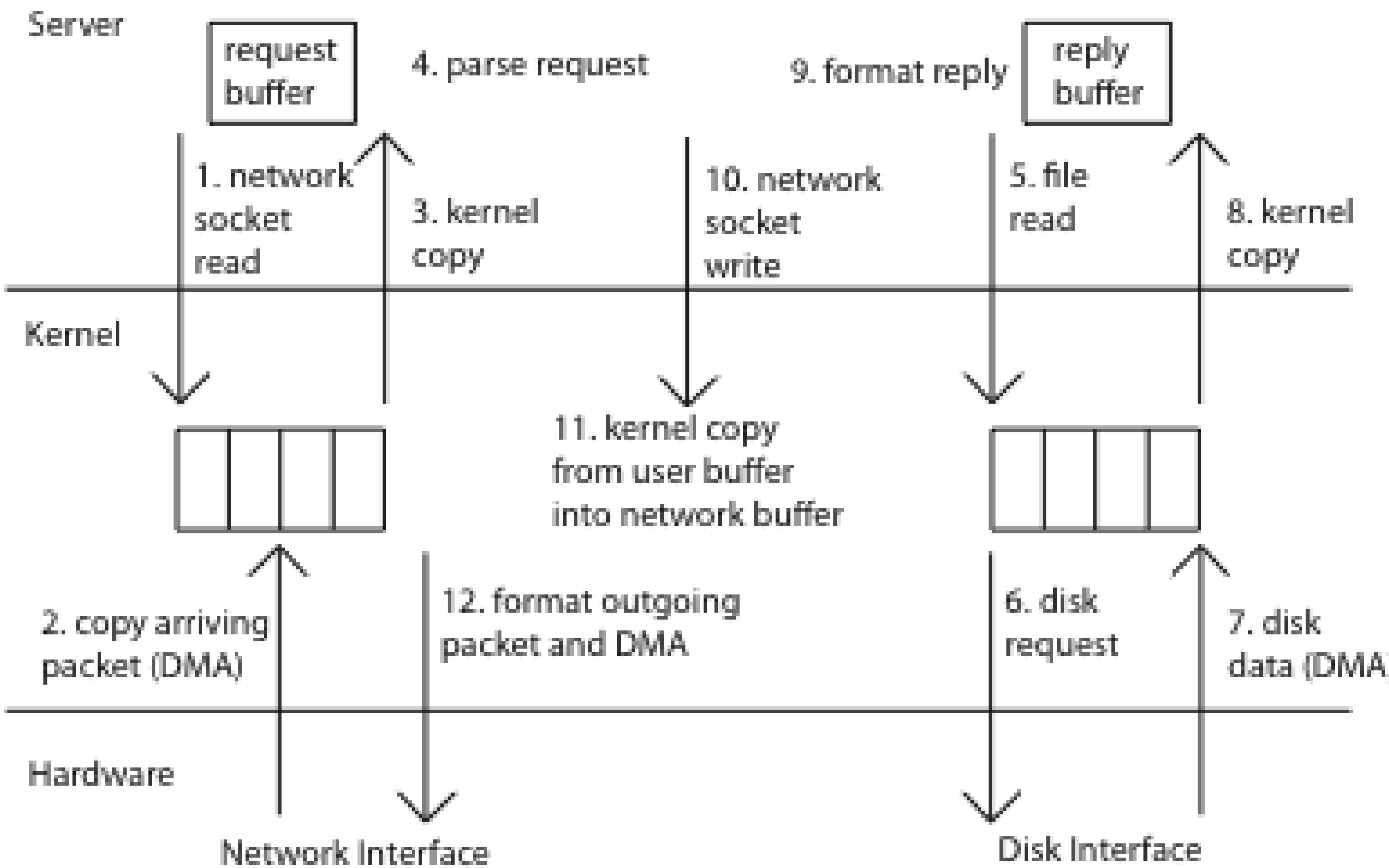
System Calls



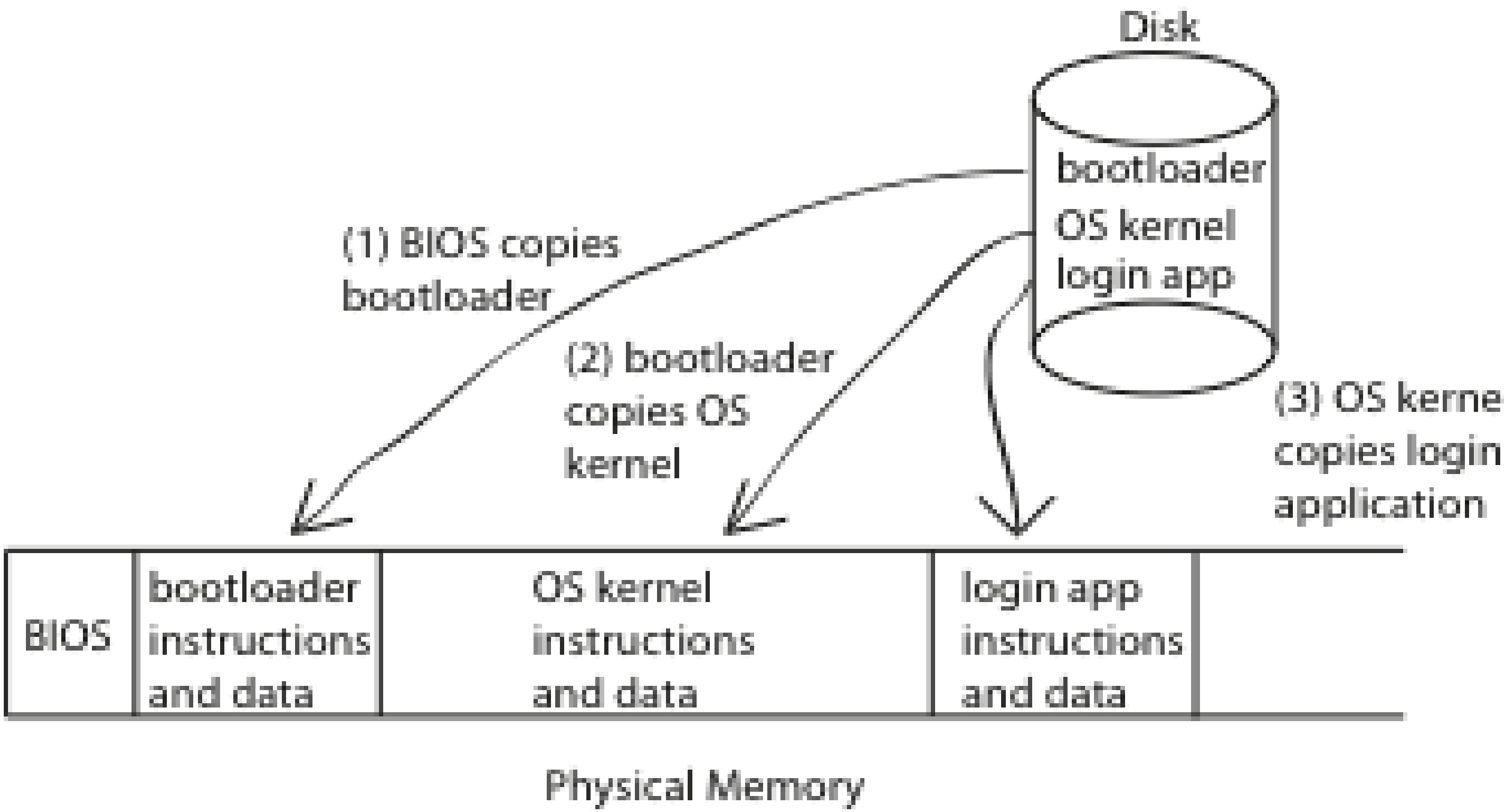
Kernel System Call Handler

- 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

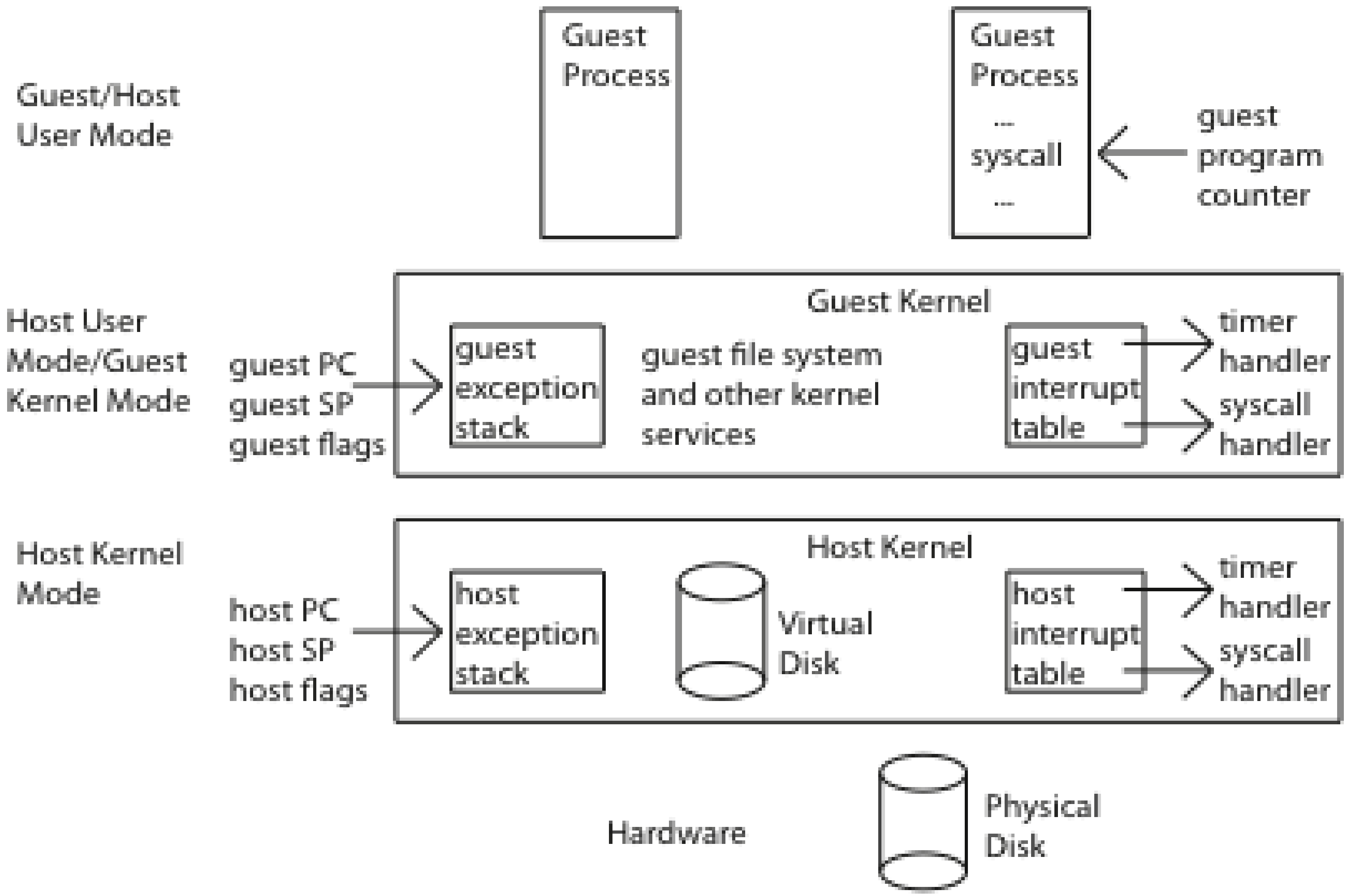
Web Server Example



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

Virtual Machine



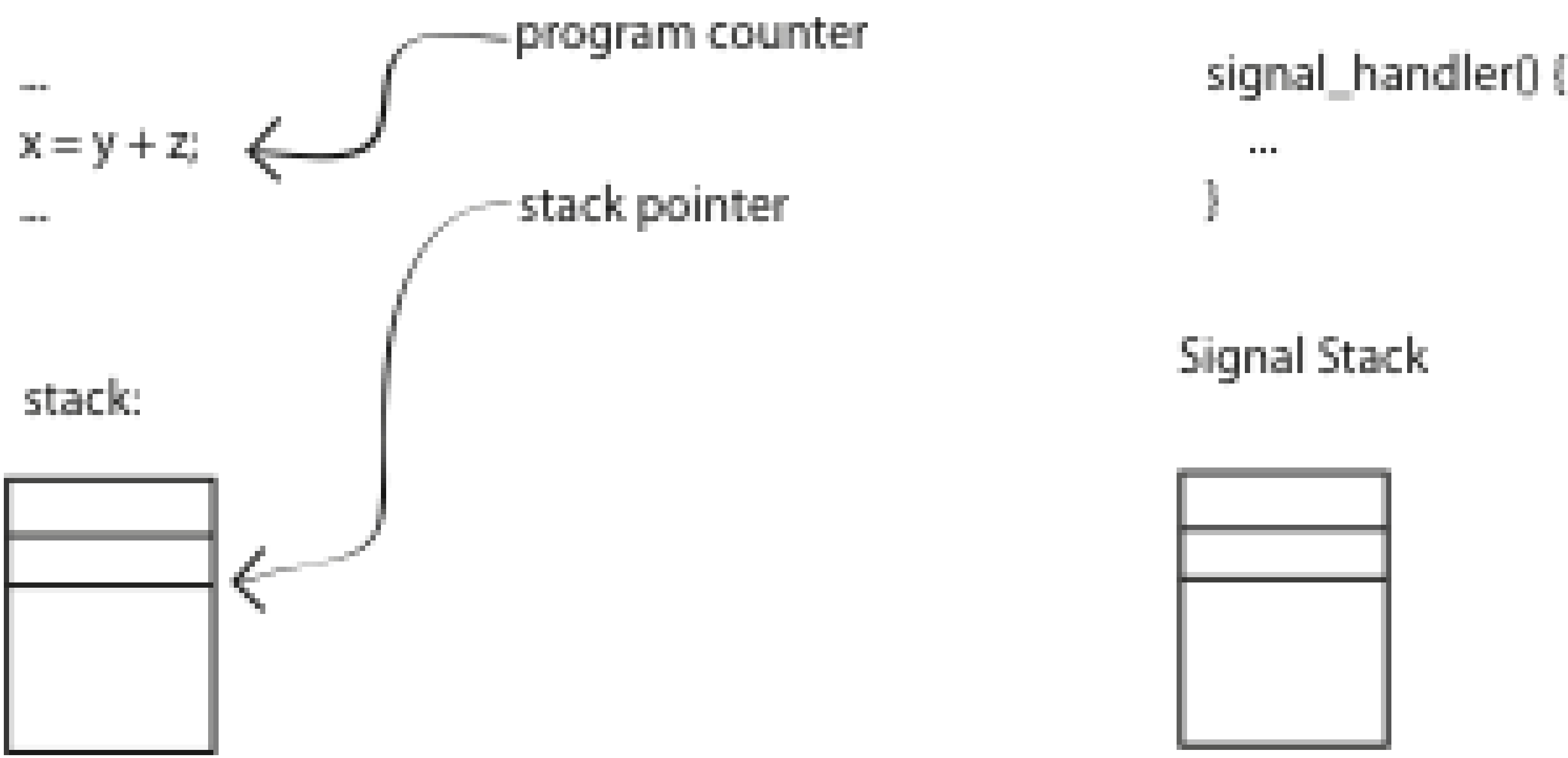
User-Level Virtual Machine

- How does VM Player work?
  - Runs as a user-level application
  - How does it catch privileged instructions, interrupts, device I/O, …
- Installs kernel driver, transparent to host kernel
  - (Requires administrator privileges!)
  - Modifies interrupt table to redirect to kernel VM code
  - If interrupt is for VM, upcall
  - If interrupt is for another process, reinstalls interrupt table and resumes kernel

Upcall: User-level interrupt (Unix "signal")

- Upcalls notify process of event that needs to be handled right away
    - Time-slice for user-level thread manager
    - Interrupt delivery for VM player
    - Die now (ctrl-C)
- 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: Before



Upcall: After