The Kernel Abstraction (part 2)

Last Time

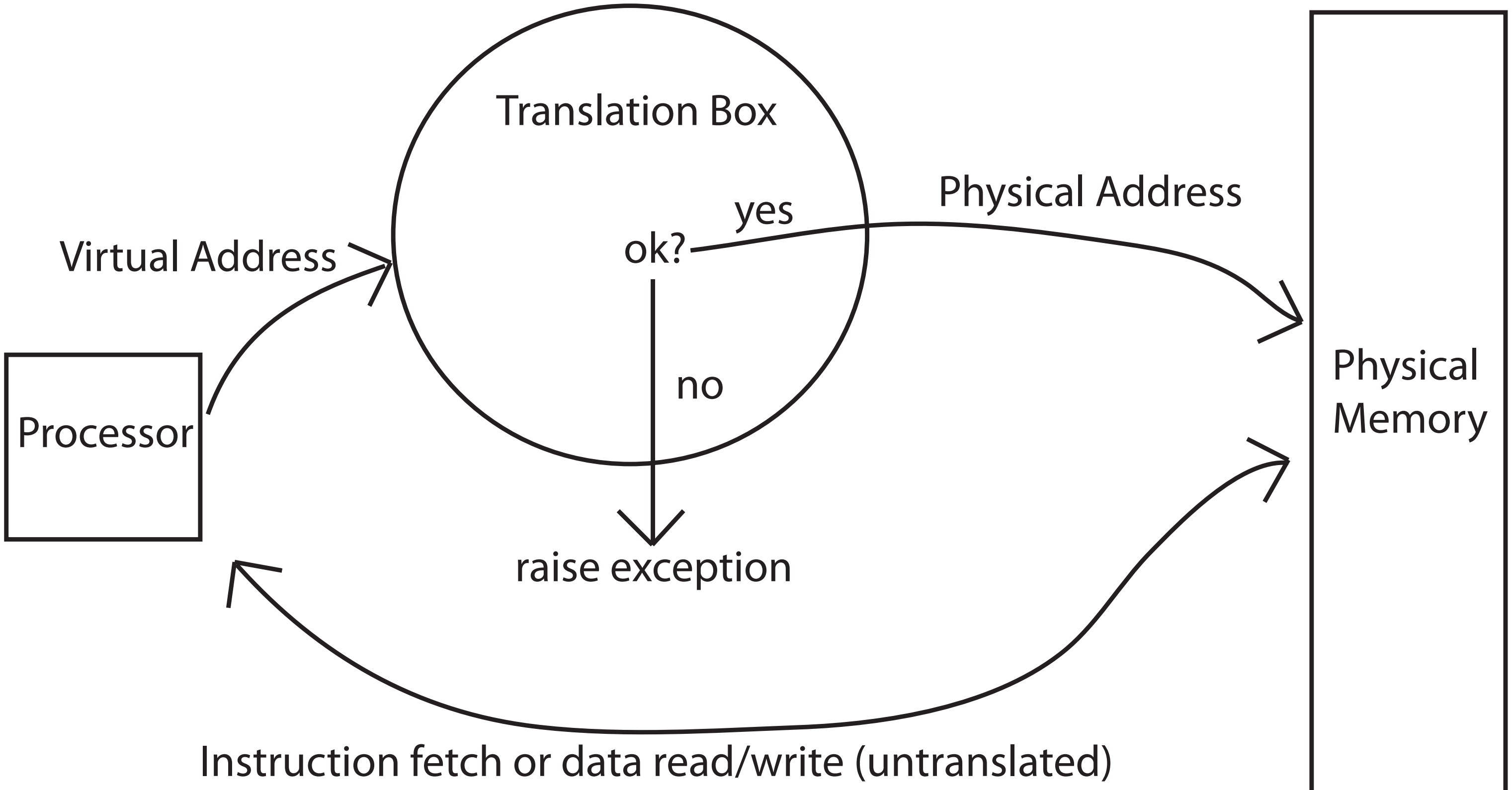
- Protection
  - prevent buggy or malicious user programs from corrupting the operating system or other apps
- Hardware support
  - Privileged instructions, not available at user-level
    - Exception trap to kernel if used at user-level
  - Memory protection
    - Virtual address -> physical address
    - If invalid virtual address, exception

Main Points

- Hardware support for dual-mode operation
  - 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?
  - Preventing misuse of control transfer

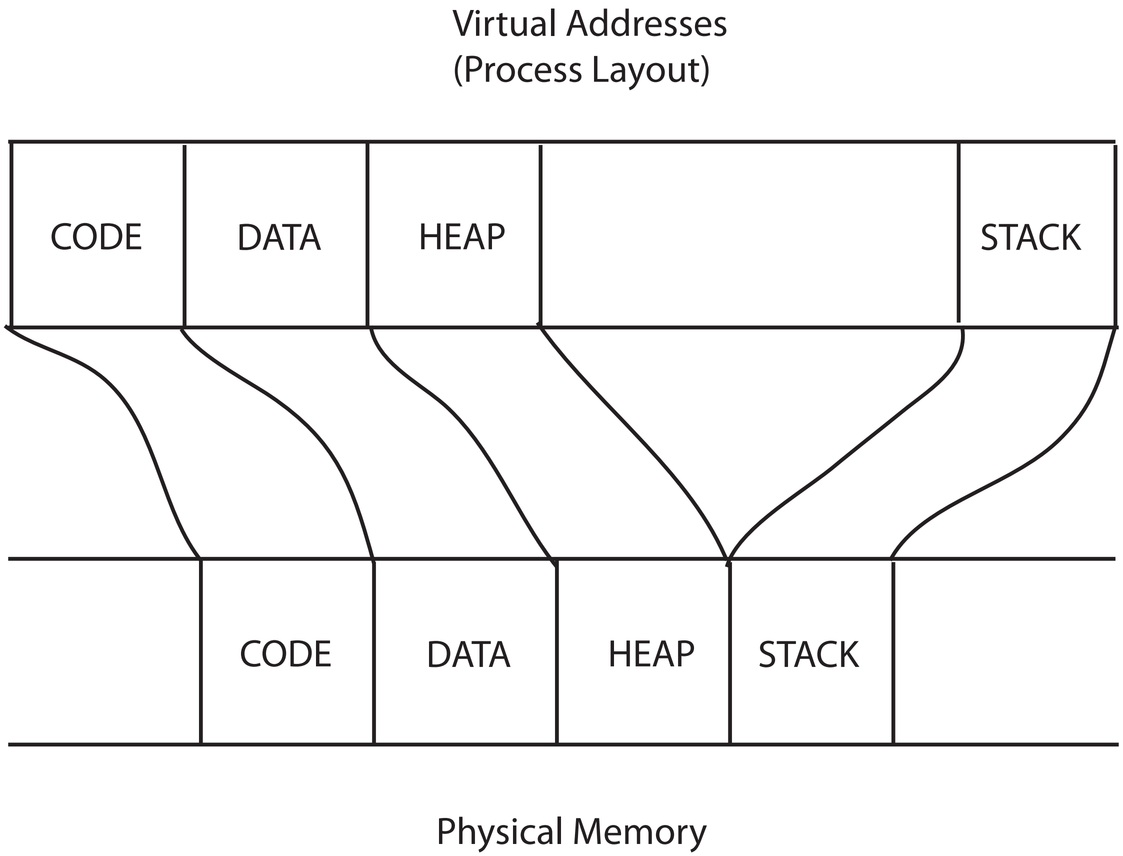
Virtual Addresses

- 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
  - Pintos assignment 1: generalize hardware timer to a software timer

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 only ran 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 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

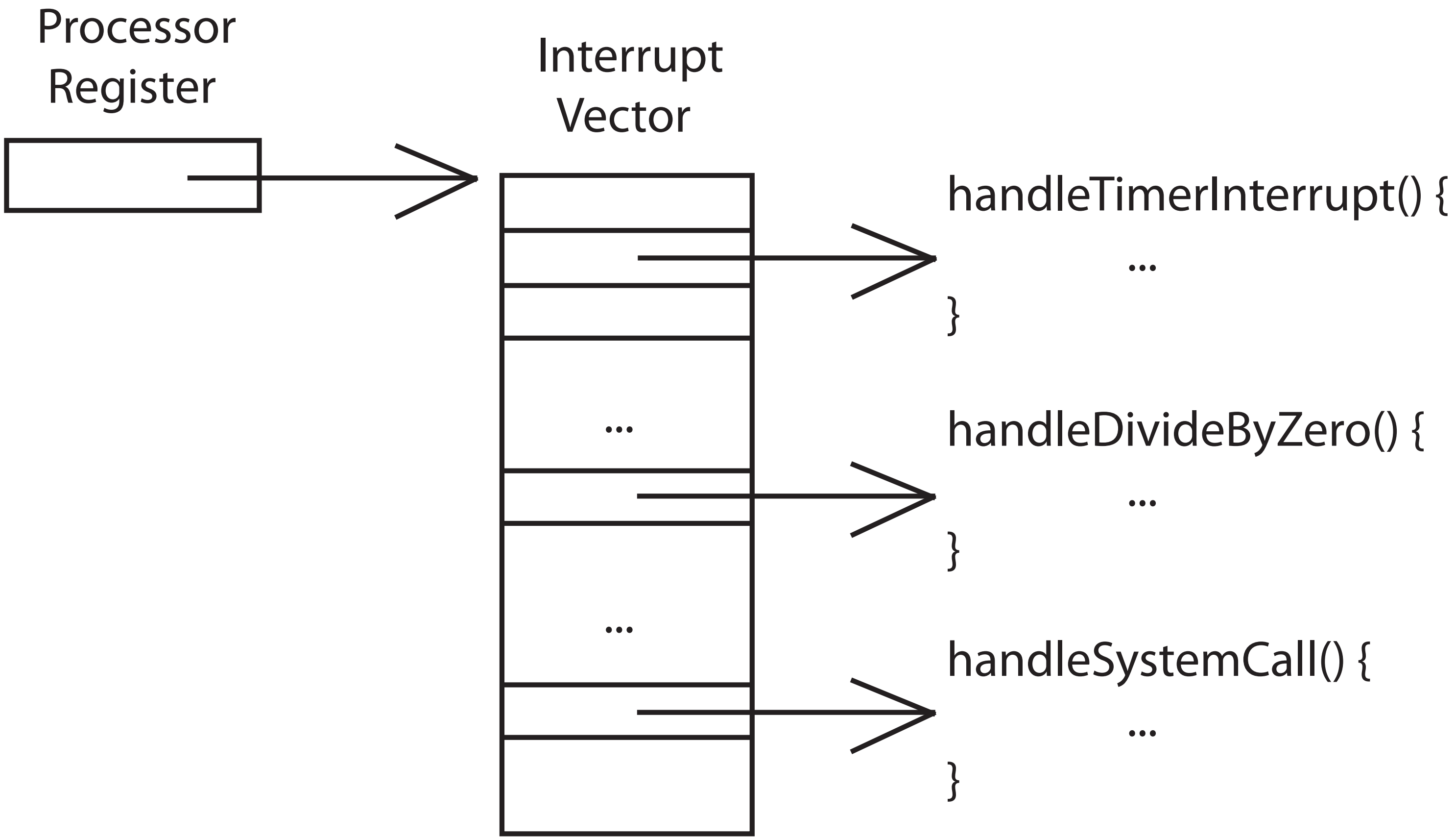
- From kernel-mode to user
  - New process/new thread start
    - Jump to first instruction in program/thread
  - Return from interrupt, exception, system call
    - Resume suspended execution
  - Process/thread context switch
    - Resume some other process
  - User-level upcall
    - Asynchronous notification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

How do we take interrupts safely?

- 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 instruction to change:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode
- Transparent restartable execution
  - User program does not know interrupt occurred

Interrupt Vector

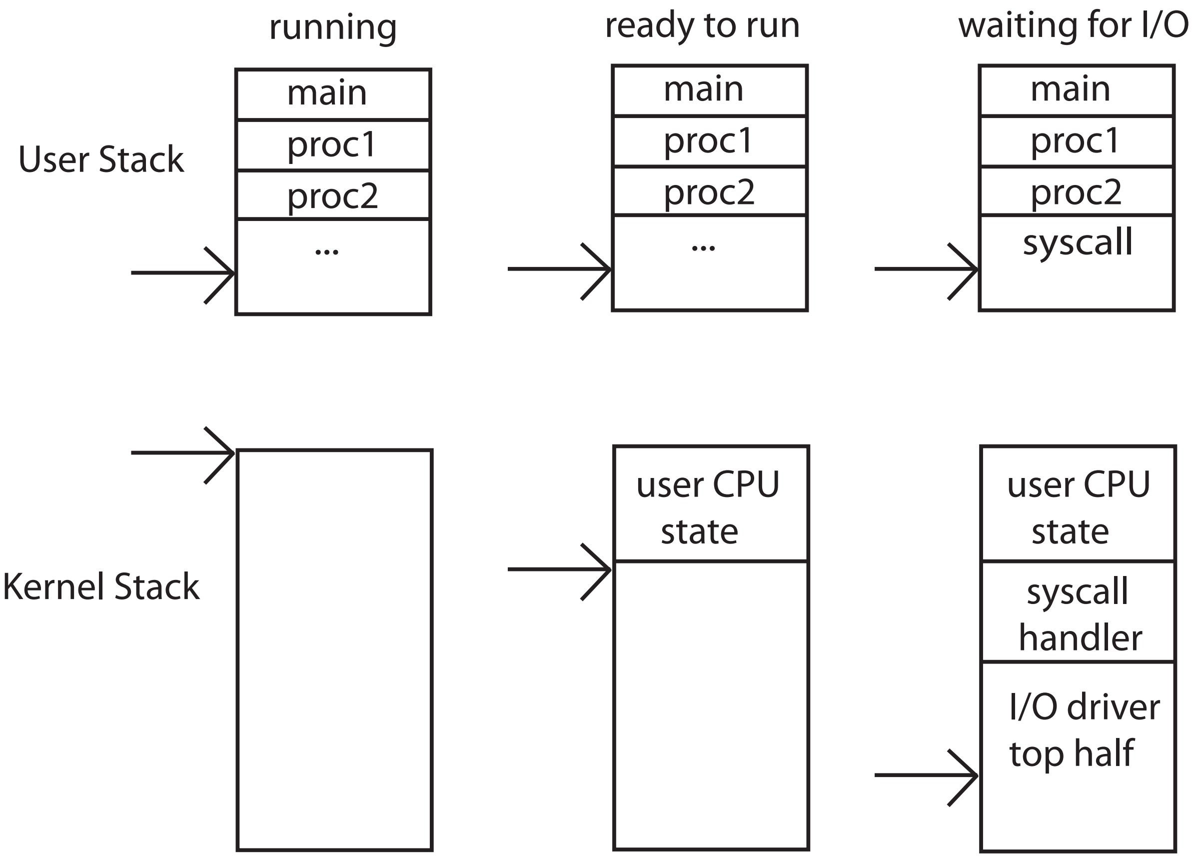
- Table set up by OS kernel; pointers to code to run on different events



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
  - 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

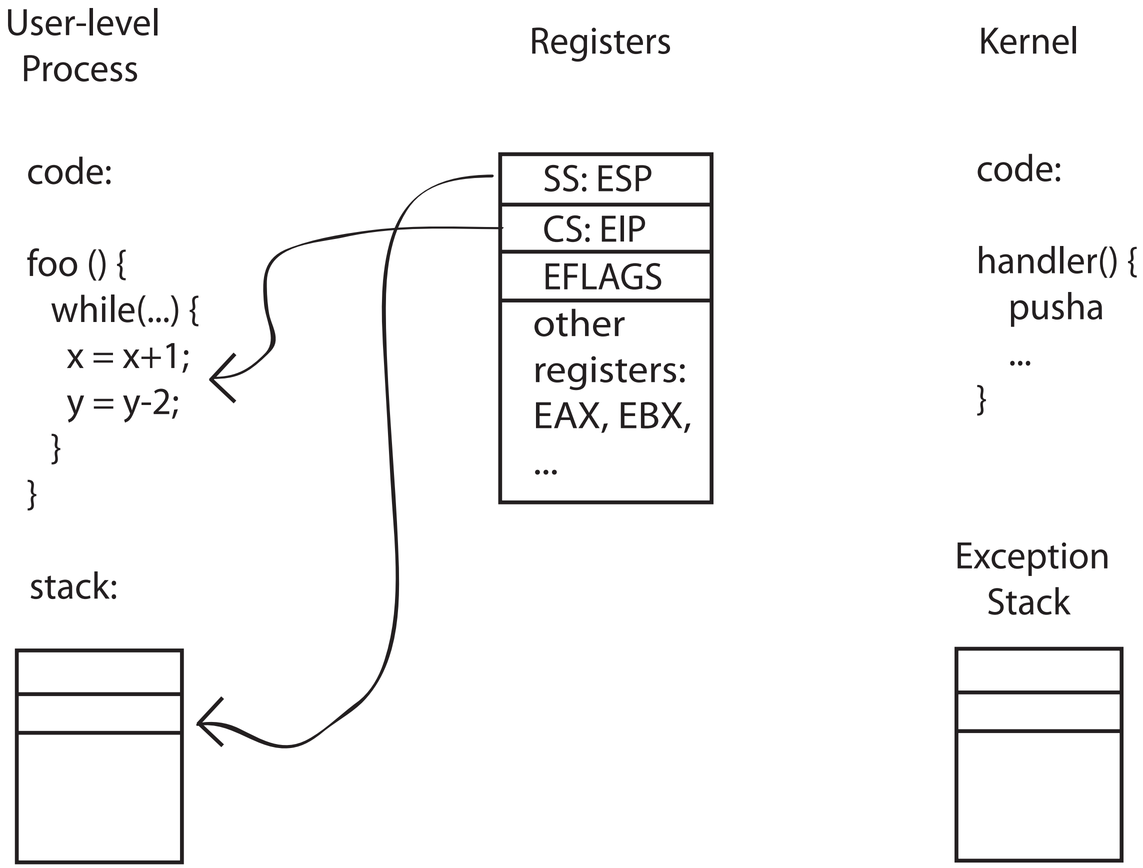
Interrupt Handlers

- 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
    - Pintos: semaphore_up
- 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

Before



User-level
Process

code:

```
foo () {
  while(...) {
    x = x+1;
    y = y-2;
  }
}
```

stack:

Registers

SS: ESP
CS: EIP
EFLAGS
other registers: EAX, EBX, ...

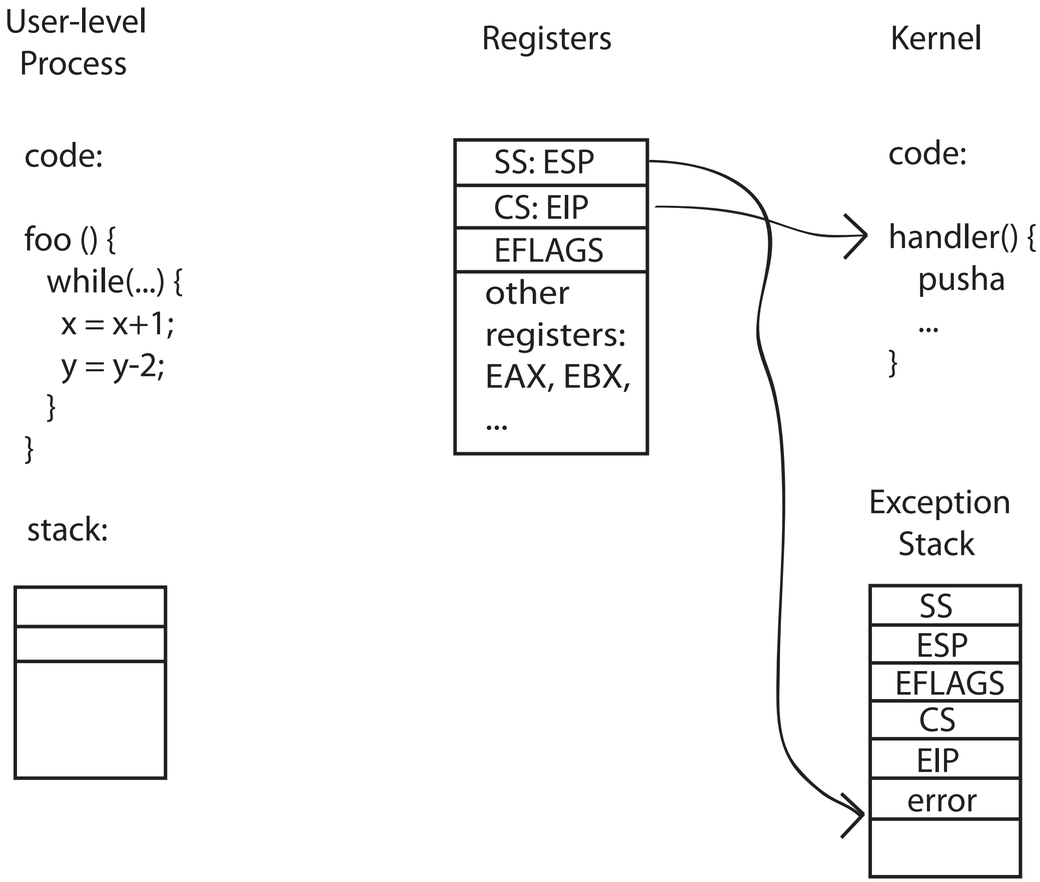
Kernel

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

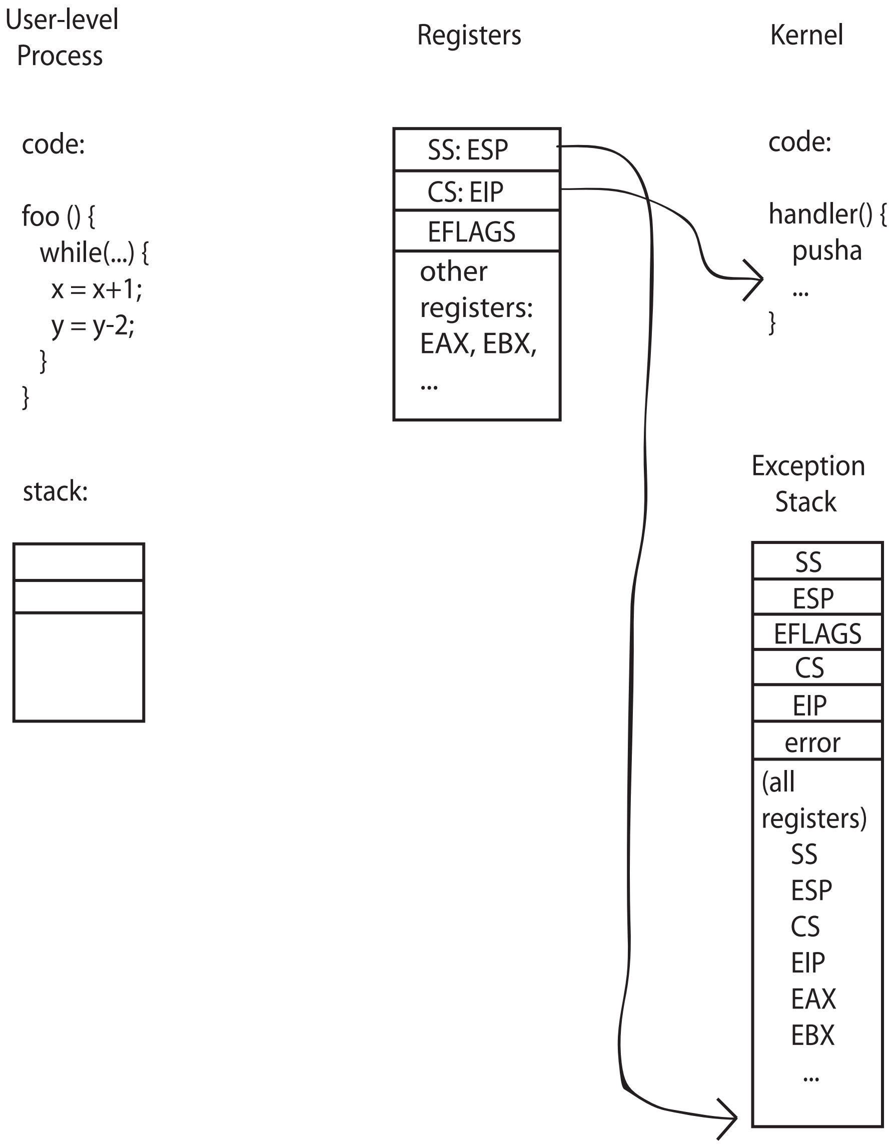
code:

```
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
   ...
}
```

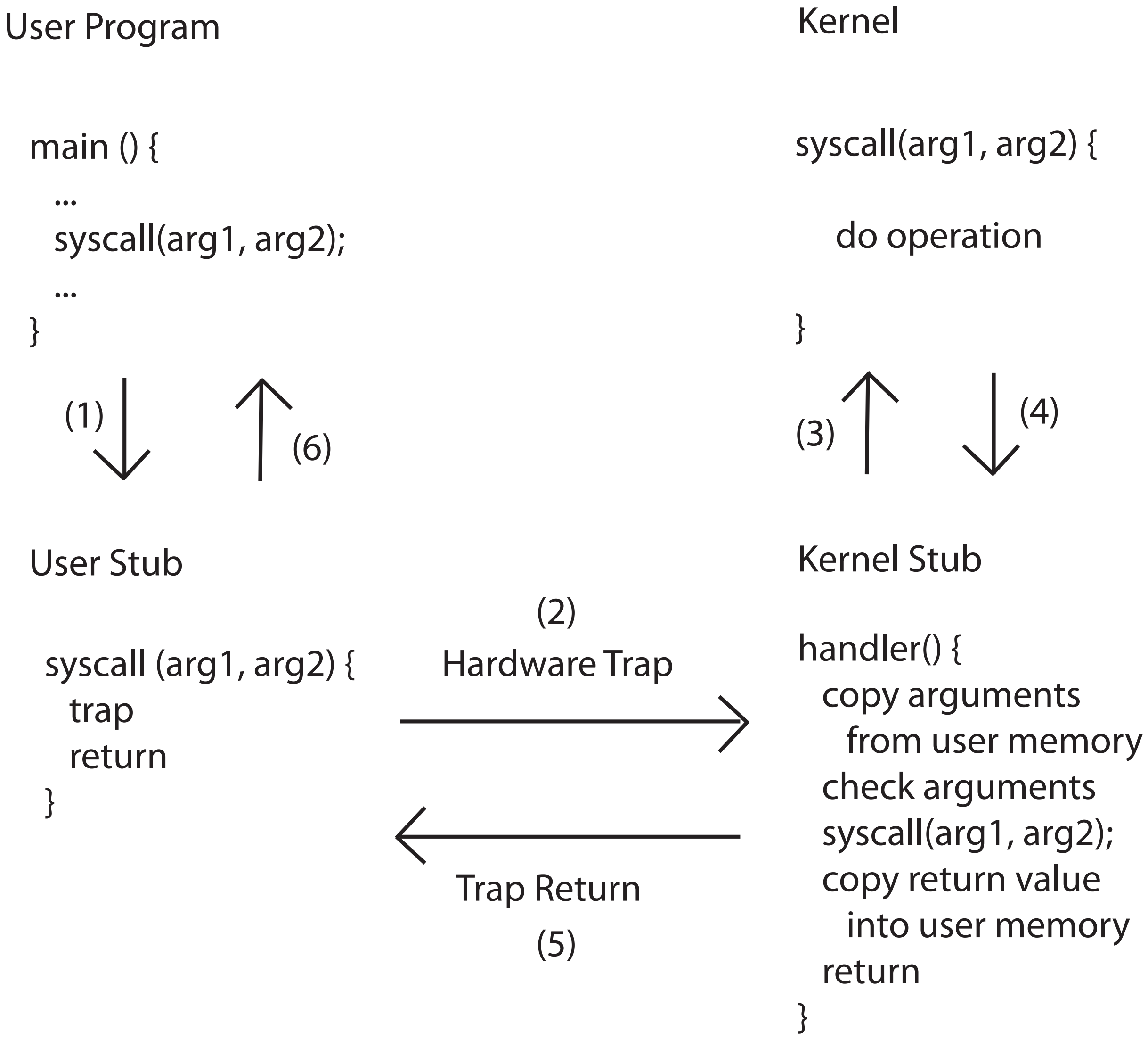
Exception
Stack

SS
ESP
EFLAGS
CS
EIP
error
(all
registers)
SS
ESP
CS
EIP
EAX
EBX
...

At end of handler

- 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



User Program

```
main () {
  ...
  syscall(arg1, arg2);
  ...
}
```

(1)

(6)

User Stub

```
syscall (arg1, arg2) {
  trap
  return
}
```

(2)
Hardware Trap

Trap Return
(5)

Kernel

```
syscall(arg1, arg2) {

  do operation

}
```

(3)

(4)

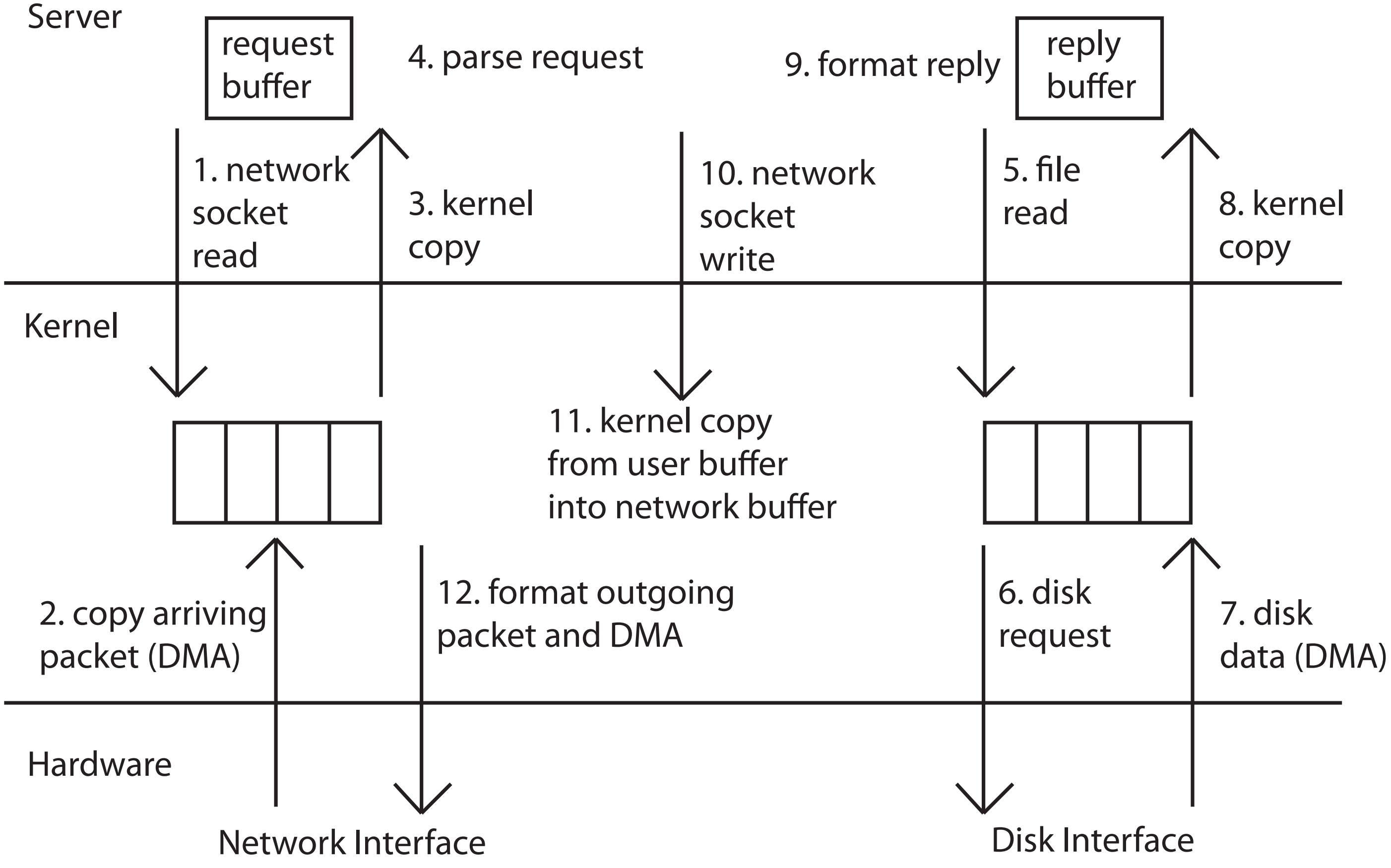
Kernel Stub

```
handler() {
  copy arguments
    from user memory
  check arguments
  syscall(arg1, arg2);
  copy return value
    into user memory
  return
}
```

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

Costs of Dual-Mode Operation



Server

request buffer

4. parse request

9. format reply

reply buffer

1. network socket read

3. kernel copy

10. network socket write

5. file read

8. kernel copy

Kernel

11. kernel copy from user buffer into network buffer

2. copy arriving packet (DMA)

12. format outgoing packet and DMA

6. disk request

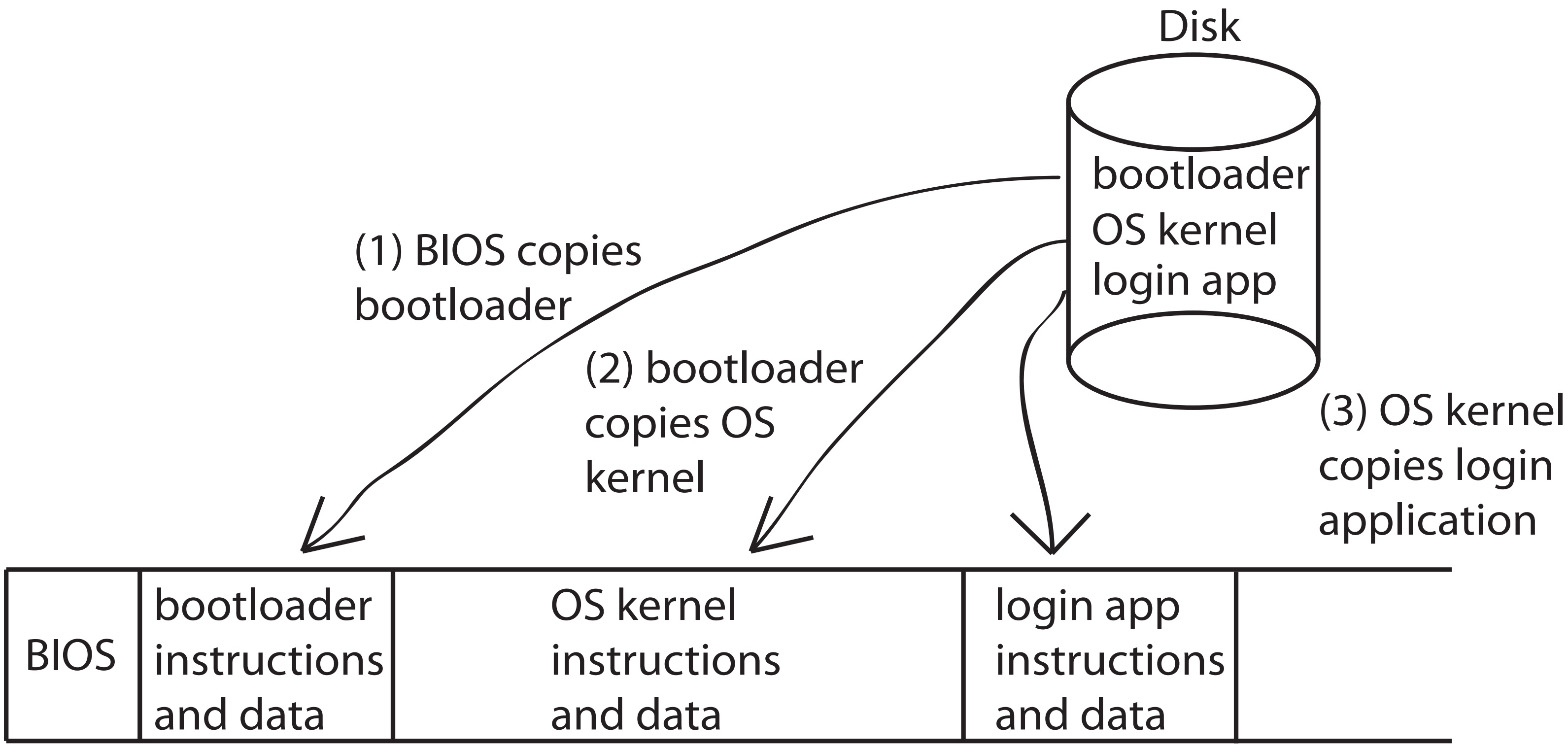
7. disk data (DMA)

Hardware

Network Interface

Disk Interface

Booting



Disk

bootloader
OS kernel
login app

(1) BIOS copies bootloader

(2) bootloader copies OS kernel

(3) OS kernel copies login application

BIOS

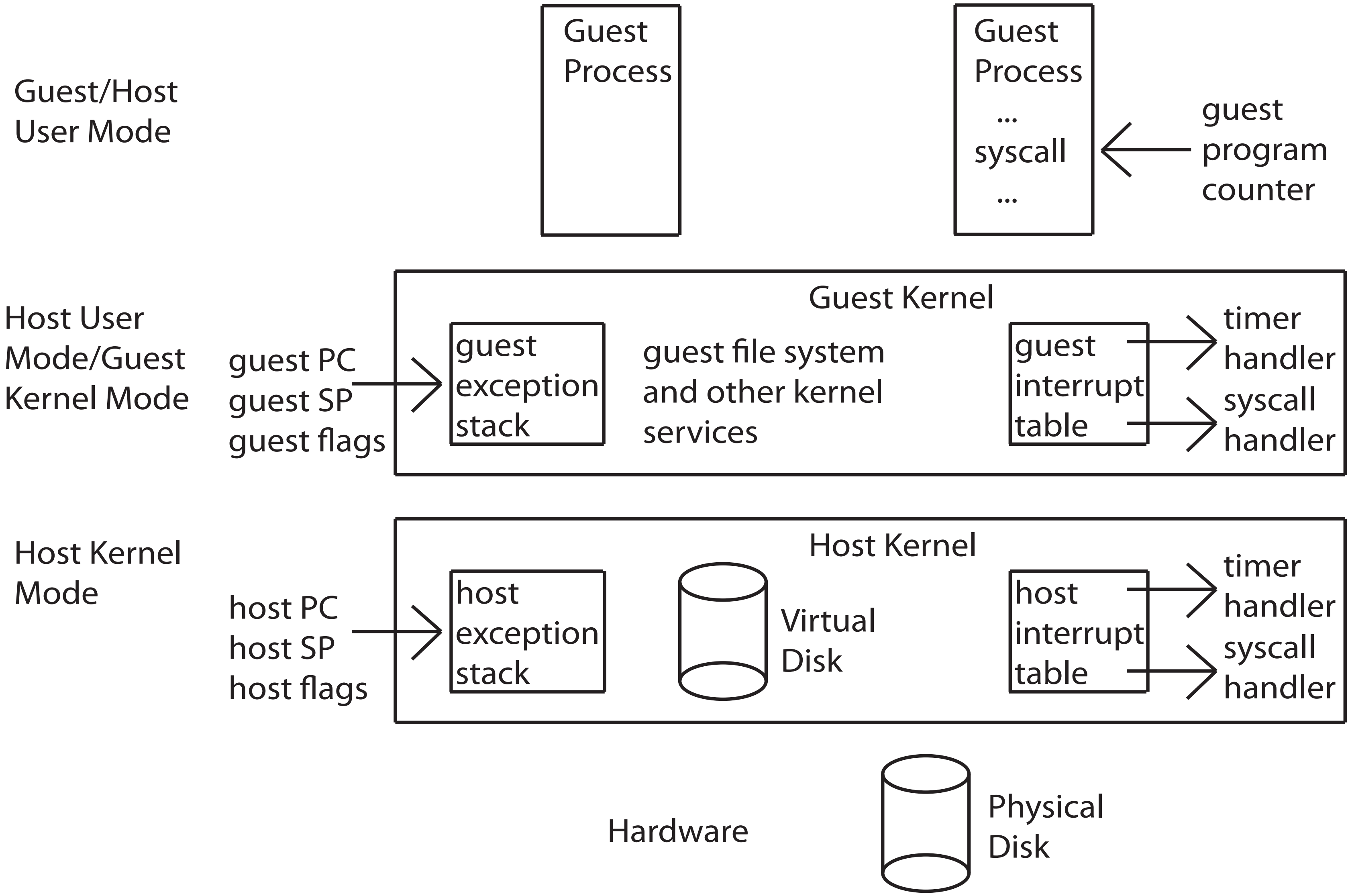
bootloader instructions and data

OS kernel instructions and data

login app instructions and data

Physical Memory

Virtual Machine



Guest/Host User Mode

Guest Process

Guest Process
…
syscall
…

guest program counter

Host User Mode/Guest Kernel Mode

guest PC
guest SP
guest flags

Guest Kernel

guest exception stack

guest file system and other kernel services

guest interrupt table

timer handler

syscall handler

Host Kernel Mode

host PC
host SP
host flags

Host Kernel

host exception stack

Virtual Disk

host interrupt table

timer handler

syscall handler

Hardware

Physical Disk