The Kernel Abstraction

Admin

- Pintos assignment 1
  - Code walk through in section
- Four slip days (for Pintos assignments, not problem sets)
  - Everything must be turned in 5pm, day before final
- Hack weeks
  - No class in week 8 and week 11, when Pintos 3 + 4 are due

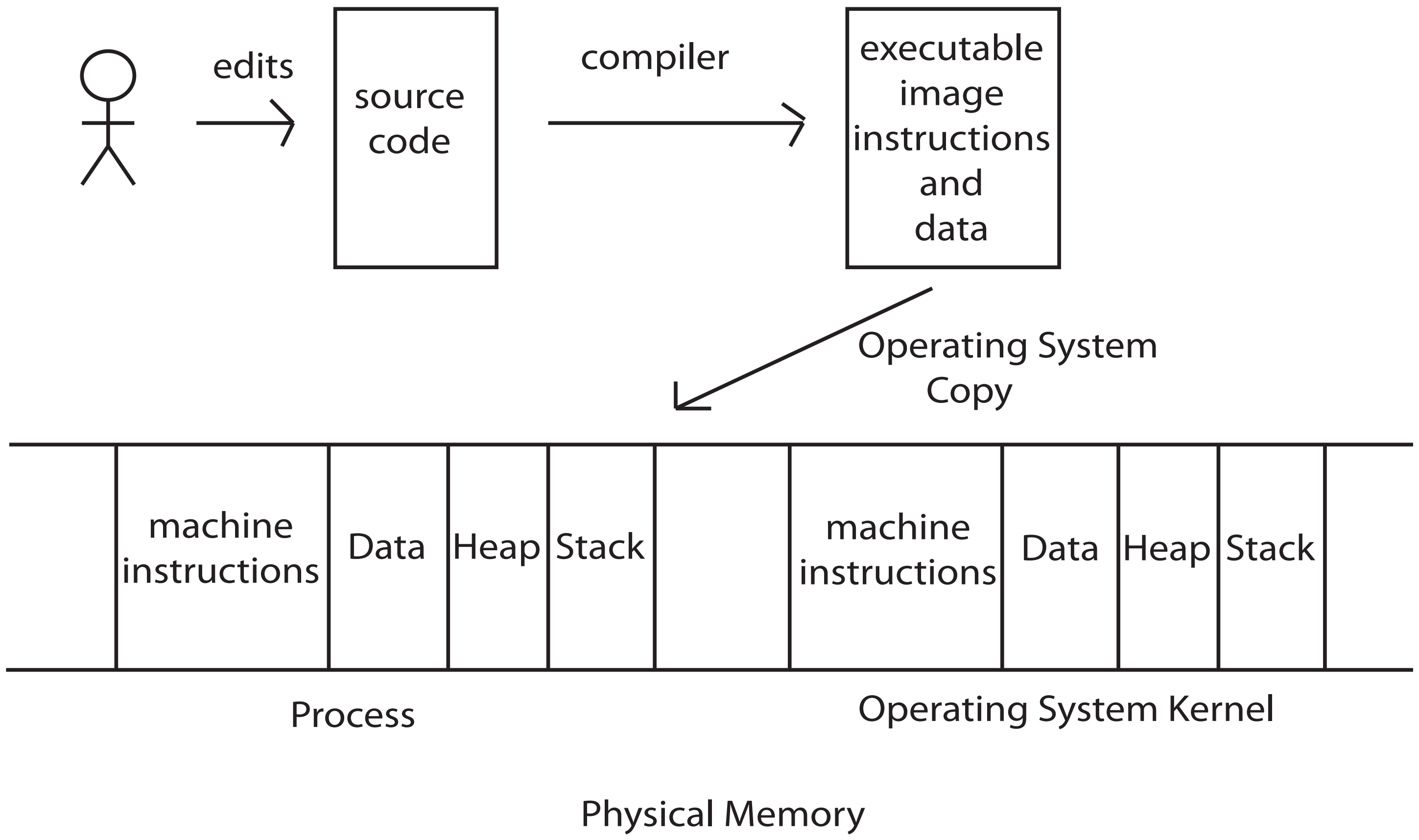
Challenge: Protection

- 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)

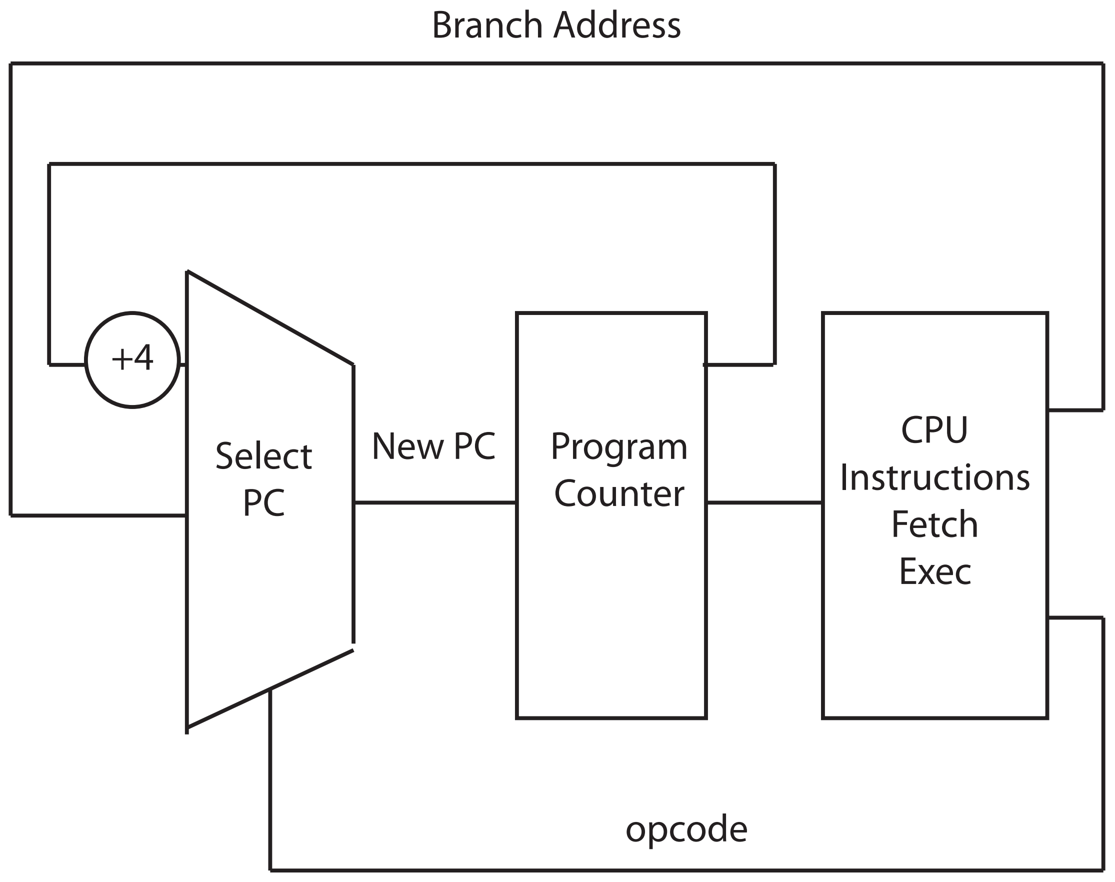
Thought Experiment

- 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?

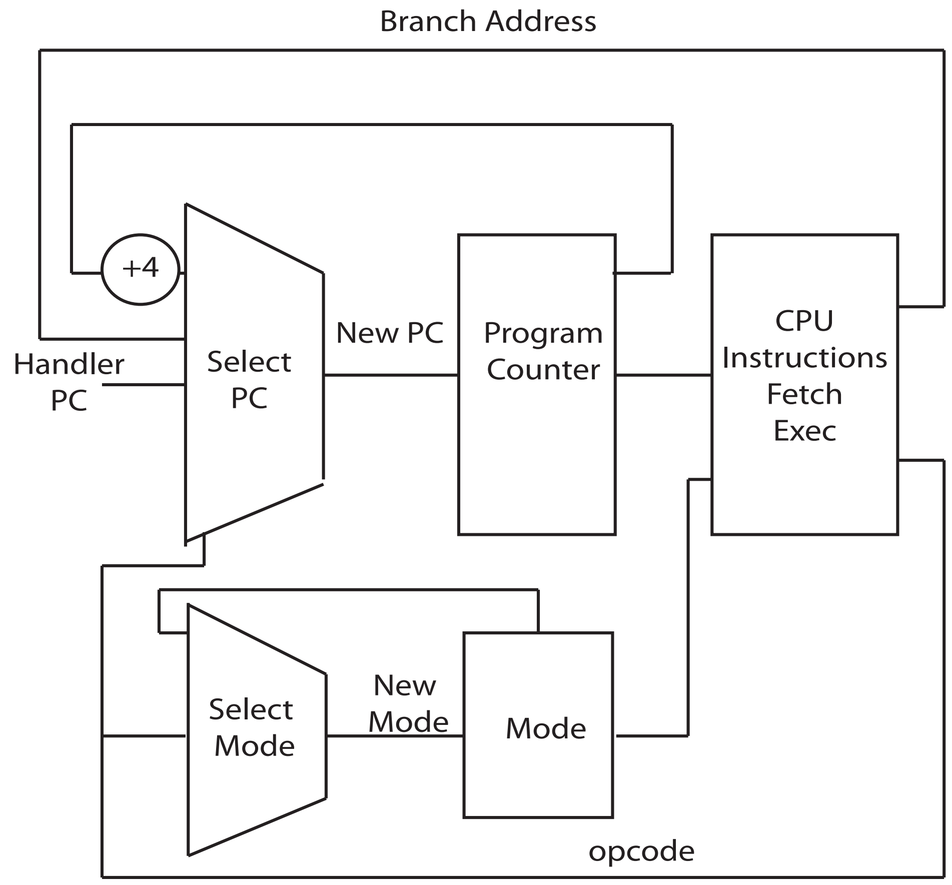
Hardware Support: Dual-Mode Operation

- 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

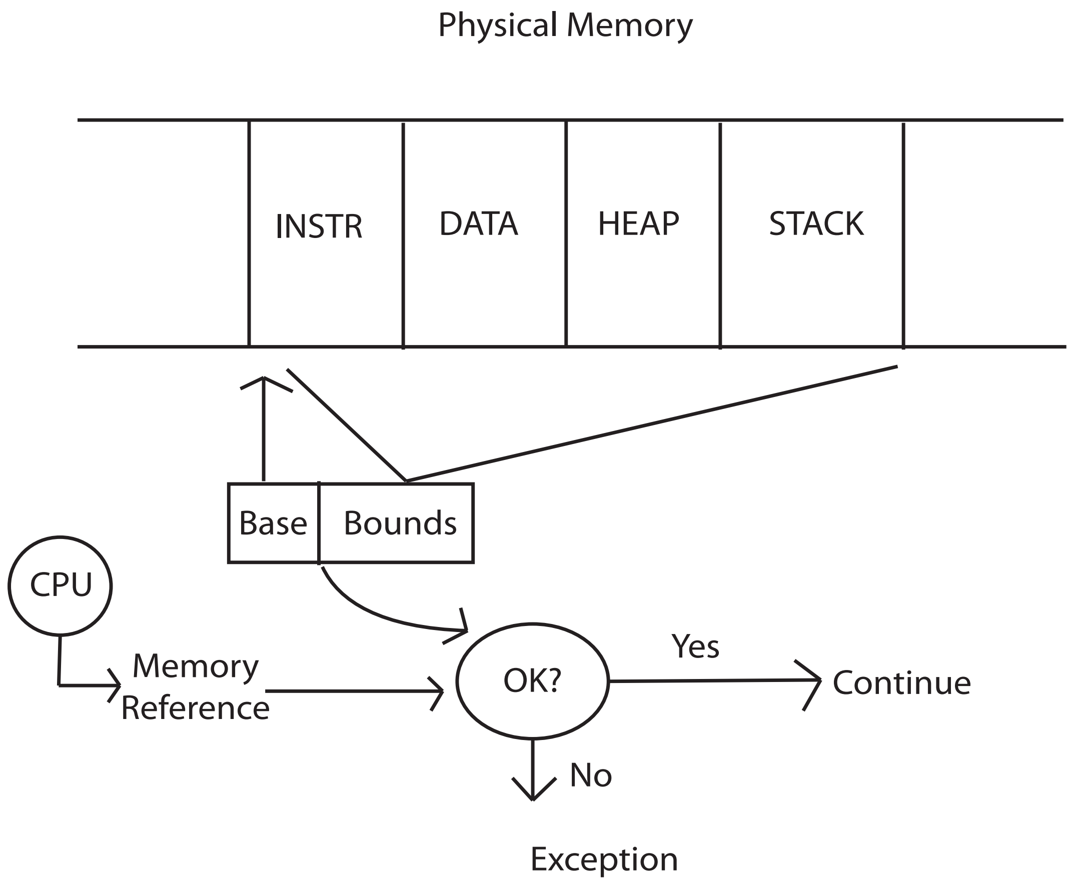
- Privileged instructions
  - Available to kernel
  - Not available to user code
- Limits on memory accesses
  - To prevent user code from overwriting the kernel
- Timer
  - To regain control from a user program in a loop
- Safe way to switch from user mode to kernel mode, and vice versa

Privileged instructions

- Examples?

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

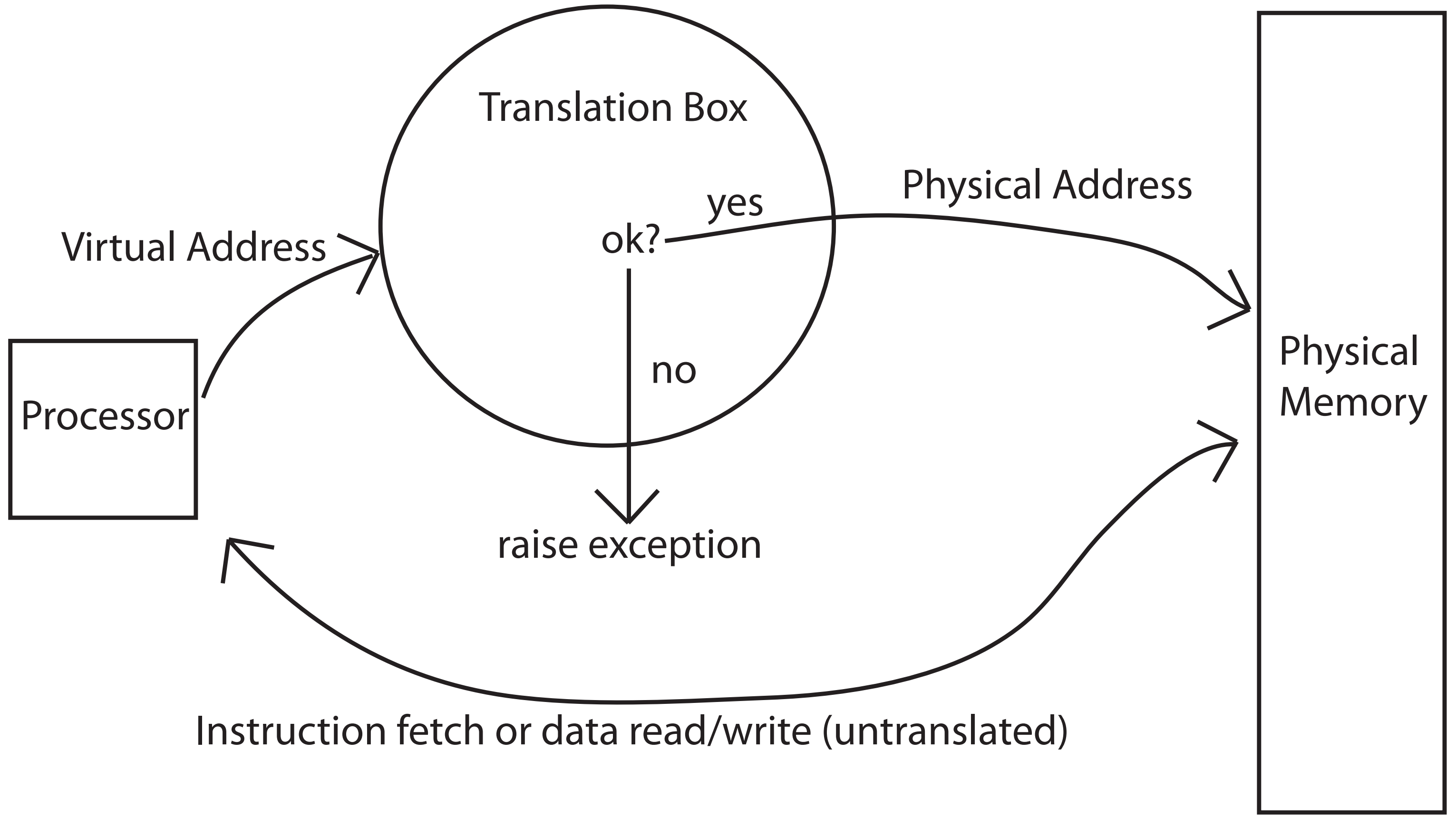
Memory Protection



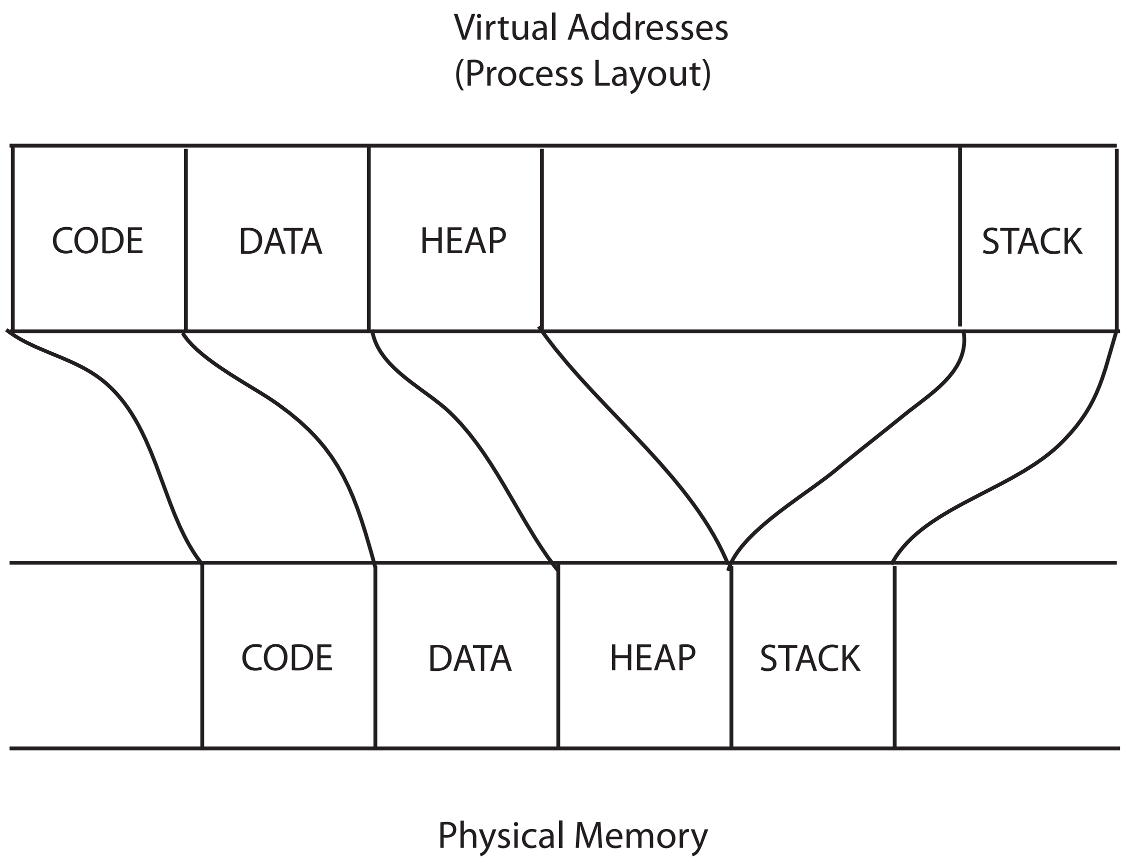
Towards Virtual Addresses

- Problems with base and bounds?

Virtual Addresses



Virtual Addresses



Example: Corrected (Why doesn’t this work?)

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

        staticVar += 1;
        localVar += 1;
        sleep(10);  /* this causes the program to wait for 10 seconds */
        printf (``static address: %x, value: %d\n'', &staticVar, staticVar);
        printf (``procedure local address: %x, value: %d\n'', &localVar, localVar);
}

> 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
  - Can be temporarily deferred (by a privileged instruction!)
    - 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

- For each of the three mechanisms for supporting dual mode operation — privileged instructions, memory protection, and timer interrupts — explain what might go wrong without that mechanism, assuming the system still had the other two.

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?

Safe Mode Switch

- From user-mode to kernel
  - Interrupts
    - Triggered by timer and I/O devices
  - Exceptions
    - Triggered by unexpected program 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

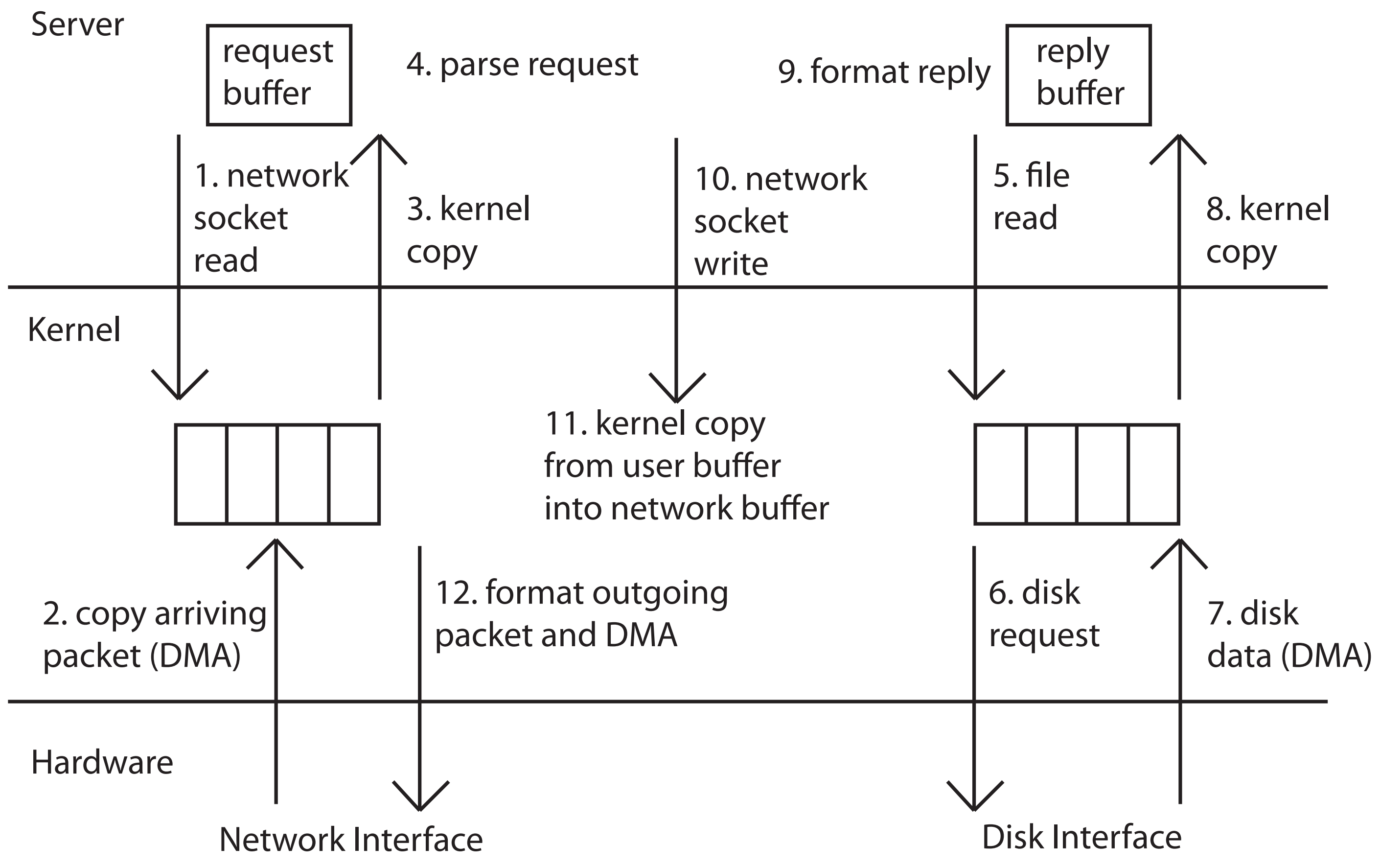
Safe Mode Switch

- From kernel-mode to user
  - New process start
    - Jump to first instruction in program
  - 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

Mode Switch Requirements

- Limited number of kernel entry points
  - Each must be protected against malicious or buggy behavior by user programs
- Atomic transfer
  - Single instruction to change:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode
- Transparent restartable execution
  - User program does not know timer interrupt occurred

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