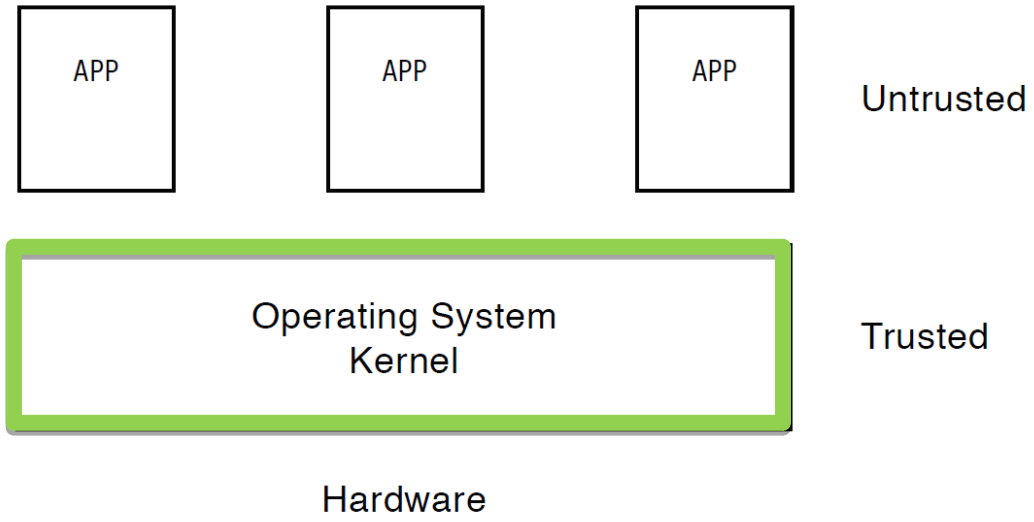
Introduction to Operating Systems

CPSC/ECE 3220 Summer 2018

Lecture Notes
OSPP Chapter 2 – Part A

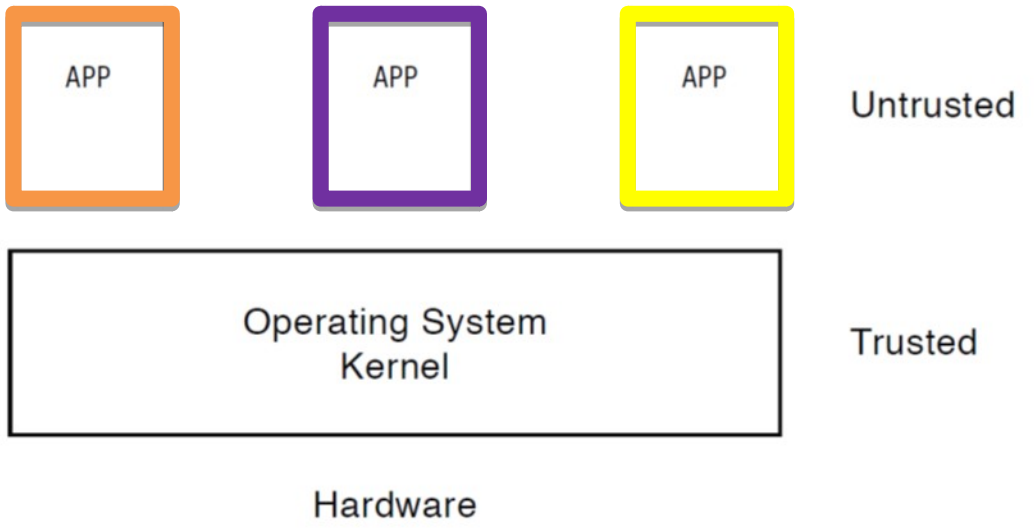
(adapted by Mark Smotherman from Tom Anderson's slides on OSPP web site)

Kernel



- Lowest level of software running on the system
- Purpose is to implement protection
- Kernel is fully trusted and has access to all the hardware

Untrusted Code



- We should restrict privileges of untrusted code
  - Should not have access to all the hardware
  - Should not have ability to modify the kernel or other applications

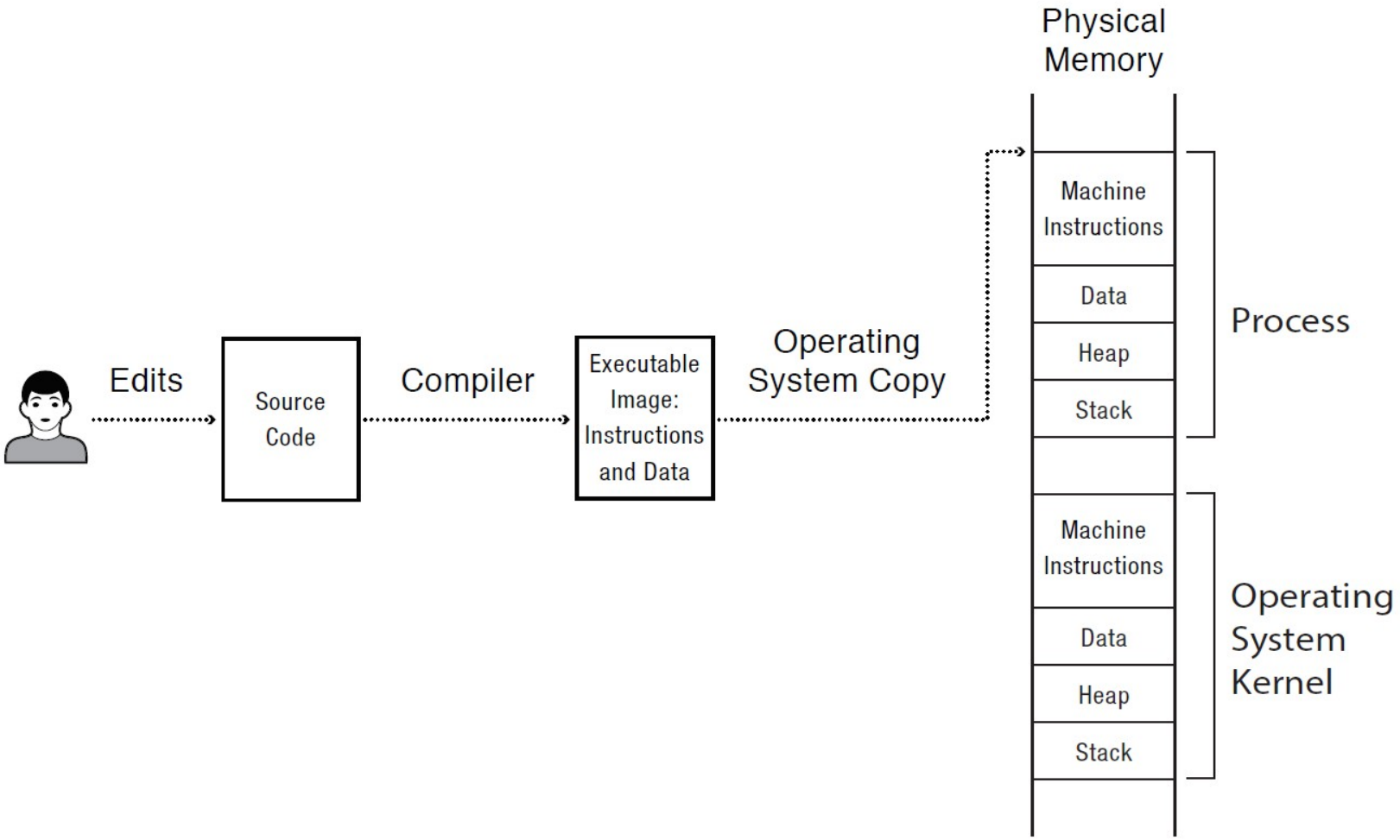
Challenge: Protection

- 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 fully tested or debugged

Stages of a Program

- Source file on disk
- Executable file on disk
  - After compiling and linking
  - Set of machine instructions (with a specified entry point) and initialized data
- Memory image
  - After loading
  - Stack, heap, and uninitialized data areas added to provide full execution environment

Compiling and Loading a Program



Main Points of Chapter 2

- Process concept
  - A process is the 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

Process Abstraction

- Process: an *instance* of a program, running with limited rights
  - Thread: an execution sequence within a process
    - Potentially many threads per process (for now 1:1)
  - Address space: the range of valid addresses
  - Allocated resources: the physical memory, files, and network connections the process can currently access
  - [in some systems] Capabilities: the permissions the process has (e.g., which system calls it can make, what objects it can access and how)

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 and other interpreted languages
- 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/rcv any packet
- User mode
  - Limited privileges
  - Only those granted by the operating system kernel
- Mode bit stored in processor status register

  (on x86, mode bits are stored in the CS register)

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
- Need safe way to switch from user mode to kernel mode, and vice versa

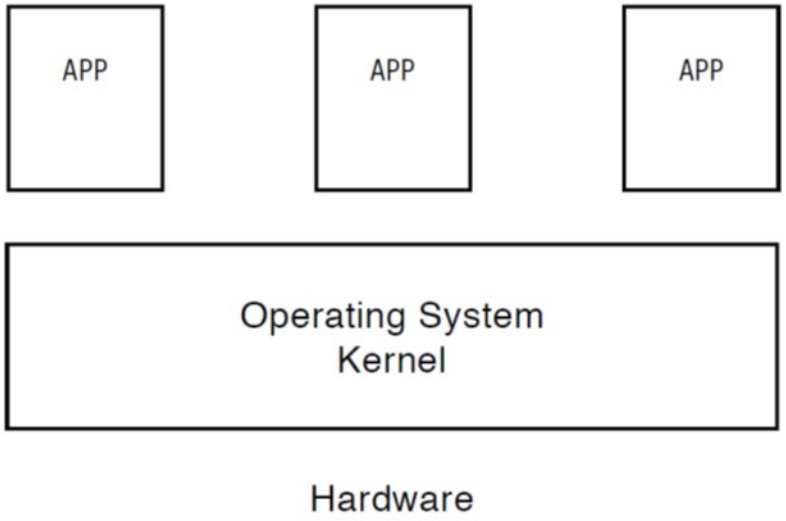
Privileged instructions

- Examples
  - Change mode bit in processor status register
  - Change which memory locations a user program can access
  - Send commands to I/O devices
  - Jump into kernel code
- What should happen if a user program attempts to execute a privileged instruction?

Non-Privileged (“Safe”) Instructions

- Examples
  - Load, store
  - Add, subtract, …
  - Conditional branch, jump to subroutine, …
- Allowed to execute in both kernel and user mode
  - OS and applications all need the ability to add numbers!
  - OS and applications all need the ability to use loops and call subroutines!

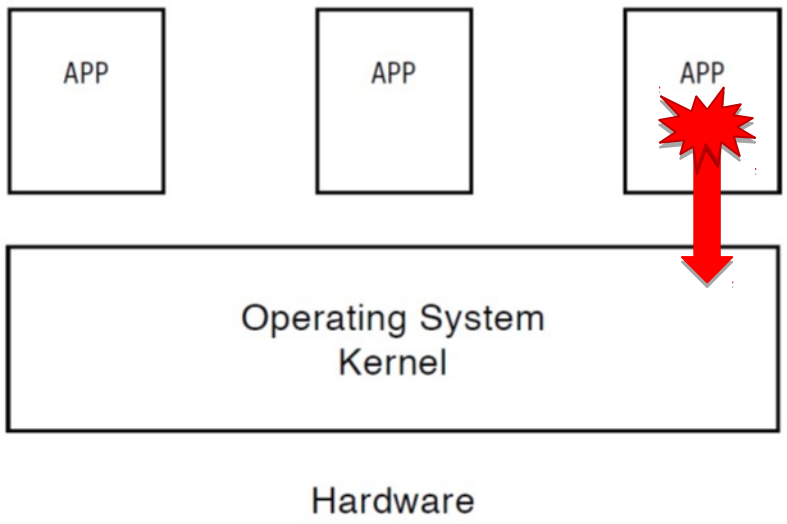
Valid Instructions



User mode: non-privileged (“safe”) instructions only

Kernel mode: both privileged and non-privileged instructions

Invalid Instructions



Attempt to execute a privileged instruction in user mode?

Stop the application and alert the OS!

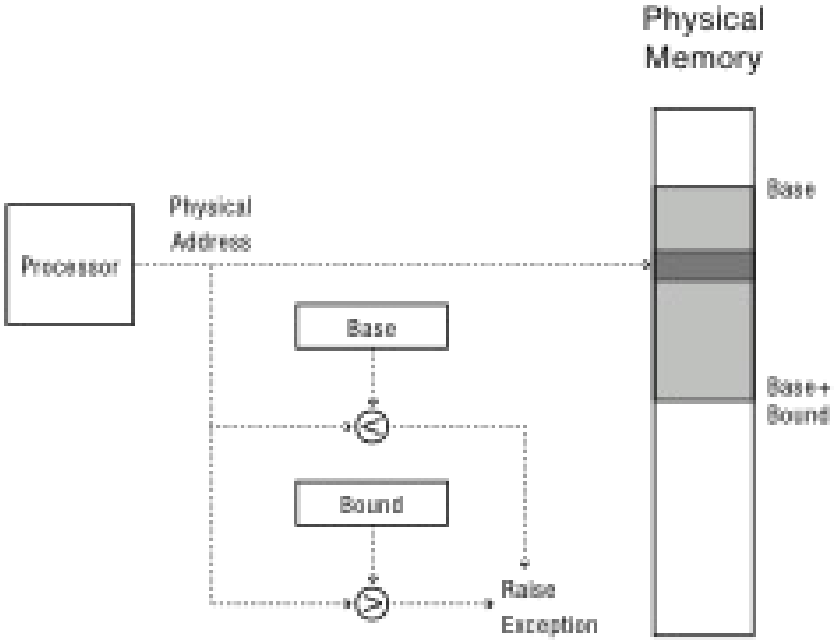
Typically the attempt is an error, but for selected instructions we will use this response to intentionally invoke the OS

(Note: could be an inefficient way to implement a virtual machine that is running a guest OS completely in user mode)

Question

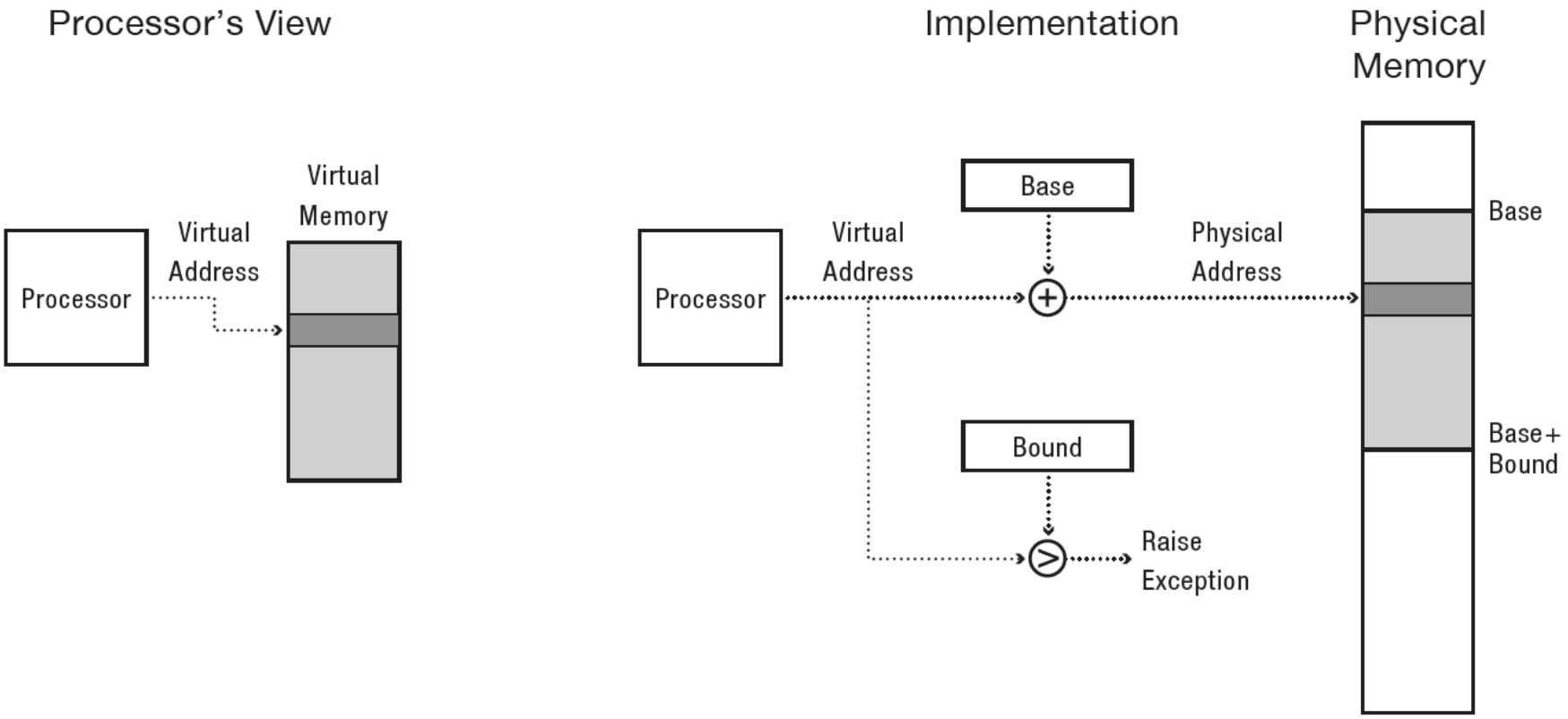
- For a “Hello world” program, the kernel must copy the string from the user program memory into the screen memory
- Why not allow the application to write directly to the screen’s buffer memory?

Simple Memory Protection



- Physical address generated by processor is range-checked against top and bottom limits

A Better Approach



- Virtual address relative to 0 checked against bound
- Virtual address added to contents of base register

Even Better Approaches!

- Problems with the additive base and bounds?
  - Memory sharing between processes? (e.g., how to share one copy of same machine instructions)
  - Memory fragmentation as processes come and go
- Paging and segmentation in Chapter 8
  - Translation done in hardware, using a table for each process
  - Table is set up and managed by kernel

Test Program – Determine if OS Runs with Virtual Addresses

```
#include<stdio.h>

int staticVar = 0;      // a static variable

main() {

    staticVar += 1;

    // sleep causes the program to wait for x seconds
    sleep(10);

    printf ("Address: %p; Value: %d\n", &staticVar, staticVar);
}
```

Script to Run Processes in Parallel

```
% cat test.script

./a.out &
./a.out &
./a.out &
wait
echo all done
```

```
% csh < test.script
[1] 9453
[2] 9454
[3] 9455
Address: 0x60104c; Value: 1
Address: 0x60104c; Value: 1
Address: 0x60104c; Value: 1
[2]    Exit 28
./a.out
[1]    Exit 28
./a.out
[3]    Exit 28
./a.out
all done
```