Introduction to OS

cCPSC/ECE 3220

Lecture Notes

OSPP Chapter 2 – Part A

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

What did we talk about last time?

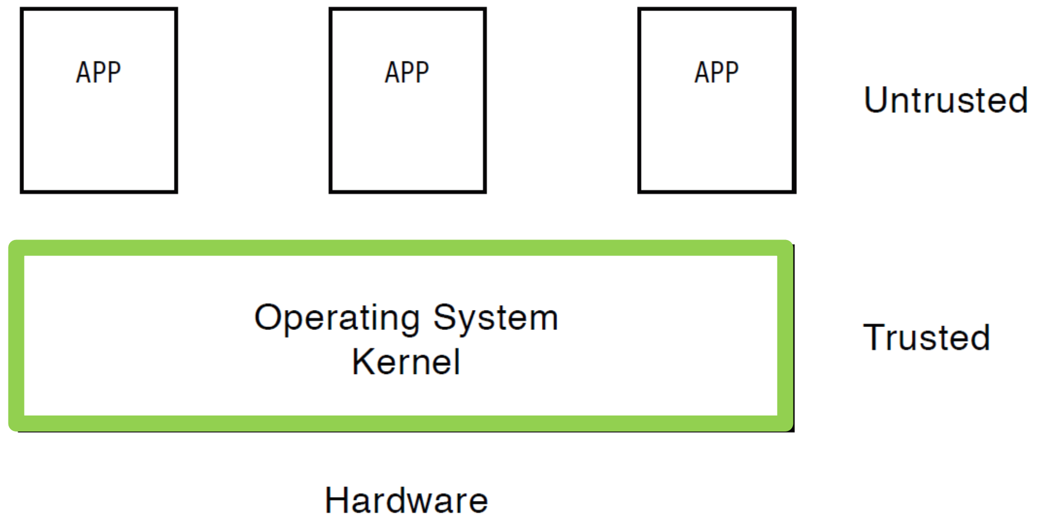
- What is an OS?
- What kinds of OS did we learn about?
- What are the three roles of an OS?

Quote of the Day

In real open source, you have the right to control your own destiny.

Linus Torvalds

Kernel

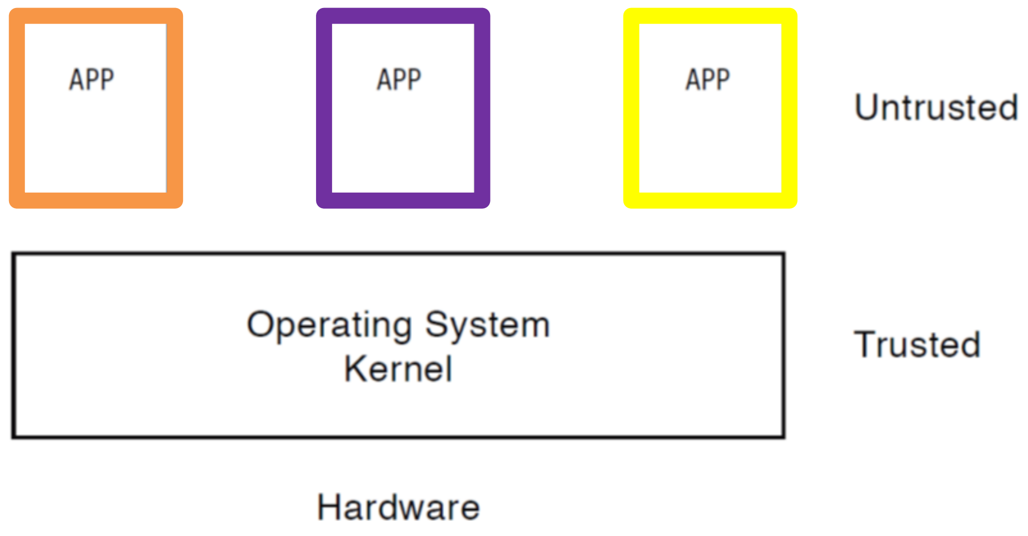


- Lowest level of OS software running on the system

- Purpose is to implement protection of users and apps from being corrupted by other users or apps

- Kernel is fully trusted and has access to all the hardware

Untrusted Code



- We should restrict privileges of untrusted code:
  - access to all the hardware
  - ability to modify the kernel or other applications

Why have a kernel?

OS should be able to preform its functions

- Reliability
  - OS has to protect itself from malicious and buggy code
- Security
  - Malicious app can write to the disk, alter code, damage files, etc.
- Privacy
  - Can only access data you are allowed to access.
- Fair resource allocation
  - Limit resources and control access to resources.

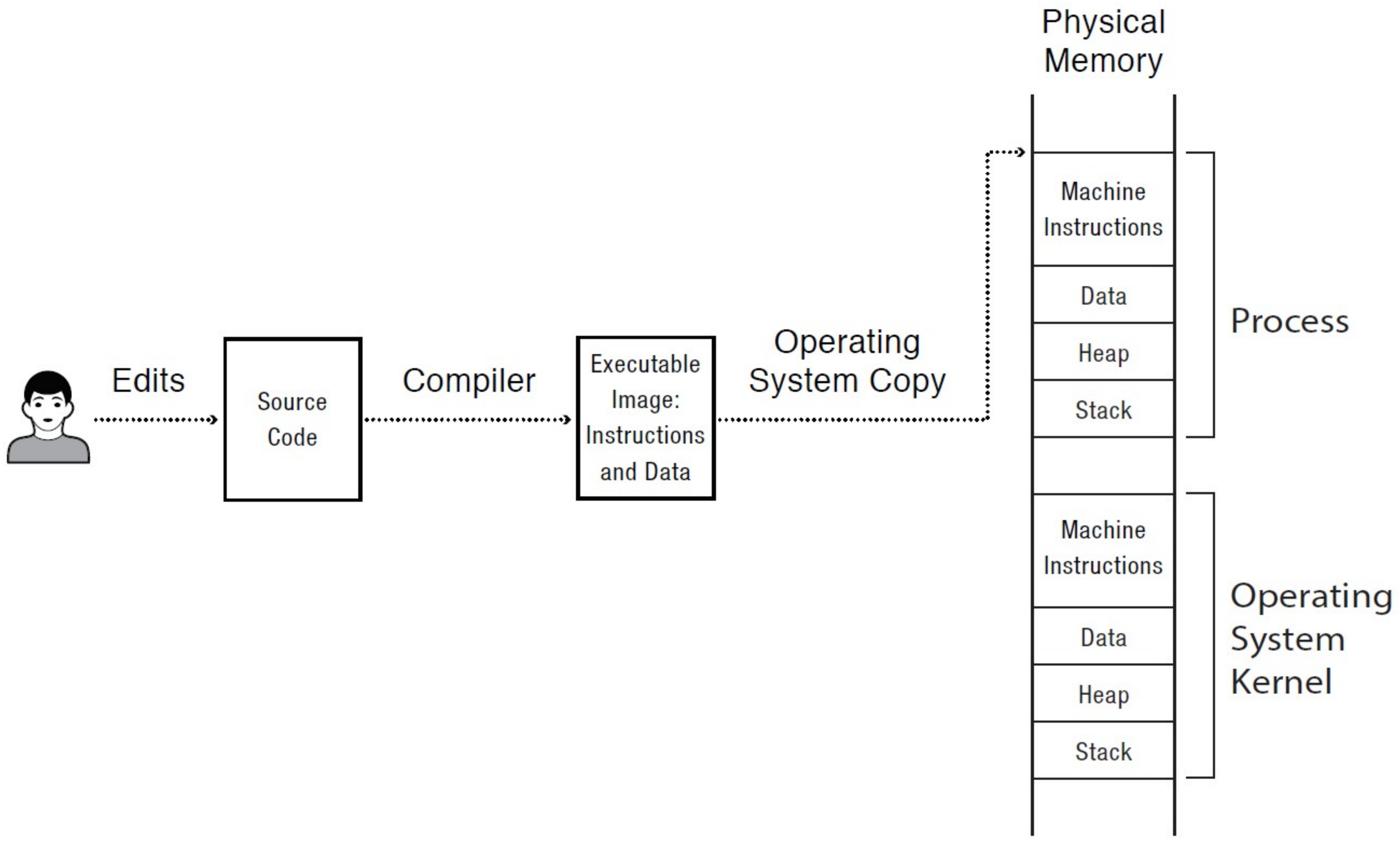
Challenge: Protection

- Why do we execute code with restricted privileges?
  - Code might be buggy
  - Code 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

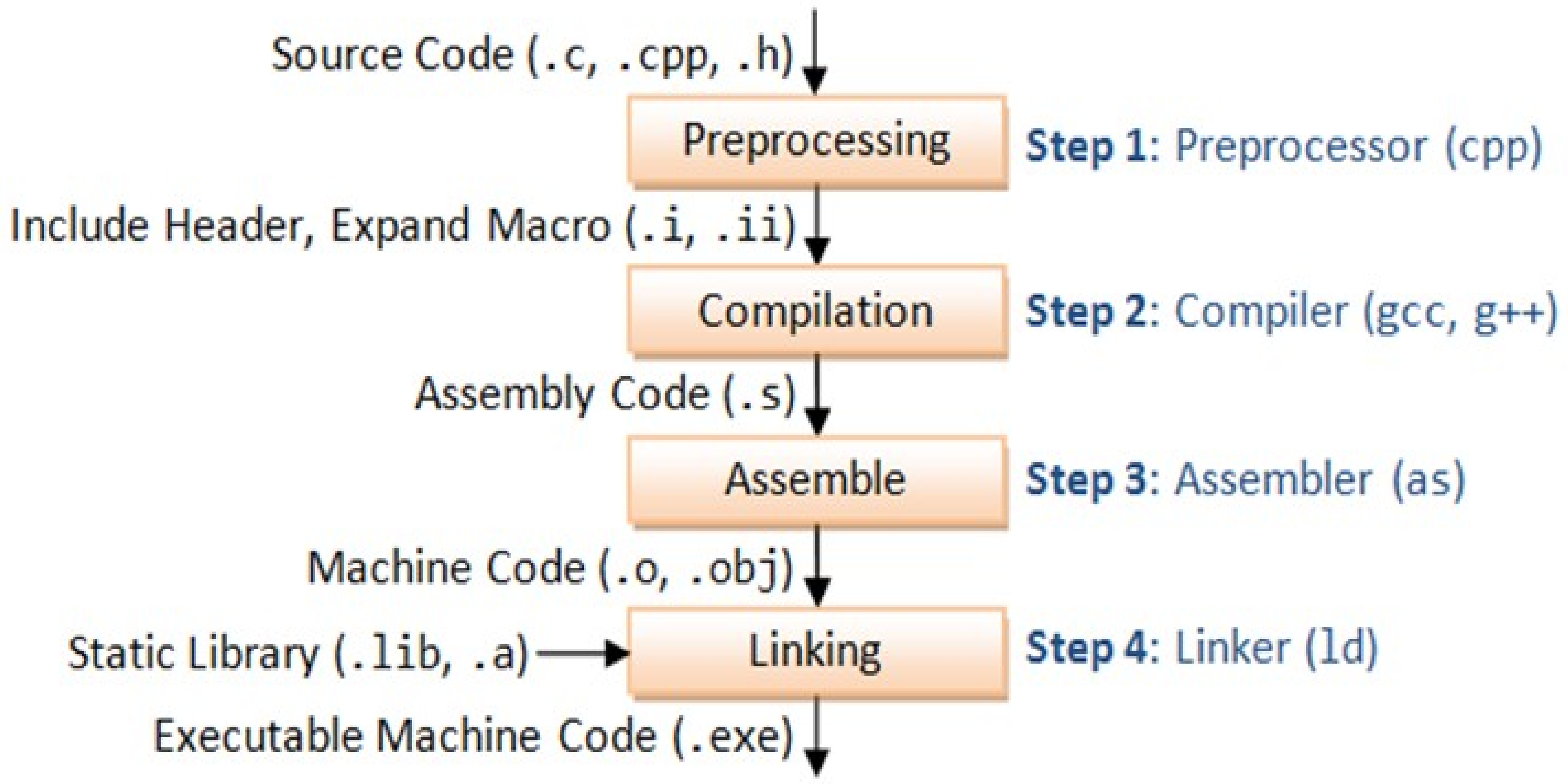
Life forms of a C Program

- Source file on a disk
- Executable file on a disk
  - After compiling and linking
  - Contains machine instructions and initialized data
- Memory image
  - After loading to memory
  - Stack, heap, and uninitialized data areas are added to provide a full execution environment

Compiling and Loading a Program



Steps in Compilation


Source Code (.c, .cpp, .h)
Preprocessing
Step 1: Preprocessor (cpp)
Include Header, Expand Macro (.i, .ii)
Compilation
Step 2: Compiler (gcc, g++)
Assembly Code (.s)
Assemble
Step 3: Assembler (as)
Machine Code (.o, .obj)
Static Library (.lib, .a)
Linking
Step 4: Linker (ld)
Executable Machine Code (.exe)

Source: C.H. Chuan

Process Abstraction

- **Program:** executable image = instructions + static data with init values.

- **Process:** an *instance* of a program, running with limited rights
  - Loaded into memory + stack + heap
  - PCB (process control block) data structure (1 per process):
    - Process location in memory, executable location on disk
    - Process id, privilege, priority
    - Process state, owner, open files, processors used, etc.

- **Thread:** an instance of execution within a process
    - Shares code and data
    - Owns stack and program counter

Linux task_struct (PCB struct)

```
struct task_struct {

  volatile long state;  /* -1 unrunnable,
                        0 runnable, >0 stopped */
  void *stack;
  int oncpu;
  int prio;
  pid_t pid;
  struct list_head children; /* list of children */
  struct files_struct *files;
  struct task_struct *real_parent; /* real parent*/
  struct task_struct *parent; /* new parent */

  .........

}
```

Checking process info in Linux

- Display running processes on a terminal:
  - **ps** displays a *snapshot* of user processes
  - **top** or **htop** display *all live* processes

- Display specific process info:
  - **/proc** directory is populated in real time
  - Look at **/proc/pid#/status** file (must be admin)
    - pid# is the id of the process of interest

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
  - Check each instruction before executing
  - Run only those granted by the kernel
  - Principle of least privilege!!!

- Process isolation
  - User mode bit stored in processor status register (PSR)
  - PSR is not accessible to user applications (only kernel)

Dual-Mode Operation

- To ensure protection from other applications and users and run directly on a CPU we need:
  - Privileged instructions
    - Available to kernel, but not user code
  - Limits on memory accesses
    - To prevent user code from accessing other process' data and 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

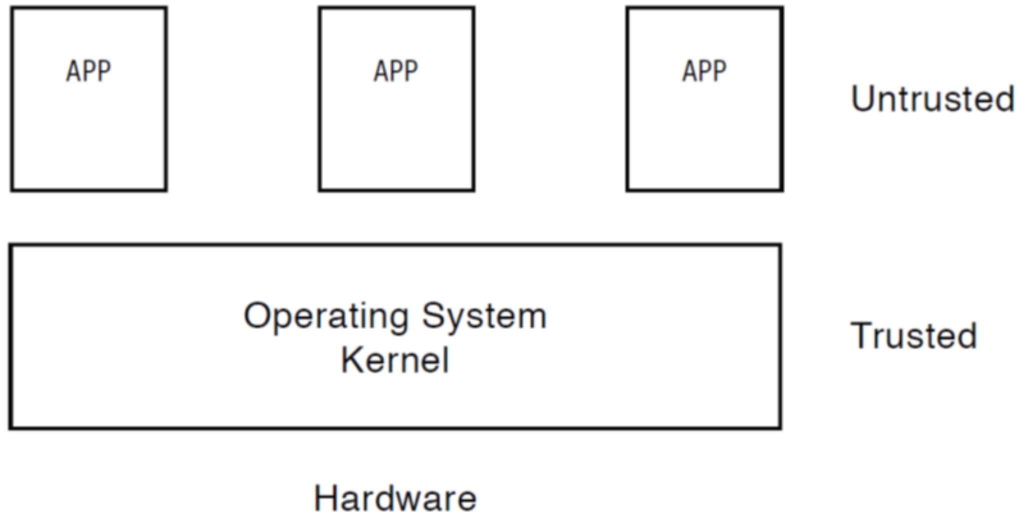
Privileged instructions

- Examples:
  - Change mode bit in PSR
  - Change memory locations that user program can access
  - Send commands to I/O devices
  - Jump into kernel code
- What happens if a user program attempts to execute a privileged instruction?
  - Processor exception
  - Control transferred to OS exception handler
  - Process holts

Non-Privileged (“Safe”) Instructions

- *Examples:*
  - Load, store
  - Add, subtract, …
  - Conditional branch, jump to subroutine, …
- Kernel also executes them:
  - OS and applications all need the ability to add numbers!
  - OS and applications all need the ability to use loops and call subroutines!

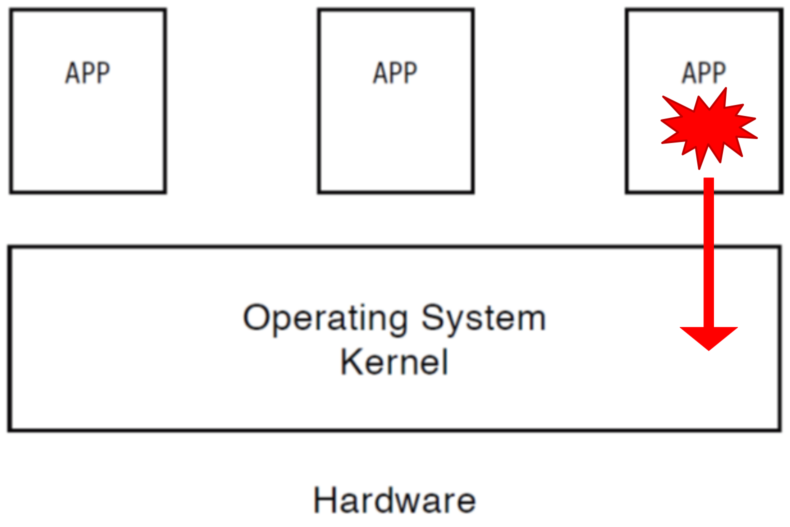
Valid Instructions



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

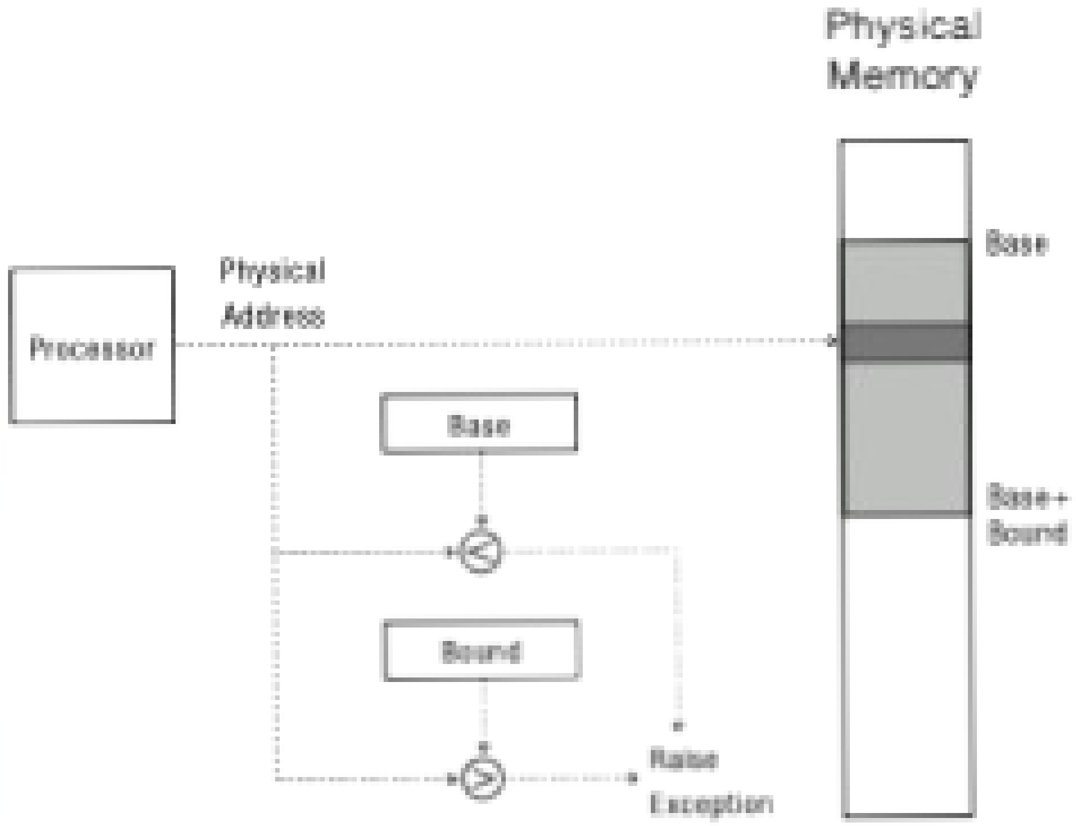
Memory Protection

- The role of OS is both security and privacy.

- Is read-only access harmless ?
  - File system buffer can contain user data
  - Memory can contain passwords

- How can you control memory access?
  - Many approaches

Early Memory Protection



- Base register – the start of memory region in physical memory
- Bounds register – the end of memory region in physical memory
- User-level code cannot change them

- Kernel executes without base/bound registers
- Physical address generated by processor is range-checked against top and bottom limits

Early Scheme's Missing Features

- Expendable Stack and Heap
  - Grow towards each other

- Memory Sharing
  - Processes running the same program
  - Processes share the same library

- Memory addresses
  - Executable file addresses start at 0
  - Addresses of procedures and global variables need to be adjusted when program is loaded into physical memory

- Memory fragmentation
  - Multiple processes running can fragment memory

A Better Approach

- Virtual addressing – a level of indirection
  - Every process’ virtual memory starts at zero
  - Hardware translates virtual to physical addresses (virtual address + base register)
  - Stack and heap can be moved to a larger chunk of physical address if needed, virtual address is still the same
  - Transparent to user process

Preview: Even Better Approaches!

- Paging and segmentation in Chapter 8
  - Translation done in hardware, using a table for each process
  - Table is set up and managed by kernel

Address Space Layout Randomization

- Most OSs randomize virtual addresses of stack and heap every time it runs
- Try to print a local variable address using the same code.