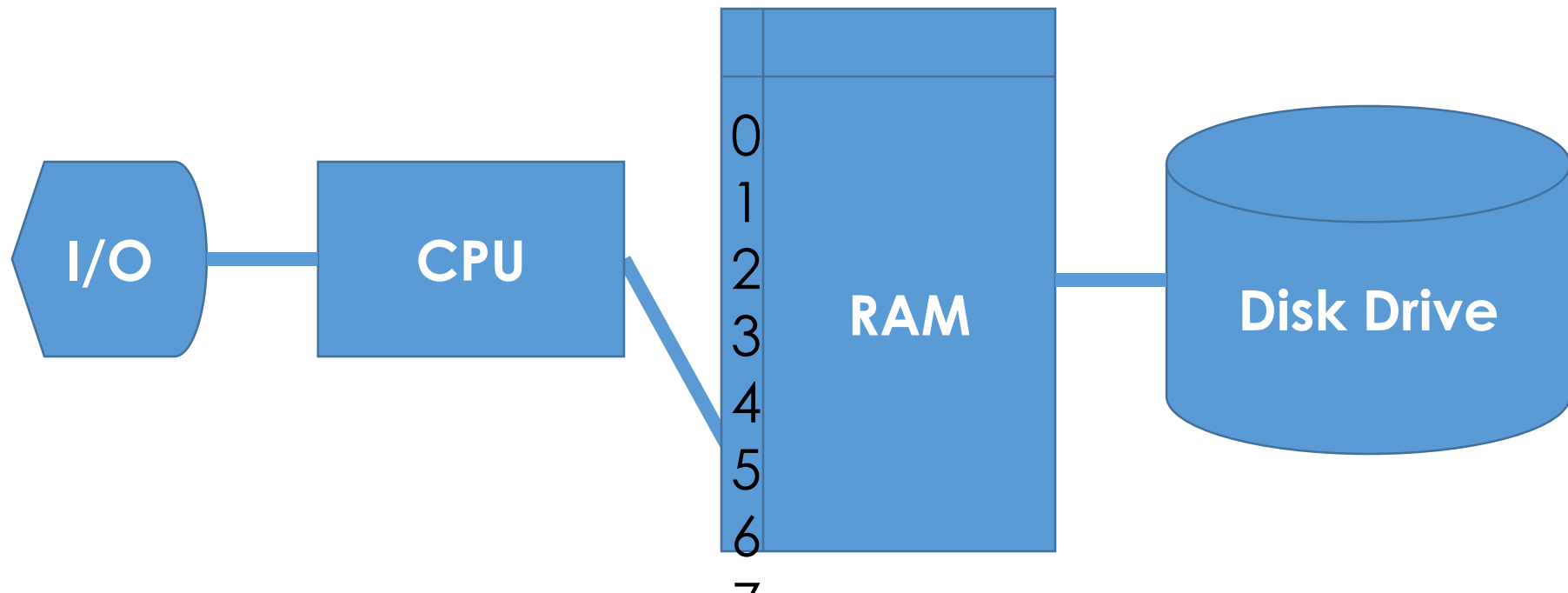

Operating Systems Concepts

A Computer Model

- An operating system has to deal with the fact that a computer is made up of a CPU, random access memory (RAM), input/output (I/O) devices, and long-term storage.



OS Concepts

- An **operating system (OS)** provides the interface between the users of a computer and that computer's hardware.
 - An operating system manages the ways applications access the resources in a computer, including its disk drives, CPU, main memory, input devices, output devices, and network interfaces.
 - An operating system manages multiple users.
 - An operating system manages multiple programs.

Multitasking

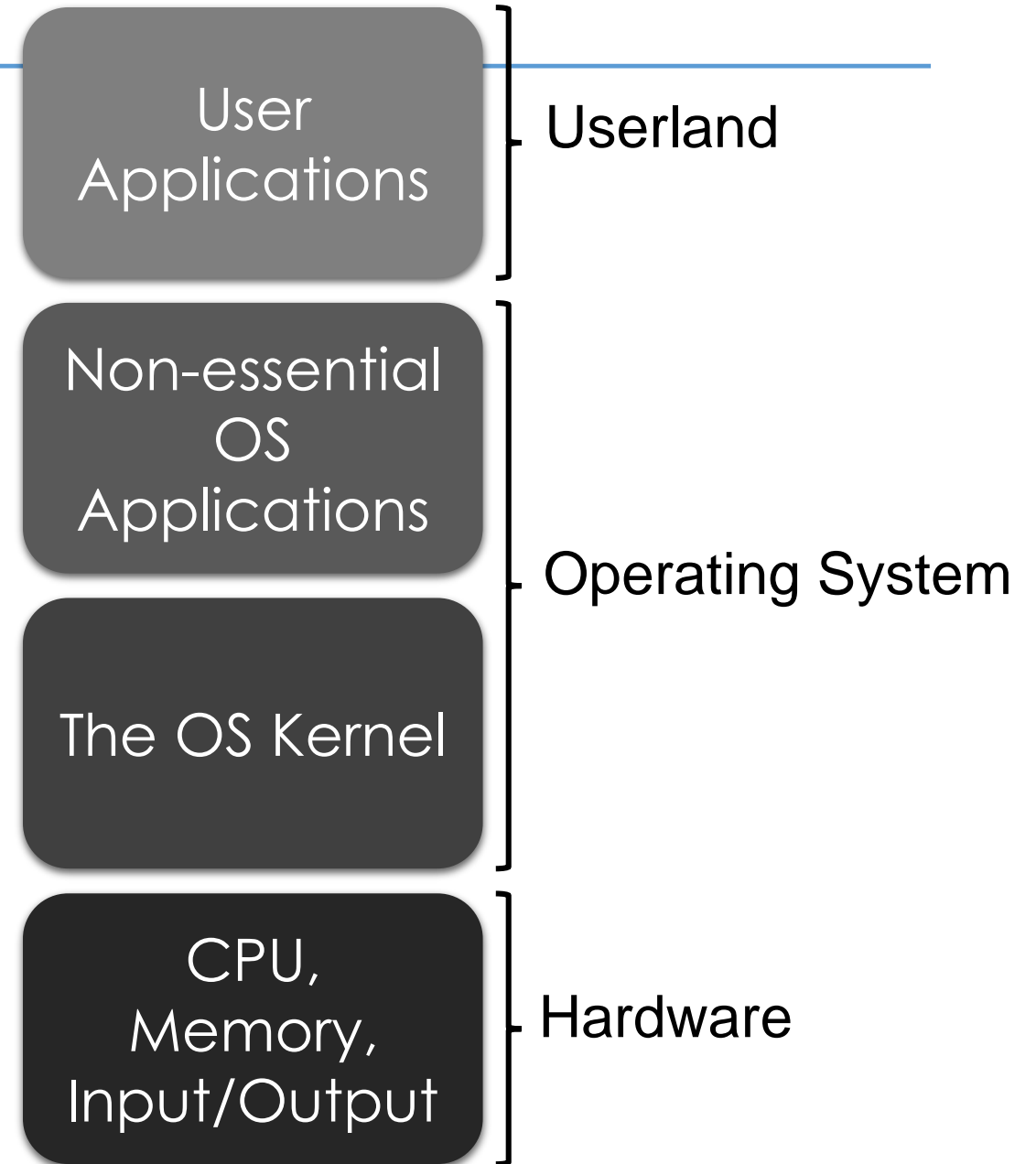
- Give each running program a “slice” of the CPU's time.
- The CPU is running so fast that to any user it appears that the computer is running all the programs simultaneously.



Public domain image from
http://commons.wikimedia.org/wiki/File:Chapters_meeting_2009_Liam_juggling.JPG

The Kernel

- The **kernel** is the core component of the operating system. It handles the management of low-level hardware resources, including memory, processors, and input/output (I/O) devices, such as a keyboard, mouse, or video display.
- Most operating systems define the tasks associated with the kernel in terms of a **layer** metaphor, with the hardware components, such as the CPU, memory, and input/output devices being on the bottom, and users and applications being on the top.



Input/Output

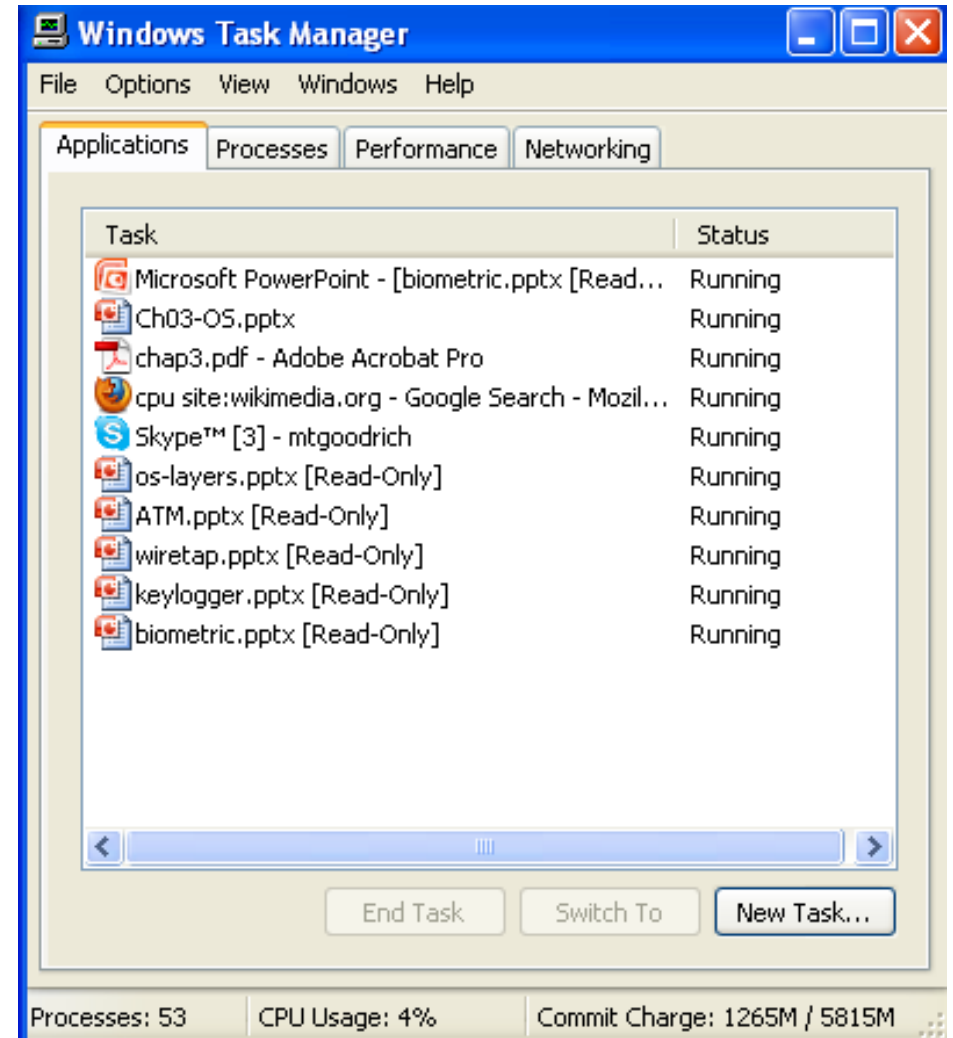
- The **input/output devices** of a computer include things like its keyboard, mouse, video display, and network card, as well as other more optional devices, like a scanner, Wi-Fi interface, video camera, USB ports, etc.
- Each such device is represented in an operating system using a **device driver**, which encapsulates the details of how interaction with that device should be done.
 - The **application programmer interface (API)**, which the device drivers present to application programs, allows those programs to interact with those devices at a fairly high level, while the operating system does the “heavy lifting” of performing the low-level interactions that make such devices actually work.

System Calls

- User applications don't communicate directly with low-level hardware components, and instead delegate such tasks to the kernel via **system calls**.
- System calls are usually contained in a collection of programs, that is, a **library** such as the C library (libc), and they provide an interface that allows applications to use a predefined series of APIs that define the functions for communicating with the kernel.
 - Examples of system calls include those for performing file I/O (open, close, read, write) and running application programs (exec).

Processes

- A **process** is an instance of a program that is currently executing.
- The actual contents of all programs are initially stored in persistent storage, such as a hard drive.
- In order to be executed, a program must be loaded into random-access memory (RAM) and uniquely identified as a process.
- In this way, multiple copies of the same program can be run as different processes.
 - For example, we can have multiple copies of MS Powerpoint open at the same time.



Process IDs

- Each process running on a given computer is identified by a unique nonnegative integer, called the **process ID (PID)**.
- Given the PID for a process, we can then associate its CPU time, memory usage, user ID (UID), program name, etc.

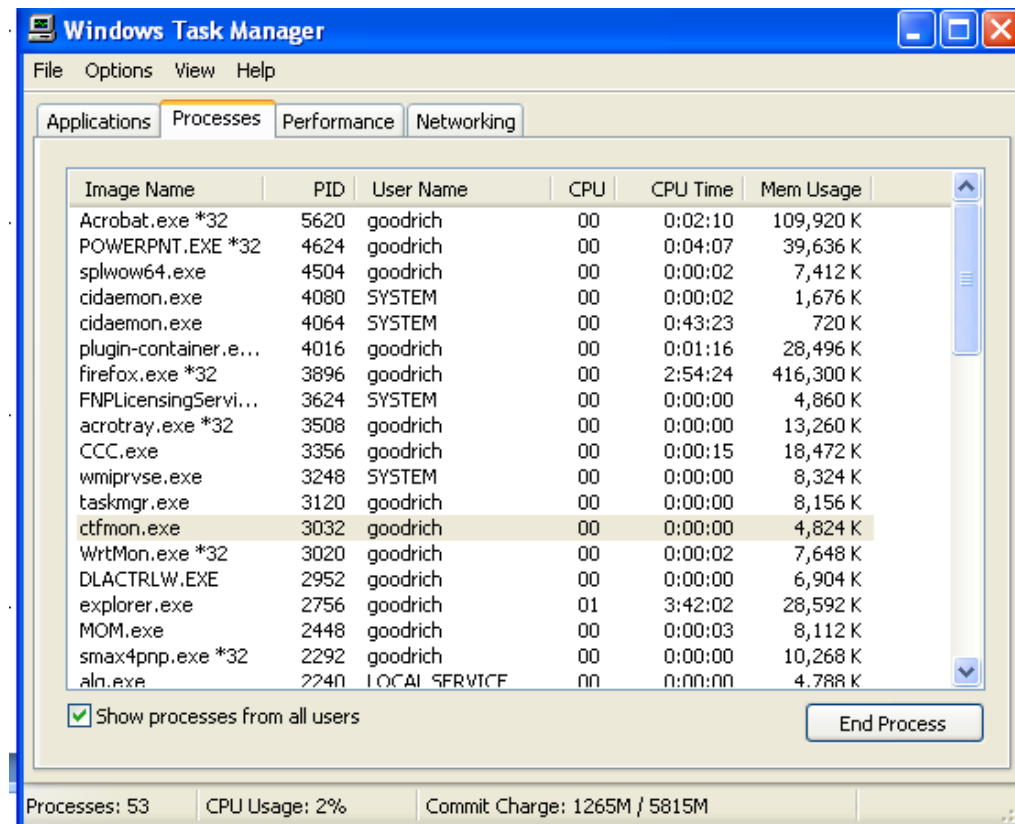


Image Name	PID	User Name	CPU	CPU Time	Mem Usage
Acrobat.exe *32	5620	goodrich	00	0:02:10	109,920 K
POWERPNT.EXE *32	4624	goodrich	00	0:04:07	39,636 K
splwow64.exe	4504	goodrich	00	0:00:02	7,412 K
cidaemon.exe	4080	SYSTEM	00	0:00:02	1,676 K
cidaemon.exe	4064	SYSTEM	00	0:43:23	720 K
plugin-container.exe	4016	goodrich	00	0:01:16	28,496 K
firefox.exe *32	3896	goodrich	00	2:54:24	416,300 K
FNPLicensingService.exe	3624	SYSTEM	00	0:00:00	4,860 K
acrotray.exe *32	3508	goodrich	00	0:00:00	13,260 K
CCC.exe	3356	goodrich	00	0:00:15	18,472 K
wmiprvse.exe	3248	SYSTEM	00	0:00:00	8,324 K
taskmgr.exe	3120	goodrich	00	0:00:00	8,156 K
ctfmon.exe	3032	goodrich	00	0:00:00	4,824 K
WrtMon.exe *32	3020	goodrich	00	0:00:02	7,648 K
DLACTRLW.EXE	2952	goodrich	00	0:00:00	6,904 K
explorer.exe	2756	goodrich	01	3:42:02	28,592 K
MOM.exe	2448	goodrich	00	0:00:03	8,112 K
smax4pnp.exe *32	2292	goodrich	00	0:00:00	10,268 K
aln.exe	2240	LOCAL SERVICE	00	0:00:00	4,788 K

☒ Show processes from all users

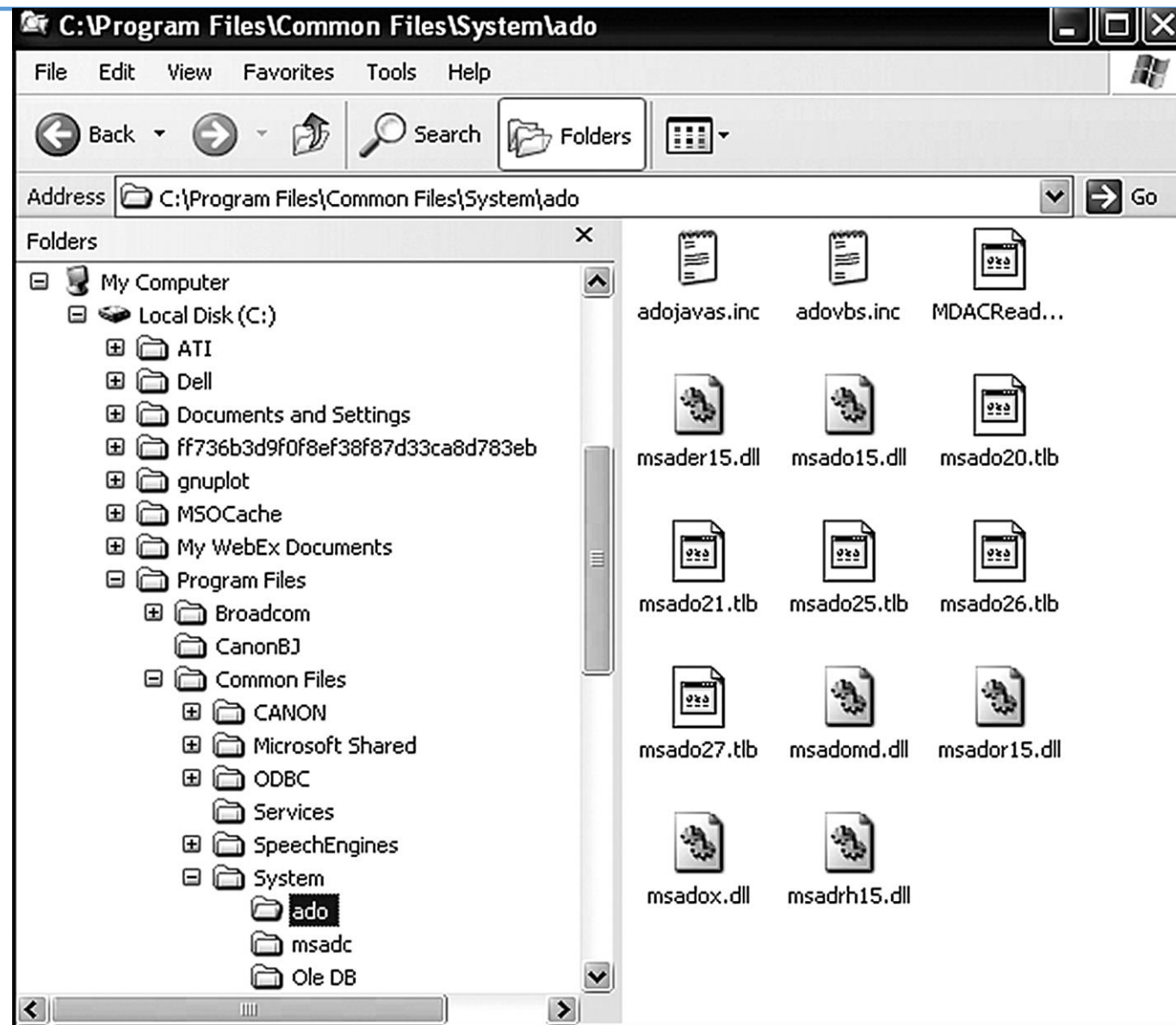
End Process

Processes: 53 CPU Usage: 2% Commit Charge: 1265M / 5815M

File Systems

- A **filesystem** is an abstraction of how the external, nonvolatile memory of the computer is organized.
- Operating systems typically organize files hierarchically into **folders**, also called **directories**.
- Each folder may contain files and/or subfolders.
- Thus, a volume, or drive, consists of a collection of nested folders that form a **tree**.
- The topmost folder is the **root** of this tree and is also called the root folder.

File System Example



File Permissions

- File permissions are checked by the operating system to determine if a file is readable, writable, or executable by a user or group of users.
- In Unix-like OS's, a **file permission matrix** shows who is allowed to do what to the file.
 - Files have **owner permissions**, which show what the owner can do, and **group permissions**, which show what some group id can do, and **world permissions**, which give default access rights.

```
rodan:~/java % ls -l
total 24
-rwxrwxrwx    1 goodrich faculty    2496 Jul 27 08:43 Floats.class
-rw-r--r--    1 goodrich faculty    2723 Jul 12  2006 Floats.java
-rw-----    1 goodrich faculty     460 Feb 25  2007 Test.java
rodan:~/java %
```

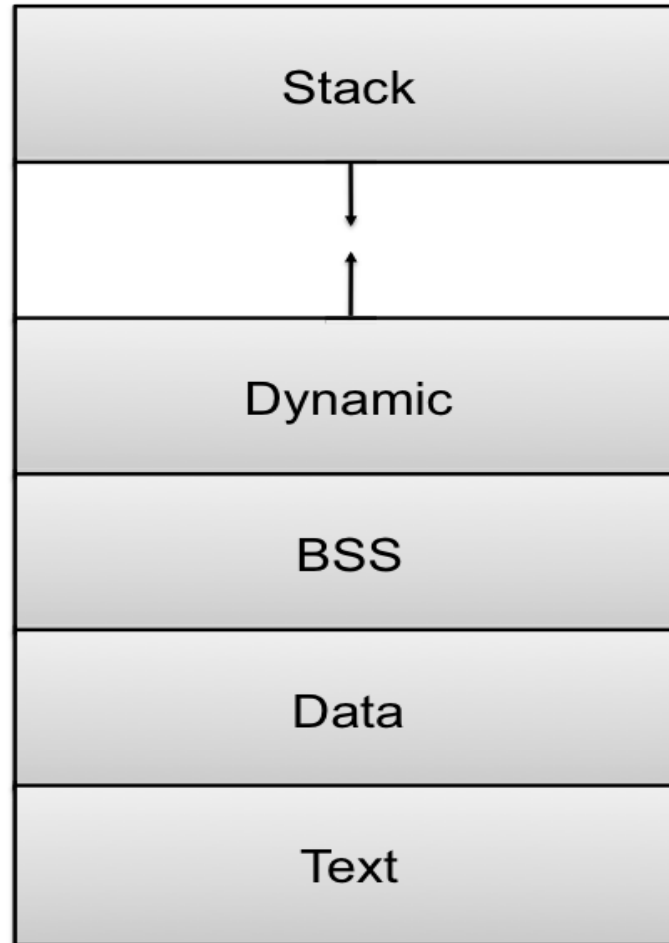
Memory Management

- The RAM memory of a computer is its **address space**.
- It contains both the code for the running program, its input data, and its working memory.
- For any running process, it is organized into different segments, which keep the different parts of the address space separate.
- As we will discuss, security concerns require that we never mix up these different segments.

Memory Organization

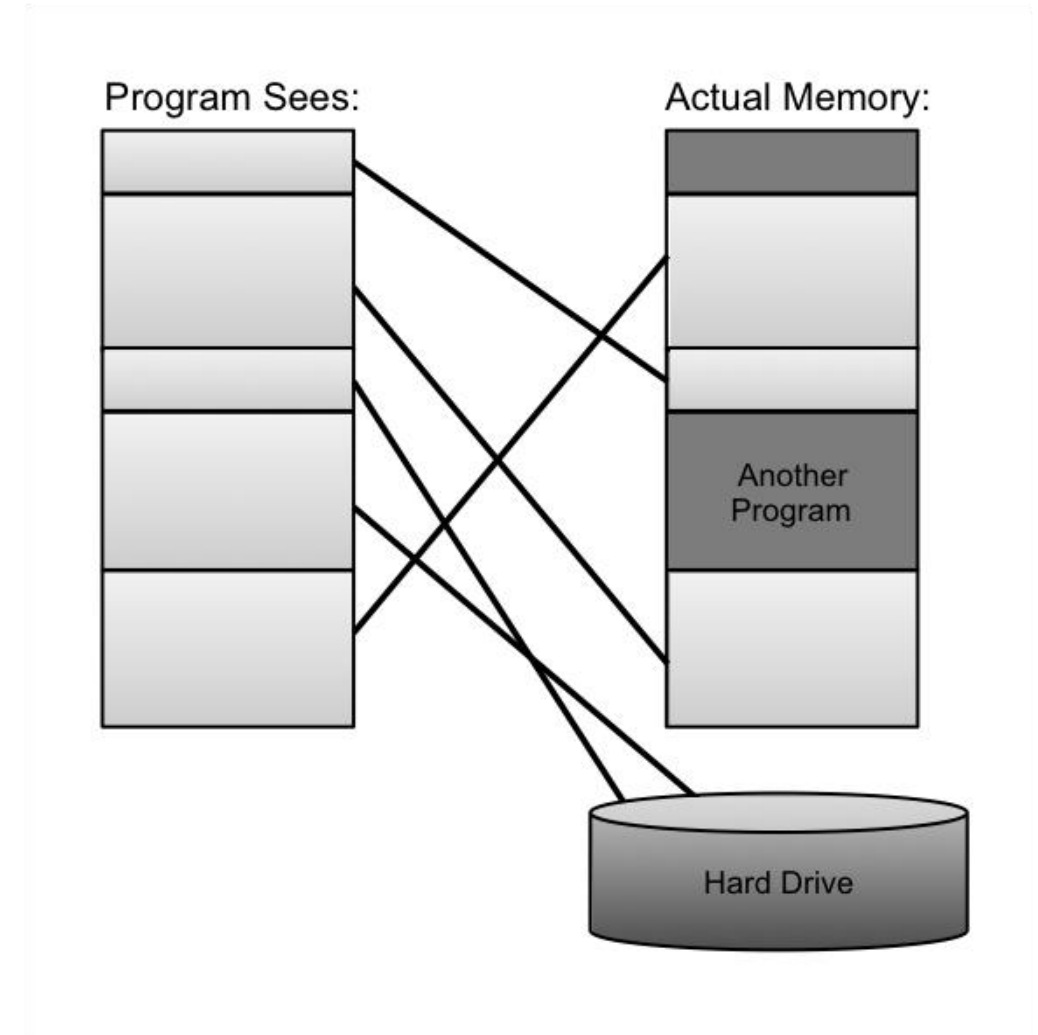
- **Text.** This segment contains the actual (binary) machine code of the program.
- **Data.** This segment contains static program variables that have been initialized in the program code.
- **BSS.** This segment, which is named for an antiquated acronym for **block started by symbol**, contains static variables that are uninitialized.
- **Heap.** This segment, which is also known as the dynamic segment, stores data generated during the execution of a process.
- **Stack.** This segment houses a stack data structure that grows downwards and is used for keeping track of the call structure of subroutines (e.g., methods in Java and functions in C) and their arguments.

Memory Layout



Virtual Memory

- There is generally not enough computer memory for the address spaces of all running processes.
- Nevertheless, the OS gives each running process the illusion that it has access to its complete (contiguous) address space.
- In reality, this view is **virtual**, in that the OS supports this view, but it is not really how the memory is organized.
- Instead, memory is divided into **pages**, and the OS keeps track of which ones are in memory and which ones are stored out to disk.



Page Faults

1. Process requests virtual address not in memory. causing a page fault.



Process

→ ***“read 0110101”***

“Page fault, let me fix that.”



Paging supervisor

2. Paging supervisor pages out an old block of RAM memory.

Blocks in RAM memory:



old



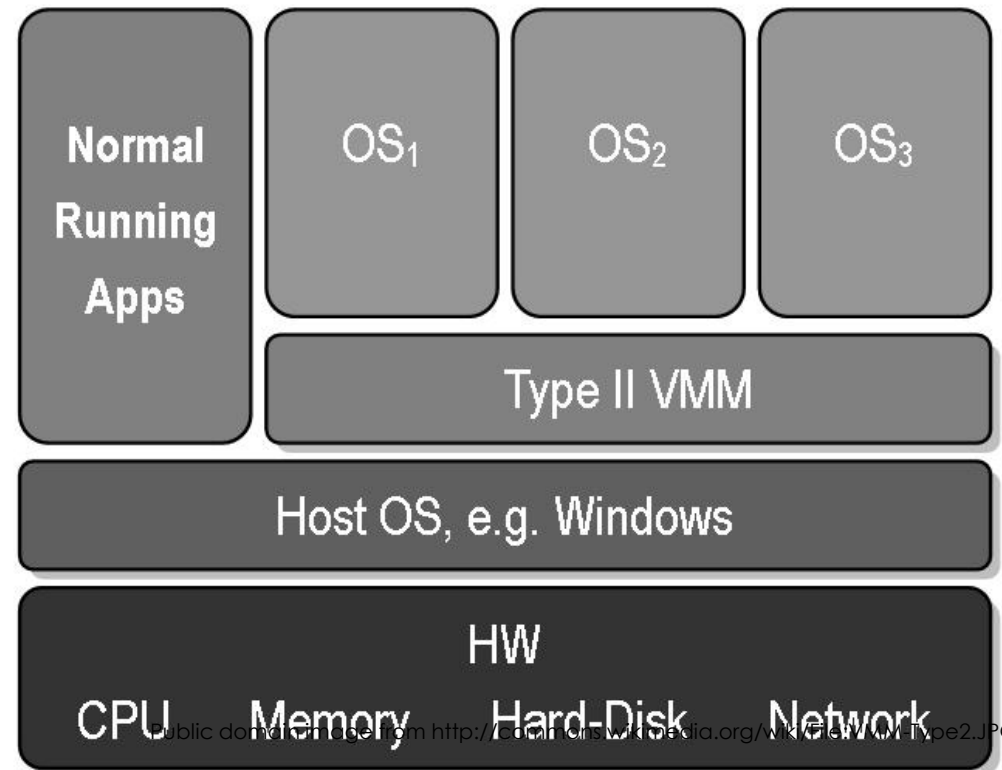
External disk

new

3. Paging supervisor locates requested block on the disk and brings it into RAM memory.

Virtual Machines

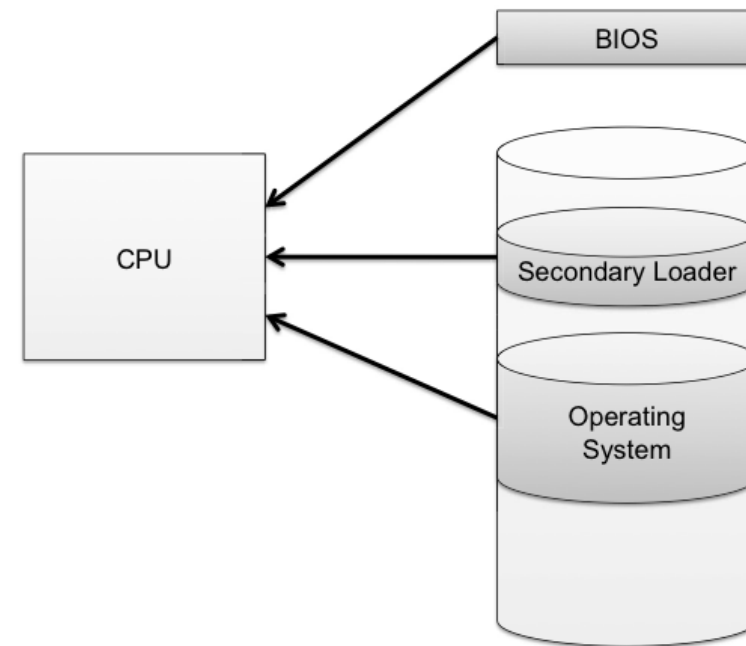
- **Virtual machine:** A view that an OS presents that a process is running on a specific architecture and OS, when really it is something else. E.g., a windows emulator on a Mac.
- **Benefits:**
 - **Hardware Efficiency**
 - **Portability**
 - **Security**
 - **Management**



Operating Systems Security

The Boot Sequence

- The action of loading an operating system into memory from a powered-off state is known as **booting** or **bootstrapping**.
- When a computer is turned on, it first executes code stored in a firmware component known as the **BIOS (basic input/output system)**.
- On modern systems, the BIOS loads into memory the **second-stage boot loader**, which handles loading the rest of the operating system into memory and then passes control of execution to the operating system.



BIOS Passwords

- A malicious user could potentially seize execution of a computer at several points in the boot process.
- To prevent an attacker from initiating the first stages of booting, many computers feature a **BIOS password** that does not allow a second-stage boot loader to be executed without proper authentication.

Hibernation

- Modern machines have the ability to go into a powered-off state known as **hibernation**.
- While going into hibernation, the OS stores the contents of machine's memory into a **hibernation file** (such as hiberfil.sys) on disk so the computer can be quickly restored later.
- But... without additional security precautions, hibernation exposes a machine to potentially invasive forensic investigation.



1. User closes a laptop computer, putting it into hibernation.

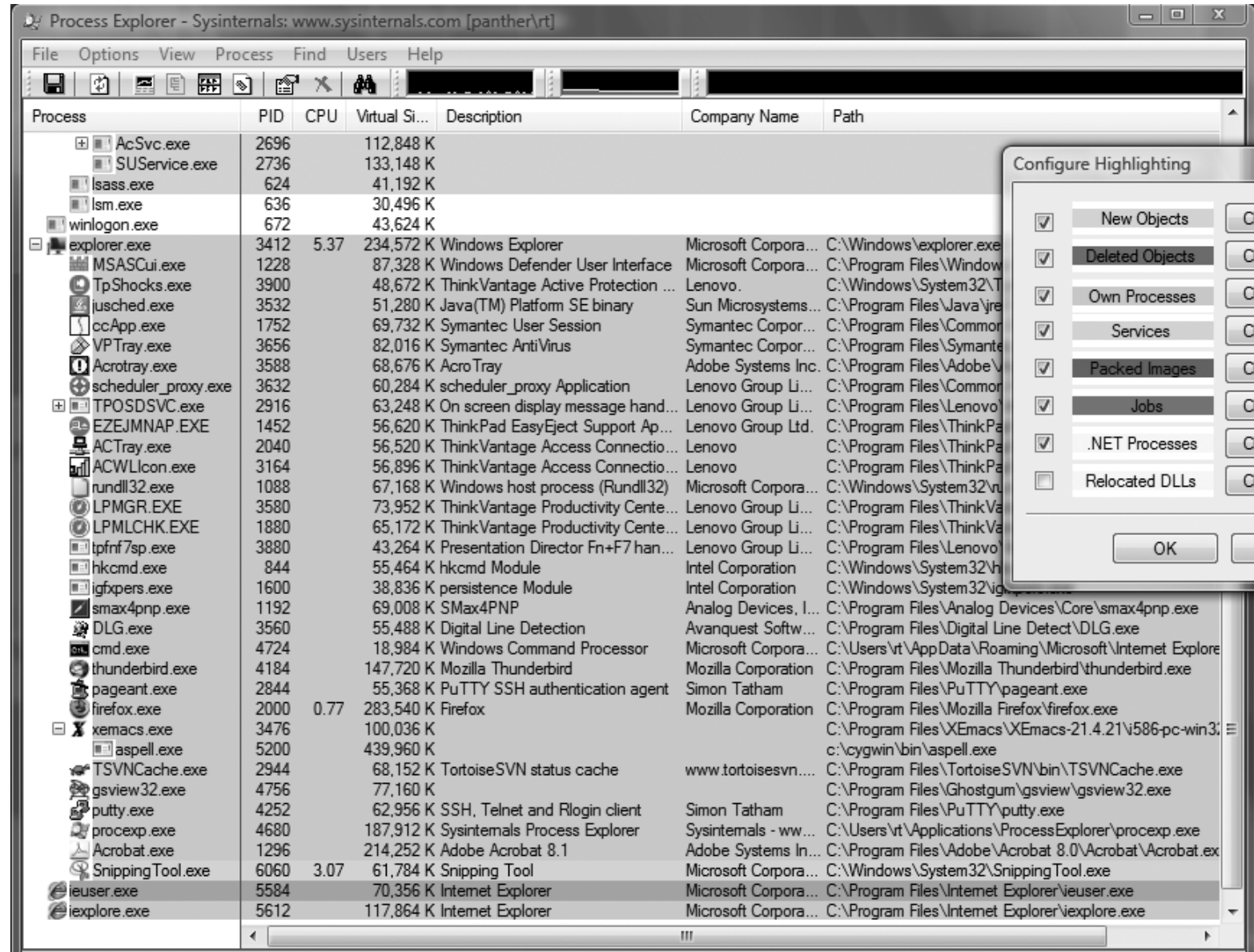


2. Attacker copies the hiberfil.sys file to discover any unencrypted passwords that were stored in memory when the computer was put into hibernation.

Event Logging

- Keeping track of what processes are running, what other machines have interacted with the system via the Internet, and if the operating system has experienced any unexpected or suspicious behavior can often leave important clues not only for troubleshooting ordinary problems, but also for determining the cause of a security breach.

Process Explorer



Memory and Filesystem Security

- The contents of a computer are encapsulated in its memory and filesystem.
- Thus, protection of a computer's content has to start with the protection of its memory and its filesystem.

Password Security

- The basic approach to guess passwords from the password file is to conduct a **dictionary attack**, where each word in a dictionary is hashed and the resulting value is compared with the hashed passwords stored in the password file.
- A dictionary of 500,000 “words” is often enough to discover most passwords.

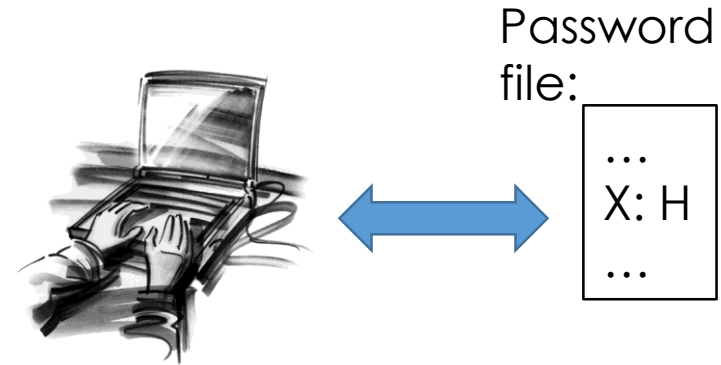
Password Salt

- One way to make the dictionary attack more difficult to launch is to use **salt**.
- Associate a random number with each userid.
- Rather than comparing the hash of an entered password with a stored hash of a password, the system compares the hash of an entered password and the salt for the associated userid with a stored hash of the password and salt.

How Password Salt Works

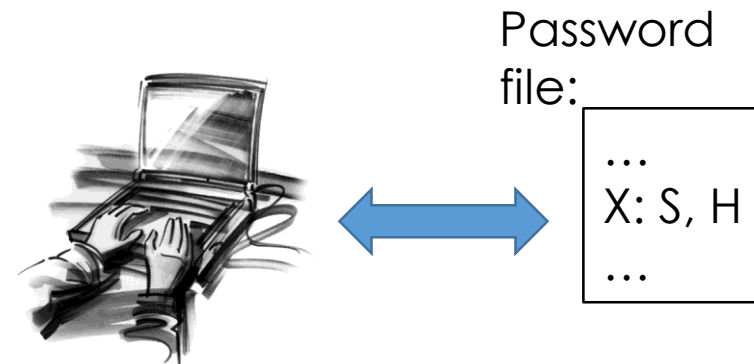
Without salt:

1. User types userid, X, and password, P.
2. System looks up H, the stored hash of X's password.
3. System tests whether $h(P) = H$.



With salt:

1. User types userid, X, and password, P.
2. System looks up S and H, where S is the random salt for userid X and H is stored hash of S and X's password.
3. System tests whether $h(S || P) = H$.



How Salt Increases Search Space Size

- Assuming that an attacker cannot find the salt associated with a userid he is trying to compromise, then the search space for a dictionary attack on a salted password is of size

$$2^B * D,$$

where B is the number of bits of the random salt and D is the size of the list of words for the dictionary attack.

- For example, if a system uses a 32-bit salt for each userid and its users pick passwords in a 500,000 word dictionary, then the search space for attacking salted passwords would be

$$2^{32} * 500,000 = 2,147,483,648,000,000,$$

which is over 2 quadrillion.

- Also, even if an attacker can find a salt password for a userid, he only learns one password.



Buffer Overflow Attacks

What is an Exploit?

- An **exploit** is any **input** (i.e., a piece of software, an argument string, or sequence of commands) that takes advantage of a bug, glitch or vulnerability in order to cause an attack
- An **attack** is an unintended or unanticipated behavior that occurs on computer software, hardware, or something electronic and that brings an advantage to the attacker

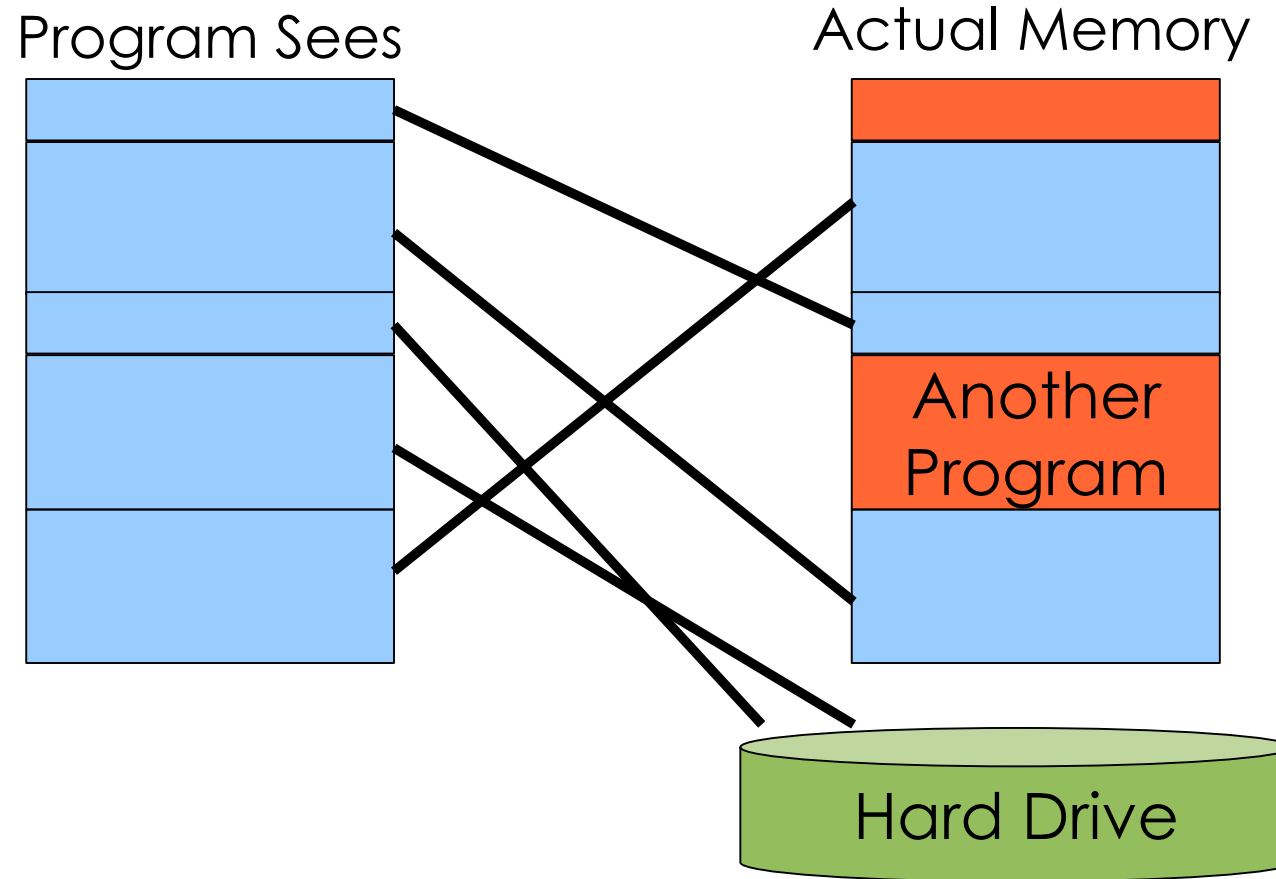
Buffer Overflow Attack

- One of the most common OS bugs is a **buffer overflow**
 - The developer fails to include code that checks whether an input string fits into its buffer array
 - An input to the running process exceeds the length of the buffer
 - The input string overwrites a portion of the memory of the process
 - Causes the application to behave improperly and unexpectedly
- Effect of a buffer overflow
 - The process can operate on malicious data or execute malicious code passed in by the attacker
 - If the process is executed as root, the malicious code will be executing with root privileges

Address Space

- Every program needs to access memory in order to run
- For simplicity sake, it would be nice to allow each process (i.e., each executing program) to act as if it owns all of memory
- The address space model is used to accomplish this
- Each process can allocate space anywhere it wants in memory
- Most kernels manage each process' allocation of memory through the **virtual memory** model
- How the memory is managed is irrelevant to the process

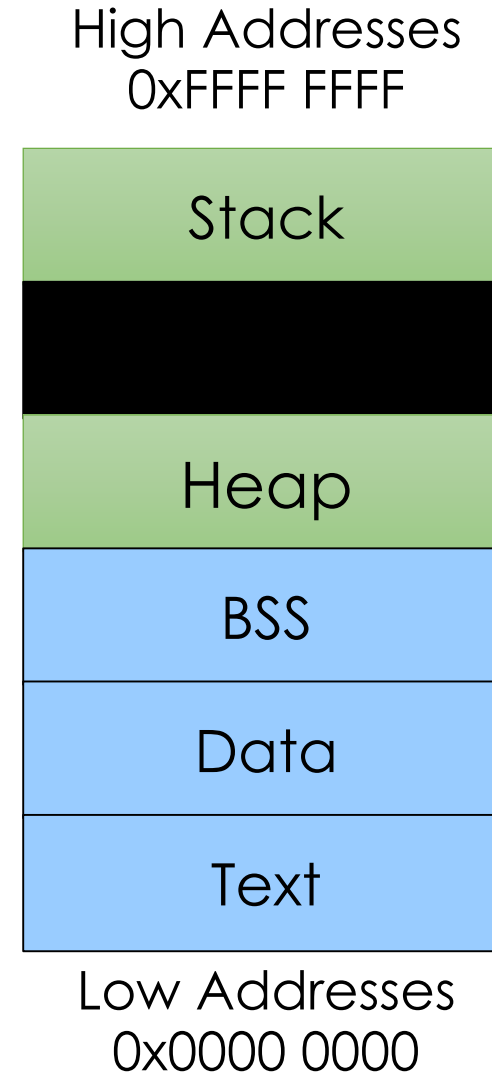
Virtual Memory



Mapping virtual addresses to real addresses

Unix Address Space

- **Text**: machine code of the program, compiled from the source code
- **Data**: static program variables initialized in the source code prior to execution
- **BSS** (block started by symbol): static variables that are uninitialized
- **Heap**: data dynamically generated during the execution of a process
- **Stack**: structure that grows downwards and keeps track of the activated method calls, their arguments and local variables



Vulnerabilities and Attack Method

- Vulnerability scenarios
 - The program has `root` privileges (`setuid`) and is launched from a shell
 - The program is part of a web application
- Typical attack method
 1. Find vulnerability
 2. Reverse engineer the program
 3. Build the exploit

Buffer Overflow Attack in a Nutshell

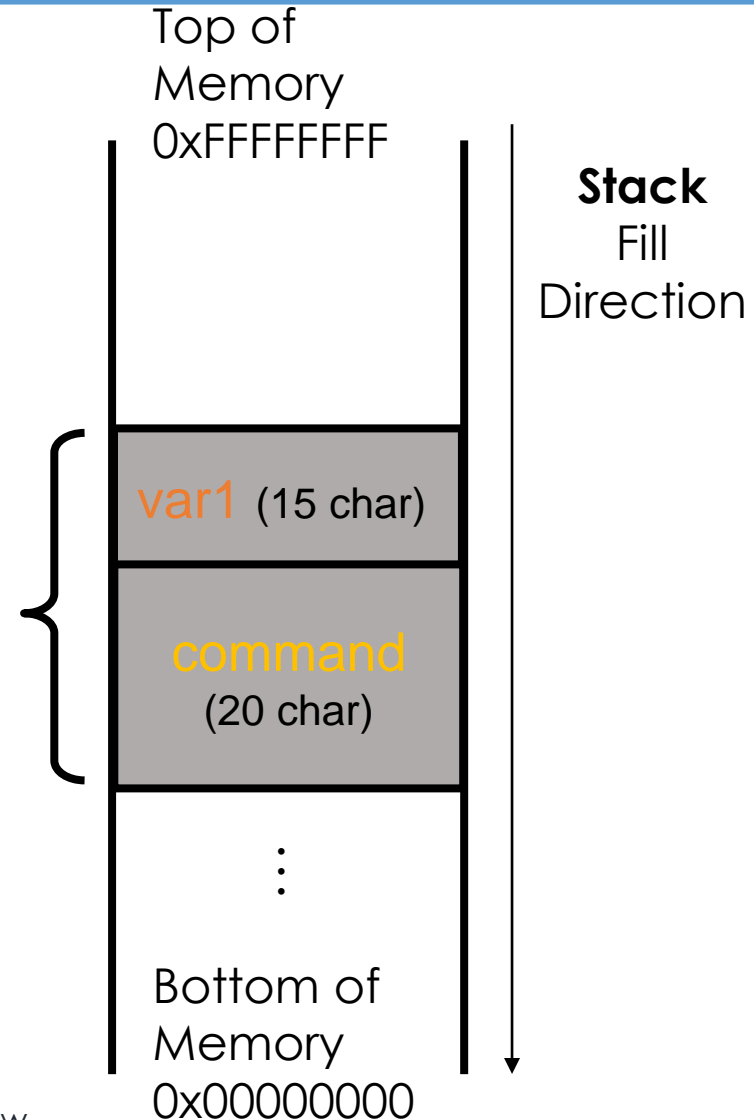
- First described in
Aleph One. Smashing The Stack For Fun And Profit. e-zine [www.Phrack.org](http://www.phrack.org)
#49, 1996
- The attacker exploits an unchecked buffer to perform a buffer overflow attack
- The ultimate goal for the attacker is getting a shell that allows to execute arbitrary commands with high privileges
- Kinds of buffer overflow attacks:
 - Heap smashing
 - Stack smashing

Buffer Overflow

domain.c

```
Main(int argc, char *argv[ ])  
/* get user_input */  
{  
    char var1 [15];  
    char command[20];  
    strcpy(command, "whois ");  
    strcat(command, argv[1]);  
    strcpy(var1, argv[1]);  
    printf(var1);  
    system(command);  
}
```

- Retrieves domain registration info
- e.g., domain brown.edu

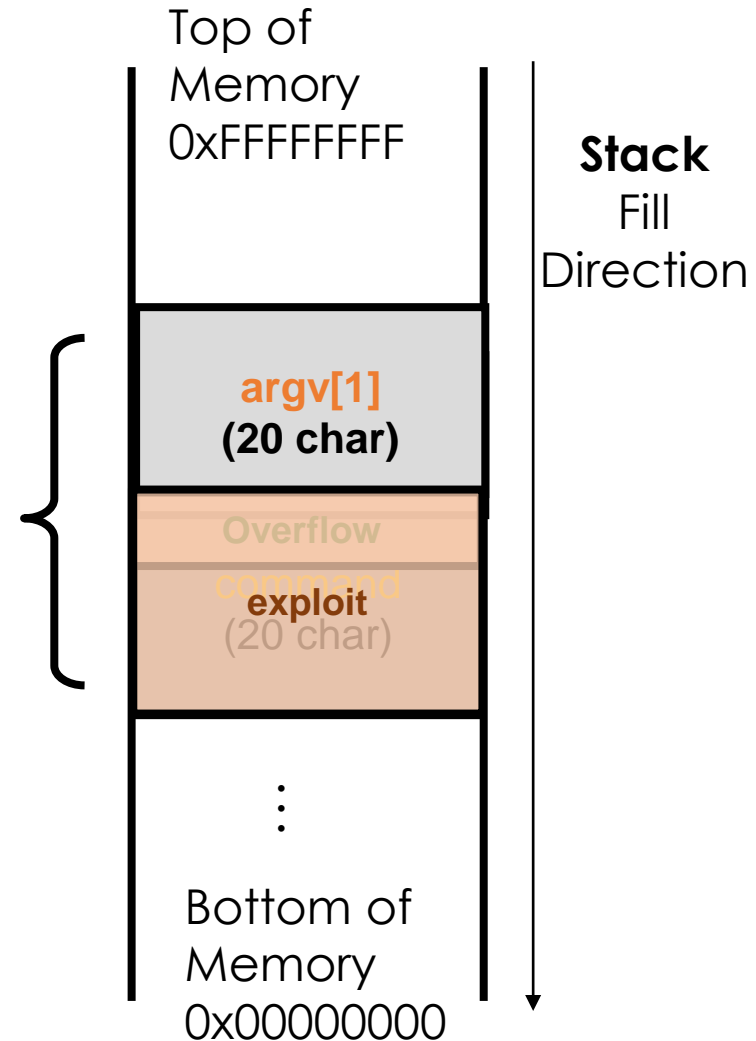


strcpy() Vulnerability

domain.c

```
Main(int argc, char *argv[])
/*get user_input*/
{
    char var1[15];
    char command[20];
    strcpy(command, "whois ");
    strcat(command, argv[1]);
    strcpy(var1, argv[1]);
    printf(var1);
    system(command);
}
```

- `argv[1]` is the user input
- `strcpy(dest, src)` does not check buffer
- `strcat(d, s)` concatenates strings



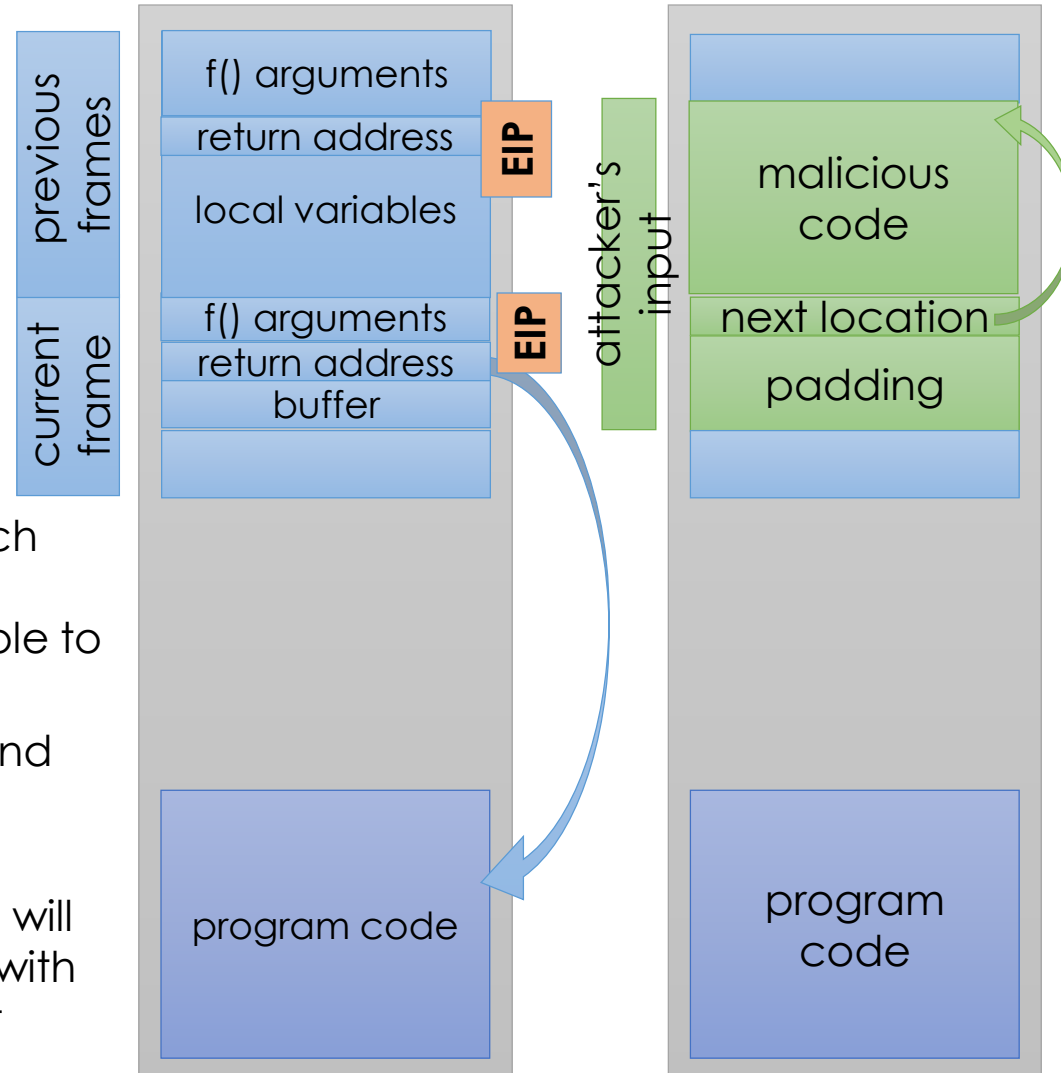
strcpy() vs. strncpy()

- Function `strcpy()` copies the string in the second argument into the first argument
 - e.g., `strcpy(dest, src)`
 - If source string > destination string, the overflow characters may occupy the memory space used by other variables
 - The **null character** is appended at the end automatically
- Function `strncpy()` copies the string by specifying the number **n** of characters to copy
 - e.g., `strncpy(dest, src, n); dest[n] = '\0'`
 - If source string is longer than the destination string, the overflow characters are discarded automatically
 - You have to place the **null character** manually

Return Address Smashing

```
void fingerd (...) {  
    char buf[80];  
    ...  
    get(buf);  
    ...  
}
```

- The Unix `fingerd()` system call, which runs as root (it needs to access sensitive files), used to be vulnerable to buffer overflow
- Write malicious code into buffer and overwrite return address to point to the malicious code
- When return address is reached, it will now execute the malicious code with the full rights and privileges of root



Unix Shell Command Substitution

- The Unix shell enables a command argument to be obtained from the standard output of another
- This feature is called **command substitution**
- When parsing command line, the shell replaces the output of a command between back quotes with the output of the command
- Example:
 - File **name.txt** contains string **farasi**
 - The following two commands are equivalent
 - **finger `cat name.txt`**
 - **finger farasi**

Shellcode Injection

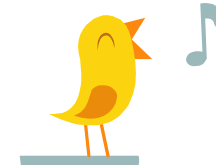
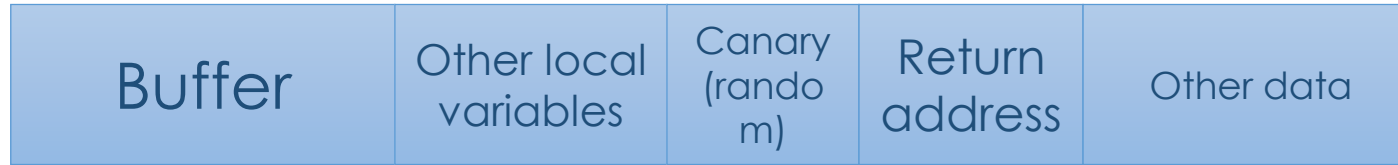
- An exploit takes control of attacked computer so injects code to “spawn a shell” or “shellcode”
- A shellcode is:
 - Code assembled in the CPU’s native instruction set (e.g. x86 , x86-64, arm, sparc, risc, etc.)
 - Injected as a part of the buffer that is overflowed.
- We inject the code directly into the buffer that we send for the attack
- A buffer containing shellcode is a “payload”

Buffer Overflow Mitigation

- We know **how** a buffer overflow happens, but **why** does it happen?
- This problem could not occur in Java; it is a C problem
 - In Java, objects are allocated dynamically on the heap (except ints, etc.)
 - Also cannot do pointer arithmetic in Java
 - In C, however, you can declare things directly on the stack
- One solution is to make the buffer dynamically allocated
- Another (OS) problem is that **fingerd** had to run as root
 - Just get rid of **fingerd**'s need for root access (solution eventually used)
 - The program needed access to a file that had sensitive information in it
 - A new world-readable file was created with the information required by **fingerd**

Stack-based buffer overflow detection using a random canary

Normal (safe) stack configuration:



Buffer overflow attack attempt:



- The canary is placed in the stack prior to the return address, so that any attempt to over-write the return address also over-writes the canary.