

# CS 370: OPERATING SYSTEMS

## [PROCESSES]

### The magic of `fork()`

To craft a process

use thy mighty `fork()`

Return in two places

Not one

The child, the parent's clone

each with IDs that

whisper which is which

Pair with `exec`

To untether and load a program

The child as if born anew

ready for the new world that now unfolds

Shrideep Pallickara  
Computer Science  
Colorado State University

COMPUTER SCIENCE DEPARTMENT



COLORADO STATE UNIVERSITY

1

## Frequently asked questions from the previous class survey

- Are kernel tasks also processes?
- Virtual memory?
- Scheduling: how does the OS manage so many processes, pause/resume them, manage/modify their priorities, etc.?
- How is the OS managing things that are developed in so many languages?
- Differences: kernel memory, user memory, physical memory, RAM
- Horizontal scaling (cloud computing); vertical scaling?
- Why are multiple cores necessary?
- What if there were no PCBs?
- Are some languages better suited for lower-level systems programming?
- Must there be a stack and a heap for every process?
- Connection between executable file name and process ID?



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.2

2

## Topics covered in this lecture

- Operations on processes
  - Creation
  - Termination
- Process groups
- Buffer Overflows
  - One of the greatest security violations of all time



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.3

3

Processes execute concurrently  
Can be created and deleted dynamically.

## OPERATIONS ON PROCESSES

COMPUTER SCIENCE DEPARTMENT



COLORADO STATE UNIVERSITY

4

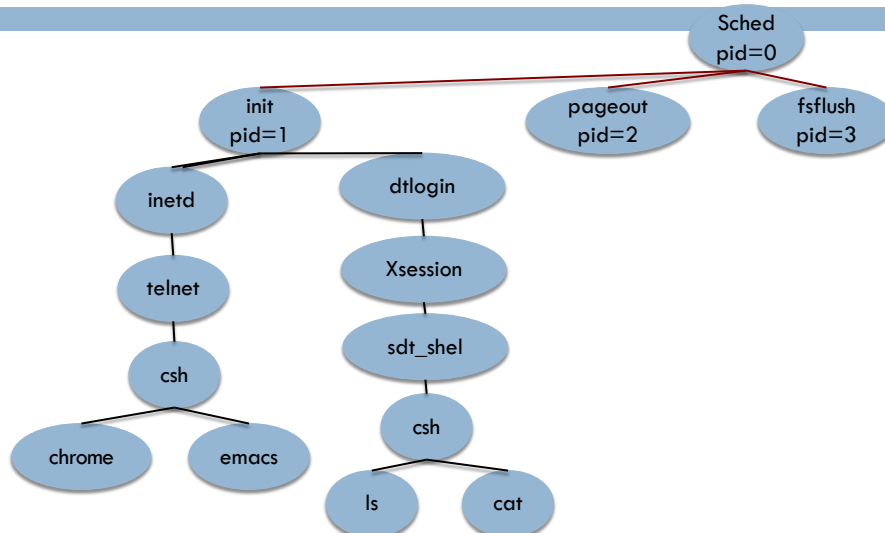
## Process Creation: A process may create new processes during its execution

- **Parent** process: The creating process
- **Child** process: New process that was created
  - May itself create processes: **Process tree**
- All processes have **unique** identifiers
  - Processes have names; in most systems, this is a number (process ID)
  - There is one ID per process



5

## Example: Process tree in Solaris



6

## Processes in UNIX

- `init` : Root parent process for all user processes
- Get a listing of processes with `ps` command
  - `ps`: List of all processes associated with user
  - `ps -a` : List of all processes associated with terminals
  - `ps -A` : List of all active processes



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.7

7

## Resource sharing between a process and its child

- Child process may obtain resources **directly from OS**
- Child may be **constrained** to a subset of parent's resources
  - Prevents any process from overloading system
- Parent process also passes along initialization data to the child
  - Physical and logical resources



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.8

8

## Parent/Child processes: Execution possibilities

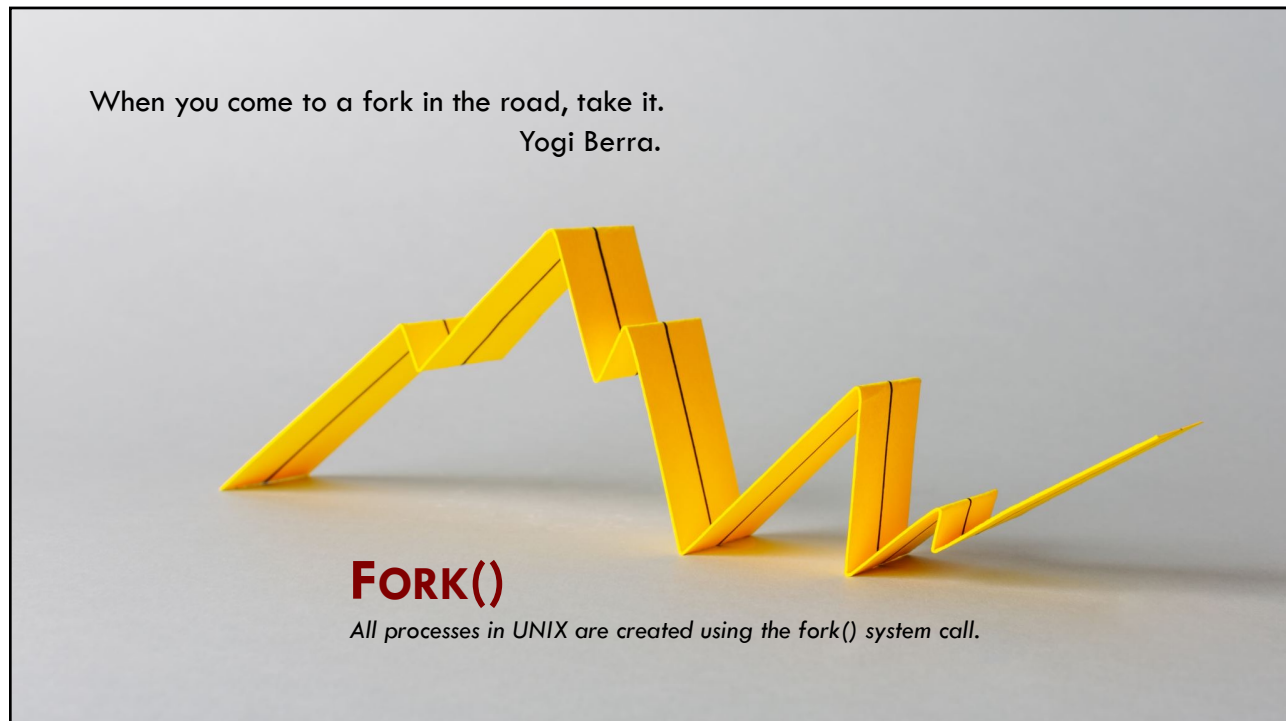
- Parent executes **concurrently** with children
- Parent **waits** until some or all of its children terminate



## Parent/Child processes: Address space possibilities

- Child is a **duplicate** of the parent
  - ▣ Same program and data as parent
- Child has a **new program** loaded into it






11

## Process creation in UNIX

- Process created using **`fork()`**
  - `fork()` copies parent's memory image
  - Includes copy of parent's address space
- Parent and child continue execution **at instruction after** `fork()`
  - Child: Return code for `fork()` is **0**
  - Parent: Return code for `fork()` is the **non-ZERO process-ID** of new child

 **COLORADO STATE UNIVERSITY** Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT PROCESSES L4.12

12

### fork() results in the creation of 2 distinct processes

Parent  
PID=abc  
...  
id = fork()  
...  
id = xyz here

Results in

Child  
PID=xyz  
...  
id = fork()  
...  
id = 0 here  
Child will execute from here

Do the parent and child process *share* a stack or heap?

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALICKARA COMPUTER SCIENCE DEPARTMENT PROCESSES L4.13

13

### Simple example:

```
#include <stdio.h>
#include <unistd.h>

int main(void) {
    int x;
    x=0;
    fork();
    x=1;
    ...
}
```

Why is this the case?

Both parent and child execute this *after* returning from fork()

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALICKARA COMPUTER SCIENCE DEPARTMENT PROCESSES L4.14

14

## Another example

```
#include <stdio.h>
#include <unistd.h>

int main () {
    printf("Hello World\n");
    fork();
    printf("Hello World\n");
}
```

Hello World  
Hello World  
Hello World

?

How many hello worlds?

```
#include <stdio.h>
#include <unistd.h>

int main () {
    printf("Hello World\n");
    if (fork()==0) {
        printf("Hello World\n");
    }
}
```

Hello World  
Hello World

?

How many hello worlds?



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.15

15

## What happens when `fork()` fails?

- No child is created
- `fork()` returns **-1** and sets `errno`
  - `errno` is a global variable in `errno.h`



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.16

16



## If a system is short on resources OR if limit on number of processes breached

- `fork()` sets `errno` to `EAGAIN`
  
- Some typical numbers for Solaris
  - `maxusers`: 2 less than number of MB of physical memory up to 1024
    - Set up to 2048 manually in `/etc/system` file
  - `mx_nprocs`: Default:  $16 \times \text{maxusers} + 10$   
min = 138, max = 30,000



## Take different paths depending on what happens with `fork()`

```
childpid = fork();
if (childpid == -1) {
    perror("Failed to fork");
    return 1;
}
if (childpid == 0) {
    .... child specific processing
} else {
    .... parent specific processing
}
```

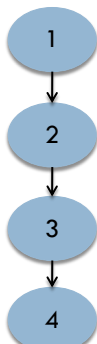
Child (any process) can use  
**`getpid()`** to retrieve  
its process ID



## Creating a chain of processes

```
for (int i=1; i < 4; i++) {  
    if (childid = fork()) {  
        break;  
    }  
}
```

value of *i*  
when process leaves loop



For each iteration:  
Parent has non-ZERO childid  
So it breaks out  
  
Child process  
Parent in NEXT iteration



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

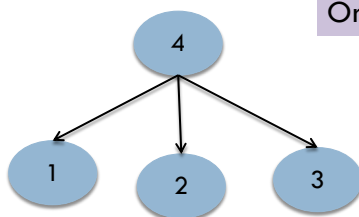
L4.19

19

## Creating a process fan

```
for (int i=1; i < 4; i++) {  
    if ((childid = fork()) <= 0) {  
        break;  
    }  
}
```

value of *i*  
when process leaves loop



Newly created process breaks out  
Original process continues



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

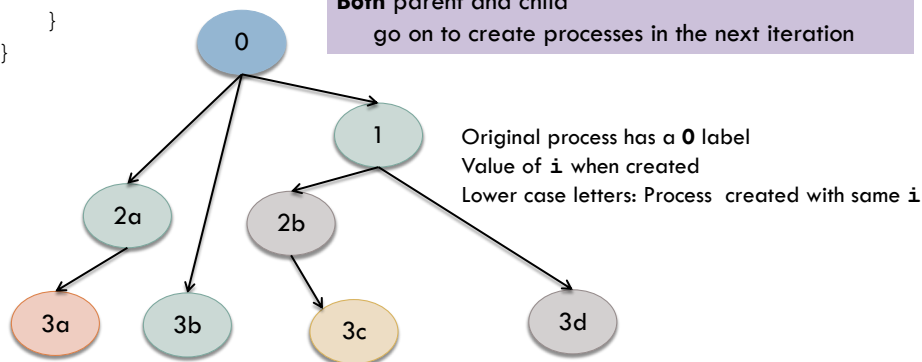
PROCESSES

L4.20

20

## Creation of a process tree

```
int i=0;
for (i=1; i < 4; i++) {
    if ((childid = fork()) == -1) {
        break;
    }
}
```



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.21

21

## Replacing a process's memory space with a new program

- Use `exec()` after the `fork()` in **one** of the two processes
- `exec()` does the following:
  - ① **Destroys** memory image of program containing the call
  - ② **Replaces** the invoking process's memory space with a new program
  - ③ Allows processes to go their **separate** ways



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.22

22

## Replacing a process's memory space with a new program

- **TRADITION:**
  - Child executes **new** program
  - Parent executes **original** code



## Launching programs using the shell is a two-step process

- Example: user types **ls** on the **shell**
  - ① Shell **forks** off a child process
  - ② Child executes **ls**



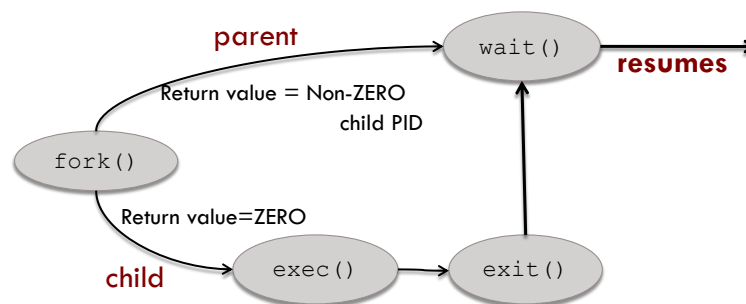
## But why is this the case?

- Allows the child to manipulate its file descriptors
  - ▣ After the `fork()`
  - ▣ But before the `exec()`
- Accomplish **redirection** of standard input, standard output, and standard error



## A parent can move itself from off the ready queue and await child's termination

- Done using the `wait()` system call.
- When child process completes, parent process resumes



`wait/waitpid` allows caller to suspend execution till  
a child's status is available

- Process status availability
  - ▣ Most commonly after termination
  - ▣ Also available if process is stopped
  
- `waitpid(pid, *stat_loc, options)`
  - `pid == -1` : any child
  - `pid > 0` : specific child
  - `pid == 0` : any child in the same **process group**
  - `pid < -1` : any child in process group `abs(pid)`



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.27

27

## PROCESS CREATION IN WINDOWS

COMPUTER SCIENCE DEPARTMENT



COLORADO STATE UNIVERSITY

28

## Process creation in Windows

- `CreateProcess` handles
  - ① Process creation
  - ② Loading in a new program
- Parent and child's address spaces are **different** from the start



## `CreateProcess` takes up to 10 parameters

- Program to be executed
- Command line parameters that feed program
- Security attributes
- Bits that control whether files are inherited
- Priority information
- Window to be created?



## Process Management on Windows

- **WIN 32** has about 100 other functions
  - Managing & Synchronizing processes



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.31

31

## PROCESS GROUPS



32



## Process groups

- Process group is a **collection** of processes
- Each process has a **process group ID**
- Process group leader?
  - Process with `pid==pgid`
- **kill** treats negative `pid` as `pgid`
  - Sends signal to all constituent processes



## Process Group IDs:

When a child is created with `fork()`

- ① **Inherits** parent's process group ID
- ② **Parent can change** group ID of child by using `setpgid`
- ③ Child can **give itself** new process group ID
  - Set process group ID = its process ID



## Process groups

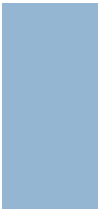
- By default, comprises:
  - ① Parent (and further ancestors)
  - ② Siblings
  - ③ Children (and further descendants)
- A process can only send **signals** to members of its process group
  - Signals are a limited form of inter-process communication used in Unix



## Windows has no concept of a process hierarchy


- The only hint of a hierarchy?
  - When a process is created, parent is given a special **token** (called **handle**)
    - Use this to control the child
- However, parent is free to **pass** this token to some other process
  - **Invalidates** hierarchy





# PROCESS TERMINATIONS

COMPUTER SCIENCE DEPARTMENT




COLORADO STATE UNIVERSITY

37

## Process terminations

- Normal exit (voluntary)
  - ▣ E.g., successful compilation of a program
- Error exit (voluntary)
  - ▣ E.g., trying to compile a file that does not exist



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.38

38

## Process terminations

- Fatal error (involuntary)
  - Program bug
    - Referencing non-existing memory, dividing by zero, etc
- Killed by another process (involuntary)
  - Execute system call telling OS to kill some other process
  - *Killer* must be authorized to do in the *killee*
  - Unix: `kill` Win32: `TerminateProcess`



## Process terminations: This can be either normal or abnormal

- OS **deallocates** the process resources
  - Cancel pending timers and signals
  - Release virtual memory resources and locks
  - Close any open files
- Updates statistics
  - Process status and resource usage
- Notifies parent in response to a `wait()`



On termination a UNIX process DOES NOT fully release resources until a parent waits for it

- If the parent is not waiting when the child terminates?
  - ▣ The process becomes a **zombie**
- Zombie is an *inactive* process
  - ▣ Still has an entry in the process table



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.41

41

## Zombies and termination

- When a process terminates, its *orphaned* children and zombies are *adopted*
  - ▣ This special system process is **init**
- Some more about **init**
  - ① Has a pid of 1
  - ② Periodically waits for children
  - ③ Eventually orphaned zombies are removed



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.42

42

## Normal termination of processes

- Return from `main`
- Implicit return from `main`
  - Function **falls off the end**
- Call to `exit`, `_Exit` or `_exit`



## Abnormal termination

- Call `abort`
- Process signal that causes termination
  - Generated by an external event: keyboard `Ctrl-C`
  - Internal errors: Accessing illegal memory location
- Consequences
  - Core dump
  - User-installed exit handler not called





45

## Protection and Security

- Control access to system resources
  - ▣ Improve reliability
- Defend against use (misuse) by unauthorized or incompetent users
- Examples
  - ▣ Ensure process executes within its own space
  - ▣ Force processes to relinquish control of CPU
  - ▣ Device-control registers accessible only to the OS
    - E.g., Why the Security of USB Is Fundamentally Broken  
<https://www.wired.com/2014/07/usb-security/>



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.46

46

## Buffer overflows:

- When? Program copies data into a variable for which it **has not allocated enough space**

```
char buf[80];  
printf("Enter your first name:");  
scanf("%s", buf);
```

If user enters string  $> 79$  bytes ?

– The string AND string terminator do not fit.



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.47

47

## Buffer Overflows: Fixing the example problem

```
char buf[80];  
printf("Enter your first name:");  
scanf("79%s", buf);
```

Program now reads at most 79 characters into buf



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.48

48



## Automatic variables (local variables)

- Allocated/deallocated automatically when program flow enters or leaves the variable's scope
- Allocated on the program stack
- Stack grows from high-memory to low-memory



COLORADO STATE UNIVERSITY

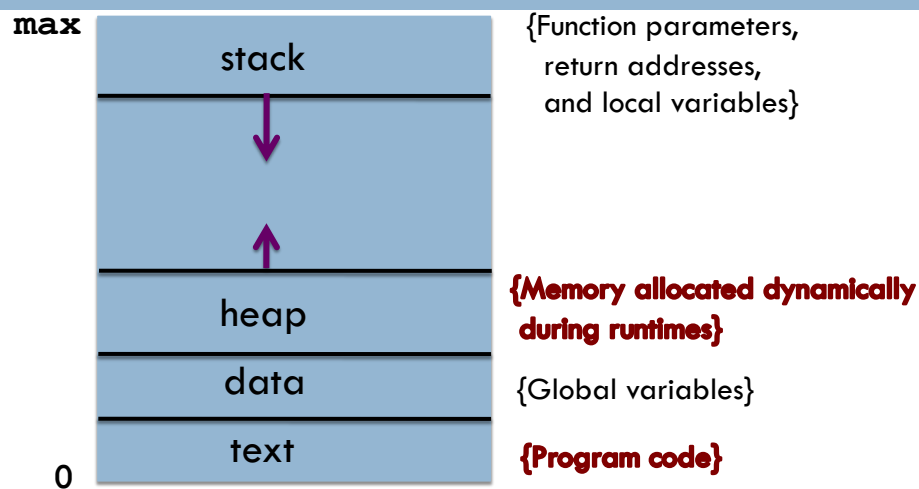
Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.49

49

## A process in memory



COLORADO STATE UNIVERSITY

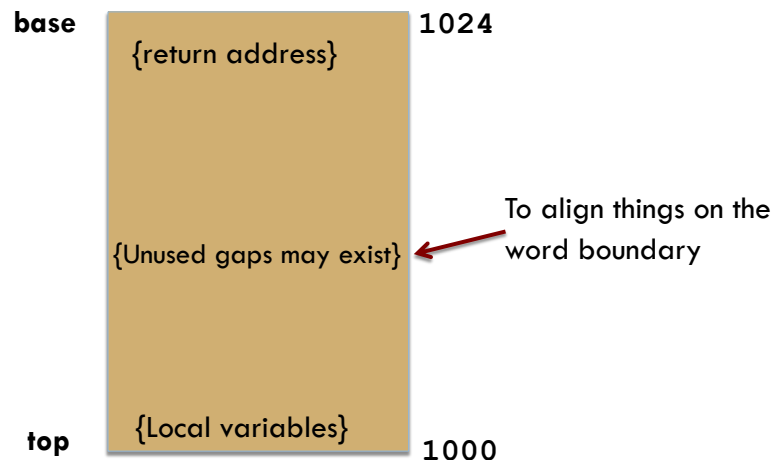
Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.50

50

## A rough anatomy of the program stack



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.51

51

## A function that checks password: Susceptible to buffer overflow

```
int checkpass(void) {  
    int x;  
    char a[9];  
    x = 0;  
    printf("Enter a short word: ");  
    scanf("%s", a);  
    if (strcmp(a, "mypass") == 0)  
        x = 1;  
    return x;  
}
```



COLORADO STATE UNIVERSITY

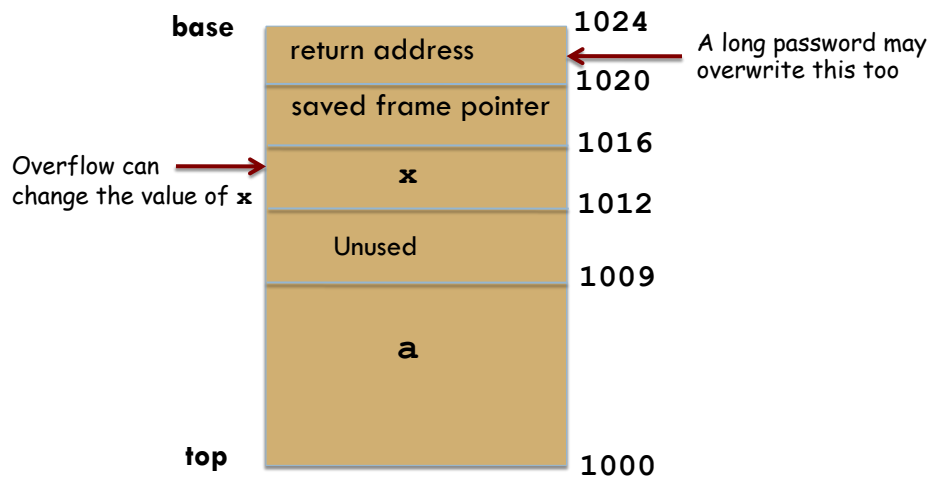
Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.52

52

## Stack layout for our unsafe function



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.53

53

## Problems with buffer overflow

- Function will try to return to an address space **outside** the program
  - Segmentation fault or core dump
  - Programs may lose unsaved data
  - In the OS, such a function can cause the OS to crash!



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.54

54

## One of the greatest security violations of all time: November 2, 1988

- ❑ Exploited 2 bugs in Berkeley UNIX
- ❑ Worm: Self replication program
- ❑ Bought down most of the Sun and VAX systems on the internet within a *few hours*



## Worm had two programs

- ① Bootstrap (99 lines of C, 11 .c)
  - ② Worm proper
- ❑ Both these programs compiled and executed on the system under attack



## Synopsis of the worm's modus operandi

- ① Spread the bootstrap to machines
- ② Once the bootstrap runs:
  - ▣ Connects back to its origins
  - ▣ Download worm proper
  - ▣ Execute worm
- ③ Worm then attempts to spread bootstrap



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.57

57

## Infecting new machines: Method 1 & 2 Violate trust

- ▣ Method 1: Run the remote shell *rsh*
  - ▣ Machines used to trust each other, and would willingly run it
  - ▣ Use this to upload the worm
- ▣ Method 2: *sendmail*



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.58

58

## Method 3: Buffer overflow in the `finger` daemon (`finger name@site`)

- `finger` daemon runs all the time on sites, and responds to queries
- The worm called `finger` with a handcrafted 536-byte string as a parameter.
  - ▣ Overflowed daemon's buffer & overwrote its stack
- Daemon did not return to `main()`, but to a procedure in the 536-bit string on stack
- Next try to get a shell by executing `/bin/sh`



## Far too many worms can grind things to a halt

- Break user passwords
- Check for copies of worm on machine
  - ▣ Exit if there is a copy 6 out of 7 times
    - This is in place to cope with a situation where sys admin starts fake worm to fool the real one
- Use of 1 in 7 caused far too worms
  - ▣ Machines ground to a halt



## Consequences

- \$10K fine, 3 years probation and 400 hours community service
- Legal costs \$150,000



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.61

61

## The contents of the slide-set are based on the following references

- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9<sup>th</sup> edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330. [Chapter 3]*
- *Andrew S Tanenbaum and Herbert Bos. Modern Operating Systems. 4<sup>th</sup> Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620 [Chapter 2]*
- *Kay Robbins & Steve Robbins. Unix Systems Programming, 2nd edition, Prentice Hall ISBN-13: 978-0-13-042411-2. [Chapters 2 & 3]*
- *CS 451: Operating Systems (Colorado State University) Help Session 2B: Forking in C by Rink Dewri. Professor: Shrideep Pallickara, GTA: Rinku Dewri*



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

PROCESSES

L4.62

62