

CS 370: OPERATING SYSTEMS
[PROCESSES]

Shrideep Pallickara
Computer Science
Colorado State University

August 30, 2018 CS370: Operating Systems [Fall 2018] L4.1
Dept. Of Computer Science, Colorado State University

Frequently asked questions from the previous class survey

- Cores
 - ▣ Since CPU clock speeds have tapered off significantly, do we rely on the kernel to do things or ... [threads/parallel programming]
- What is the executable image?
- Processes:
 - ▣ Can a process have multiple parents?
 - ▣ Do you have to recompile the Kernel? NO!
- PCB: Nothing fancy about this data structure?

August 30, 2018 CS370: Operating Systems [Fall 2018] L4.2
Professor: SHRIDEEP PALLICKARA Dept. Of Computer Science, Colorado State University

Frequently asked questions from the previous class survey

- Memory? [Main memory, RAM, Physical memory, DRAM]
- Is there one giant Stack and Heap for ALL processes? NO!
 - ▣ How is the stack and heap connected to main memory?
- FSMs: How can a process go from Waiting for I/O to Ready without using the CPU?
- Lots of Memory Management Questions
 - ▣ Access limits, Paging, etc

August 30, 2018 CS370: Operating Systems [Fall 2018] L4.3
Professor: SHRIDEEP PALLICKARA Dept. Of Computer Science, Colorado State University

Topics covered in this lecture

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

August 30, 2018 CS370: Operating Systems [Fall 2018] L4.4
Professor: SHRIDEEP PALLICKARA Dept. Of Computer Science, Colorado State University

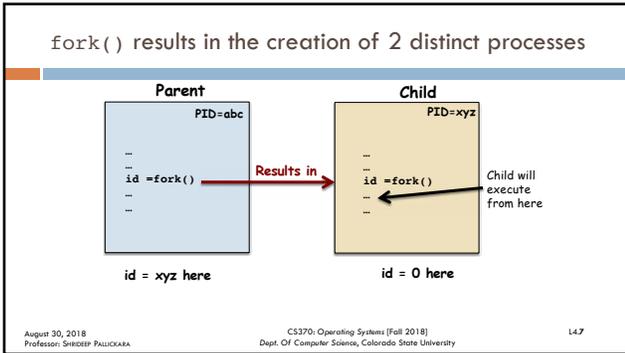
FORK()
All processes in UNIX are created using the `fork()` system call.

August 30, 2018 CS370: Operating Systems [Fall 2018] L4.5
Dept. Of Computer Science, Colorado State University

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

August 30, 2018 CS370: Operating Systems [Fall 2018] L4.6
Professor: SHRIDEEP PALLICKARA Dept. Of Computer Science, Colorado State University



Simple example:

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

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

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

August 30, 2018
 Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University

L4.8

Another example

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

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

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

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

Hello World
 Hello World
 Hello World

Hello World
 Hello World

August 30, 2018
 Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University

L4.9

- ### What happens when fork() fails?
- No child is created
 - fork() returns -1 and sets errno
 - errno is a global variable in errno.h
- August 30, 2018
 Professor: SHRIDEEP PALICKARA
- CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University
- L4.10

- ### 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 x maxusers + 10
 min = 138, max = 30,000
- August 30, 2018
 Professor: SHRIDEEP PALICKARA
- CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University
- L4.11

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

August 30, 2018
 Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University

L4.12

Creating a chain of processes

```

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

For each iteration:
 Parent has non-ZERO childid
 So it breaks out

Child process
 Parent in NEXT iteration

value of i
 when process leaves loop

August 30, 2018
 Professor: SHRIDEEP PALLICKARA
 CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University
 L4.13

Creating a process fan

```

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

Newly created process breaks out
 Original process continues

value of i
 when process leaves loop

August 30, 2018
 Professor: SHRIDEEP PALLICKARA
 CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University
 L4.14

Creation of a process tree

```

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

Both parent and child
 go on to create processes in the next iteration

Original process has a 0 label
 Value of i when created
 Lower case letters: Process created with same i

August 30, 2018
 Professor: SHRIDEEP PALLICKARA
 CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University
 L4.15

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:
 - D**estroys memory image of program containing the call
 - R**eplaces the invoking process's memory space with a new program
 - A**llows processes to go their **s**eparate ways

August 30, 2018
 Professor: SHRIDEEP PALLICKARA
 CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University
 L4.16

Replacing a process's memory space with a new program

- T**RADITION:
 - Child executes **n**ew program
 - Parent executes **o**riginal code

August 30, 2018
 Professor: SHRIDEEP PALLICKARA
 CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University
 L4.17

Launching programs using the shell is a two-step process

- Example: user types **s**ort on the shell
 - Shell **f**orks off a child process
 - Child executes **s**ort

August 30, 2018
 Professor: SHRIDEEP PALLICKARA
 CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University
 L4.18

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

August 30, 2018
 Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University

L4.19

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

```

    graph TD
        Parent((parent)) -- fork() --> Child((child))
        Child -- exec() --> Exit((exit()))
        Parent -- wait() --> Resumes((resumes))
        Child -- "Return value = ZERO" --> Parent
        Parent -- "Return value = Non-ZERO child PID" --> Child
    
```

August 30, 2018
 Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University

L4.20

`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)`

August 30, 2018
 Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University

L4.21

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**

August 30, 2018
 Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University

L4.22

`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?

August 30, 2018
 Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University

L4.23

Process Management on Windows

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

August 30, 2018
 Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University

L4.24

PROCESS GROUPS

August 30, 2018 CS370: Operating Systems [Fall 2018] L4.25
Dept. Of Computer Science, Colorado State University

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

August 30, 2018 CS370: Operating Systems [Fall 2018] L4.26
Professor: SHRIDEEP PALLICKARA Dept. Of Computer Science, Colorado State University

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

August 30, 2018 CS370: Operating Systems [Fall 2018] L4.27
Professor: SHRIDEEP PALLICKARA Dept. Of Computer Science, Colorado State University

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.

August 30, 2018 CS370: Operating Systems [Fall 2018] L4.28
Professor: SHRIDEEP PALLICKARA Dept. Of Computer Science, Colorado State University

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

August 30, 2018 CS370: Operating Systems [Fall 2018] L4.29
Professor: SHRIDEEP PALLICKARA Dept. Of Computer Science, Colorado State University

PROCESS TERMINATIONS

August 30, 2018 CS370: Operating Systems [Fall 2018] L4.30
Dept. Of Computer Science, Colorado State University

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

August 30, 2018
Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

L4.31

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 *killed*
 - ▣ Unix: **kill** Win32: **TerminateProcess**

August 30, 2018
Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

L4.32

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()`

August 30, 2018
Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

L4.33

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

- When 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

August 30, 2018
Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

L4.34

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

August 30, 2018
Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

L4.35

Normal termination of processes

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

August 30, 2018
Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

L4.36

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

August 30, 2018
Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

L4.37

PROTECTION & SECURITY

August 30, 2018

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

L4.38

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/>

August 30, 2018
Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

L4.39

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.

August 30, 2018
Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

L4.40

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

August 30, 2018
Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

L4.41

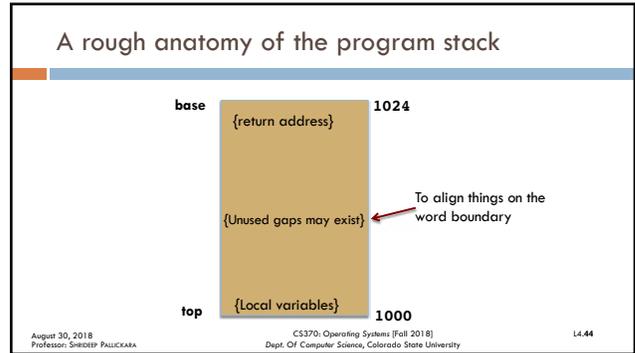
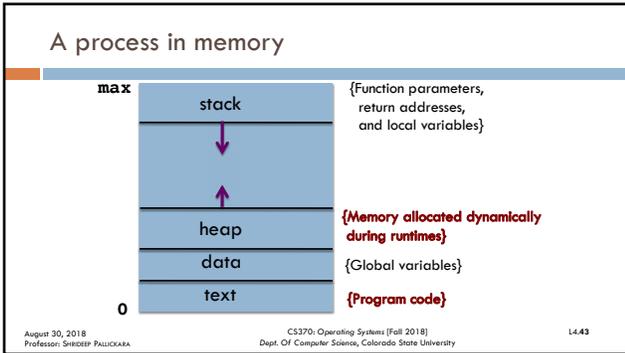
Automatic variables (local variables)

- 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

August 30, 2018
Professor: SHRIDEEP PALICKARA

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

L4.42



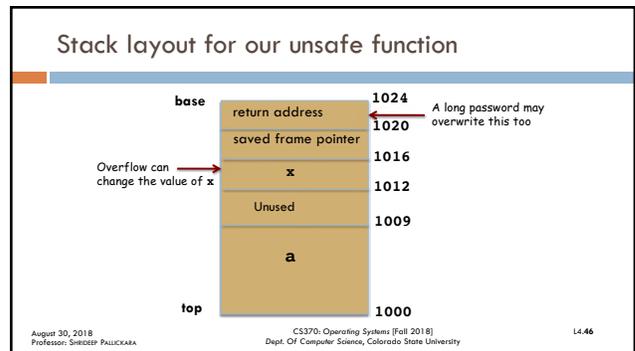
A function that checks password: Susceptible to buffer overflow

```
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;
}
```

August 30, 2018
 Professor: SHRIDEEP PALLICKARA

CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University

L4.45



- ### 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!
- August 30, 2018
 Professor: SHRIDEEP PALLICKARA
- CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University
- L4.47

- ### 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
- August 30, 2018
 Professor: SHRIDEEP PALLICKARA
- CS370: Operating Systems [Fall 2018]
 Dept. Of Computer Science, Colorado State University
- L4.48

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

August 30, 2018
Professor: SHRIDEEP PALLICKARA

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

L4.49

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

August 30, 2018
Professor: SHRIDEEP PALLICKARA

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

L4.50

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*

August 30, 2018
Professor: SHRIDEEP PALLICKARA

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

L4.51

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*

August 30, 2018
Professor: SHRIDEEP PALLICKARA

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

L4.52

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

August 30, 2018
Professor: SHRIDEEP PALLICKARA

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

L4.53

Consequences

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

August 30, 2018
Professor: SHRIDEEP PALLICKARA

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

L4.54

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

- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9th edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330. [Chapter 3]*
- *Andrew S Tanenbaum and Herbert Bos. Modern Operating Systems, 4th 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]*