# Processes: Introduction

CS 241
February 13, 2012

# Announcements

MP2 due tomorrow

Deadline and contest cutoff 11:59 p.m.

Fabulous prizes on Wednesday

MP3 out Wednesday: Shell (1 week)

Code from this lecture posted after class

# Processes

Definition: A *process* is an instance of a running program.

- One of the most profound ideas in computer science
- Not the same as "program" or "processor"

Process provides each program with two key abstractions:

- Logical control flow
  - Each program seems to have exclusive use of the CPU
- Private virtual address space
  - Each program seems to have exclusive use of main memory

How are these illusions maintained?

- Process executions interleaved (multitasking) or run on separate cores
- Address spaces managed by virtual memory system

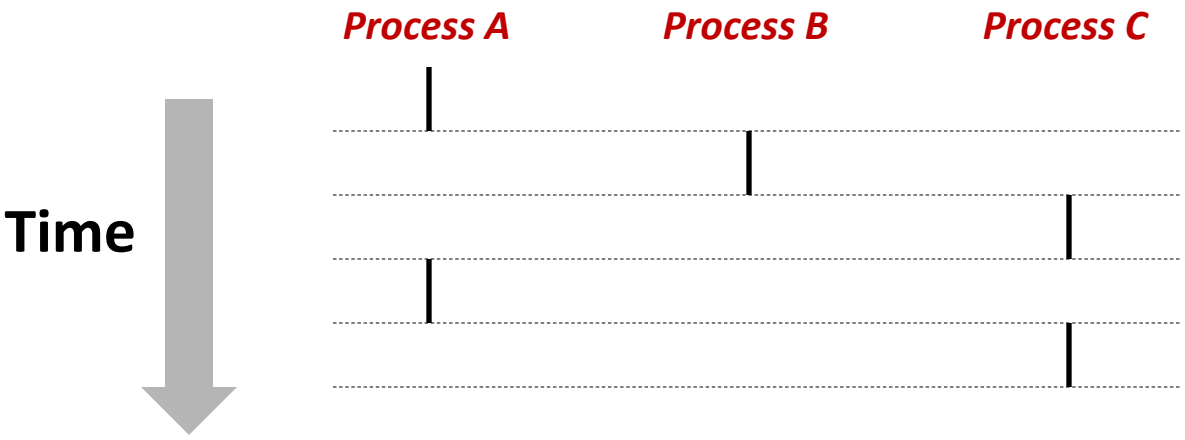
# Concurrent Processes

Two processes *run concurrently* (*are concurrent)* if their flows overlap in time

Otherwise, they are *sequential*

Examples (running on single core):

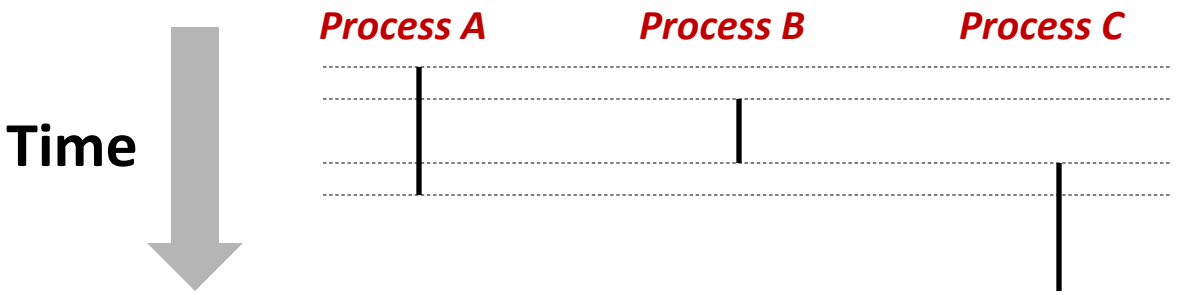
- Concurrent: A & B, A & C
- Sequential: B & C



# User View of Concurrent Processes

Control flows for concurrent processes are physically disjoint in time

However, we can think of concurrent processes as running in parallel with each other

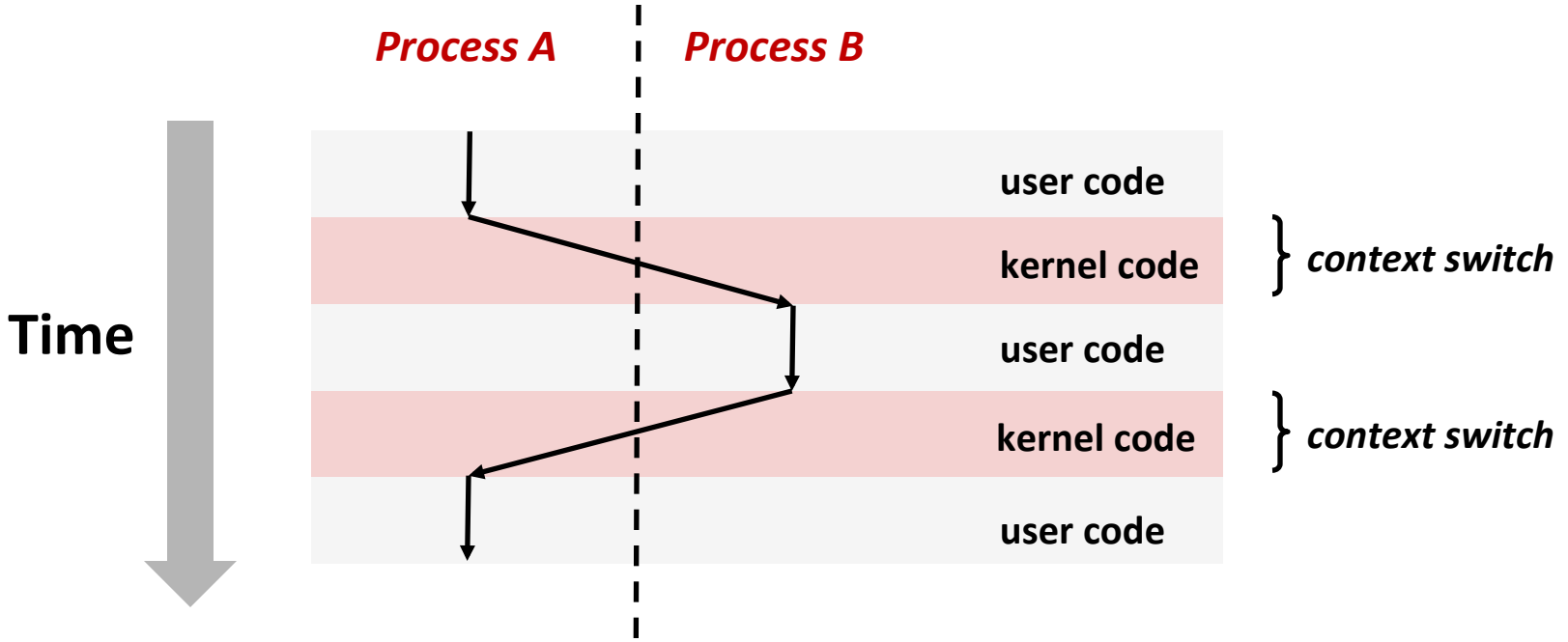


# Context Switching

Processes are managed by a shared chunk of OS code called the *kernel*

- Important: the kernel is not a separate process, but rather runs as part of some user process

Control flow passes from one process to another via a *context switch*



# `fork`: Creating New Processes

**`int fork(void)`**

- creates a new process (child process) that is identical to the calling process (parent process)
- returns 0 to the child process
- returns child's **`pid`** (process id) to the parent process

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Fork is interesting (and often confusing) because it is called *once* but returns *twice*

# Understanding fork

*Process n*

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

*Child Process m*

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

`hello from parent` *Which one is first?* `hello from child`

# Fork Example #1

Parent and child both run same code

- Distinguish parent from child by return value from **`fork`**

Start with same state, but each has private copy

- Including shared output file descriptor
- Relative ordering of their print statements undefined

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

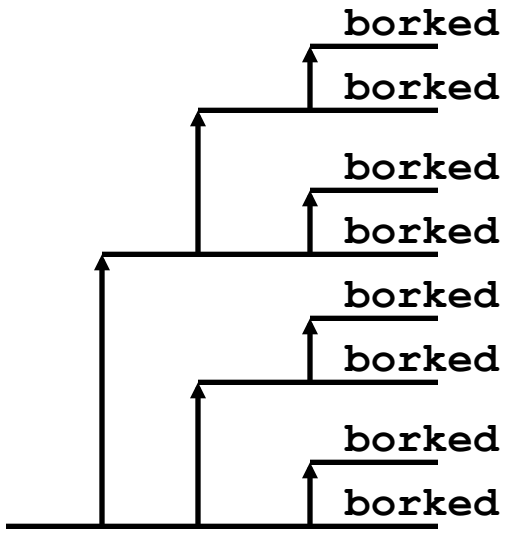
# Fork Example #2

```
#define bork fork

void fork3()
{
    bork(); bork(); bork();
    printf(”borked\n");
}
```

# Fork Example #2

Three consecutive forks

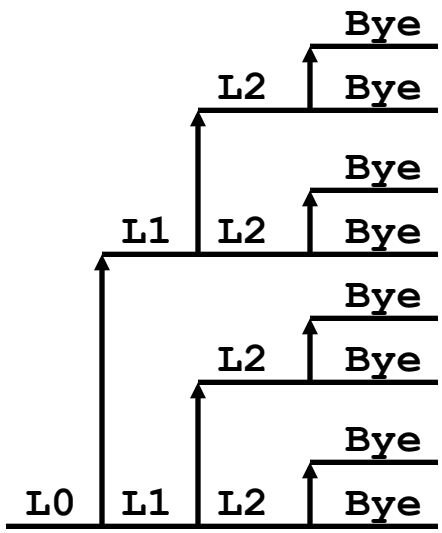
```
#define bork fork

void fork3()
{
    bork(); bork(); bork();
    printf("borked\n");
}
```



# Fork Example #3

Three consecutive forks

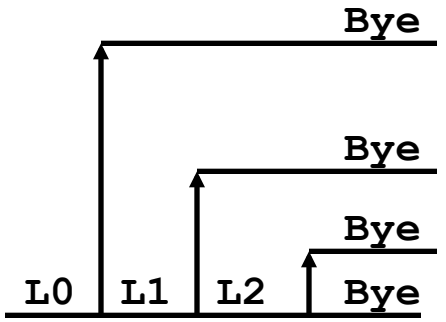
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



# Fork Example #4

Nested forks in parent

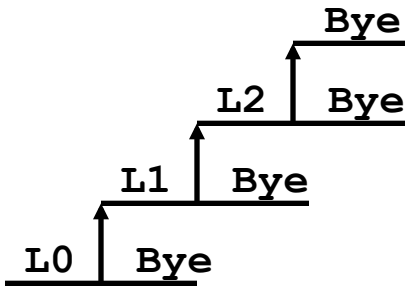
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
       printf("L1\n");
       if (fork() != 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



# Fork Example #5

Nested forks in children

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
       printf("L1\n");
       if (fork() == 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



# `exit`: Ending a process

**`void exit(int status)`**

- exits a process
  - Normally return with status 0
- **`atexit()`** registers functions to be executed upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

# Zombies

What happens on termination?

- When process terminates, still consumes system resources
- Various tables & info maintained by OS

Called a “zombie”

- Living corpse, half alive and half dead

# Zombies

What happens on termination?

- When process terminates, still consumes system resources
- Various tables & info maintained by OS

Called a "zombie"

- Living corpse, half alive and half dead

Reaping

- Performed by parent on terminated child (using **`wait`** or **`waitpid`**)
- Parent is given exit status information
- Kernel discards process

What if parent doesn't reap?

- If any parent terminates without reaping a child, then child will be reaped by **`init`** process (pid == 1)
- So, only need explicit reaping in long-running processes
  - e.g., shells and servers

# Zombie Example

**Fig. 1. Exemplary Zombies.**

# Zombie Example

```
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

**ps** shows child process as "defunct"

Killing parent allows child to be reaped by **init**

# Nonterminating Child Example

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

Child process still active even though parent has terminated

- Child is an *orphan*

Must kill explicitly, or else will keep running indefinitely

# `wait`: synchronizing with children

Parent reaps child by calling the wait function

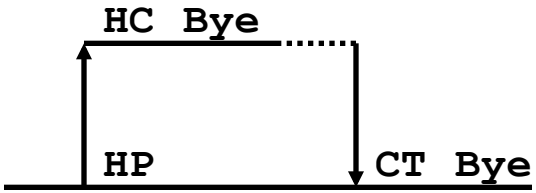
**`int wait(int *child_status)`**

- suspends current process until one of its children terminates
- return value is the **`pid`** of the child process that terminated
- if **`child_status != NULL`**, then the object it points to will be set to a status indicating why the child process terminated

Professional code uses signal handler (CS241 later lecture) for signal **`SIGCHLD`** which issues a **`wait()`** call

# `wait`: synchronizing with children

```
void fork9() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



# `wait()` Example

If multiple children completed, will take in arbitrary order

Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

# `waitpid()`: Waiting for a Specific Process

**`waitpid(pid, &status, options)`**

- suspends current process until specific process terminates
- various options (see man page or textbook)

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

# `execve`: Loading and Running Programs

```
int execve(
   char *filename,
   char *argv[],
   char *envp[]
)
```

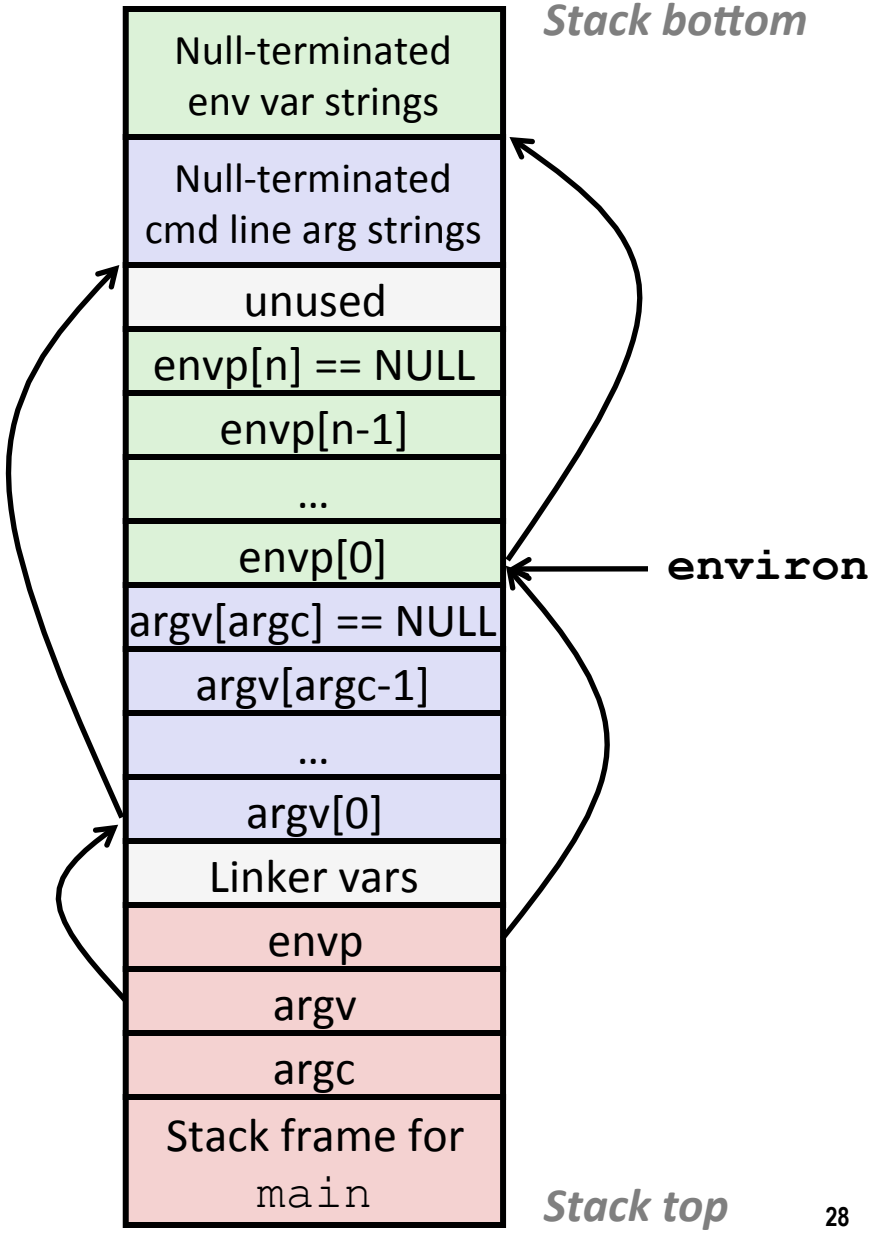
Loads and runs in current process:

- Executable **`filename`**
- With argument list **`argv`**
- And environment variable list **`envp`**

Does not return (unless error)

Overwrites code, data, and stack

- keeps pid, open files and signal context



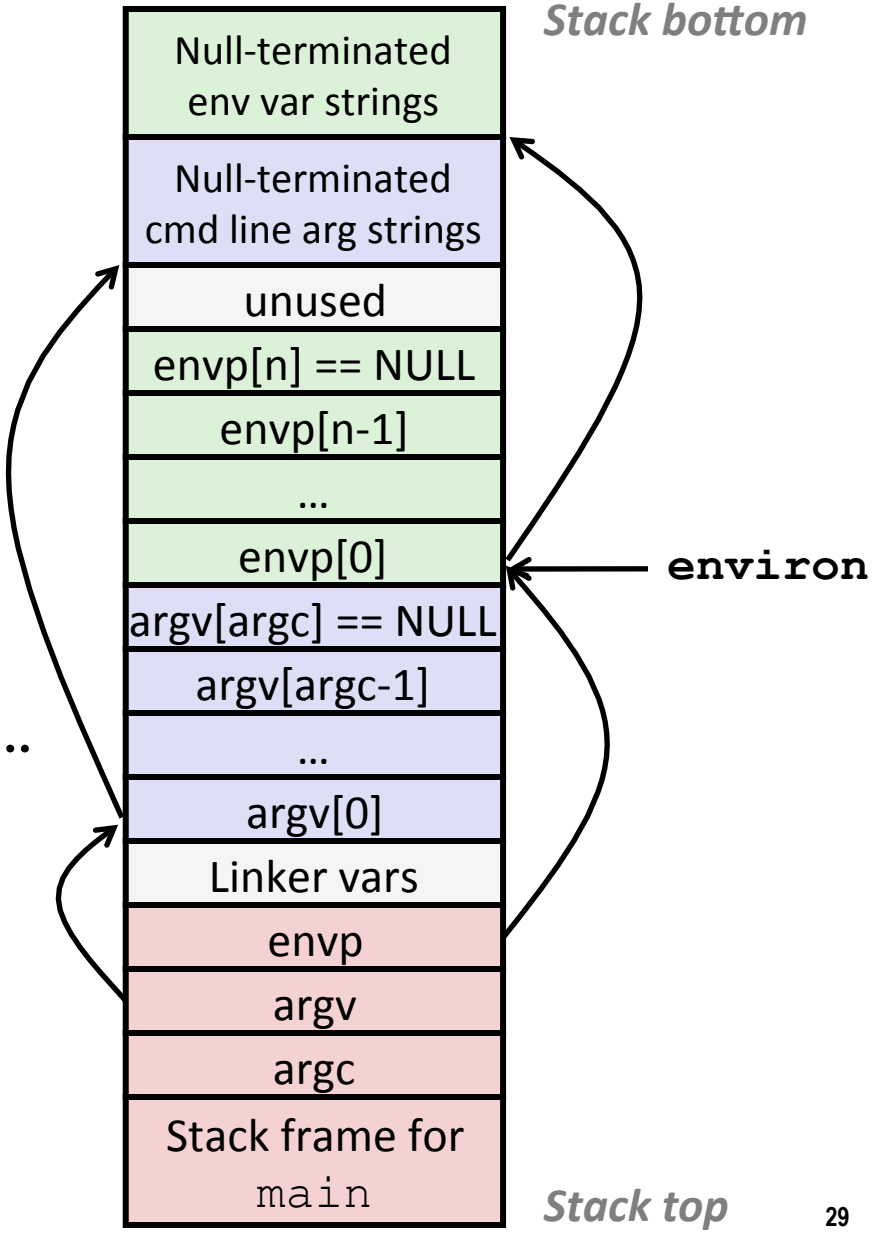
# `execve`: Loading and Running Programs

```
int execve(
   char *filename,
   char *argv[],
   char *envp[]
)
```

Environment variables:

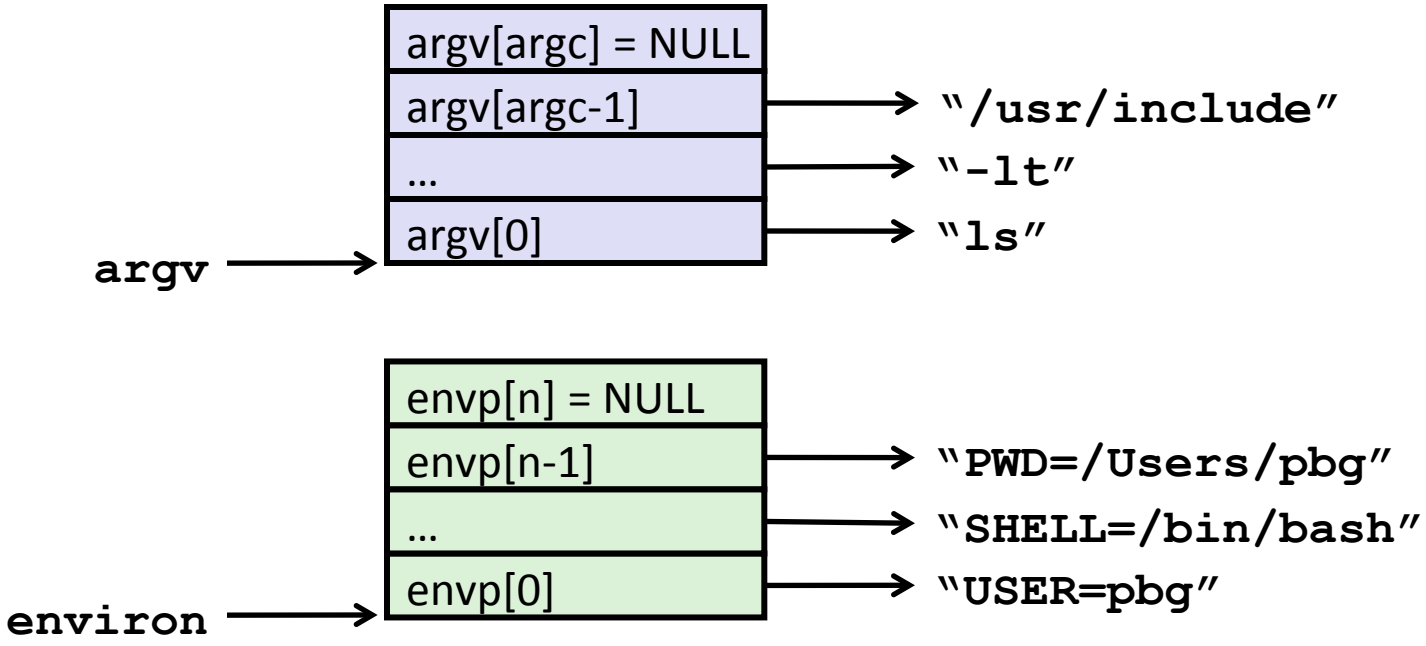
- "name=value" strings
- **`getenv`** and **`putenv`**

Simpler variants: **`execlp`**, **`execv`**, ...



# execve Example

```
if ((pid = fork()) == 0) { /* Child runs user job */
    if (execve(argv[0], argv, env) < 0) {
        printf("%s: Command not found.\n", argv[0]);
        exit(0);
    }
}
```



# Summary

Processes

- At any given time, system has multiple active processes
- But only one can execute at a time on a single core
- Each process appears to have total control of processor + private memory space

# Summary (cont.)

Spawning processes

- Call **`fork`**
- One call, two returns

Process completion

- Call **`exit`**
- One call, no return

Reaping and waiting for processes

- Call **`wait`** or **`waitpid`**

Loading and running programs

- Call **`execve`** (or "front-end" variant)
- One call, (normally) no return