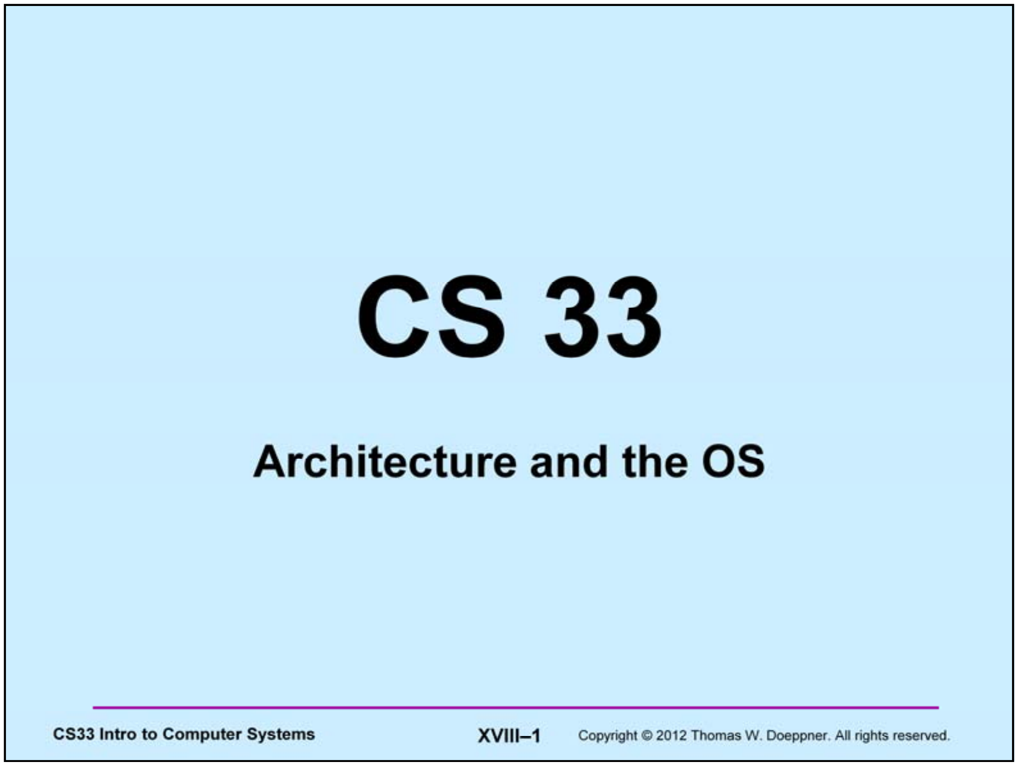
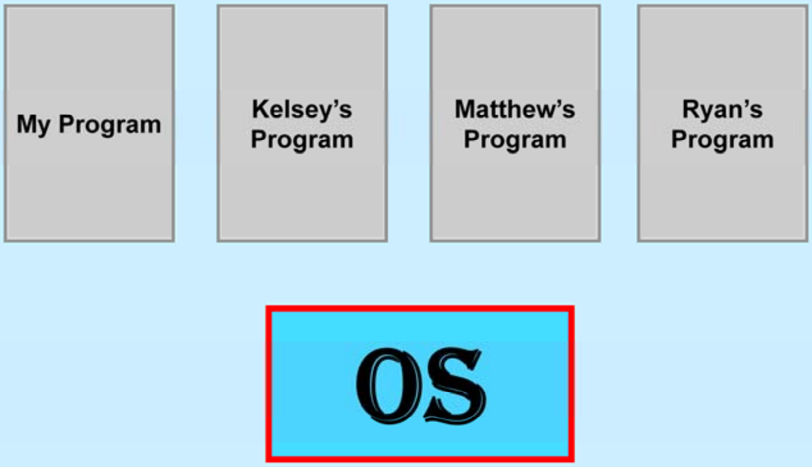
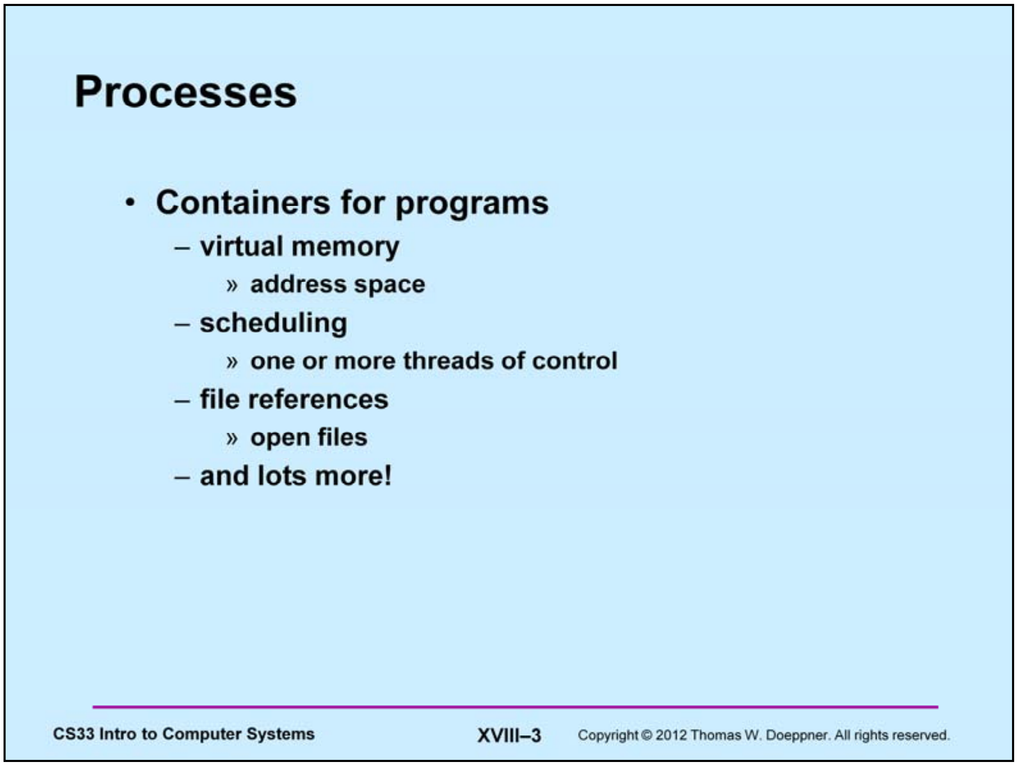


CS 33

Architecture and the OS

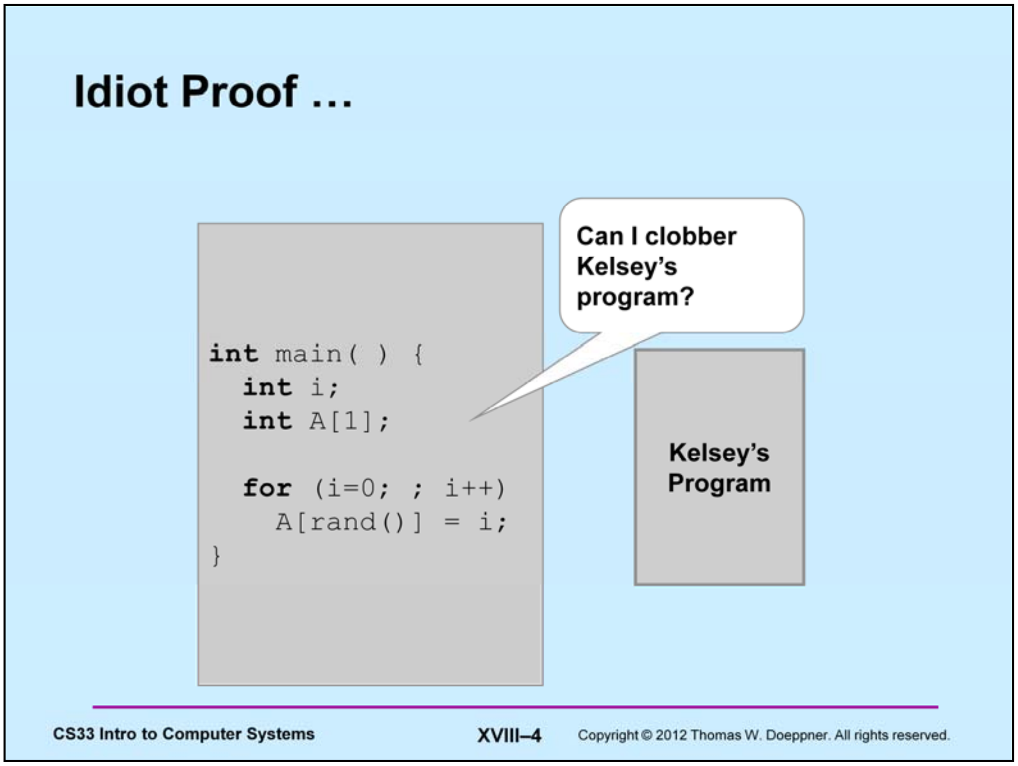
The Operating System





Processes

- Containers for programs
  - virtual memory
    - address space
  - scheduling
    - one or more threads of control
  - file references
    - open files
  - and lots more!



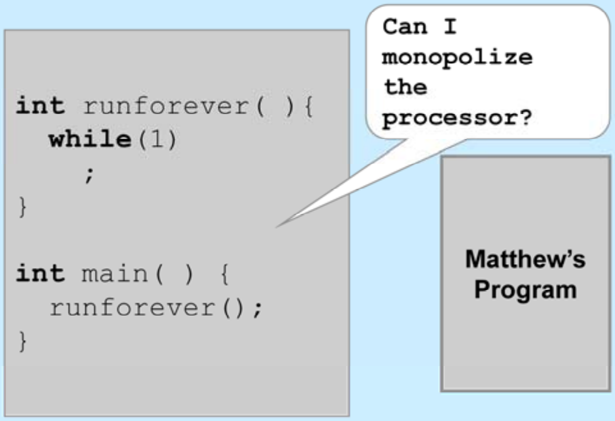
Idiot Proof …

```
int main( ) {
  int i;
  int A[1];

  for (i=0; ; i++)
    A[rand()] = i;
}
```

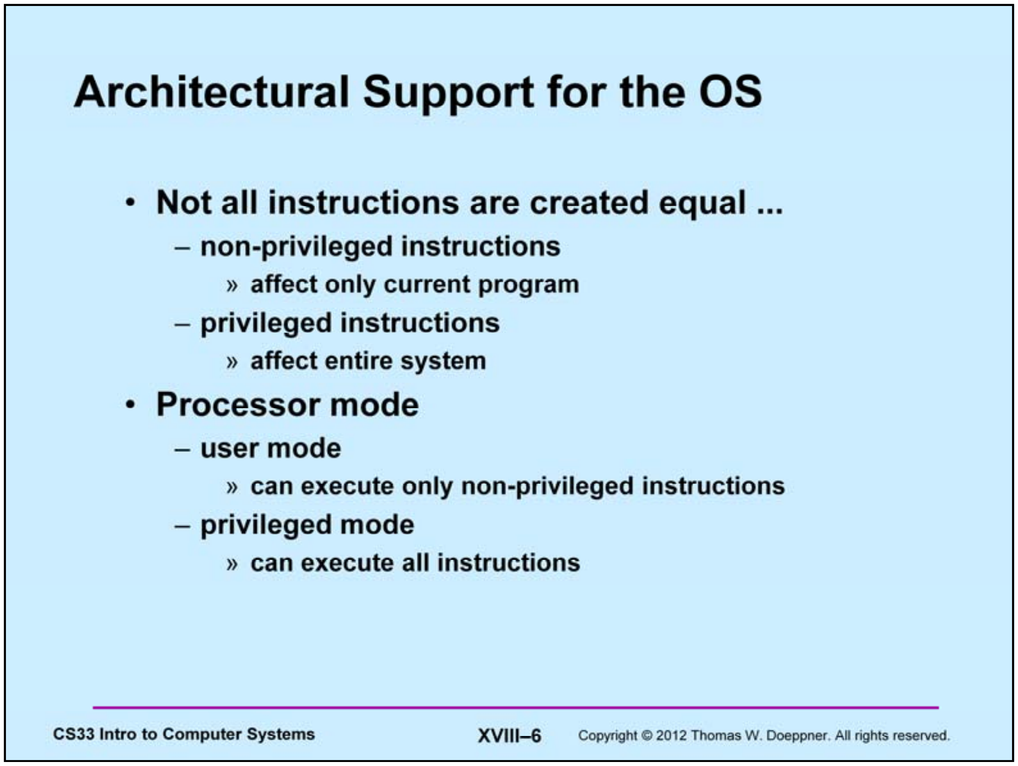
Can I clobber Kelsey's program?

Kelsey's Program

Fair Share

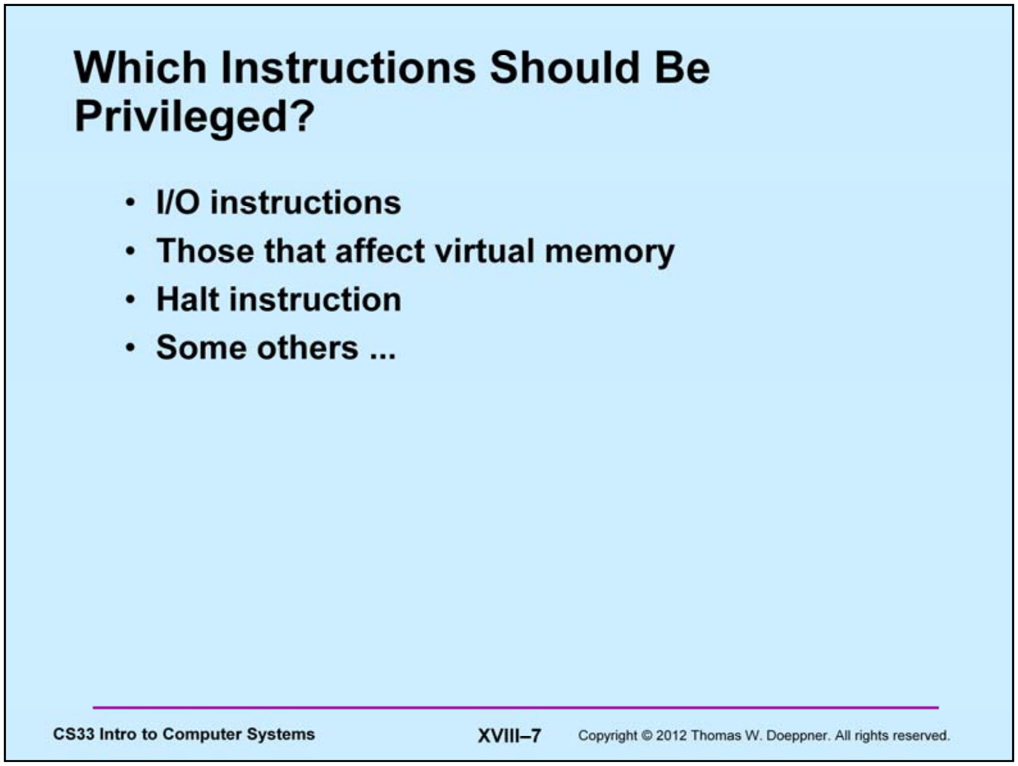


```
int runforever( ){
  while(1)
    ;
}

int main( ) {
  runforever();
}
```



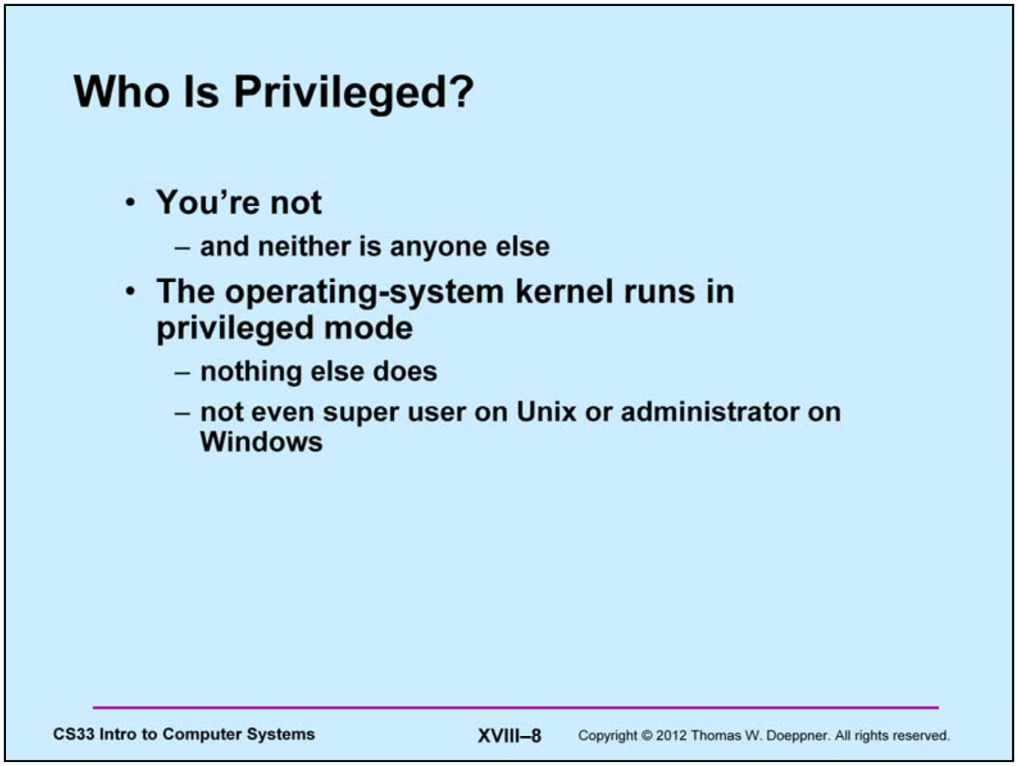
Architectural Support for the OS

- Not all instructions are created equal ...
  - non-privileged instructions
    - affect only current program
  - privileged instructions
    - affect entire system
- Processor mode
  - user mode
    - can execute only non-privileged instructions
  - privileged mode
    - can execute all instructions



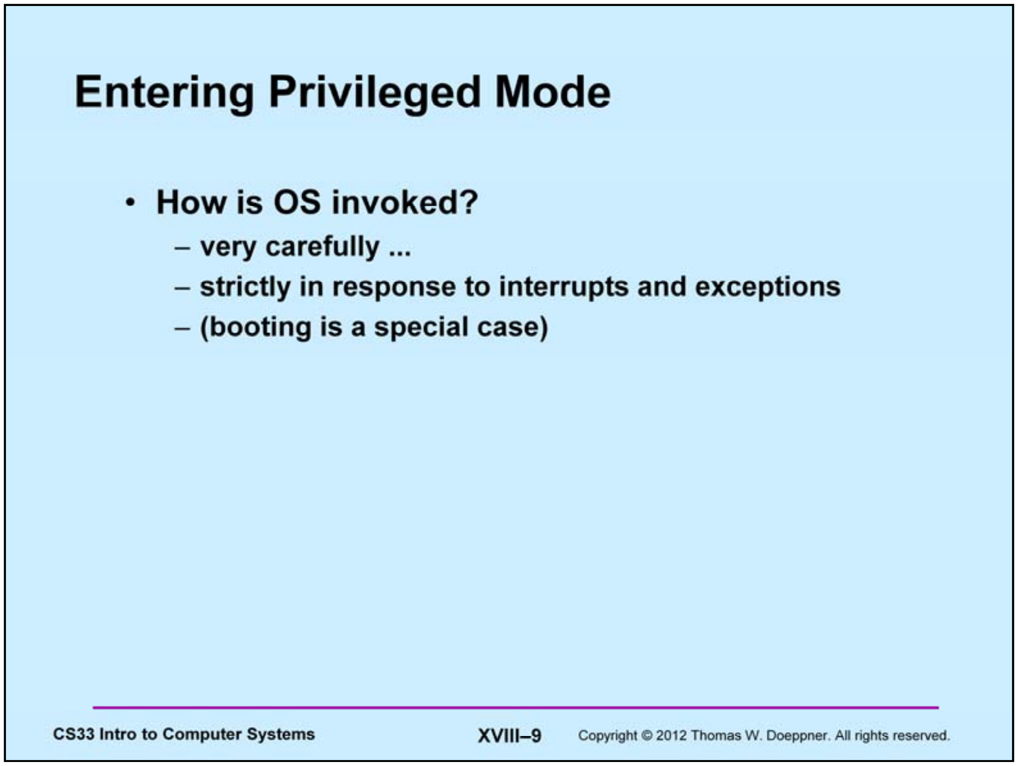
Which Instructions Should Be Privileged?

- I/O instructions
- Those that affect virtual memory
- Halt instruction
- Some others ...



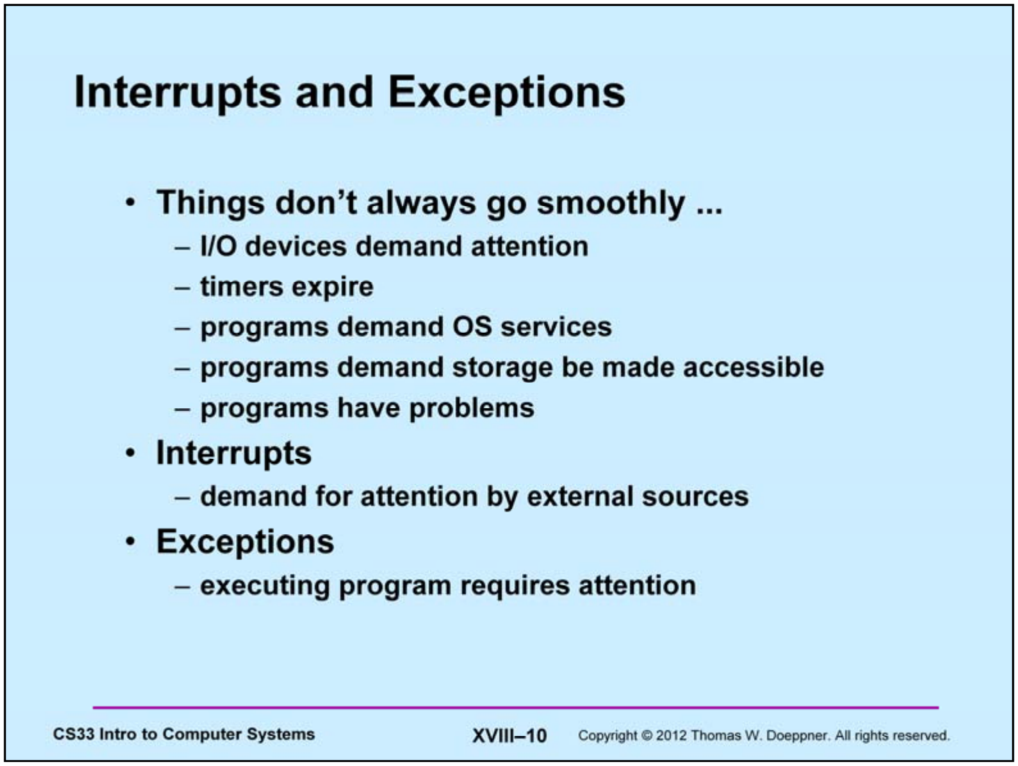
Who Is Privileged?

- You're not
  - and neither is anyone else
- The operating-system kernel runs in privileged mode
  - nothing else does
  - not even super user on Unix or administrator on Windows



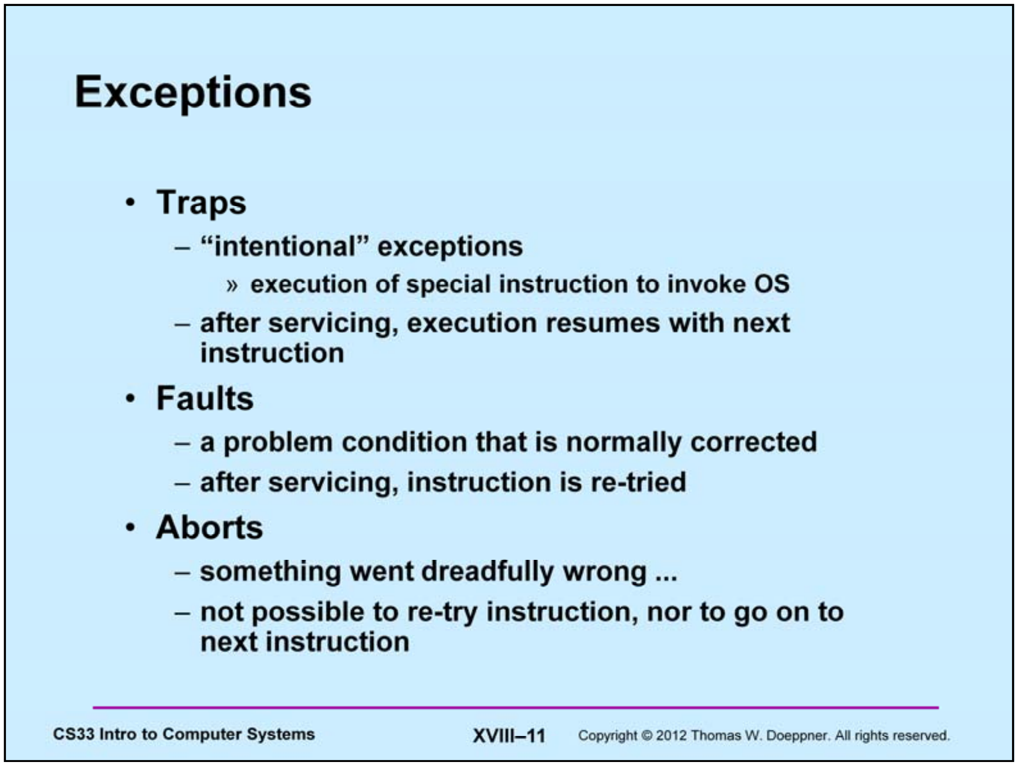
Entering Privileged Mode

- How is OS invoked?
  - very carefully ...
  - strictly in response to interrupts and exceptions
  - (booting is a special case)



Interrupts and Exceptions

- Things don’t always go smoothly ...
  - – I/O devices demand attention
  - – timers expire
  - – programs demand OS services
  - – programs demand storage be made accessible
  - – programs have problems
- Interrupts
  - – demand for attention by external sources
- Exceptions
  - – executing program requires attention



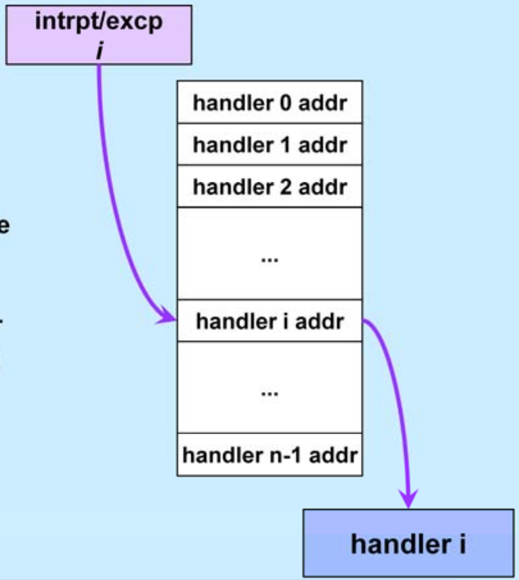
Exceptions

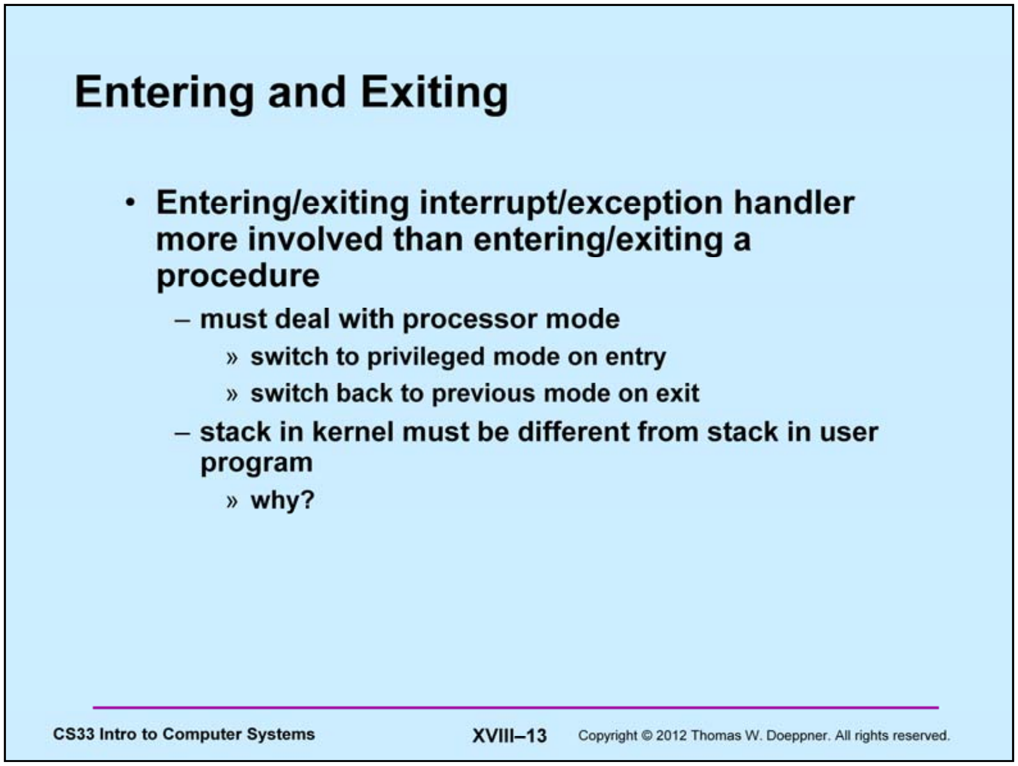
- Traps
  - “intentional” exceptions
    - » execution of special instruction to invoke OS
  - after servicing, execution resumes with next instruction
- Faults
  - a problem condition that is normally corrected
  - after servicing, instruction is re-tried
- Aborts
  - something went dreadfully wrong ...
  - not possible to re-try instruction, nor to go on to next instruction

These definitions follow those given in “Intel® 64 and IA-32 Architectures Software Developer’s Manual” and are generally accepted even outside of Intel.

Interrupt and Exception Handling

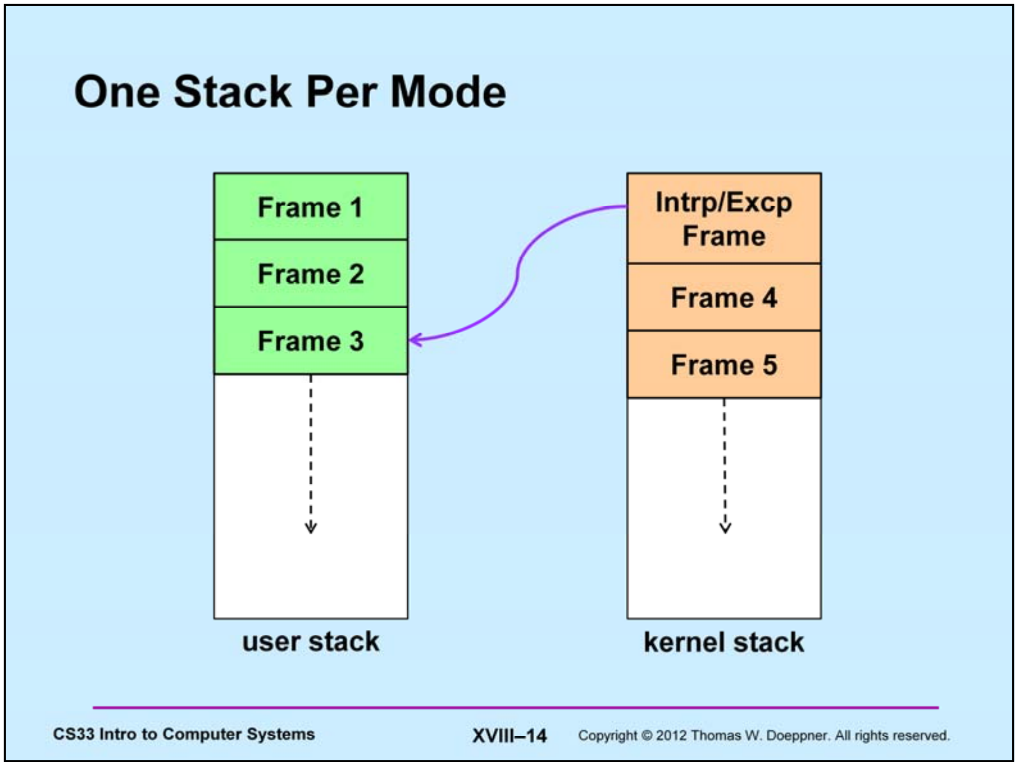
- Interrupt or exception invokes handler (in OS)
  - via interrupt and exception vector
    - » one entry for each possible interrupt/exception
      - contains
        - address of handler
  - code executed in privileged mode
    - » but code is part of the OS



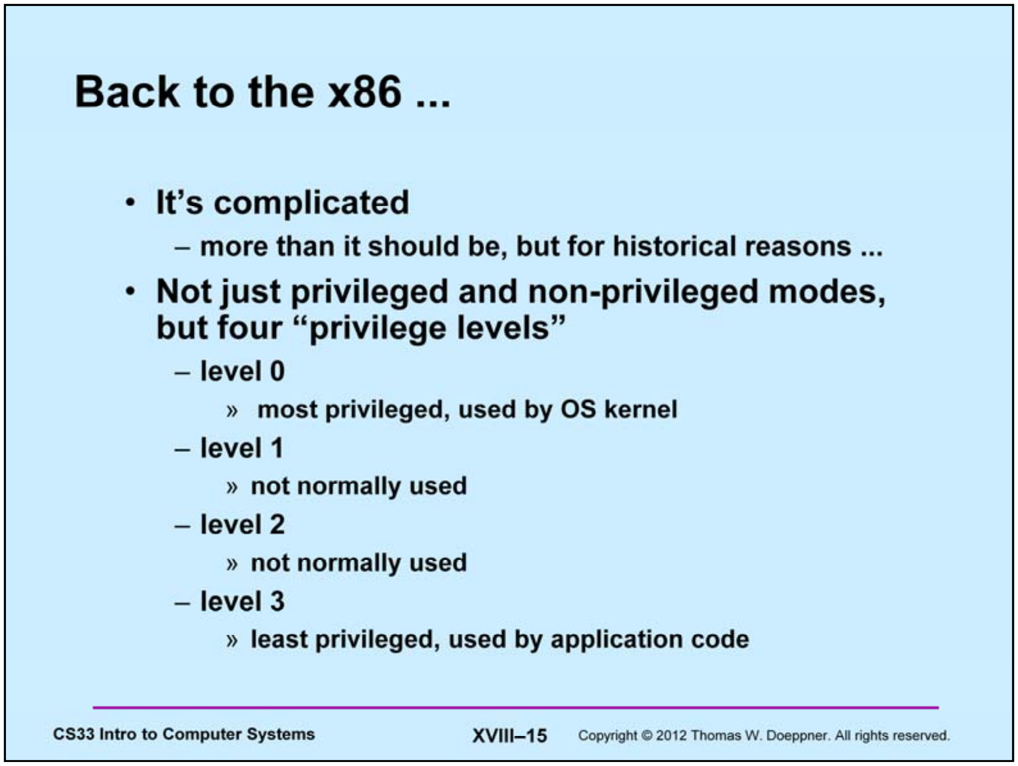


Entering and Exiting

- Entering/exiting interrupt/exception handler more involved than entering/exiting a procedure
  - must deal with processor mode
    - switch to privileged mode on entry
    - switch back to previous mode on exit
  - stack in kernel must be different from stack in user program
    - why?

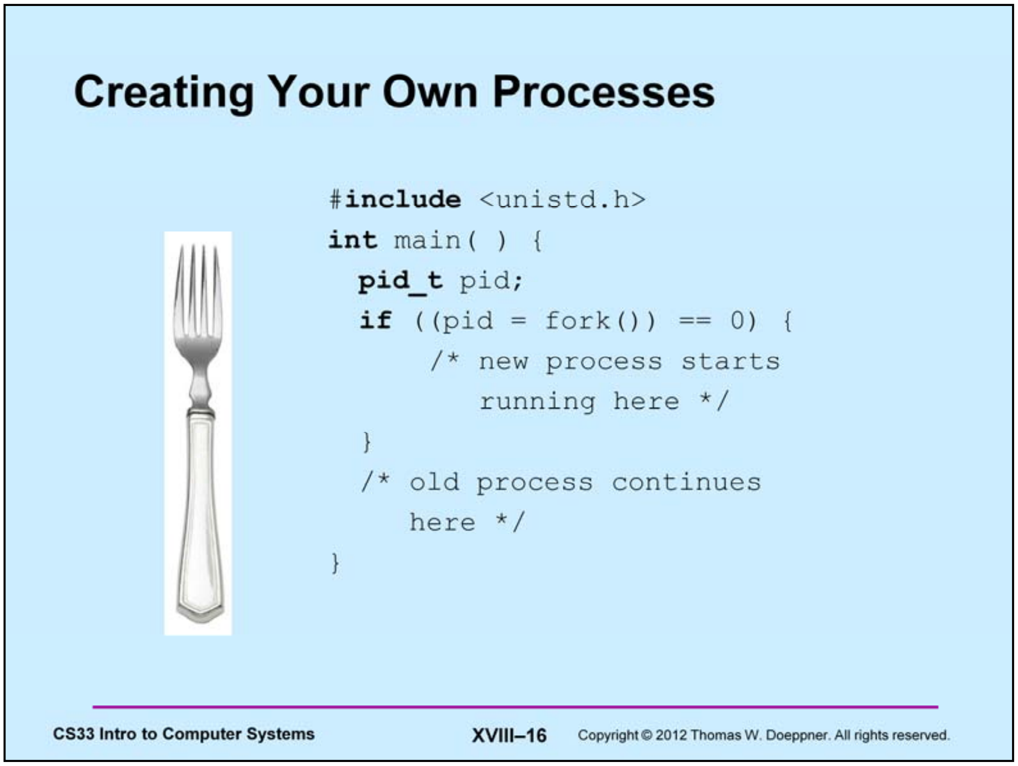


One Stack Per Mode



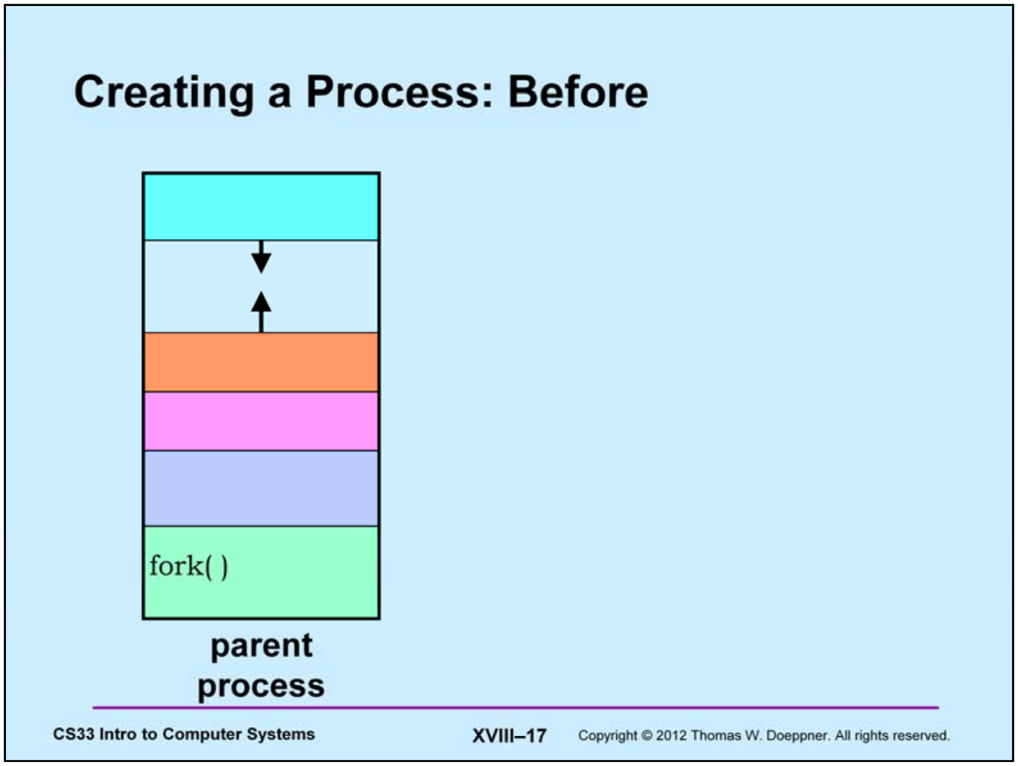
Back to the x86 ...

- It’s complicated
  - more than it should be, but for historical reasons ...
- Not just privileged and non-privileged modes, but four “privilege levels”
  - level 0
    - » most privileged, used by OS kernel
  - level 1
    - » not normally used
  - level 2
    - » not normally used
  - level 3
    - » least privileged, used by application code



Creating Your Own Processes

```
#include <unistd.h>
int main( ) {
  pid_t pid;
  if ((pid = fork()) == 0) {
      /* new process starts
         running here */
  }
  /* old process continues
     here */
}
```

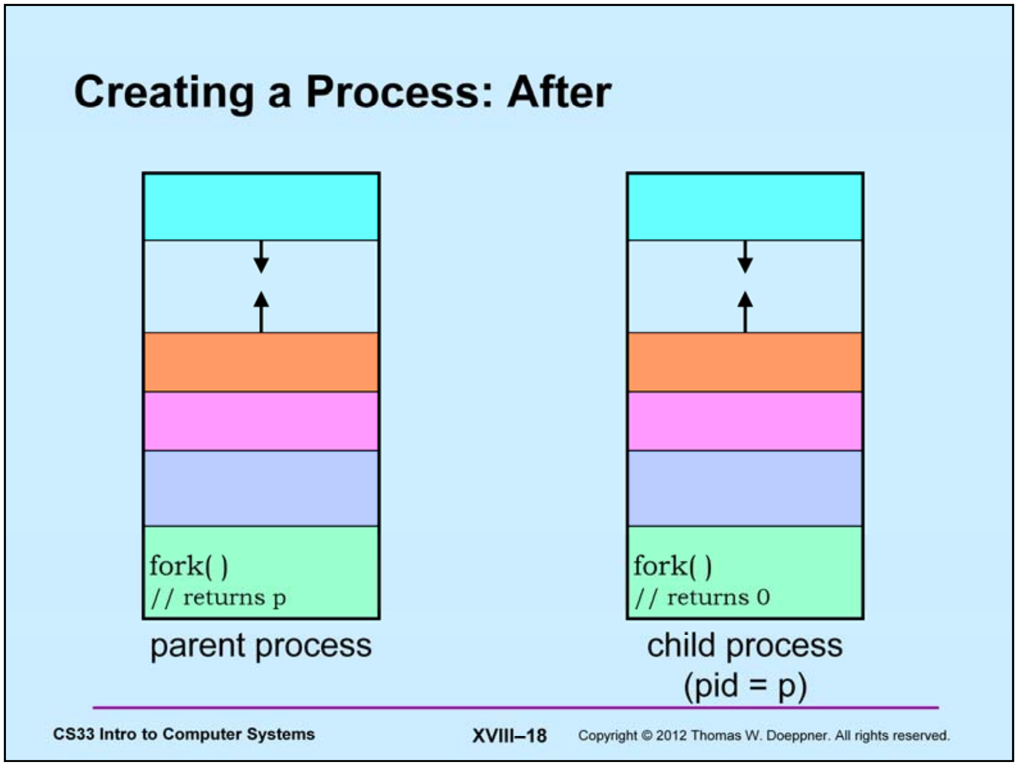


Creating a Process: Before

fork( )

parent process

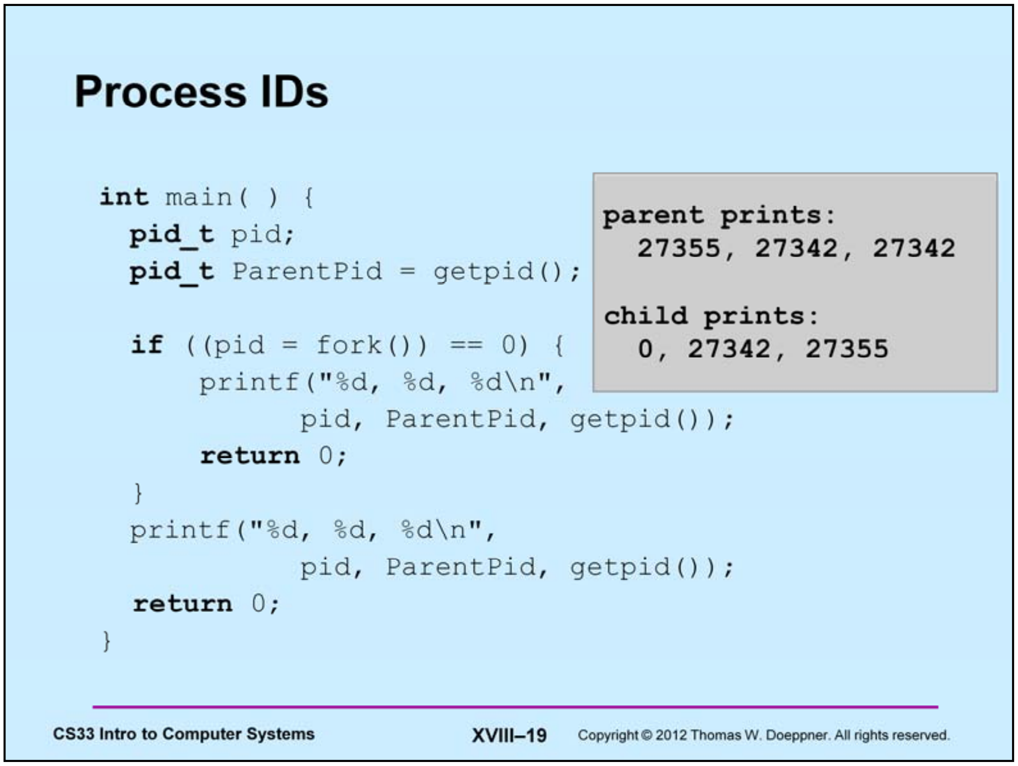
The only way to create a new process is to use the *fork* system call.



Creating a Process: After

By executing *fork* the parent process creates an almost exact clone of itself which we call the child process. This new process executes the same text as its parent, but contains a copy of the data and a copy of the stack. This copying of the parent to create the child can be very time-consuming. We discuss later how it is optimized.

Fork is a very unusual system call: one thread of control flows into it but two threads of control flow out of it, each in a separate address space. From the parent's point of view, fork does very little: nothing happens to the parent except that fork returns the process ID (PID—an integer) of the new process. The new process starts off life by returning from fork. It always views fork as returning a zero.

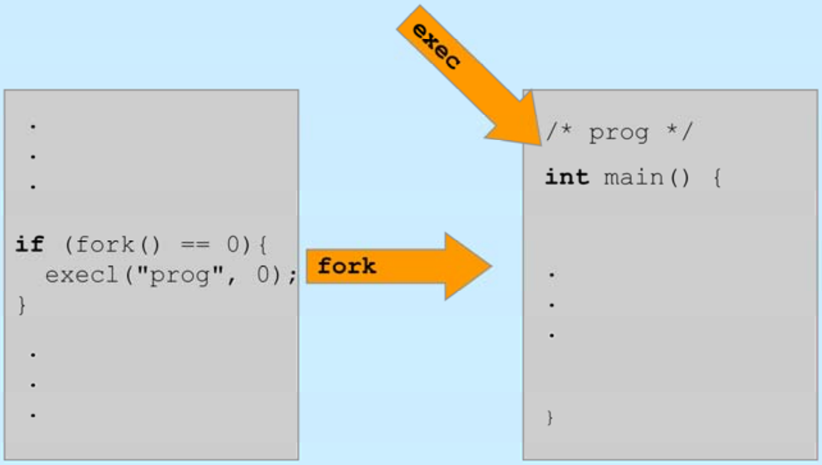


Process IDs

```
int main( ) {
  pid_t pid;
  pid_t ParentPid = getpid();

  if ((pid = fork()) == 0) {
      printf("%d, %d, %d\n",
             pid, ParentPid, getpid());
      return 0;
  }
  printf("%d, %d, %d\n",
             pid, ParentPid, getpid());
  return 0;
}
```

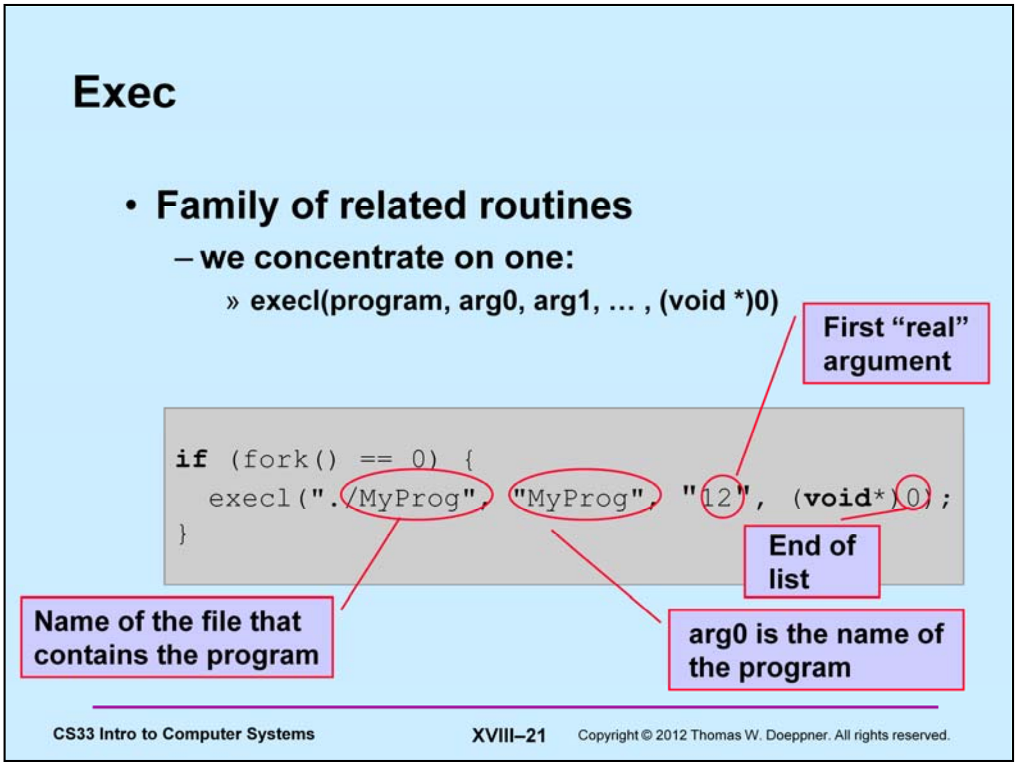
```
parent prints:
  27355, 27342, 27342

child prints:
  0, 27342, 27355
```

Putting Programs into Processes



```
 .
 .
 .

if (fork() == 0){
  execl("prog", 0);
}

 .
 .
 .
```

```
/* prog */
int main() {


 .
 .
 .


}
```



Exec

- Family of related routines
  - we concentrate on one:
    - » execl(program, arg0, arg1, … , (void *)0)

```
if (fork() == 0) {
  execl("./MyProg", "MyProg", "12", (void*)0);
}
```

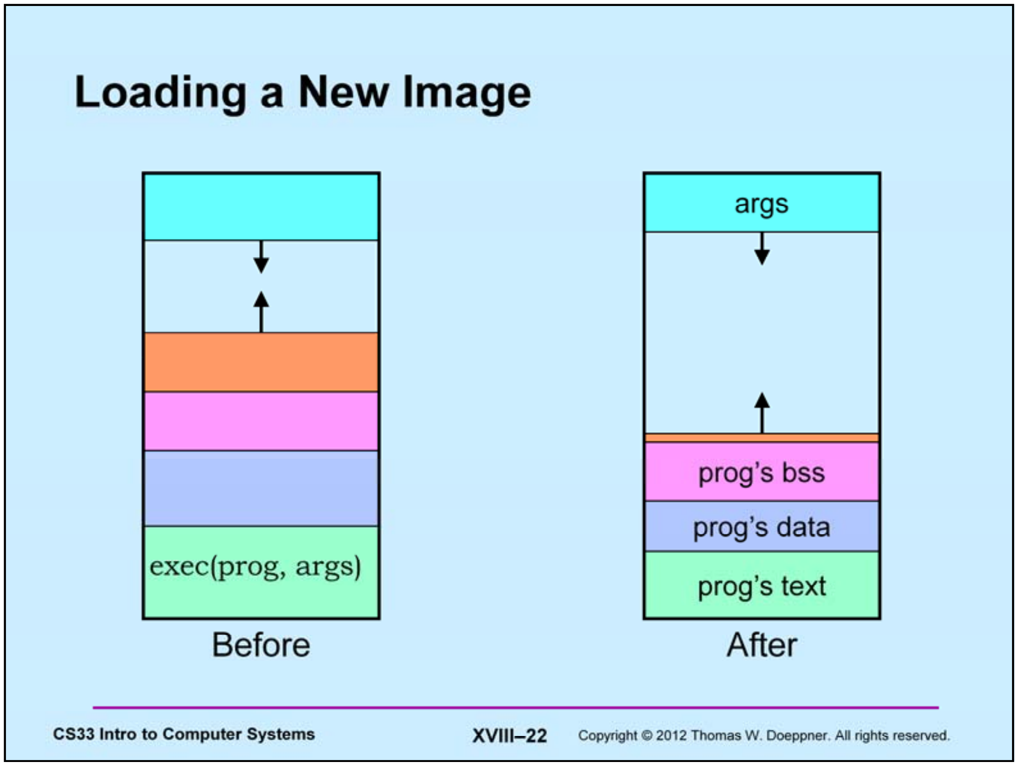
First “real” argument

End of list

Name of the file that contains the program

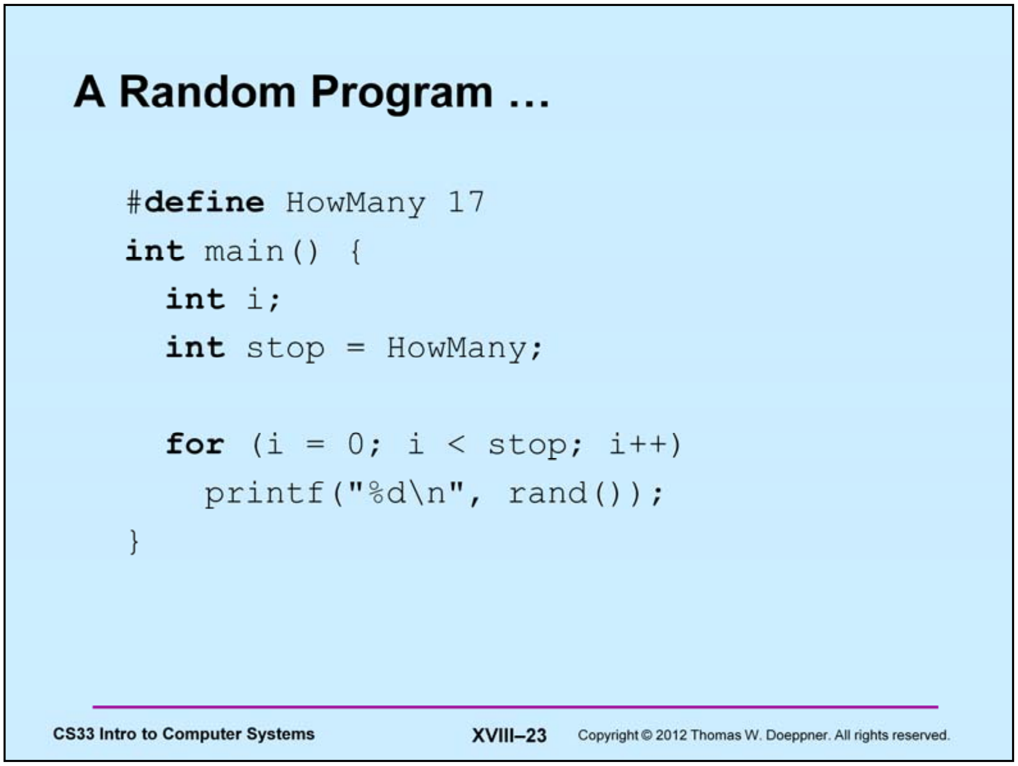
arg0 is the name of the program

Note that the end-of-list designator, known as a *sentinel*, must be cast as a void *.



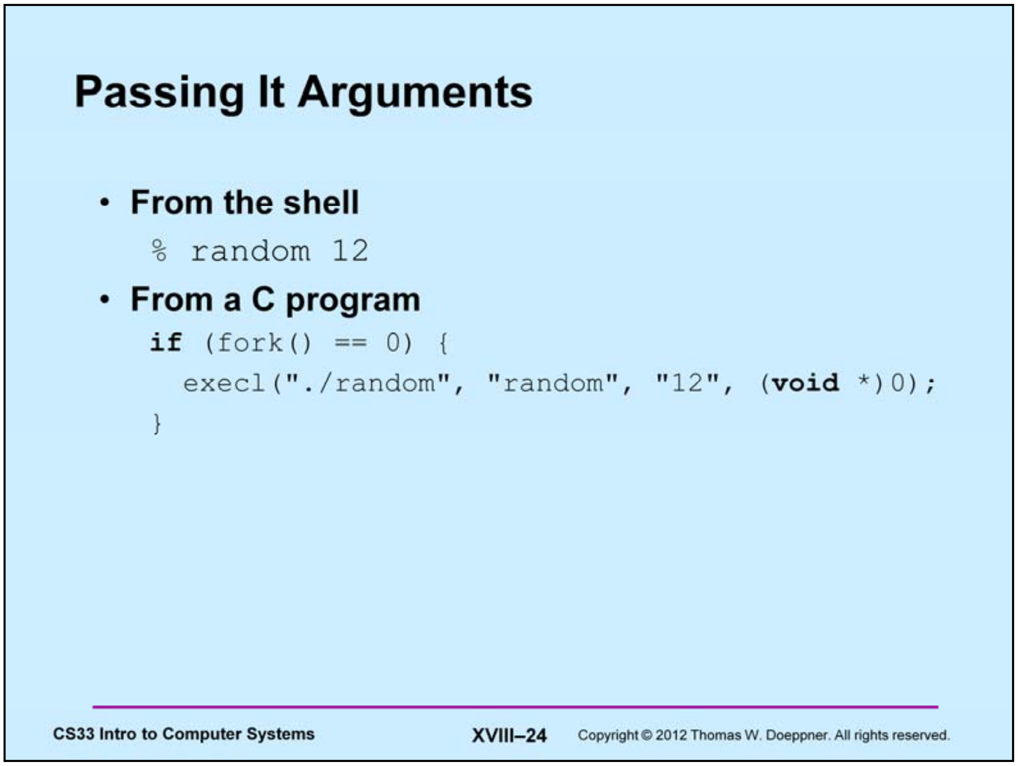
Loading a New Image

Most of the time the purpose of creating a new process is to run a new (i.e., different) program. Once a new process has been created, it can use the *exec* system call to load a new program image into itself, replacing the prior contents of the process's address space. Exec is passed the name of a file containing a fully relocated program image (which might require further linking via a runtime linker). The previous text region of the process is replaced with the text of the program image. The data, BSS and dynamic areas of the process are "thrown away" and replaced with the data and BSS of the program image. The contents of the process's stack are replaced with the arguments that are passed to the main procedure of the program.



A Random Program …

```
#define HowMany 17
int main() {
  int i;
  int stop = HowMany;

  for (i = 0; i < stop; i++)
    printf("%d\n", rand());
}
```



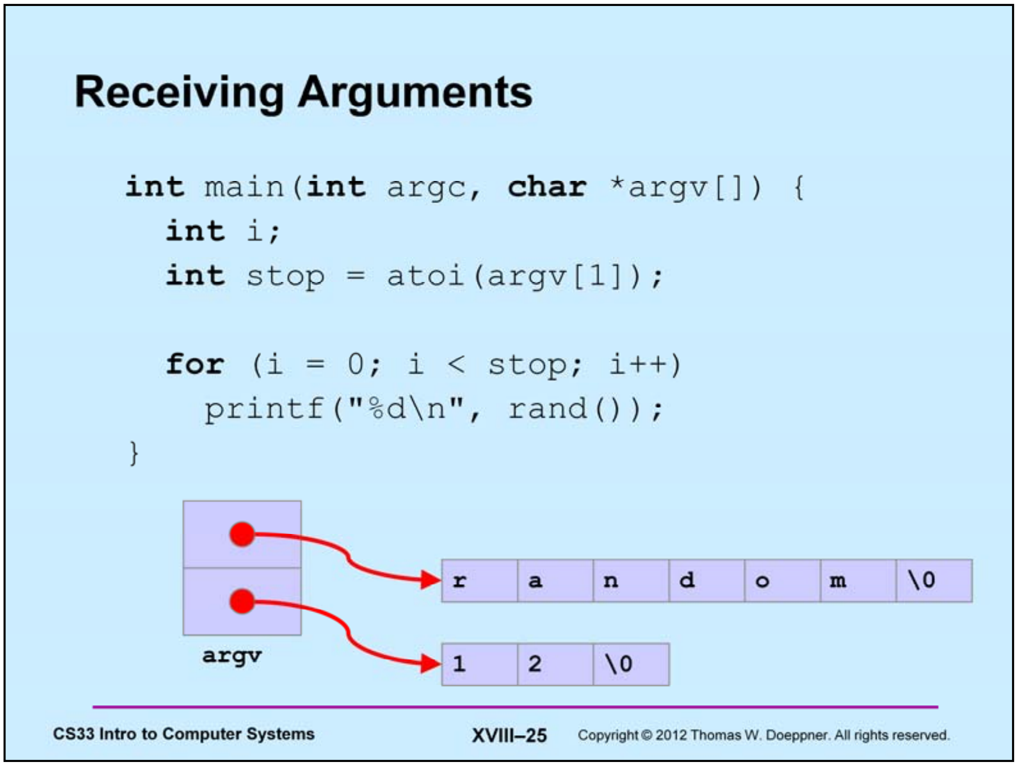
Passing It Arguments

- From the shell

```
% random 12
```

- From a C program

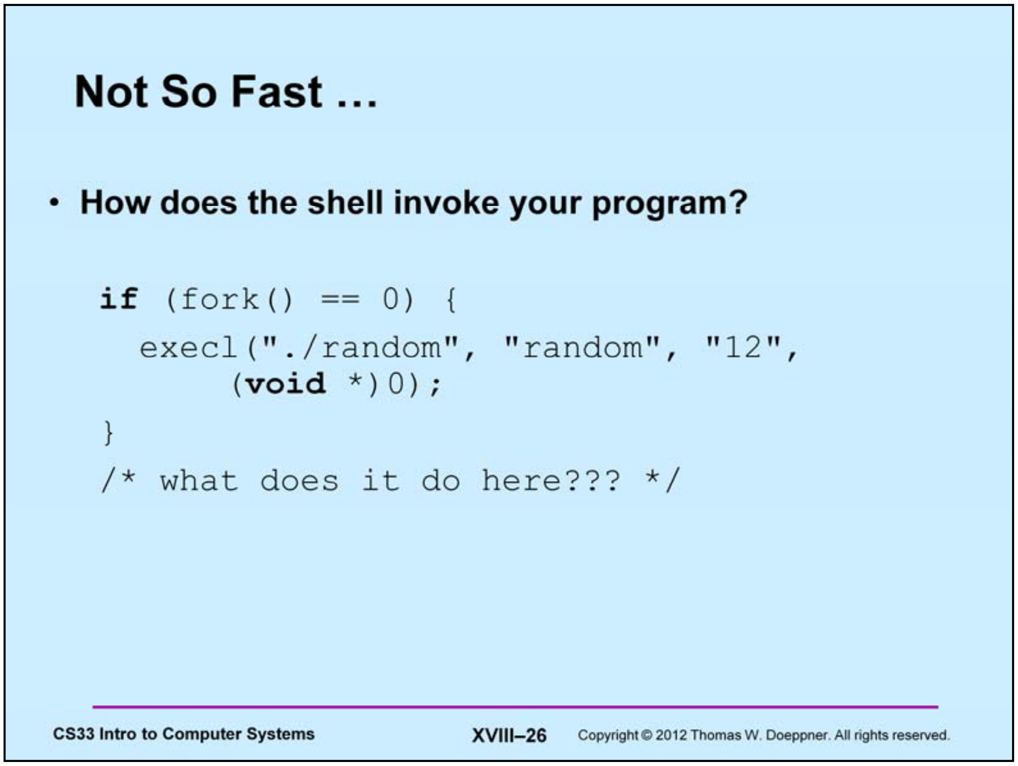
```
if (fork() == 0) {
  execl("./random", "random", "12", (void *)0);
}
```



Receiving Arguments

```
int main(int argc, char *argv[]) {
  int i;
  int stop = atoi(argv[1]);

  for (i = 0; i < stop; i++)
    printf("%d\n", rand());
}
```

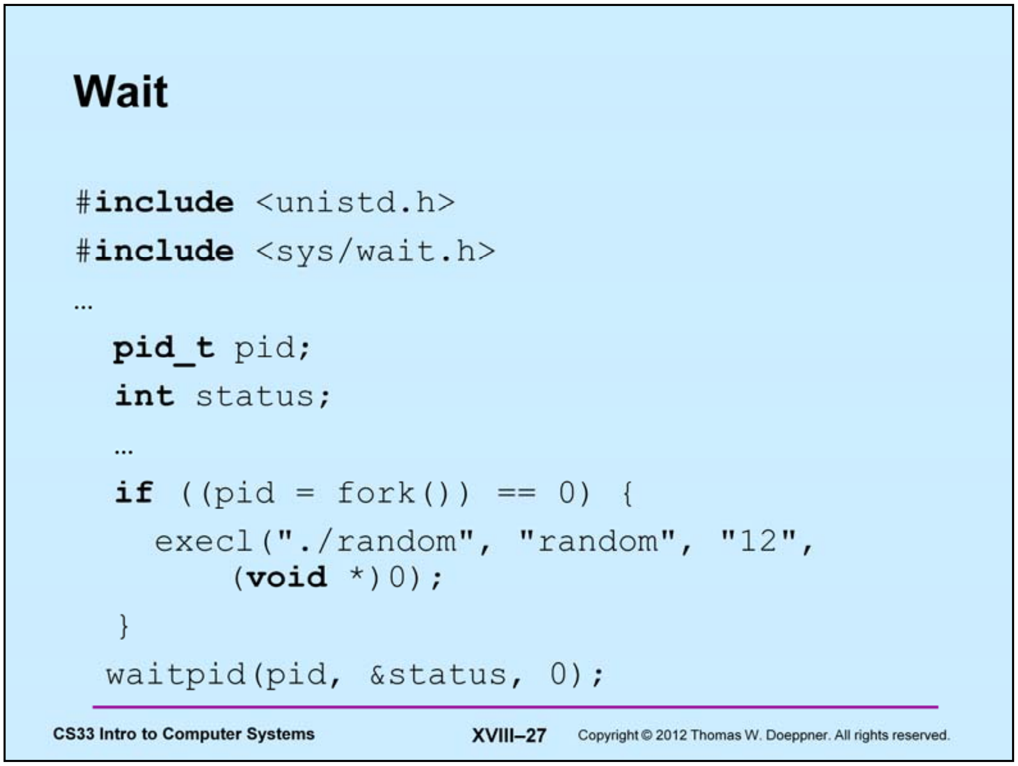
r
a
n
d
o
m
\0
1
2
\0
argv



Not So Fast …

- **How does the shell invoke your program?**

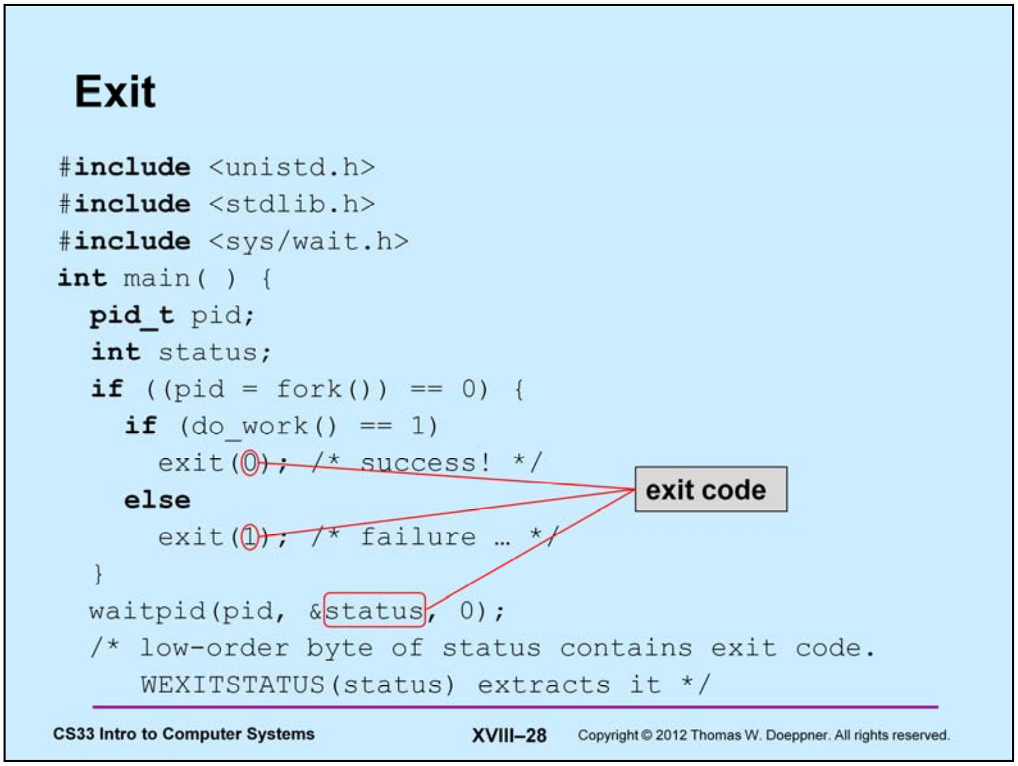
```
if (fork() == 0) {
  execl("./random", "random", "12",
       (void *)0);
}
/* what does it do here??? */
```



Wait

```
#include <unistd.h>
#include <sys/wait.h>
…
  pid_t pid;
  int status;
  …
  if ((pid = fork()) == 0) {
    execl("./random", "random", "12",
        (void *)0);
  }
  waitpid(pid, &status, 0);
```

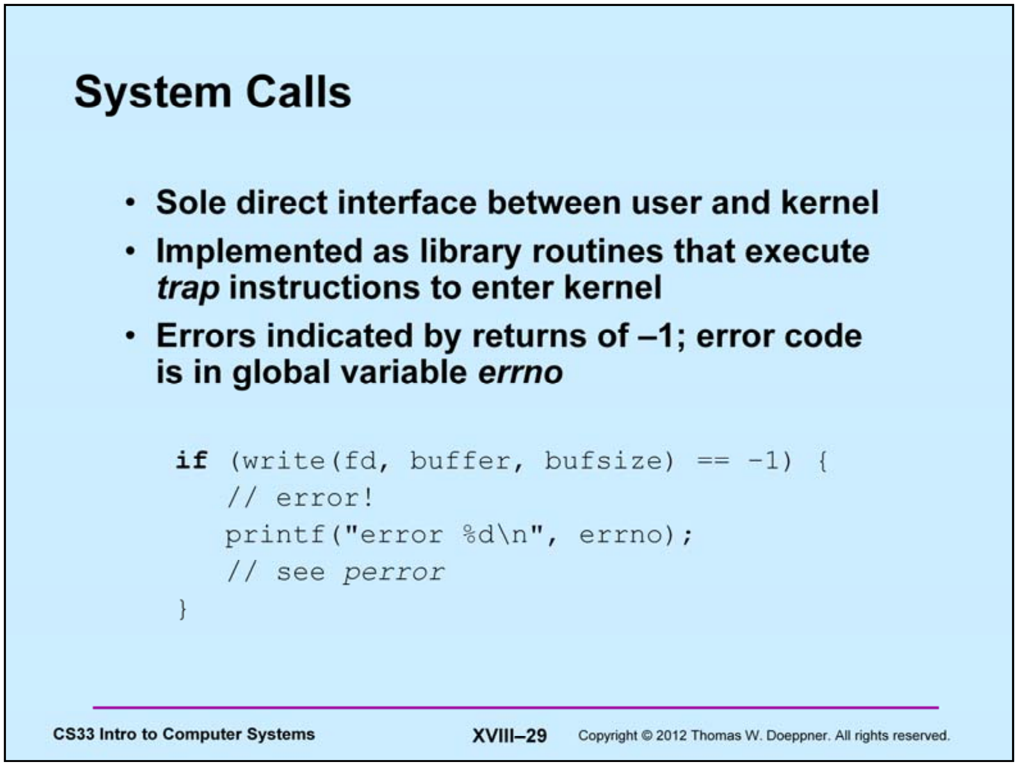
There’s a variant of *waitpid*, called *wait*, that waits for any child of the current process to terminate.



Exit

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main( ) {
  pid_t pid;
  int status;
  if ((pid = fork()) == 0) {
    if (do_work() == 1)
      exit(0); /* success! */
    else
      exit(1); /* failure … */
  }
  waitpid(pid, &status, 0);
  /* low-order byte of status contains exit code.
     WEXITSTATUS(status) extracts it */
```

exit code



System Calls

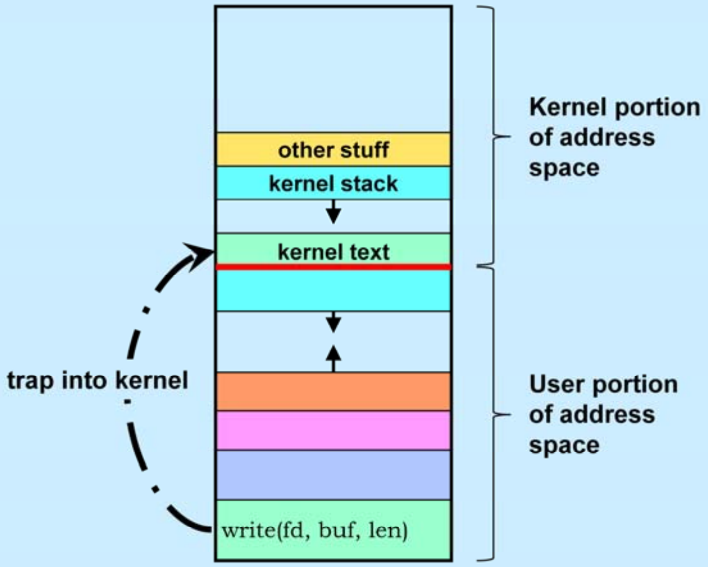
- **Sole direct interface between user and kernel**
- **Implemented as library routines that execute *trap* instructions to enter kernel**
- **Errors indicated by returns of –1; error code is in global variable *errno***

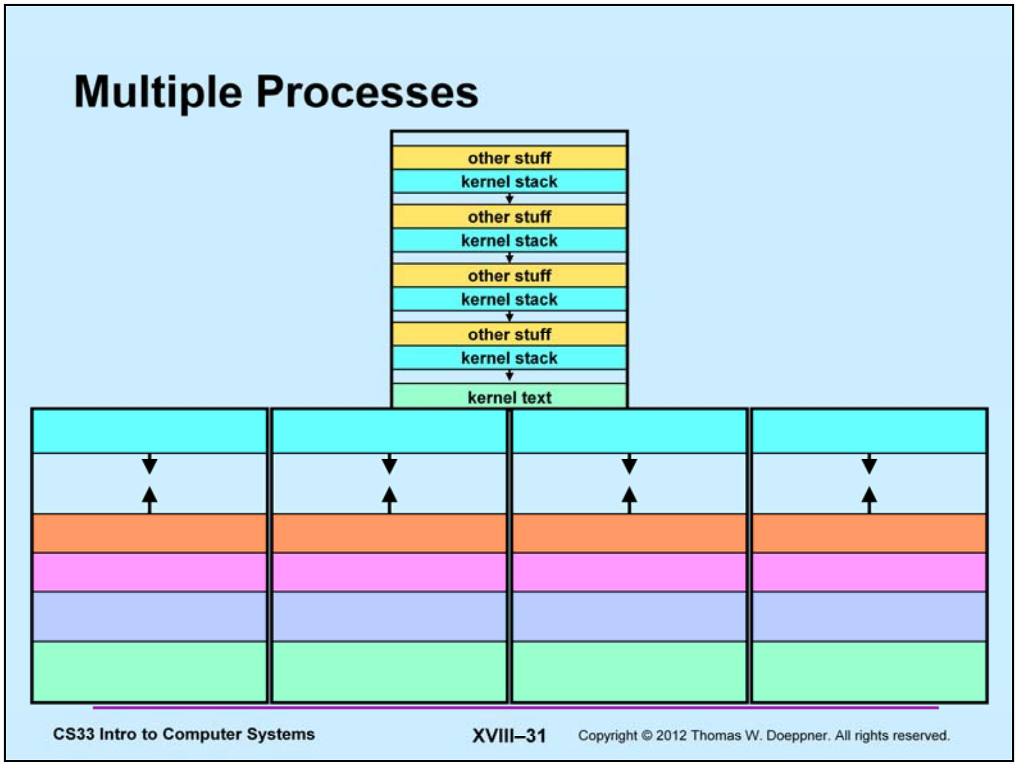
```
if (write(fd, buffer, bufsize) == -1) {
   // error!
   printf("error %d\n", errno);
   // see perror
}
```

System calls, such as *fork*, *execv*, *read*, *write*, etc., are the only means for application programs to communicate directly with the kernel: they form an API (application program interface) to the kernel. When a program calls such a routine, it is actually placing a call to a subroutine in a system library. The body of this subroutine contains a hardware-specific trap instruction which transfers control and some parameters to the kernel. On return to this library return, the kernel provides an indication of whether or not there was an error and what the error was. The error indication is passed back to the original caller via the functional return value of the library routine. If there was an error, a positive-integer code identifying it is stored in the global variable *errno*. Rather than simply print this code out, as shown in the slide, one might instead print out an informative error message. This can be done via the *perror* routine.

The "hardware-specific trap instruction" is (or used to be) the "int" (interrupt) instruction on the x86. However, it's considered too expensive for such performance-critical operations as system calls. A new facility, known as "sysenter/sysexit" was introduced with the Pentium II processors (in 1997) and has been used by operating systems (including Windows and Linux) ever since. Its description is beyond the scope of this course.

System Calls



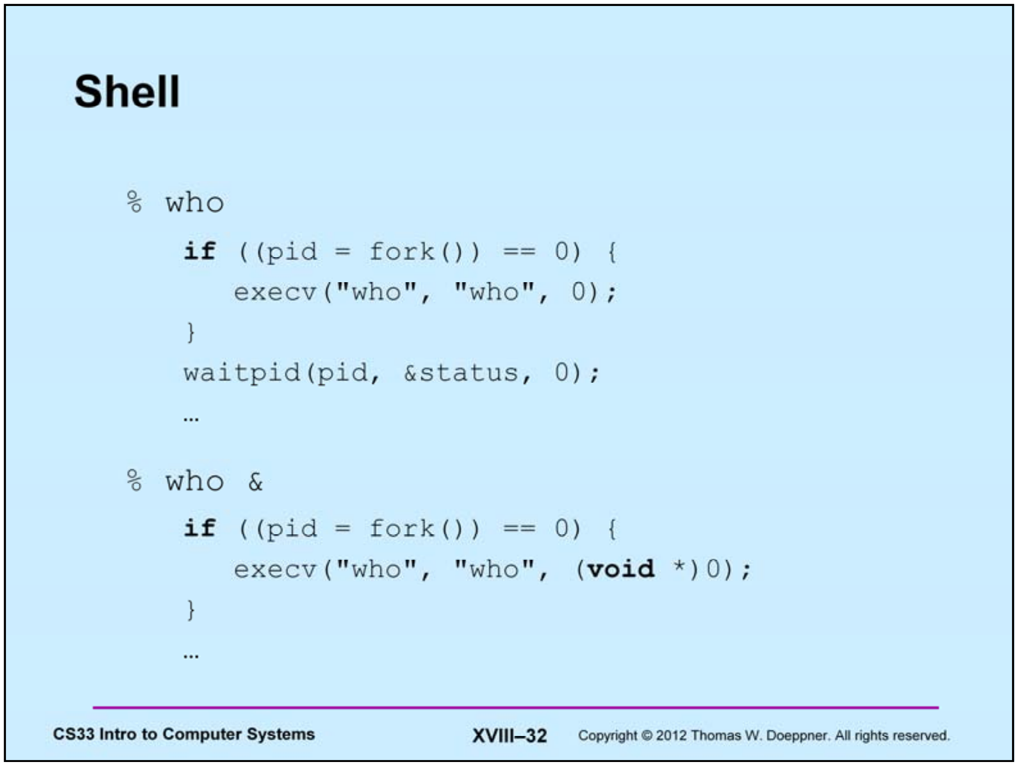


Multiple Processes

CS33 Intro to Computer Systems

XVIII–31

Each process has its own user address space, but there’s a single kernel address space. It contains context information for each user process, including the stacks used by each process when executing system calls.



Shell

```
% who
    if ((pid = fork()) == 0) {
        execv("who", "who", 0);
    }
    waitpid(pid, &status, 0);
    …

% who &
    if ((pid = fork()) == 0) {
        execv("who", "who", (void *)0);
    }
    …
```