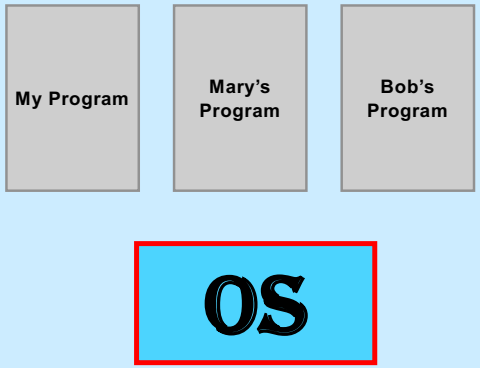
CS 33

Architecture and the OS

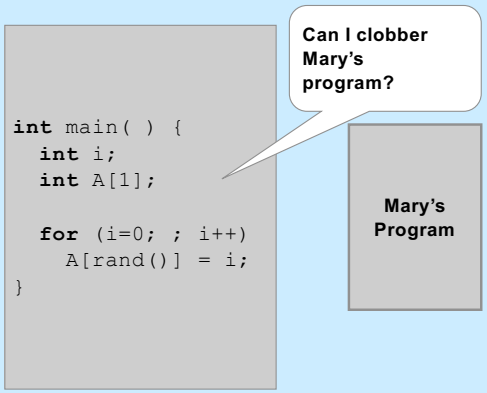
The Operating System



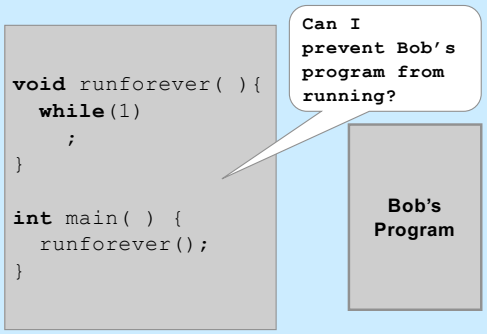
Processes

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

Idiot Proof …



Fair Share



Architectural Support for the OS

- Not all instructions are created equal ...
  - non-privileged instructions
    - » can affect only current program
  - privileged instructions
    - » may 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 how memory is mapped
- Halt instruction
- Some others ...

Who Is Privileged?

- No one
  - user code always runs in user mode
- 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 from 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

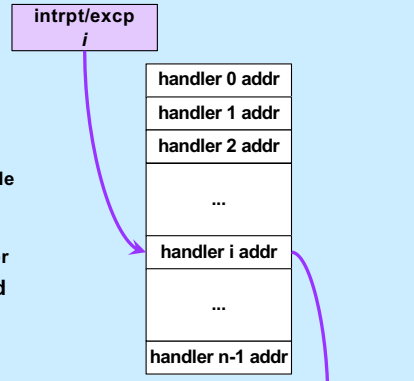
These definitions follow those given in "Intel® 64 and IA-32 Architectures Software Developer's Manual" and are generally accepted even outside of Intel.

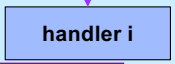
Actions for Interrupts and Exceptions

- When interrupt or exception occurs
  - processor saves state of current thread/process on stack
  - processor switches to privileged mode (if not already there)
  - invokes handler for interrupt/exception
  - if thread/process is to be resumed (typical action after interrupt)
    - » thread/process state is restored from stack
  - if thread/process is to re-execute current instruction
    - » thread/process state is restored, after backing up instruction pointer
  - if thread/process is to terminate
    - » it's terminated

Interrupt and Exception Handlers

- 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

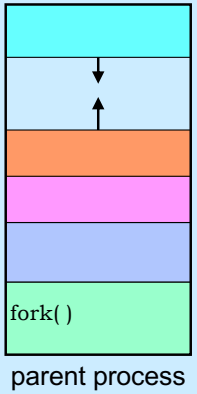




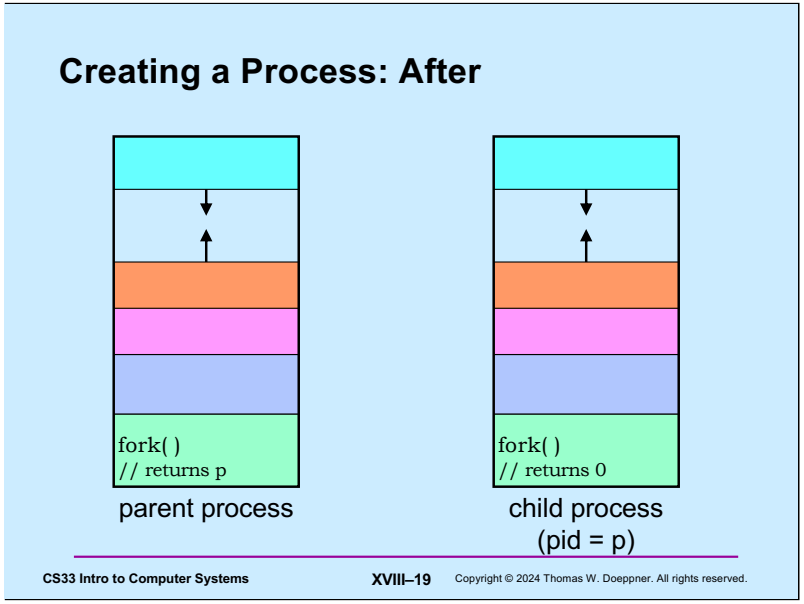
Creating Your Own Processes

```
#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



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



By executing **fork** the parent process creates an almost exact clone of itself that 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 if done naively. Some tricks are employed to make it much less so.

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, which it sees as returning a zero.

Quiz 1

The following program

a) runs forever

b) terminates quickly

```
int flag;
int main() {
  while (flag == 0) {
    if (fork() == 0) {
      // in child process
      flag = 1;
      exit(0);  // causes process to terminate
    }
  }
}
```


Process IDs

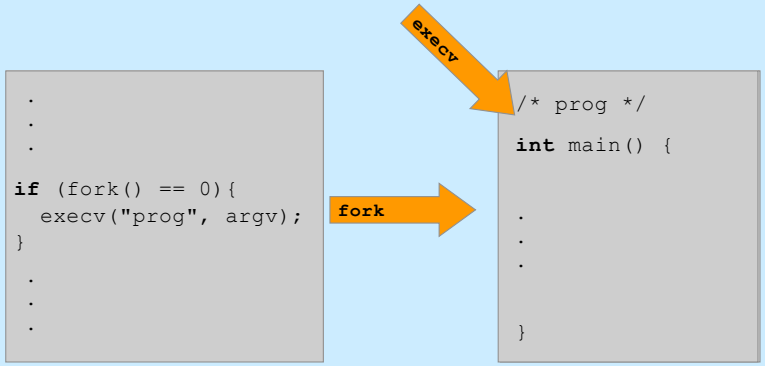
```
int main( ) {
  pid_t pid;
  pid_t ParentPid = getpid();

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

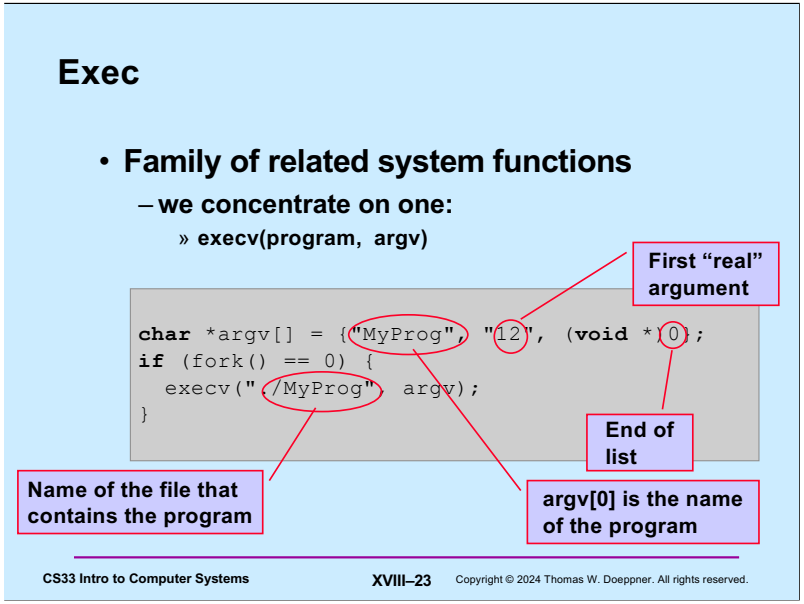
```
parent prints:
  27355, 27342, 27342

child prints:
  0, 27342, 27355
```


The **getpid** function returns the caller's process ID.

The parent process executes the second **printf**; the child process executes the first **printf**.

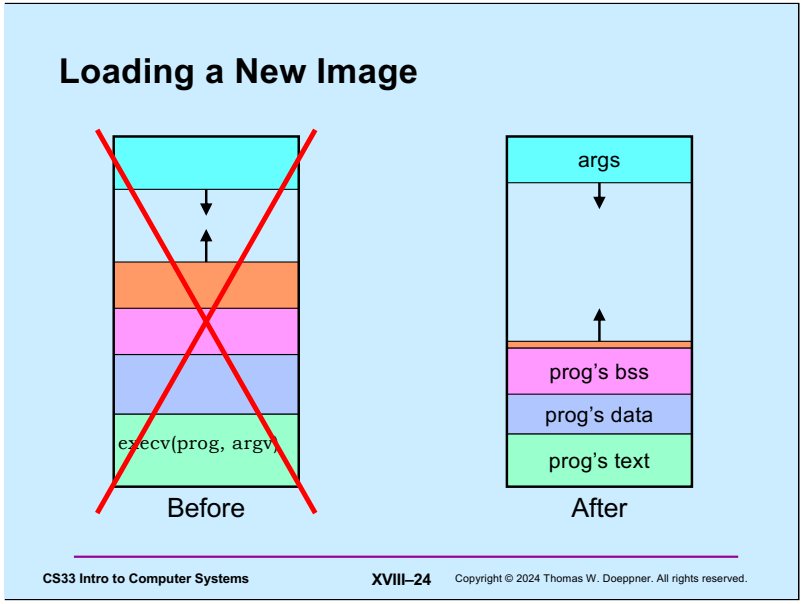
Putting Programs into Processes





We will use the convention that the name of the program, as given in argv[0] is the last component of the file's pathname.

Note that a null pointer, termed a **sentinel**, is used to indicate the end of the list of arguments.



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 one of the *exec* system calls 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 an executable program image. 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 …

```
int main(int argc, char *argv[]) {
 if (argc != 2) {
    fprintf(stderr, "Usage: random count\n");
    exit(1);
  }
  int stop = atoi(argv[1]);
  for (int i = 0; i < stop; i++)
    printf("%d\n", rand());
  return 0;
}
```


The argument **argv** is what was provided to **execv**. The argument **argc** is the number of elements of **argv** (i.e., the number of arguments, including argv[0]).

Passing It Arguments

- **From the shell**

```
$ random 12
```

- **From a C program**

```
if (fork() == 0) {
  char *argv[] = {"random", "12", (void *)0};
  execv("./random", argv);
}
```


Quiz 2

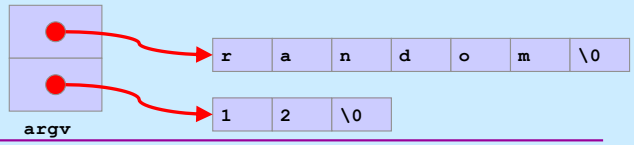
```
if (fork() == 0) {
  char *argv[] = {"random", "12", (void *)0};
  execv("./random", argv);
  printf("random done\n");
}
```

The *printf* statement will be executed

a) always
b) only if execv fails
c) only if execv succeeds

Receiving Arguments

```
int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "Usage: random count\n");
    exit(1);
  }
  int stop = atoi(argv[1]);
  for (int i = 0; i < stop; i++)
    printf("%d\n", rand());

  return 0;
}
```



Note that argv[0] is the name by which the program is invoked. argv[1] is the first "real" argument. In this program, argv[2] will contain the NULL pointer (0). arrgc is two, indicating two arguments (argv[0] and argv[1]).

Not So Fast …

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

```
if (fork() == 0) {
  char *argv = {"random", "12", (void *)0};
  execv("./random", argv);
}
/* what does the shell do here??? */
```

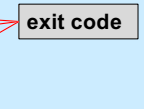

Wait

```
#include <unistd.h>
#include <sys/wait.h>
…
  pid_t pid;
  int status;
  …
  if ((pid = fork()) == 0) {
    char *argv[] = {"random", "12", (void *)0};
    execv("./random", argv);
  }
  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 */
```



The exit code is used to indicate problems that might have occurred while running a program. The convention is that an exit code of 0 means success; other values indicate some sort of error. Note that if the main function returns, it returns to code that calls exit; thus, returning from main is equivalent to calling **exit**. The argument passed to **exit** in this case is the value returned by main.

Shell: To Wait or Not To Wait ...

```
$ who
    if ((pid = fork()) == 0) {
       char *argv[] = {"who", 0};
       execv("who", argv);
    }
    waitpid(pid, &status, 0);
    …
$ who &
    if ((pid = fork()) == 0) {
       char *argv[] = {"who", 0};
       execv("who", argv);
    }
    …
```


System Calls

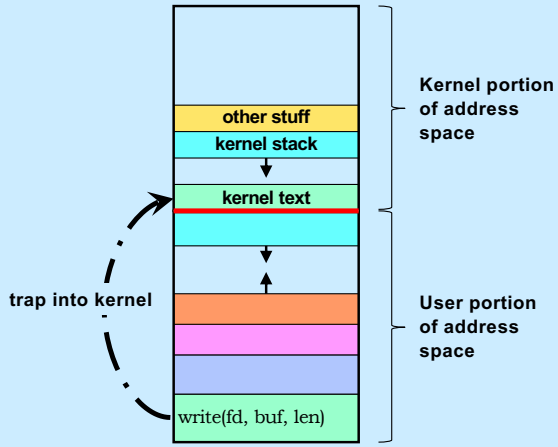
- **Sole direct interface between user and kernel**
- **Implemented as library functions that execute *trap* instructions to enter kernel**
- **Errors indicated by returns of –1; error code is in global variable *errno***

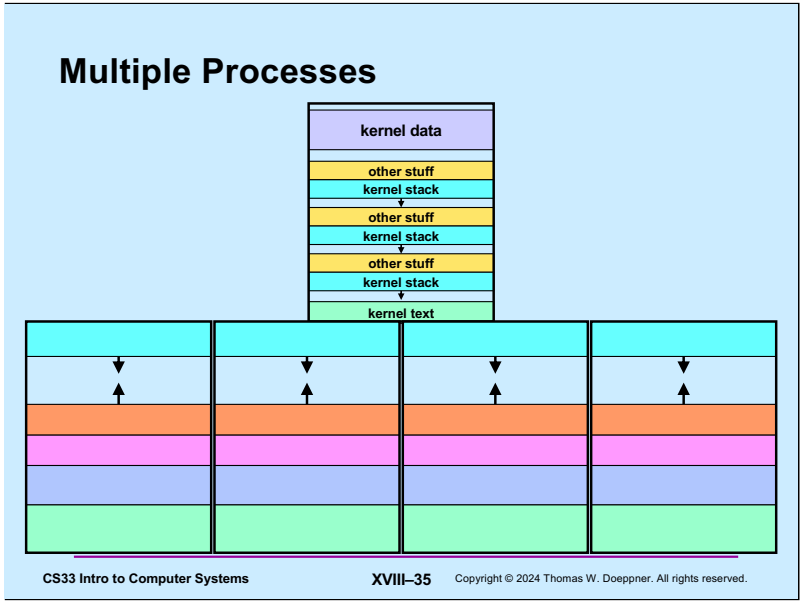
```
if (write(fd, buffer, bufsize) == -1) {
   // error!
   printf("error %d\n", errno);
   // see perror
}
```


System calls, such as **fork**, **execv**, **read**, **write**, etc., are the only means for application programs to communicate directly with the OS kernel: they form an API (application program interface) to the kernel. When a program calls such a function, it is actually placing a call to a function in a system library. The body of this function contains a hardware-specific trap instruction that transfers control and some parameters to the kernel. On return to the library function, 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 function: If there was an error, the function returns -1 and a positive-integer code identifying the error 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** function.

The "hardware-specific trap instruction" is (or used to be) the "int" (interrupt) instruction on the x86. However, this instruction is now considered too expensive for such performance-critical operations as system calls. A new facility, known as "syscall/sysret" 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





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.

CS 33

Shells and Files

Shells

- Command and scripting languages for Unix
- First shell: Thompson shell
  - sh, developed by Ken Thompson
  - released in 1971
- Bourne shell
  - also sh, developed by Steve Bourne
  - released in 1977
- C shell
  - csh, developed by Bill Joy
  - released in 1978
  - tcsh, improved version by Ken Greer

This information is from Wikipedia.

More Shells

- Bourne-Again Shell
  - bash, developed by Brian Fox
  - released in 1989
  - found to have a serious security-related bug in 2014
    - » shellshock
- Almquist Shell
  - ash, developed by Kenneth Almquist
  - released in 1989
  - similar to bash
  - dash (debian ash) used for scripts in Debian Linux
    - » faster than bash
    - » less susceptible to shellshock vulnerability

This information is also from Wikipedia.

CS Department computers run Debian Linux (and thus weren't affected by shellshock).

Our examples use bash syntax.

Roadmap

- We explore the file abstraction
  - what are files
  - how do you use them
  - how does the OS represent them
- We explore the shell
  - how does it launch programs
  - how does it connect programs with files — shell 1
  - how does it control running programs — shell 2

The File Abstraction

- A file is a simple array of bytes
- A file is made larger by writing beyond its current end
- Files are named by paths in a naming tree
- System calls on files are synchronous
- Files are permanent

Most programs perform file I/O using library code layered on top of system calls. In this section we discuss just the kernel aspects of file I/O, looking at the abstraction and the high-level aspects of how this abstraction is implemented.

The Unix file abstraction is very simple: files are simply arrays of bytes. Some systems have special system calls to make a file larger. In Unix, you simply write where you've never written before, and the file "magically" grows to the new size (within limits). The names of files are equally straightforward — just the names labeling the path that leads to the file within the directory tree. Finally, from the programmer's point of view, all operations on files appear to be synchronous — when an I/O system call returns, as far as the process is concerned, the I/O has completed. (Things are different from the kernel's point of view.) Another important property of files is permanence: they continue to exist until explicitly deleted.

Note that there are numerous issues in implementing the Unix file abstraction that we do not cover in this course. In particular, we do not discuss what is done to lay out files on disks (both rotating and solid-state) so as to take maximum advantage of their architectures. Nor do we discuss the issues that arise in coping with failures and crashes. What we concentrate on here are those aspects of the file abstraction that are immediately relevant to application programs.

Naming

- (almost) everything has a path name
  - files
  - directories
  - devices (known as *special files*)
    - keyboards
    - displays
    - disks
    - etc.

The notion that almost everything in Unix has a path name was a startlingly new concept when Unix was first developed; one that has proved to be important. We discuss this in more detail in the next lecture.

I/O System Calls

- `int file_descriptor = open(pathname, mode [, permissions])`
- `int close(file_descriptor)`
- `ssize_t count = read(file_descriptor, buffer_address, buffer_size)`
- `ssize_t count = write(file_descriptor, buffer_address, buffer_size)`
- `off_t position = lseek(file_descriptor, offset, whence)`

Given the name of a file, one uses **open** to get a **file descriptor** that will refer to that file when performing operations on it. One calls close to tell the system one is no longer using that file descriptor. The **read** and **write** system calls perform the indicated operation on the file, using a buffer described by their second two arguments. By default, **read** and **write** operations go through a file from beginning to end sequentially. The **lseek** system call is used to specify where in a file the next read or write will take place.

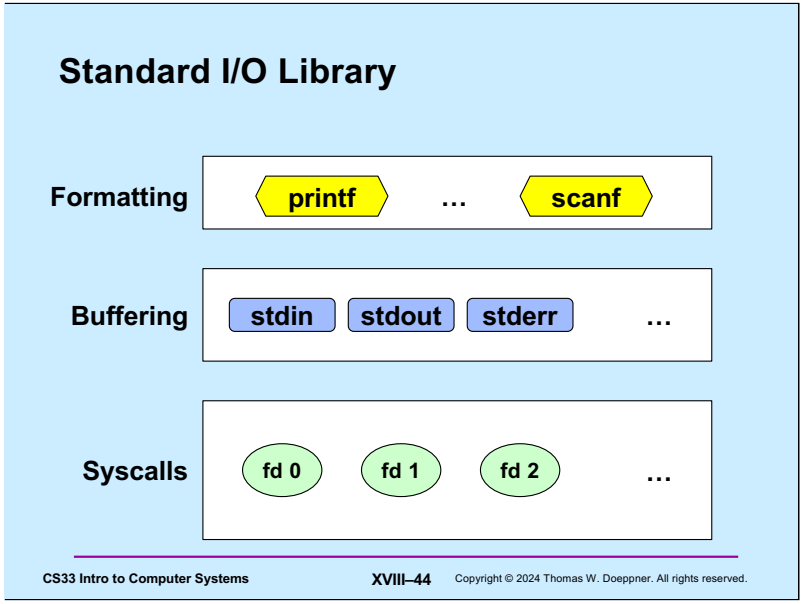
ssize_t ("signed size") is a typedef for **long** and represents the number of bytes that were transferred. It's signed so as to allow -1 as a return value, which indicates an error. **off_t** is also a typedef for **long** and represents an offset from some position in the file (the starting position is given by the **whence** argument to **lseek**).

Standard File Descriptors

```
int main( ) {
  char buf[BUFSIZE];
  int n;
  const char *note = "Write failed\n";

  while ((n = read(0, buf, sizeof(buf))) > 0)
    if (write(1, buf, n) != n) {
          write(2, note, strlen(note));
          exit(1);
    }
  return(0);
}
```


The file descriptors 0, 1, and 2 are set up before a process starts. File descriptor 0 refers to input (the keyboard, by default). Descriptors 1 and 2 are for output: normal output goes to file descriptor 1, error messages go to file descriptor 2. By default, this output goes to the current window.

We'll soon see a way to print more informative error messages than the one given here.

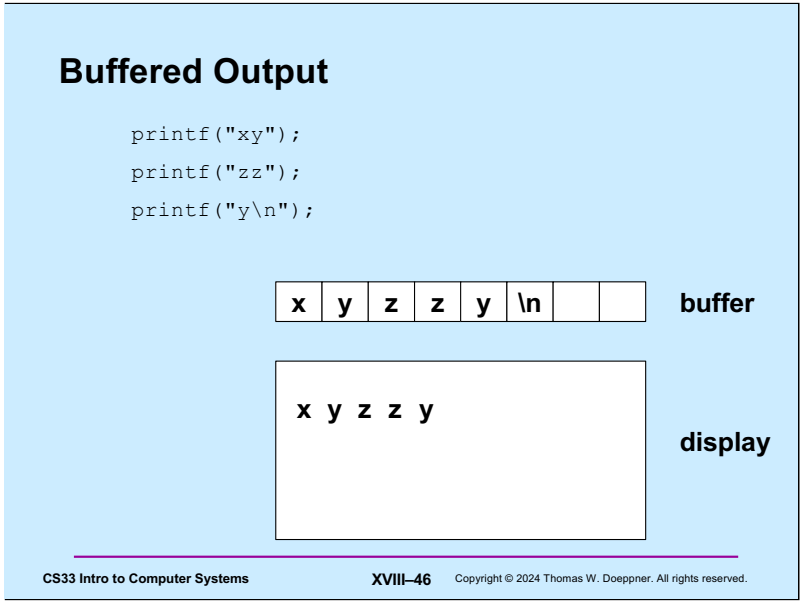


C programs often do I/O via the standard I/O library (known as **stdio**), which provides both buffering and formatting.

Standard I/O

```
FILE *stdin;        // declared in stdio.h
FILE *stdout;       // declared in stdio.h
FILE *stderr;       // declared in stdio.h

scanf("%d", &in);   // read via f.d. 0
printf("%d\n", in); // write via f.d. 1
fprintf(stderr, "there was an error\n");
     // write via f.d. 2
```

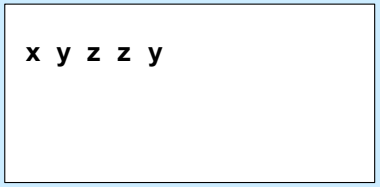

The **streams** stdin, stdout, and stderr are automatically set up to refer to data from/to file descriptors 0, 1, and 2, respectively.



The **stdout** stream is buffered. This means that characters written to **stdout** are copied into a buffer. Only when either a newline is output or the capacity of the buffer is reached are the characters actually written to the display (via a call to **write**). The reason for doing things this way is to reduce the number of (relatively expensive) calls to **write**.

Unbuffered Output

```
fprintf(stderr, "xy");
fprintf(stderr, "zz");
fprintf(stderr, "y\n");
```



display

The **stderr** stream is not buffered. Thus characters output to it are immediately written to the display.