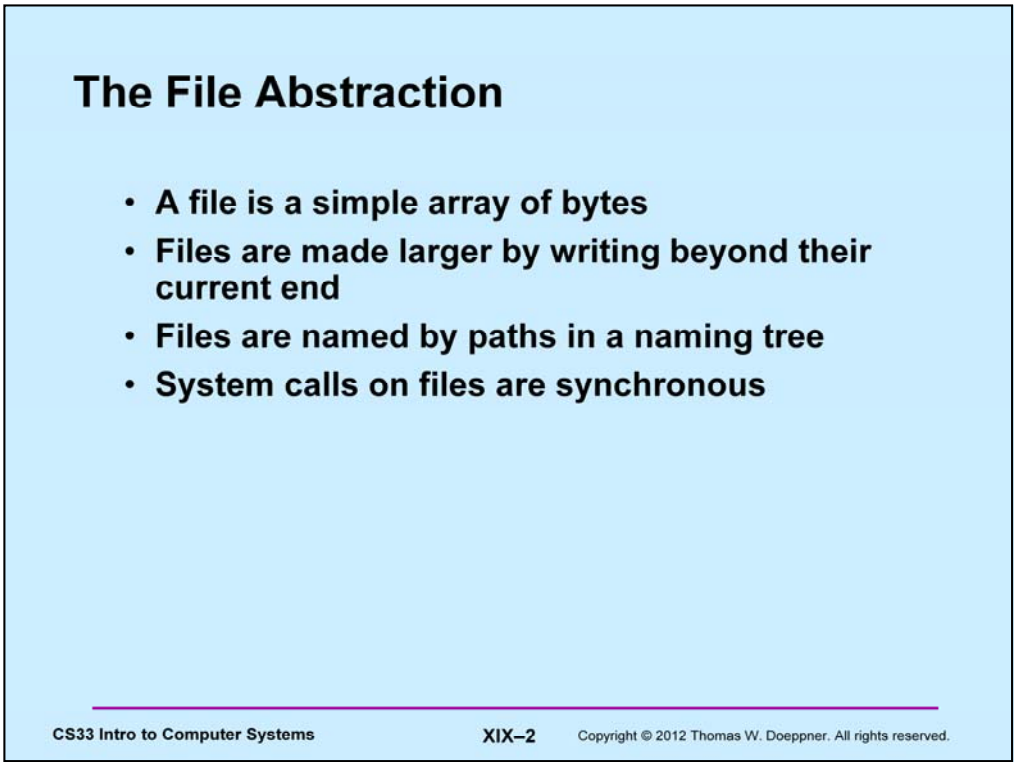
CS 33

Files and Signals



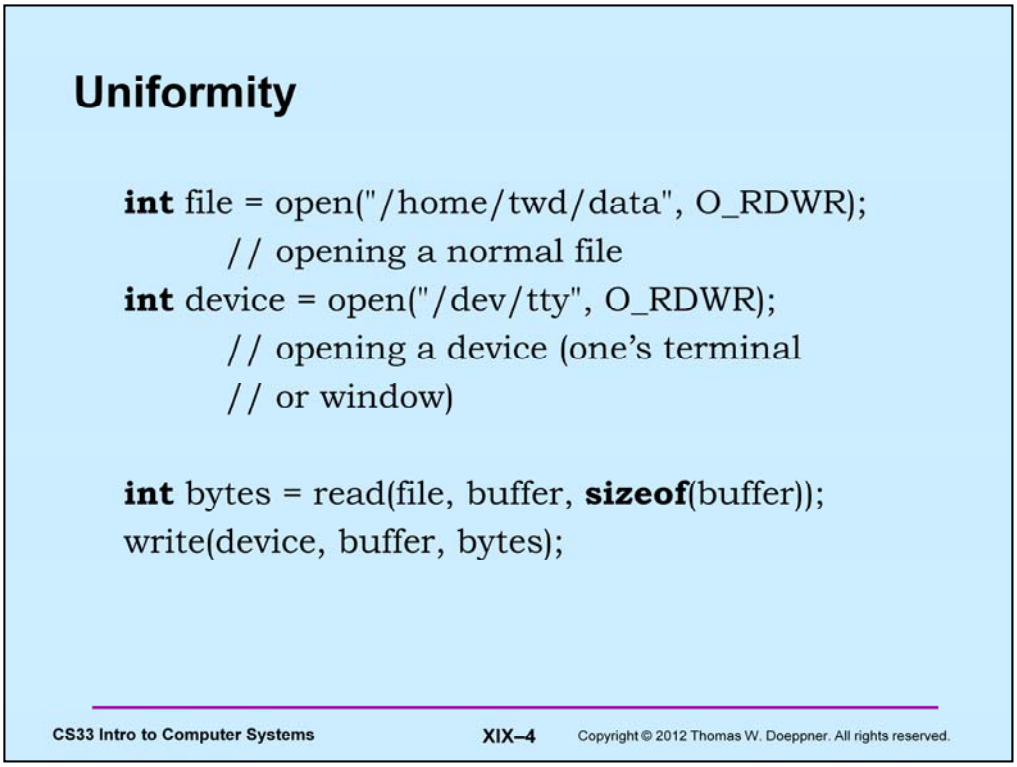
Most programs perform file I/O using library code layered on top of kernel code. In this section we discuss just the kernel aspects of file I/O, looking at the abstraction and the high-level aspects of how this abstraction is implemented.

The Unix file abstraction is very simple: files are simply arrays of bytes. Many 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, as discussed later.)

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 proven to be important.



This notion that everything has a path name facilitates a uniformity of interface. Reading and writing a normal file involves a different set of internal operations than reading and writing a device, but they are named in the same style and the I/O system calls treat them in the same way. What we have is a form of polymorphism (though the term didn't really exist when the original Unix developers came up with this way of doing things).

Note that the *open* system call returns an integer called a *file descriptor,* used in subsequent system calls to refer to the file.

Working Directory

- Maintained in kernel for each process
  - paths not starting from "/" start with the working directory
  - changed by use of the *chdir* system call
  - displayed (via shell) using "pwd"
    - » how is this done?

The *working directory* is maintained (as the inode number (explained subsequently) of the directory) in the kernel for each process. Whenever a process attempts to follow a path that doesn't start with "/", it starts at its working directory (rather than at "/").

Standard File Descriptors

```
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) {
          (void)write(2, note, strlen(note));
          exit(EXIT_FAILURE);
    }
  return(EXIT_SUCCESS);
}
```


The file descriptors 0, 1, and 2 are opened to access your window when you log in, and are preserved across forks, unless redirected.

Back to Primes …

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
   …
   for (i=1; i<nprimes; i++) {
      …
   }
   if (write(1, prime, nprimes*sizeof(int)) == -1) {
      perror("primes output");
      exit(1);
   }
   return(0);
}
```


Human-Readable Output

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
   …
   for (i=1; i<nprimes; i++) {
      …
   }
   for (i=0; i<nprimes; i++) {
      printf("%d\n", prime[i]);
   }
   return(0);
}
```

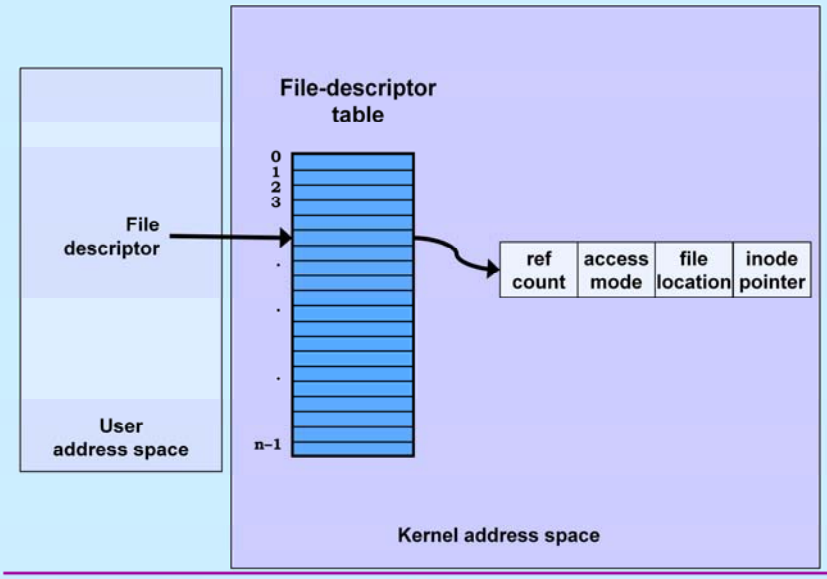
Running It

```
if (fork() == 0) {
   /* set up file descriptor 1 in the child process */
   close(1);
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      perror("/home/twd/Output");
      exit(1);
   }
   execl("/home/twd/bin/primes", "primes", "300", (void *)0);
   exit(1);
}

/* parent continues here */

while(pid != wait(0))      /* ignore the return code */
   ;
```


Here we arrange so that file descriptor 1 (standard output) refers to /home/twd/Output.

The *wait* system call is similar to *waitpid*, except that it waits for any child process to terminate, not just some particular one. Its argument is the address of where return status is to be stored. In this case, by specifying zero, we're saying that we're not interested — status info should not be stored.

File-Descriptor Table



Allocation of File Descriptors

- **Whenever a process requests a new file descriptor, the lowest numbered file descriptor not already associated with an open file is selected; thus**

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

close(0);
fd = open("file", O_RDONLY);
```

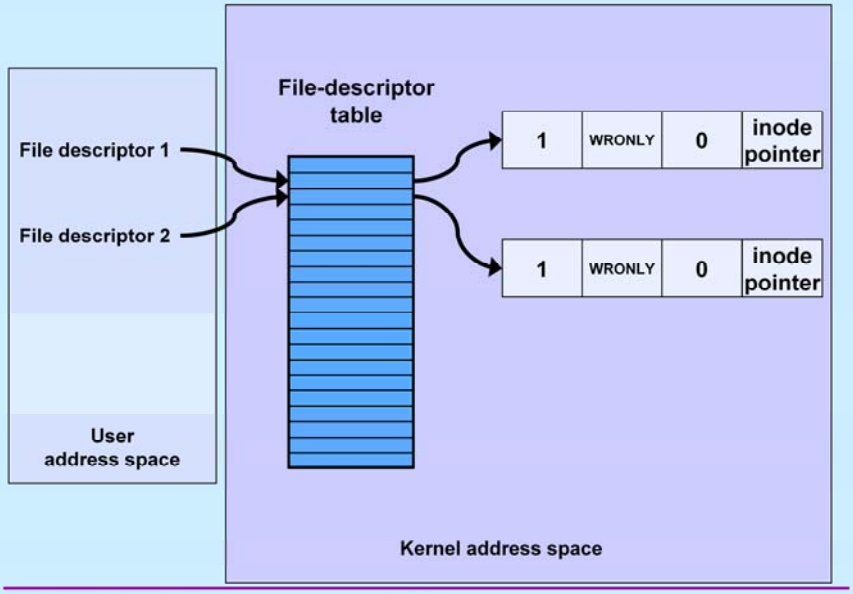
  - **will always associate *file* with file descriptor 0 (assuming that the *open* succeeds)**

One can depend on always getting the lowest available file descriptor.

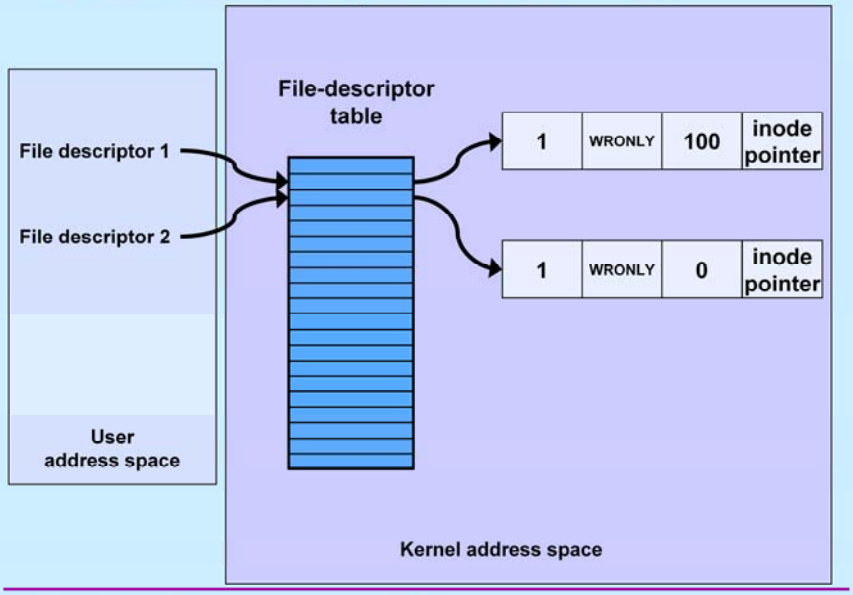
Redirecting Output … Twice

```
if (fork() == 0) {
   /* set up file descriptors 1 and 2 in the child process */
   close(1);
   close(2);
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      exit(1);
   }
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      exit(1);
   }
   execl("/home/twd/bin/program", "program", (void *)0);
   exit(1);
}

/* parent continues here */
```

Redirected Output



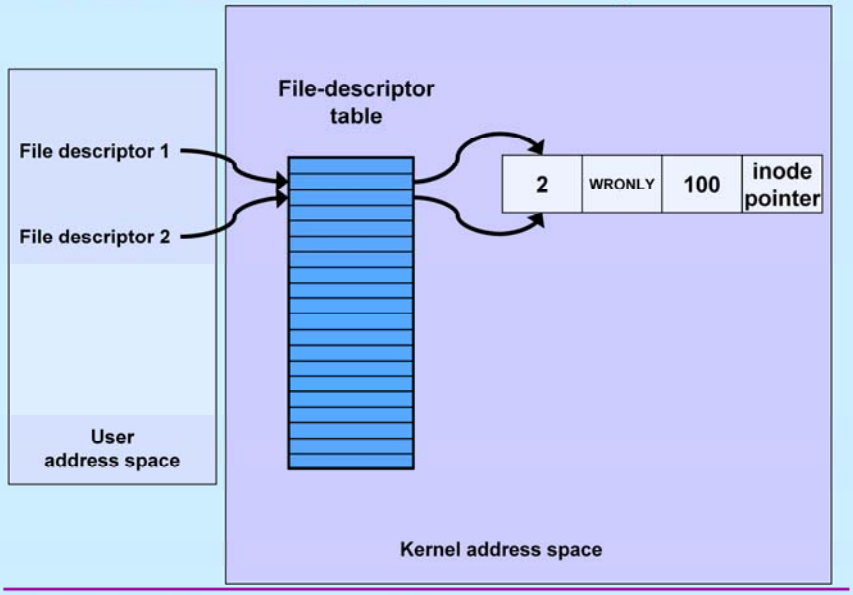
Redirected Output After Write



Sharing Context Information

```
if (fork() == 0) {
   /* set up file descriptors 1 and 2 in the child process */
   close(1);
   close(2);
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      exit(1);
   }
   dup(1); /* set up file descriptor 2 as a duplicate of 1 */
   execl("/home/twd/bin/program", "program", (void *)0);
   exit(1);
}
/* parent continues here */
```

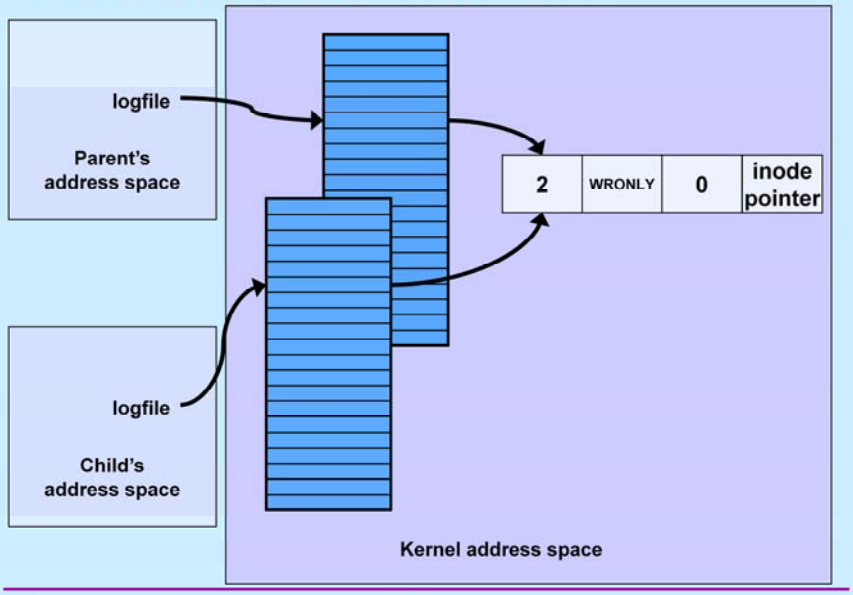
Redirected Output After Dup

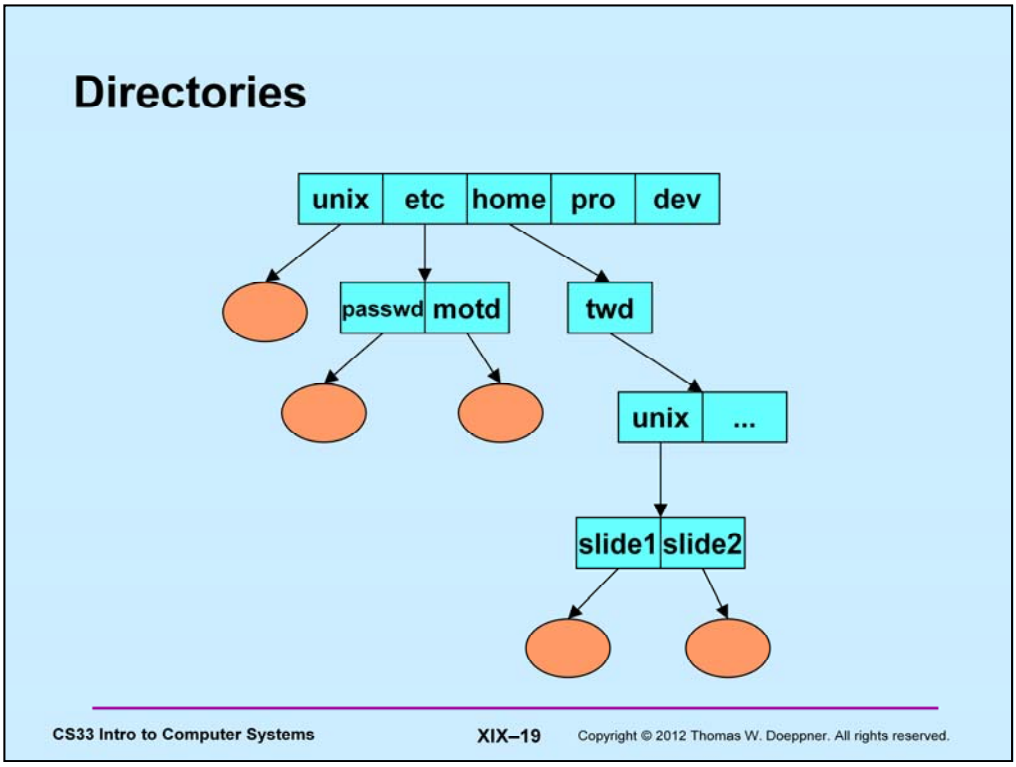


Fork and File Descriptors

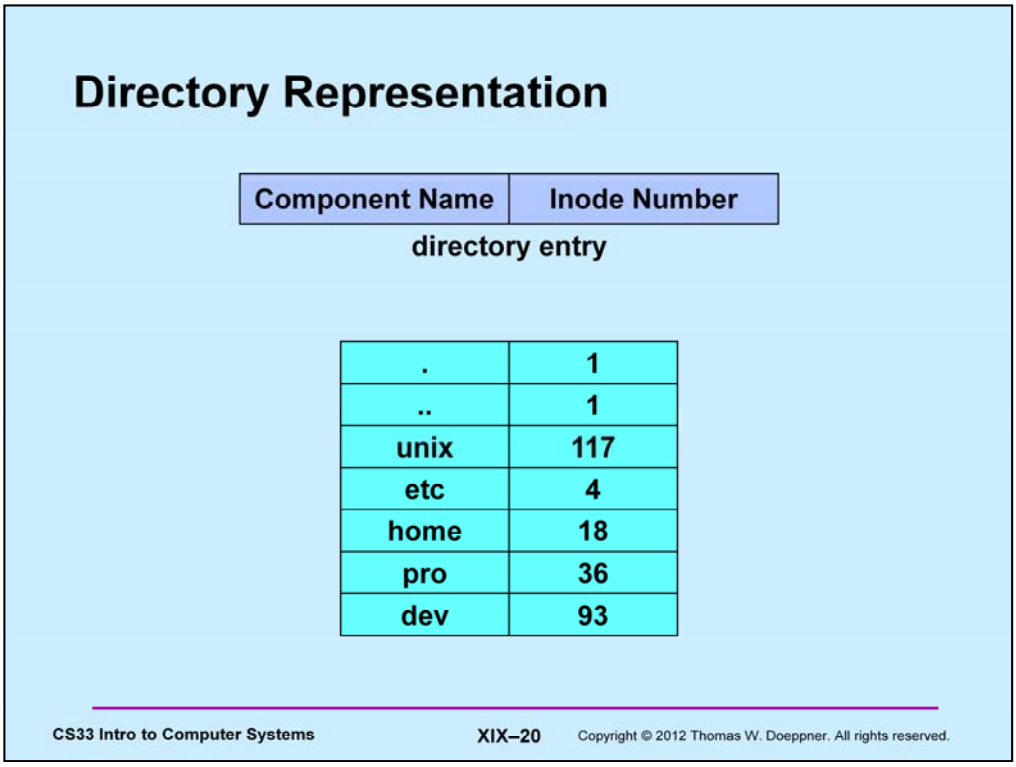
```
int logfile = open("log", O_WRONLY);
if (fork() == 0) {
   /* child process computes something, then does: */
   write(logfile, LogEntry, strlen(LogEntry));
   …
   exit(0);
}

/* parent process computes something, then does: */

write(logfile, LogEntry, strlen(LogEntry));
…
```

File Descriptors After Fork

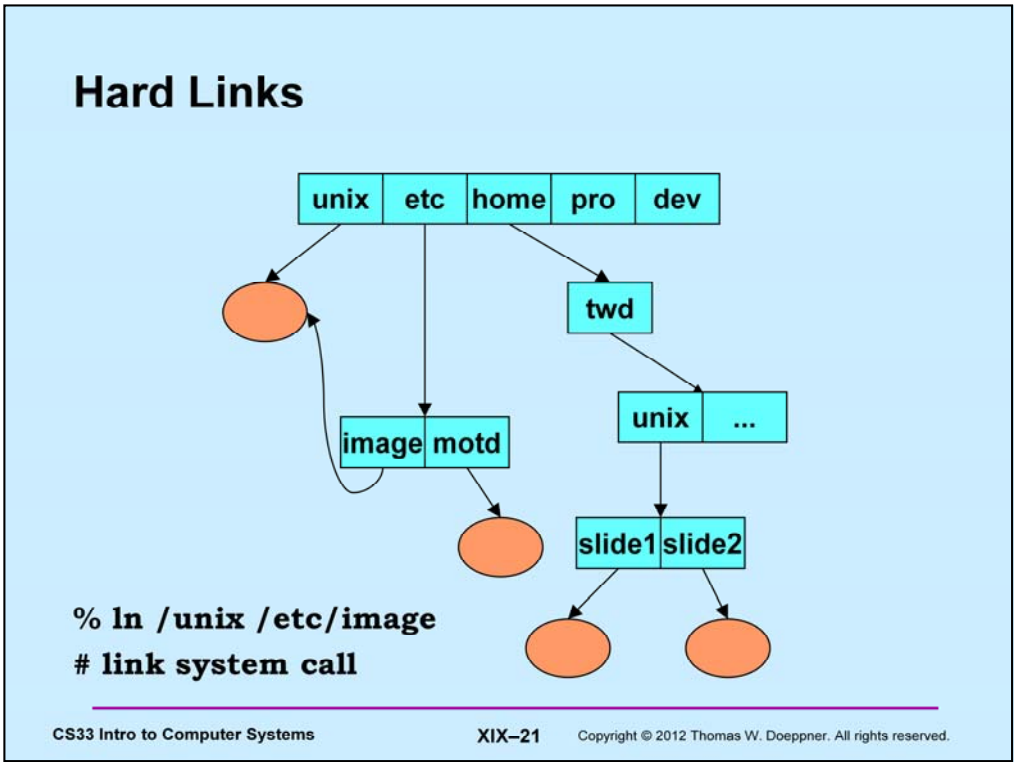




Here is a portion of a Unix directory tree. The ovals represent files, the rectangles represent directories (which are really just special cases of files).

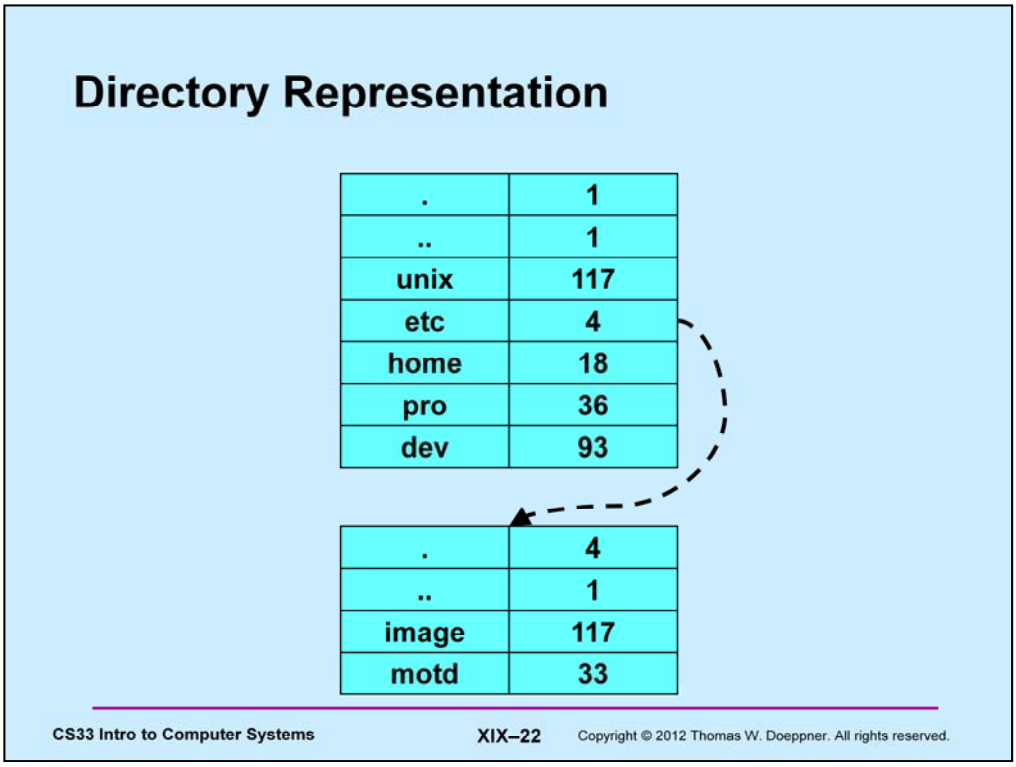


A directory consists of an array of pairs of *component name* and *inode number*, where the latter identifies the target file's *inode* to the operating system (an inode is data structure maintained by the operating system that represents a file). Note that every directory contains two special entries, "." and "..". The former refers to the directory itself, the latter to the directory's parent (in the case of the slide, the directory is the root directory and has no parent, thus its ".." entry is a special case that refers to the directory itself).



Here are two directory entries referring to the same file. This is done, via the shell, through the *ln* command which creates a (hard) link to its first argument, giving it the name specified by its second argument.

The shell's "ln" command is implemented using the link system call.

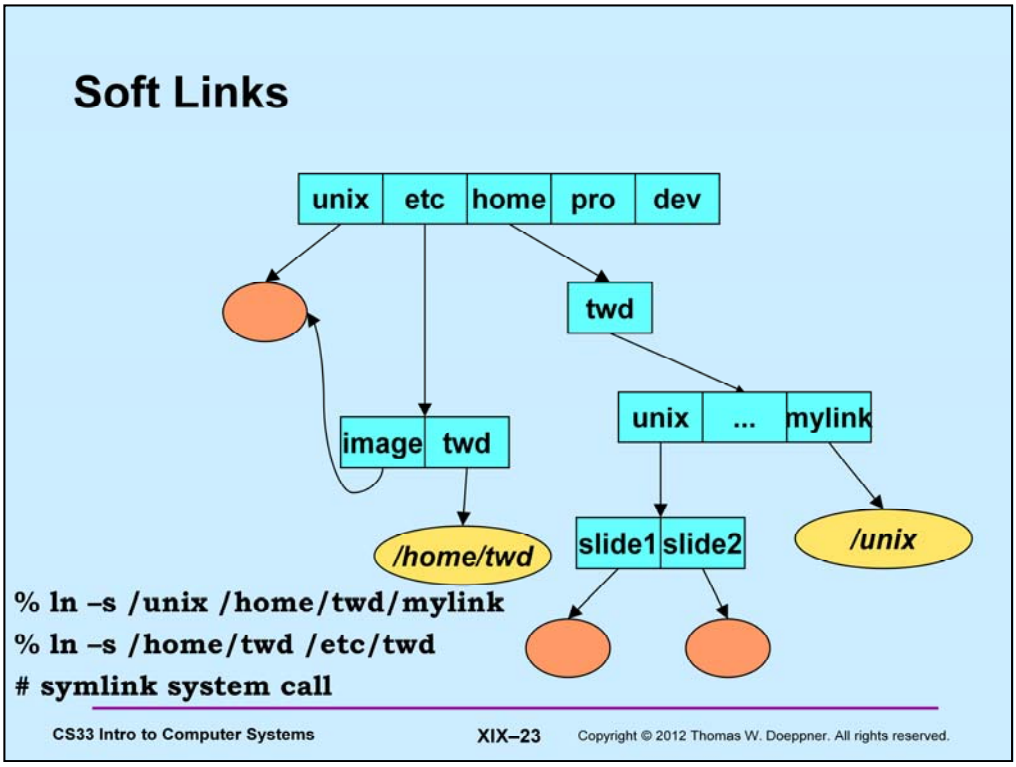


Here are the (abbreviated) contents of both the *root* (*/*) and */etc* directories, showing how */unix* and */etc/image* are the same file. Note that if the directory entry */unix* is deleted (via the shell's "rm" command), the file (represented by inode 117) continues to exist, since there is still a directory entry referring to it. However if */etc/image* is also deleted, then the file has no more links and is removed. To implement this, the file's inode contains a link count, indicating the total number of directory entries that refer to it. A file is actually deleted only when its inode's link count reaches zero.

Note: suppose a file is open, i.e. is being used by some process, when its link count becomes zero. Rather than delete the file while the process is using it, the file will continue to exist until no process has it open. Thus the inode also contains a reference count indicating how many times it is open: in particular, how many system file table entries point to it. A file is deleted when and only when both the link count and this reference count become zero.

The shell's "rm" command is implemented using the *unlink* system call.

Note that */etc/..* refers to the root directory.



Differing from a hard link, a soft link (or symbolic link) is a special kind of file containing the name of another file. When the kernel processes such a file, rather than simply retrieving its contents, it makes use of the contents by replacing the portion of the directory path that it has already followed with the contents of the soft-link file and then following the resulting path. Thus referencing */home/twd/mylink* results in the same file as referencing */unix*. Referencing */etc/twd/unix/slide1* results in the same file as referencing */home/twd/unix/slide1*.

The shell's "ln" command with the "-s" flag is implemented using the *symlink* system call.

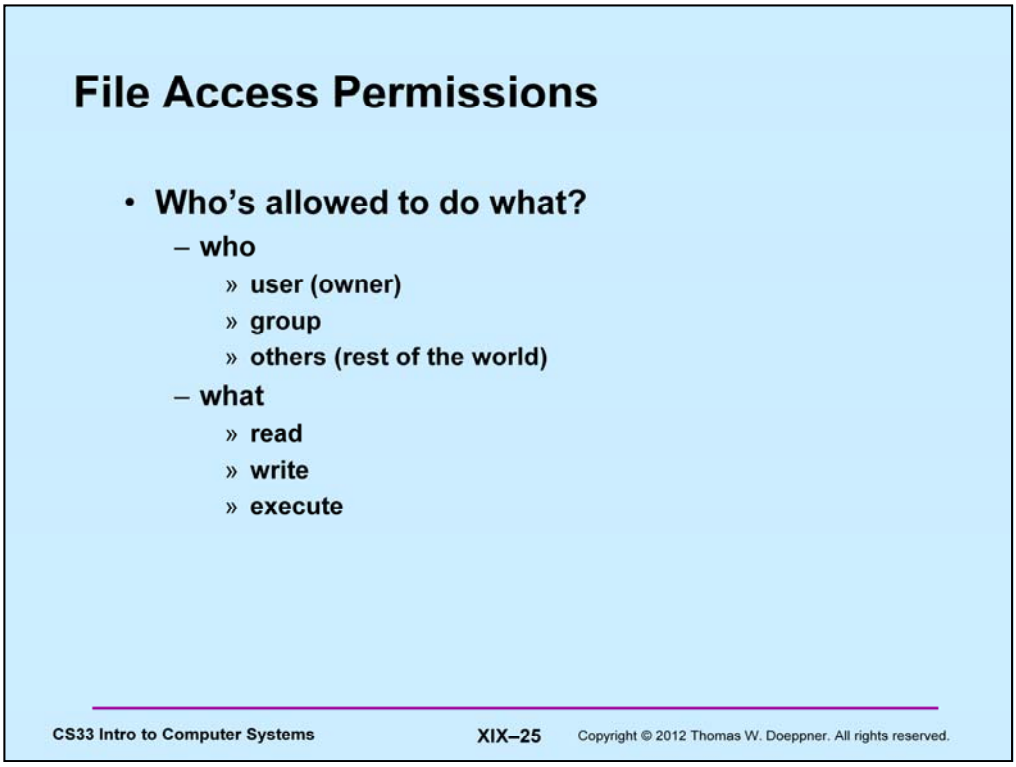
Open

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int options [, mode_t mode])
```

– options

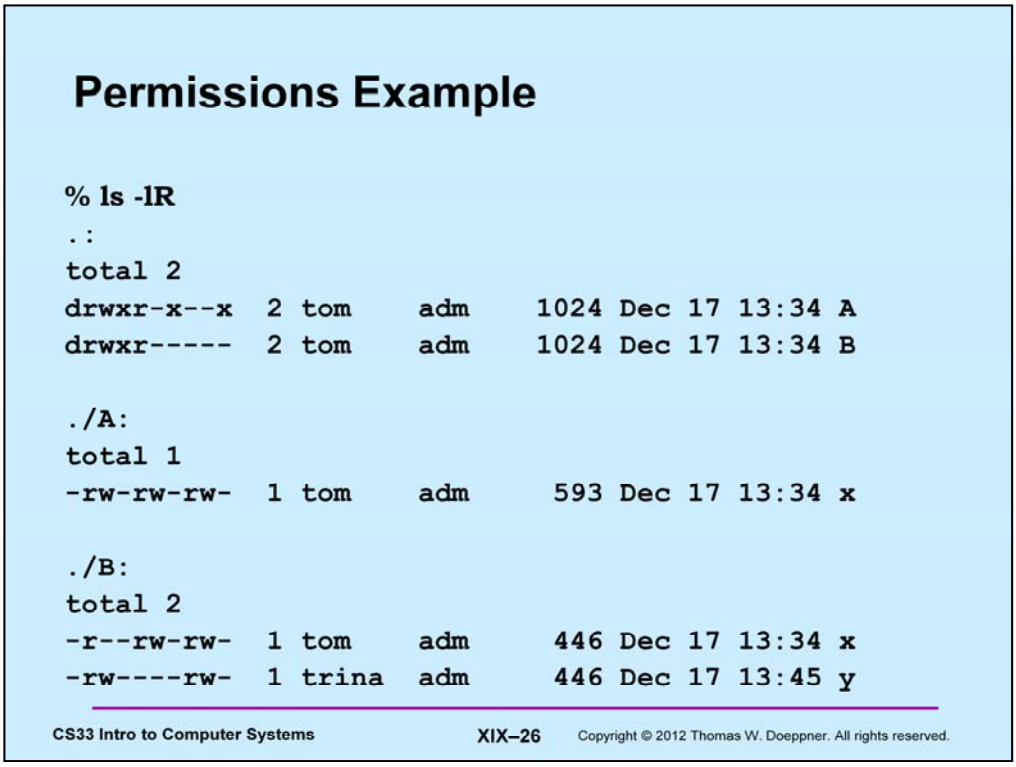
» O_RDONLY	open for reading only
» O_WRONLY	open for writing only
» O_RDWR	open for reading and writing
» O_APPEND	set the file offset to *end of file* prior to each *write*
» O_CREAT	if the file does not exist, then create it, setting its mode to *mode* adjusted by *umask*
» O_EXCL	if O_EXCL and O_CREAT are set, then *open* fails if the file exists
» O_TRUNC	delete any previous contents of the file
» O_NONBLOCK	don't wait if I/O can't be done immediately

Here's a partial list of the options available as the second argument to open. (Further options are often available, but they depend on the version of Unix.) Note that the first three options are mutually exclusive: one, and only one, must be supplied. We discuss the third argument to *open*, mode, shortly.



Each file has associated with it a set of access permissions indicating, for each of three classes of principals, what sorts of operations on the file are allowed. The three classes are the owner of the file, known as *user*, the group owner of the file, known simply as *group*, and everyone else, known as *others*. The operations are grouped into the classes *read*, *write*, and *execute*, with their obvious meanings. The access permissions apply to directories as well as to ordinary files, though the meaning of execute for directories is not quite so obvious: one must have execute permission for a directory file in order to follow a path through it.

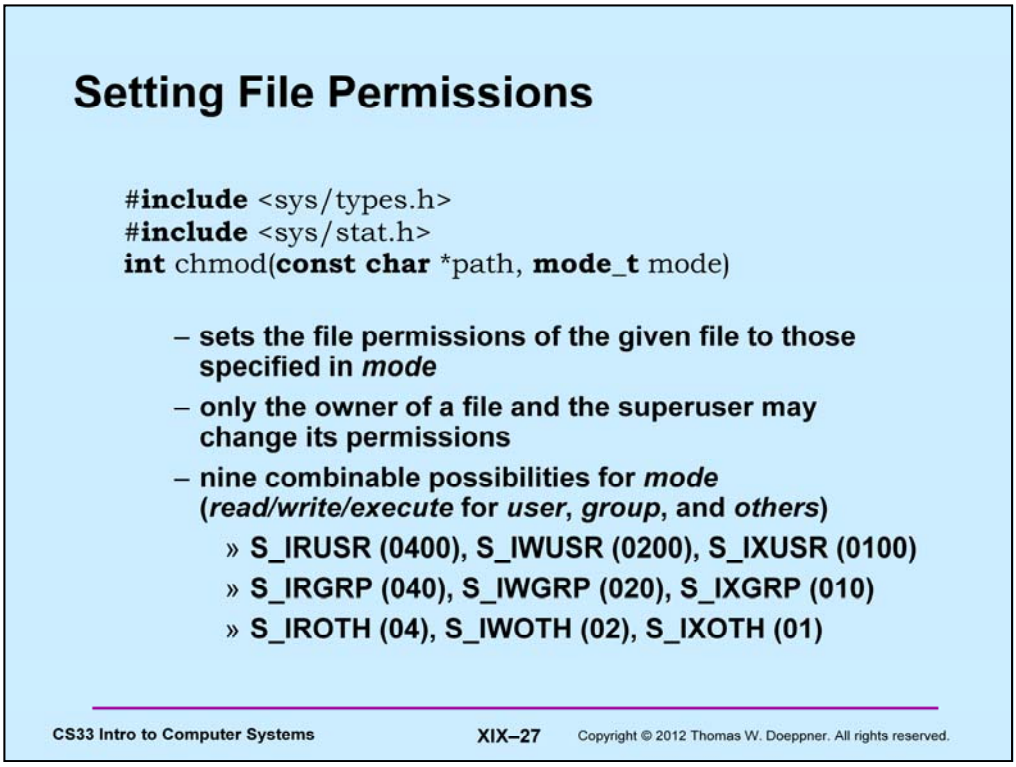
The system, when checking permissions, first determines the smallest class of principals the requester belongs to: user (smallest), group, or others (largest). It then, within the chosen class, checks for appropriate permissions.



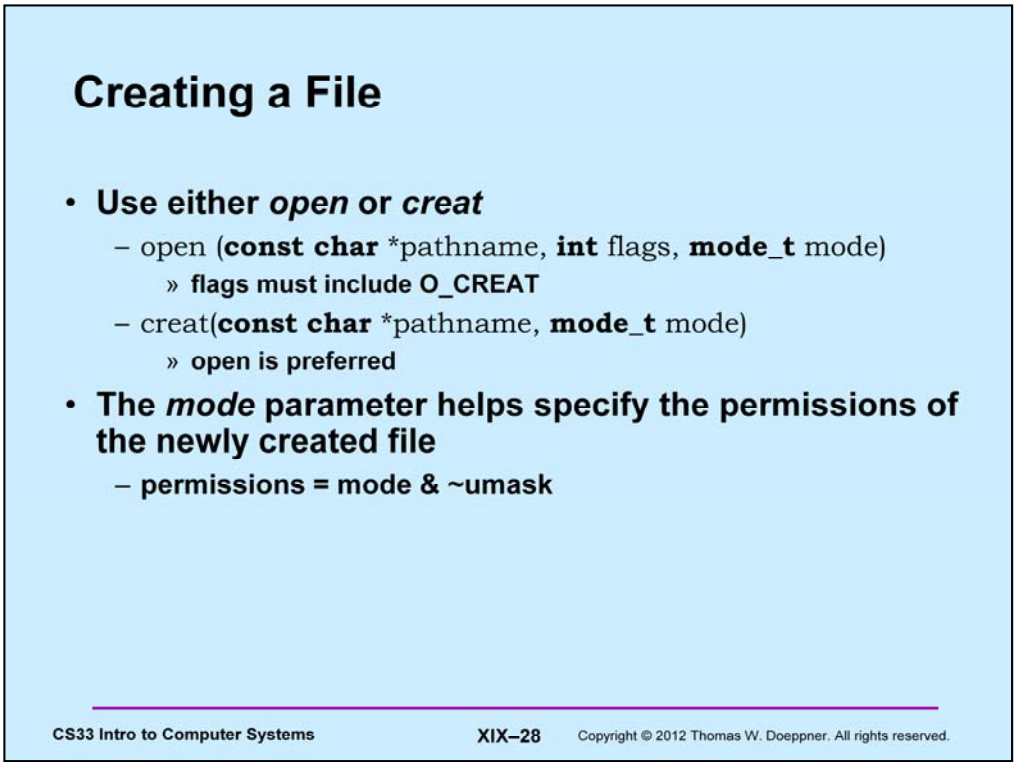
In the current directory are two subdirectories, *A* and *B*, with access permissions as shown in the slide. Note that the permissions are given as a string of characters: the first character indicates whether or not the file is a directory, the next three characters are the permissions for the owner of the file, the next three are the permissions for the members of the file's group's members, and the last three are the permissions for the rest of the world.

Quiz: the users *tom* and *trina* are members of the *adm* group; *andy* is not.

- May *andy* list the contents of directory *A*?
- May *andy* read *A/x*?
- May *trina* list the contents of directory *B*?
- May *trina* modify *B/y*?
- May *tom* modify *B/x*?
- May *tom* read *B/y*?

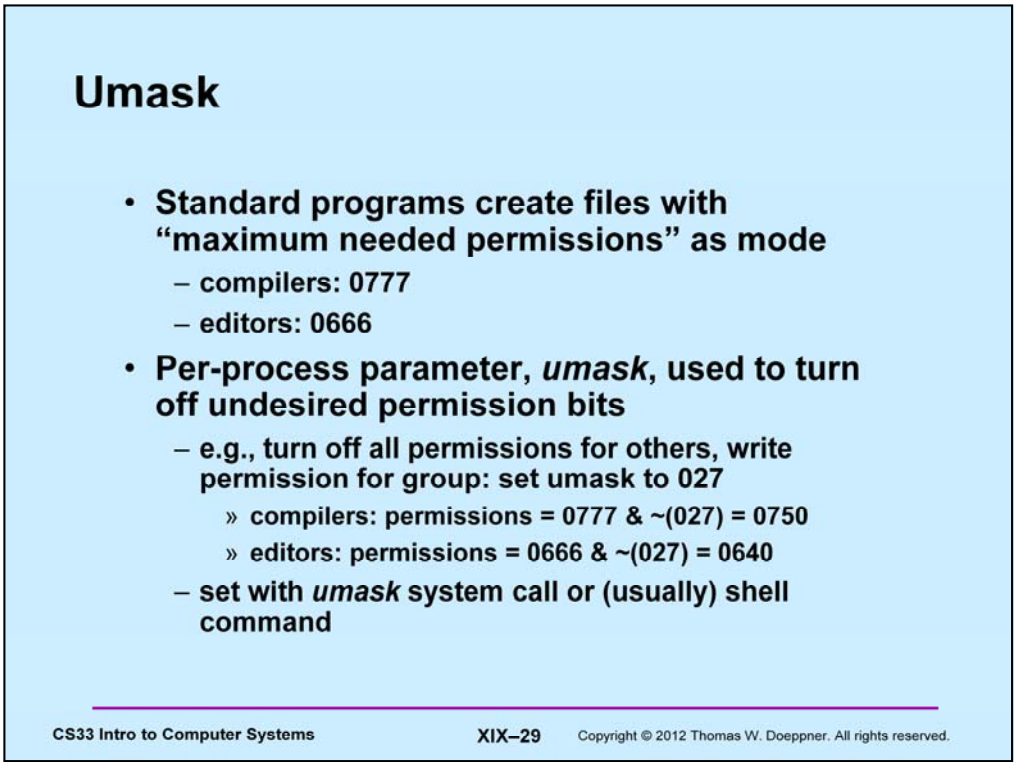


The *chmod* system call (and the similar *chmod* shell command) are used to change the permissions of a file. Note that the symbolic names for the permissions are rather cumbersome; what is often done is to use their numerical equivalents instead. Thus the combination of read/write/execute permission for the user (0700), read/execute permission for the group (050), and execute-only permission for others (01) can be specified simply as 0751.



Originally in Unix one created a file only by using the *creat* system call. A separate O_CREAT flag was later given to *open* so that it, too, can be used to create files. The *creat* system call fails if the file already exists. For *open*, what happens if the file already exists depends upon the use of the flags O_EXCL and O_TRUNC. If O_EXCL is included with the flags (e.g., *open("newfile", O_CREAT|O_EXCL, 0777)),* then, as with *creat*, the call fails if the file exists. Otherwise, the call succeeds and the (existing) file is opened. If O_TRUNC is included in the flags, then, if the file exists, its previous contents are eliminated and the file (whose size is now zero) is opened.

When a file is created by either *open* or *creat*, the file's initial access permissions are the bitwise AND of the mode parameter and the complement of the process's umask (explained in the next slide).



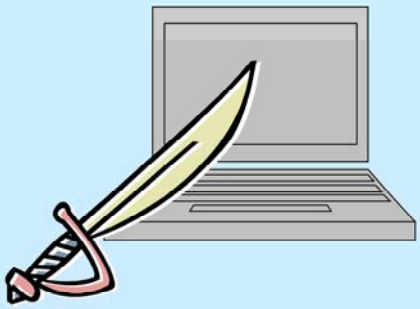
The *umask* (often called the "creation mask") allows programs to have wired into them a standard set of maximum needed permissions as their file-creation modes. Users then have, as part of their environment (via a per-process parameter that is inherited by child processes from their parents), a limit on the permissions given to each of the classes of security principals. This limit (the *umask*) looks like the 9-bit permissions vector associated with each file, but each one-bit indicates that the corresponding permission is not to be granted. Thus, if *umask* is set to 022, then, whenever a file is created, regardless of the settings of the mode bits in the *open* or *creat* call, writepermission for group and others is not to be included with the file's access permissions.

You can determine the current setting of *umask* by executing the *umask* shell command without any arguments.

Whoops …

```
% SometimesUsefulProgram xyz
Are you sure you want to proceed? Y
Are you really sure? Y
Reformatting of your hard drive will
begin in 3 seconds.
Everything you own will be deleted.
There's little you can do about it.
Too bad …
```

One Approach …



A Gentler Approach

- Signals
  - get a process's attention
    - » send it a signal
  - process must either deal with it or be terminated
    - » in some cases, the latter is the only option

Stepping Back …

- What are we trying to do?
  - interrupt the execution of a program
    - » cleanly terminate it

      or
    - » cleanly change its course
  - not for the faint of heart
    - » it's difficult
    - » it gets complicated
    - » (not done in Windows)

Signals

```
int x, y;

x = 0;
…
y = 16/x;
```

```
for (;;)
   keep_on_trying( );
```


Signals are a kernel-supported mechanism for reporting events to user code and forcing a response to them. There are actually two sorts of such events, to which we sometimes refer as *exceptions* and *interrupts*. The former occur typically because the program has done something wrong. The response, the sending of a signal, is immediate; such signals are known as *synchronous* signals. The latter are in response to external actions, such as a timer expiring, an action at the keyboard, or the explicit sending of a signal by another process. Signals send in response to these events can seemingly occur at any moment and are referred to as *asynchronous* signals.

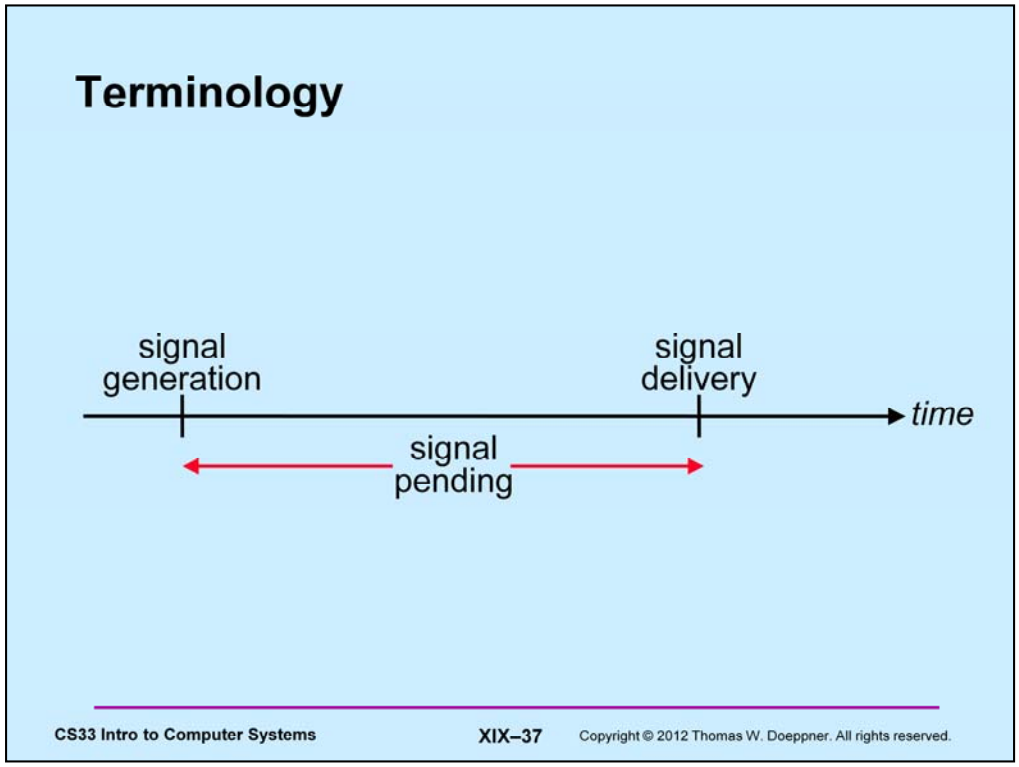
The OS to the Rescue

- Signals
  - generated (by OS) in response to
    - » exceptions (e.g., arithmetic errors, addressing problems)
    - » external events (e.g., timer expiration, certain keystrokes, actions of other processes)
  - effect on process:
    - » termination (possibly after producing a core dump)
    - » invocation of a procedure that has been set up to be a signal handler
    - » suspension of execution
    - » resumption of execution

Processes react to signals using the actions shown in the slide. The action taken depends partly on the signal and partly on arrangements made in the process beforehand.

Signal Actions

- For each signal type, a process can specify an action:
  - *abort* (with or without a core dump)
  - *ignore*
  - *hold*: temporarily ignore (delay delivery)
  - *catch*: call a signal handler function
- For each signal type, there is a default action



A signal is *generated* for (or sent to) a process when the event that causes the signal first occurs; the same event may generate signals for multiple processes. A signal is *delivered* to a process when the appropriate action for the process and signal is taken. In the period between the generation of the signal and its delivery the signal is *pending*.

Much like how hardware-generated interrupts can be masked by the processor, (software-generated) signals can be *blocked* from delivery to the process. Associated with each process is a vector indicated which signals are blocked. A signal that's been generated for a process remains pending until after it's been unblocked.

Signal Types

SIGABRT	*abort* called	term, core
SIGALRM	alarm clock	term
SIGCHLD	death of a child	ignore
SIGCONT	continue after stop	cont
SIGFPE	erroneous arithmetic operation	term, core
SIGHUP	hangup on controlling terminal	term
SIGILL	illegal instruction	term, core
SIGINT	interrupt from keyboard	term
SIGKILL	kill	forced term
SIGPIPE	write on pipe with no one to read	term
SIGQUIT	quit	term, core
SIGSEGV	invalid memory reference	term, core
SIGSTOP	stop process	forced stop
SIGTERM	software termination signal	term
SIGTSTP	stop signal from keyboard	stop
SIGTTIN	background read attempted	stop
SIGTTOU	background write attempted	stop
SIGUSR1	application-defined signal 1	stop
SIGUSR2	application-defined signal 2	stop

This slide shows the complete list of signals required by POSIX 1003.1. In addition, many Unix systems support other signals, some of which we'll mention in the course. The third column of the slide lists the default actions in response to each of the signals. *term* means the process is terminated, *core* means there is also a core dump; *ignore* means that the signal is ignored; *stop* means that the process is stopped (suspended); *cont* means that a stopped process is resumed (continued); *forced* means that the default action cannot be changed and that the signal cannot be blocked.

Sending a Signal

- `int kill(pid_t pid, int sig)`
  - send signal *sig* to process *pid*
- Also
  - *kill* shell command
  - type ctrl-c
    - » sends signal 2 (SIGINT) to current process
  - do something illegal
    - » bad address, bad arithmetic, etc.

Handling Signals

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signo,
    sighandler_t handler);

sighandler_t OldHandler;

OldHandler = signal(SIGINT, NewHandler);
```


Special Handlers

- SIG_IGN
  - ignore the signal
  - `signal(SIGINT, SIG_IGN);`
- SIG_DFL
  - use the default handler
    - » usually terminates the process
  - `signal(SIGINT, SIG_DFL);`

Example

```
int main() {
  void handler(int);

  signal(SIGINT, handler);
  while(1)
    ;
  return 1;
}
void handler(int signo) {
  printf("I received signal %d. "
     "Whoopee!!\n", signo);
}
```


Note that the C compiler implicitly concatenates two adjacent strings, as done in printf above.

Signals and Handlers

- What happens when signal is delivered to process that has a handler?
  - original Unix: current handler is called, but for subsequent occurrences, handler set to default
  - BSD (1981): new system call, sigset, introduced
    - » signal/handler association is permanent
    - » signal is blocked (masked) while handler is running
  - Sun Solaris (~1992): meanings of *signal* and *sigset* switched
  - Linux: ???

See the Linux man-page entry for signal (in particular, under “Portability” for a detailed explanation).

sigaction

```
int sigaction(int sig, const struct sigaction *new,
              struct sigaction *old);
struct sigaction {
   void (*sa_handler)(int);
   void (*sa_sigaction)(int, siginfo_t *, void *);
   sigset_t sa_mask;
   int sa_flags;
};

int main() {
   struct sigaction act; void sighandler(int);
   sigemptyset(&act.sa_mask);
   act.sa_flags = 0;
   act.sa_handler = sighandler;
   sigaction(SIGINT, &act, NULL);
   …
}
```


The *sigaction* system call is the primary means for establishing a process's response to a particular signal. Its first argument is the signal for which a response is being specified, the second argument is a pointer to a *sigaction* structure defining the response, and the third argument is a pointer to memory in which a *sigaction* structure will be stored containing the specification of what the response was prior to this call. If the third argument is null, the prior response is not returned.

The *sa_handler* member of *sigaction* is either a pointer to a user-defined handler function for the signal or one of SIG_DFL (meaning that the default action is taken) or SIG_IGN (meaning that the signal is to be ignored). The *sig_action* member is an alternative means for specifying a handler function; we discuss it starting in the next slide.

When a a user-defined signal-handler function is entered in response to a signal, the signal itself is masked until the function returns. Using the *sa_mask* member, one can specify additional signals to be masked while the handler function is running. On return from the handler function, the process's previous signal mask is restored.

The *sa_flags* member is used to specify various other things which we describe in upcoming slides.

Example

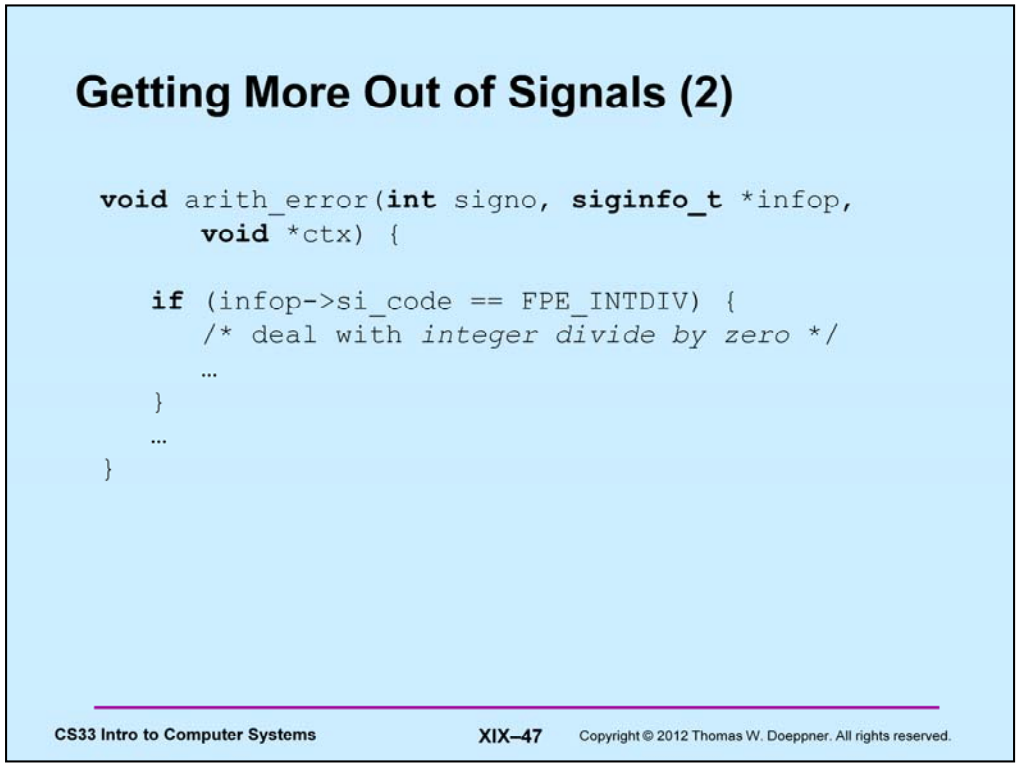
```
int main() {
  void handler(int);
  struct sigaction act;
  act.sa_handler = handler;
  act.sa_mask = 0;
  act.sa_flags = 0;
  sigaction(SIGINT, &act, 0);

  while(1)
    ;
  return 1;
}
void handler(int signo) {
  printf("I received signal %d. "
     "Whoopee!!\n", signo);
}
```


This has behavior identical to the previous example; we're using sigaction rather than signal to set up the signal handler.

Getting More Out of Signals (1)

- Getting more than the signal number
  - for example, which arithmetic problem caused a SIGFPE?
- Use sa_sigaction rather than sa_handler

```
struct sigaction act;
act.sa_sigaction = arith_error;
   /* not sa_handler! */
act.sa_mask = 0;
act.sa_flags = SA_SIGINFO;
   /* means that we're using sa_sigaction */
sigaction(SIGFPE, &act, 0);
```

The slide illustrates the signature of the handler procedure used with *siginfo*, as well as a partial example of its use. The third parameter is, on some implementations (but not on Linux), a pointer to a structure of type *ucontext_t* and contains the register context of the process at the time of interruption by the signal. We won't be discussing it further in this course, but information about its use can be found in the man page for *ucontext*.

The *siginfo* structure (of type *siginfo_t*) contains the following:

```
int           si_signo    /* signal number */
int           si_errno    /* error number */
int           si_code     /* signal code */
union sigval  si_value    /* signal value */
```

- if *si_errno* is not zero, it contains whatever error code is associated with the signal
- if *si_code* is positive, the signal was generated in response to a kernel-detected event (such as an arithmetic exception) indicated by *si_code* (there are a great number of possibilities—see *siginfo*'s man page for details)
  - *si_value* may contain other useful information, such as the problem address in the case of SIGSEGV and SIGBUS, and the child's PID and status in the case of SIGCHLD
- if *si_code* is less than or equal to zero, then the signal was generated by a user process via the *kill* system call
  - *si_value* contains the signaller's PID and UID, which can be referenced as:

```
pid_t    si_pid
uid_t    si_uid
```

Waiting for a Signal …

```
signal(SIGALRM, DoSomethingInteresting);

…

struct timeval waitperiod = {0, 1000};
      /* seconds, microseconds */
struct timeval interval = {0, 0};
struct itimerval timerval;
timerval.it_value = waitperiod;
timerval.it_interval = interval;

setitimer(ITIMER_REAL, &timerval, 0);
      /* SIGALRM sent in ~one millisecond */
pause();  /* wait for it */
```


Here we use the *setitimer* system call to arrange so that a SIGALRM signal is generated in one millisecond. (The system call takes three arguments: the first indicates how time should be measured; what's specified here is to use real time. See its man page for other possibilities. The second argument contains a *struct itimerval* that itself contains two *struct timeval*s. One (named *it_value*) indicates how much time should elapse before a SIGALRM is generated for the process. The other (named *it_interval*), if non-zero, indicates that a SIGALRM should be sent again, repeatedly, every it_interval period of time. Each process may have only one pending timer, thus when a process calls setitimer, the new value replaces the old. If the third argument to setitimer is non-zero, the old value is stored at the location it points to.)

The *pause* system call causes the process to block and not resume until *any* signal that is not ignored is delivered.

Note that there is a race condition here: it's possible that the SIGALRM might be delivered after the process calls *setitimer*, but before it calls *pause* (the system might be very busy). If this were to happen, then the process might get "stuck" within *pause*, since no other signals are forthcoming.

Doing It Safely

```
sigset_t set, oldset;
sigemptyset(&set);
sidaddset(&set, SIGALRM);
sigprocmask(SIG_BLOCK, &set, &oldset);
      /* SIGALRM now masked */
setitimer(ITIMER_REAL, &timerval, 0);
      /* SIGALRM sent in ~one millisecond */

sigsuspend(&oldset);    /* wait for it safely */
/* SIGALRM masked again */
…

sigprocmask(SIG_SET, &oldset, (sigset_t *)0);
      /* SIGALRM unmasked */
```


Here's a safer way of doing what was attempted in the previous slide. We mask the SIGALRM signal before calling *setitimer*. Then, rather than calling *pause*, we call *sigsuspend*, which sets the set of masked signals to its argument and, at the same instant, blocks the calling process. Thus if the SIGALRM is generated before our process calls *sigsuspend*, it won't be delivered right away. Since the call to *sigsuspend* reinstates the previous mask (which, presumably, did not include SIGALRM), the SIGALRM signal will be delivered and the process will return (after invoking the handler). When *sigsuspend* returns, the signal mask that was in place just before it was called (*set*) is restored. Thus we have to restore *oldset* explicitly.

As with *pause*, *sigsuspend* returns only if an unmasked signal that is not ignored is delivered.

Signal Sets

- To clear a set:

```
int sigemptyset(sigset_t *set);
```

- To add or remove a signal from the set:

```
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
```

- Example: to refer to both SIGHUP and SIGINT:

```
sigset_t set;

sigemptyset(&set);
sigaddset(&set, SIGHUP);
sigaddset(&set, SIGINT);
```


A number of signal-related operations involve sets of signals. These sets are normally represented by a bit vector of type *sigset_t*.

Masking (Blocking) Signals

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set,
      sigset_t *old);
```

- used to examine or change the signal mask of the calling process
  - » *how* is one of three commands:
    - SIG_BLOCK
      - the new signal mask is the union of the current signal mask and set
    - SIG_UNBLOCK
      - the new signal mask is the intersection of the current signal mask and the complement of set
    - SIG_SETMASK
      - the new signal mask is set

In addition to ignoring signals, you may specify that they are to be blocked (that is, held pending or masked). When a signal type is masked, signals of that type remains pending and do not interrupt the process until they are unmasked. When the process unblocks the signal, the action associated with any pending signal is performed. This technique is most useful for protecting critical code that should not be interrupted. Also, as we've already seen, when the handler for a signal is entered, subsequent occurrences of that signal are automatically masked until the handler is exited, hence the handler never has to worry about being invoked to handle another instance of the signal it's already handling.

Timed Out!

```
int TimedInput( ) {
   signal(SIGALRM, timeout);
   …
   alarm(30);     /* send SIGALRM in 30 seconds */
   GetInput();    /* possible long wait for input */
   HandleInput();
   return(0);
nogood:
   return(1);
}

void timeout( ) {
   goto nogood;   /* not legal but straightforward */
}
```

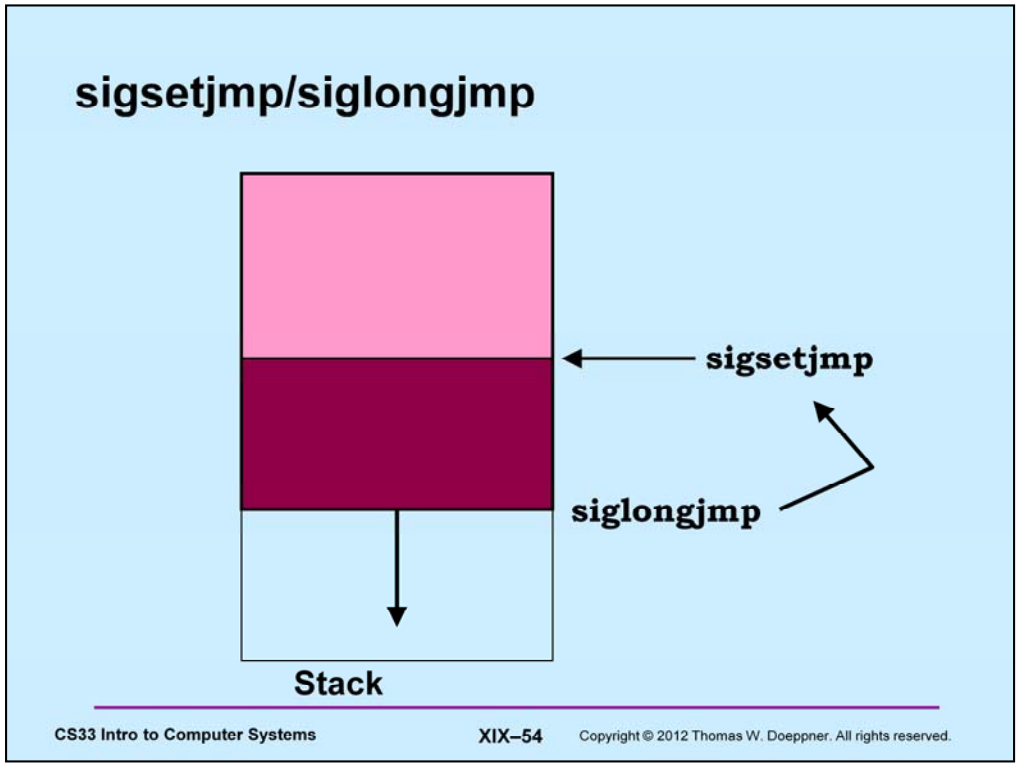

This slide sketches something that one might want to try to do: give a user a limited amount of time (in this case, 30 seconds) to provide some input, then, if no input, notify the caller of a problem. Here we'd like our timeout handler to transfer control to someplace else in the program, but we can't do this. (Note also that we should cancel the call to *alarm* if there is input. So that we can fit all the code in the slide, we've left this part out.)

Doing It Legally

```
sigjmp_buf context;
int TimedInput( ) {
   signal(SIGALRM, timeout);
   if (sigsetjmp(&context, 1) == 0) {
      alarm(30);
      GetInput(); /* possible long wait for input */
      HandleInput();
      return(0);
   } else
      return(1);
}

void timeout() {
   siglongjmp(&context, 1);   /* legal but weird */
}
```


To get around the problem of not being able to use a *goto* statement to get out of a signal handler, we introduce the setjmp/longjmp facility, also known as the *nonlocal goto*. A call to *sigsetjmp* stores context information (about the current locus of execution) that can be restored via a call to *siglongjmp*. A bit more precisely: *sigsetjmp* stores into its first argument the values of its program-counter (instruction-pointer), stack-pointer, frame-pointer, and other registers representing the process's current execution context. If the second argument is non-zero, the current signal mask is saved as well. The call returns 0. When *siglongjmp* is called with a pointer to this context information as its first argument, the current register values are replaced with those that were saved. If the signal mask was saved, that is restored as well. The effect of doing this is that the process resumes execution where it was when the context information was saved: inside of sigsetjmp. However, this time, rather than returning zero, it returns the second argument passed to *siglongjmp* (1 in the example).

To use this facility, you must include the header file *setjmp.h*.



The effect of *sigsetjmp* is, roughly, to save a description of the current stack location. A subsequent call to *siglongjmp* restores the stack to where it was at the time of the call to *sigsetjmp*. Note that *siglongjmp* should be called only from a stack frame that is farther on the stack than the one in which *sigsetjmp* was called.

Dealing with Failure

- *fork*, *execl*, *wait*, *kill* directly invoke the operating system
- Sometimes the OS says no
  - usually because you did something wrong
  - sometimes because the system has run out of resources
  - such "system calls" typically return -1 to signify a problem

Reporting Failure

- Integer error code placed in global variable *errno*

```
extern int errno;
```

  - "man errno" lists all possible error codes and meanings
  - to print out most recent error

```
perror("message");
```

Fork

```
int main( ) {
  while(1) {
    if (fork() == -1) {
      perror("fork");
      exit(1);
  }
}
```


Exec

```
int main( ) {
  if (fork() == 0) {
      execl("/garbage", "garbage", 0);
      /* if we get here, there was an
         error! */
      perror("exec");
      exit(1);
  }
}
```


Signals and Blocking System Calls

- What if a signal is generated while a process is blocked in a system call?
  1) deal with it when the system call completes
  2) interrupt the system call, deal with signal, resume system call
  3) interrupt system call, deal with signal, return from system call with indication that something happened

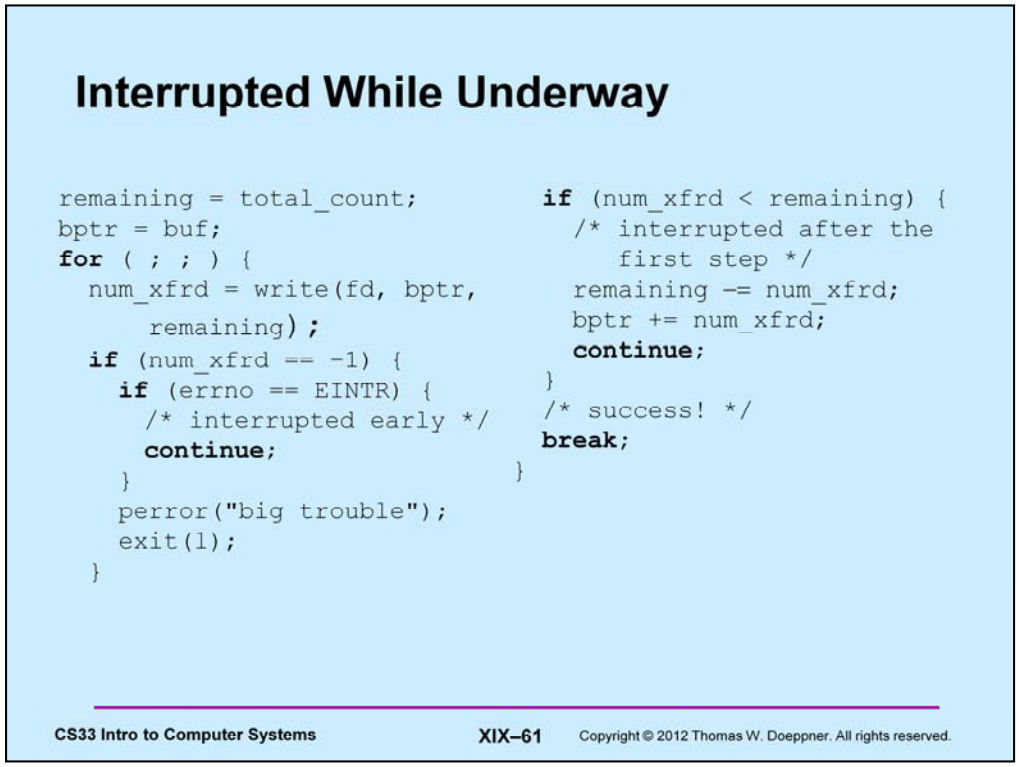
The kernel normally checks for pending, unmasked signals when a process is returning to user mode from privileged mode. However, if a process is blocked in a system call, it might be a long time until it returns and notices the signal. If the blocking time is guaranteed to be short (e.g., waiting for a disk operation to complete), then it makes sense to postpone handling the signal until the system call completes. Such waits and system calls are termed "non-interruptible." But if the wait could take a long time (e.g., waiting for something to be typed at the keyboard), then the signal should be dealt with as quickly as possible, which means that the process should be forced out of the system call.

What happens to the system call after the signal handling completes (assuming that the process has not been terminated)? One possibility is for the system to automatically restart it. However, it's not necessarily the case that it should be restarted—the signal may have caused the program to lose interest. Thus what's normally done for such "interruptible" system calls is that some indication of what has happened is passed to the program, as is shown in the next slide.

Interrupted System Calls

```
while(read(fd, buffer, buf_size) == -1) {
  if (errno == EINTR) {
    /* interrupted system call – try again */
    continue;
  }
  /* the error is more serious */
  perror("big trouble");
  exit(1);
}
```


If a system call is interrupted by a signal, the call fails and the error code is set to *errno*. The process then executes the signal handler and then returns to the point of the interrupt, which causes it to (finally) return from the system call with the error.



However, the actions of some system calls are broken up into discrete steps. For example, if one issues a system call to write a megabyte of data to a file, the write will actually be split by the kernel into a number of smaller writes. If the system call is interrupted by a signal after the first component write has completed (but while there are still more to be done), it would not make sense for the call to return an error code: such an error return would convince the program that none of the write had completed and thus all should be redone. Instead, the call completes successfully: it returns the number of bytes actually transferred, the signal handler is invoked, and, on return from the signal handler, the user program receives the successful return from the system call.

Automatic Restart

```
struct sigaction act;

act.sa_handler = reap_child;
act.sa_mask = 0;
act.sa_flags = SA_RESTART;
sigaction(SIGCHLD, &act, 0);
```

```
remaining = total_count;
bptr = buf;
while ((num_xfrd =
    write(fd, bptr, remaining))
      != remaining) {
  if (num_xfrd == -1) {
    /* no EINTR from SIGCHLD */
    …
    break;
  }
  /* still must deal with
     partial completions */
  remaining -= num_xfrd;
  bptr += num_xfrd;
}
```


Sometime it's convenient to specify that system calls be automatically restarted when a particular signal occurs. For example, in the slide we've done this for the SIGCHLD signal by setting the SA_RESTART flag in the *sigaction* structure. However, automatic restart applies only if the system call was interrupted before any transfer took place. We still must deal with the case of a partial completion.

Asynchronous Signals (1)

```
main( ) {
   void handler(int);
   signal(SIGINT, handler);

   ...   /* long-running buggy code */

}

void handler(int sig) {
   ...   /* die gracefully */
   exit(1);
}
```


Let's look at some of the typical uses for asynchronous signals. Perhaps the most common is to force the termination of the process. When the user types control-C, the program should terminate. There might be a handler for the signal, so that the program can clean up and then terminate.

Asynchronous Signals (2)

```
computation_state_t  state;

main( ) {
  void handler(int);

  signal(SIGINT, handler);

  long_running_procedure( );
}
```

```
long_running_procedure( ) {
  while (a_long_time) {
    update_state(&state);
    compute_more( );
  }
}

void handler(int sig) {
  display(&state);
}
```


Here we are using a signal to send a request to a running program: when the user types control-C, the program prints out its current state and then continues execution. If synchronization is necessary so that the state is printed only when it is stable, it must be provided by appropriate settings of the signal mask.

Asynchronous Signals (3)

```
main( ) {
  void handler(int);

  signal(SIGINT, handler);

  ... /* complicated program */

  myput("important message\n");

  ... /* more program */

}
```

```
void handler(int sig) {

  ... /* deal with signal */

  myput("equally important "
      "message\n";
}
```


In this example, both the mainline code and the signal handler call *myput*, which is similar to the standard-I/O routine *puts*. It's possible that the signal invoking the handler occurs while the mainline code is in the midst of the call to *myput*. Could this be a problem?

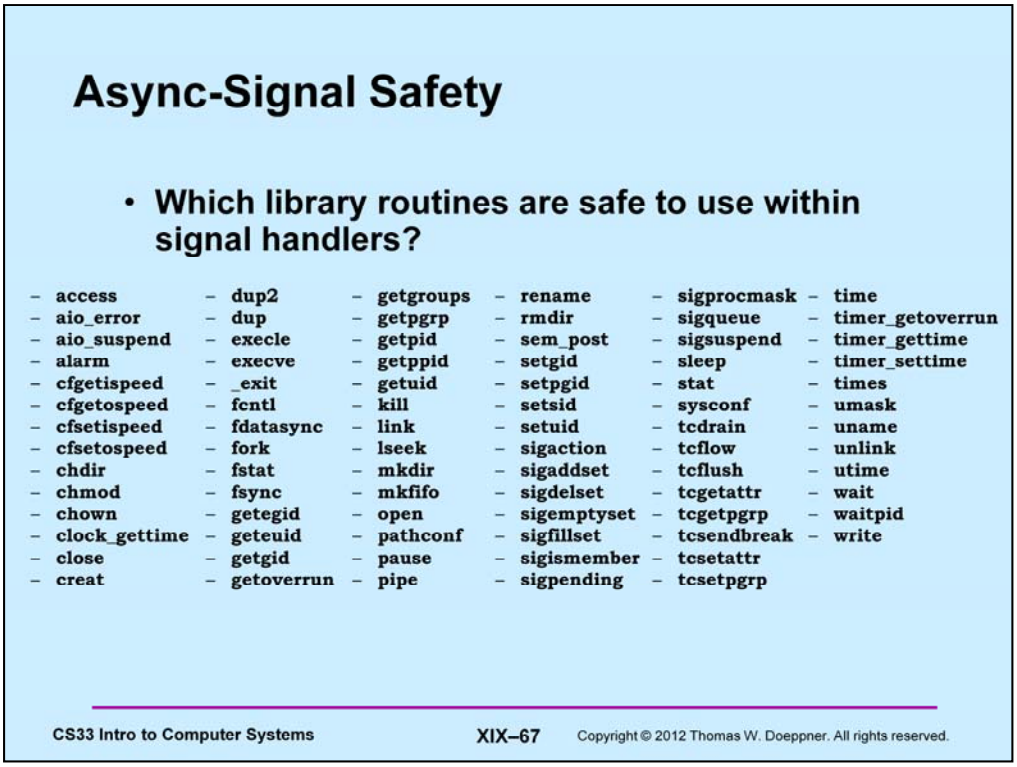
Asynchronous Signals (4)

```
void myput(char *str) {
  static char buf[BSIZE];
  static int pos=0;
  int i;
  int len = strlen(str);
  for (i=0; i<len; i++, pos++) {
    buf[pos] = str[i];
    if ((buf[pos] == '\n') || (pos == BSIZE-1)) {
      write(1, buf, pos+1);
      pos = -1;
    }
  }
}
```


Here's the implementation of *myput*, used in the previous slide. What it does is copy the input string, one character at a time, into buf, which is of size BSIZE. Whenever a newline character is encountered, the current contents of buf up to that point are written to standard output, then subsequent characters are copied starting at the beginning of buf. Similarly, if buf is filled, its contents are written to standard output and subsequent characters are copied starting at the beginning of buf. Since buf is static, characters not written out may be written after the next call to myput.

The point of *myput* is to minimize the number of calls to write, so that write is called only when we have a complete line of text or when its buffer is full.

However, consider what happens if execution is in the middle of *myput* when a signal occurs, as in the previous slide. Among the numerous problem cases, suppose *myput* is interrupted just after *pos* is set to -1 (if the code hadn't have been interrupted, *pos* would be soon incremented by 1). The signal handler now calls *myput*, which copies the first character of *str* into *buf[pos]*, which, in this case, is *buf[-1]*. Thus the first character "misses" the buffer. At best it simply won't be printed, but there might well be serious damage done to the program.



To deal with the problem on the previous page, we must arrange that signal handlers cannot destructively interfere with the operations of the mainline code. Unless we are willing to work with signal masks (which can be expensive), this means we must restrict what can be done inside a signal handler. Routines that, when called from a signal handler, do not interfere with the operation of the mainline code, no matter what that code is doing, are termed *async-signal-safe*. The POSIX 1003.1 spec defines a number of these, as shown in the slide.

Note that POSIX specifies only those routines that must be async-signal safe. Implementations may make other routines async-signal safe as well. In particular, everything is async-signal-safe in Solaris (but this is not the case in Linux).