

CS 33

Files Part 4

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

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`, in the next few slides.

Appending Data to a File (1)

```
int fd = open("file", O_WRONLY);
lseek(fd, 0, SEEK_END);
    // sets the file location to the end
write(fd, buffer, bsize);
    // does this always write to the
    // end of the file?
```

We'd like to write data to the end of a file. One approach, shown here, is to use the **lseek** system call to set the file location in the file context structure to the end of the file. Once this is done, then when we write to the file, we're writing to its end and thus are appending data to the file.

However, this assumes that no other program is writing data to the end of the file at the same time. If another program were doing this, then the file could grow between our calls to **lseek** and **write**. If this happens, then the write would no longer be to the end of the file but would overwrite the data written by the other program.

Appending Data to a File (2)

```
int fd = open("file", O_WRONLY | O_APPEND);  
write(fd, buffer, bsize);  
    // this is guaranteed to write to the  
    // end of the file
```

By using the `O_APPEND` option of `open`, we make certain that writes on this file descriptor are always to the end of file. If another program is doing this at the same time, the operating system makes certain that one doesn't start until after the other ends.

In the Shell ...

% program >> file

The ">>" operator tells the shell to open file with the O_APPEND flag so that writes are always to the end of the file.

File Access Permissions

- **Who's allowed to do what?**
 - **who**
 - » **user (owner)**
 - » **group**
 - » **others (rest of the world)**
 - **what**
 - » **read**
 - » **write**
 - » **execute**

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.

Permissions Example

adm group:
joe, angie

```
$ ls -lR
.:
total 2
drwxr-x--x  2 joe    adm    1024 Dec 17 13:34 A
drwxr----- 2 joe    adm    1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 joe    adm     593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-  1 joe    adm     446 Dec 17 13:34 x
-rw----rw-  1 angie  adm     446 Dec 17 13:45 y
```

The `ls -lR` command lists the contents of the current directory, its subdirectories, their subdirectories, etc. in long format (the `l` causes the latter, the `R` the former).

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 **joe** and **angie** are members of the **adm** group; **leo** is not.

- May **leo** list the contents of directory *A*?
- May **leo** read *A/x*?
- May **angie** list the contents of directory *B*?
- May **angie** modify *B/y*?
- May **joe** modify *B/x*?

- May **joe** read B/y ?

Setting File Permissions

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode)
```

- sets the file permissions of the given file to those specified in *mode*
- only the owner of a file and the superuser may change its permissions
- nine combinable possibilities for *mode* (*read/write/execute for user, group, and others*)
 - » `S_IRUSR (0400)`, `S_IWUSR (0200)`, `S_IXUSR (0100)`
 - » `S_IRGRP (040)`, `S_IWGRP (020)`, `S_IXGRP (010)`
 - » `S_IROTH (04)`, `S_IWOTH (02)`, `S_IXOTH (01)`

The **chmod** system call (and the similar **chmod** shell command) is 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, for example, 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.

Permission Bits

- It's worth your while to remember this!
 - read: 4
 - write: 2
 - execute: 1
 - read/write: 6
 - read/write/execute: 7
- user:group:others
 - » 0751
 - rwx for user, rx for group, x for others
 - » 0640
 - rw for user, r for group, nothing for others

For each category (user, group, other), three bits represent their permissions. Thus these are usually viewed as octal digits.

Umask

- Standard programs create files with “maximum needed permissions” as mode
 - compilers: 0777
 - editors: 0666
- Per-process parameter, *umask*, used to turn off undesired permission bits
 - e.g., turn off all permissions for others, write permission for group: set umask to 027
 - » compilers: permissions = $0777 \& \sim(027) = 0750$
 - » editors: permissions = $0666 \& \sim(027) = 0640$
 - set with *umask* system call or (usually) shell command

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, write permission 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.

(Recall that numbers written with a leading 0 are in octal (base-8) notation.)

Quiz 1

You get the following message when you attempt to execute `./program` (a file that you own):

```
bash: ./program: Permission denied
```

You're first response should be:

- a) execute the shell command
`chmod 0644 program`
- b) execute the shell command
`chmod 0755 program`
- c) find the source code for program and recompile it
- d) make an Ed post

Creating a File

- Use either *open* or *creat*

- `open(const char *pathname, int flags, mode_t mode)`
 - » flags must include `O_CREAT`
- `creat(const char *pathname, mode_t mode)`
 - » *open* is preferred

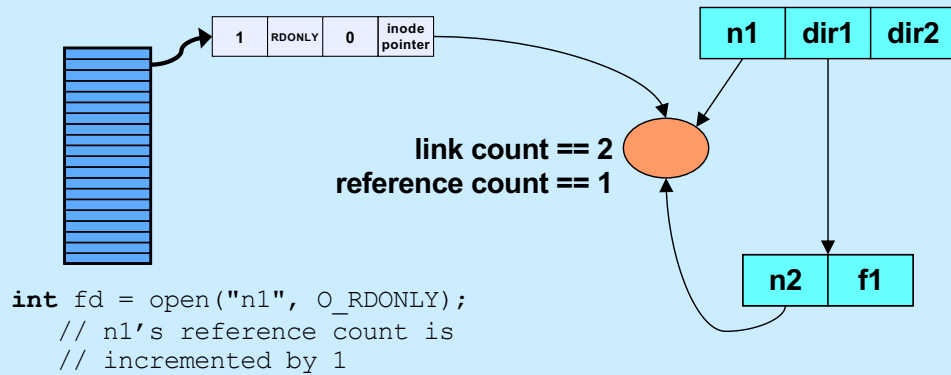
- The *mode* parameter helps specify the permissions of the newly created file

- `permissions = mode & ~umask`

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.

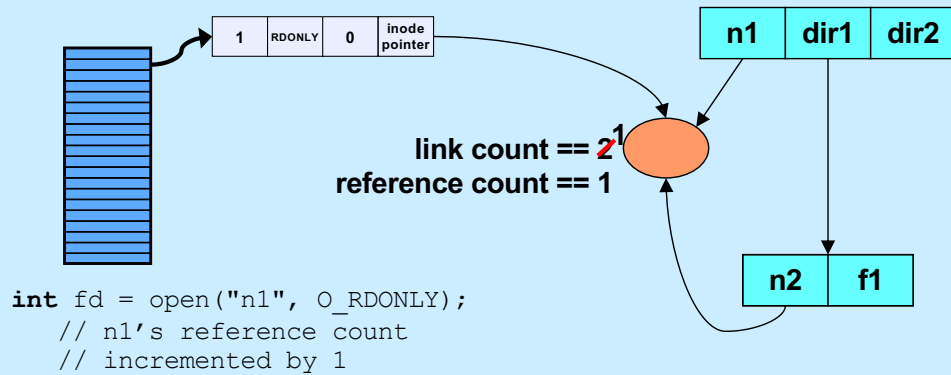
When a file is created by either **open** or **creat**, the file's initial access permissions are the bitwise AND of the mode parameter and the complement of the process's **umask** (explained in the previous slide).

Link and Reference Counts



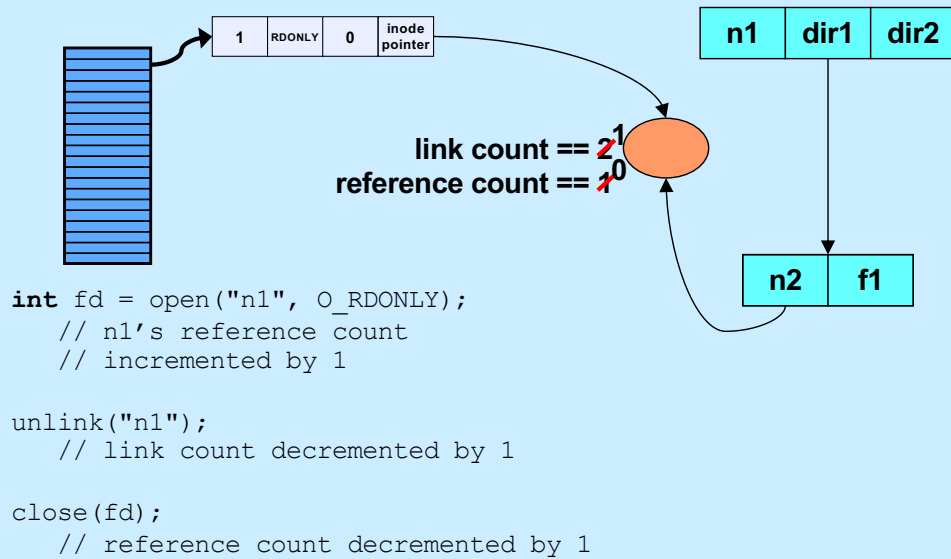
A file's link count is the number of directory entries that refer to it. There's a separate reference count that's the number of file context structures that refer to it (via the inode pointer – see slide XX-13). These counts are maintained in the file's inode, which contains all information used by the operating system to refer to the file (on disk).

Link and Reference Counts



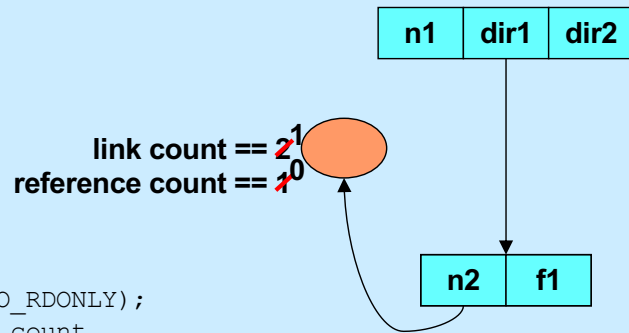
Note that the shell's `rm` command is implemented using `unlink`; it simply removes the directory entry, reducing the file's link count by 1.

Link and Reference Counts



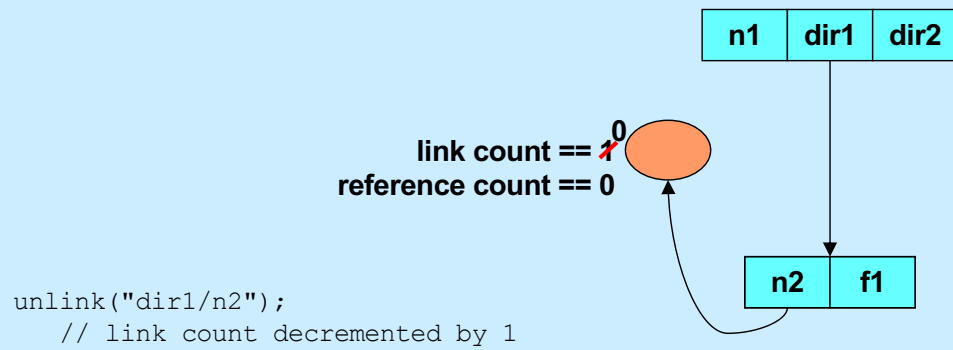
Link and Reference Counts

```
int fd = open("n1", O_RDONLY);  
    // n1's reference count  
    // incremented by 1  
  
unlink("n1");  
    // link count decremented by 1  
  
close(fd);  
    // reference count decremented by 1
```



A file is deleted if and only if both its link and reference counts are zero.

Link and Reference Counts



A file is deleted if and only if both its link and reference counts are zero.

Quiz 2

```
int main() {  
    int fd = open("file", O_RDWR|O_CREAT, 0666);  
    unlink("file");  
    PutStuffInFile(fd);  
    GetStuffFromFile(fd);  
    return 0;  
}
```

Assume that *PutStuffInFile* writes to the given file, and *GetStuffFromFile* reads from the file.

- a) The file will be deleted when the program terminates
- b) This program is doomed to failure, since the file is deleted before it's used
- c) Because the file is used after the unlink call, it won't be deleted

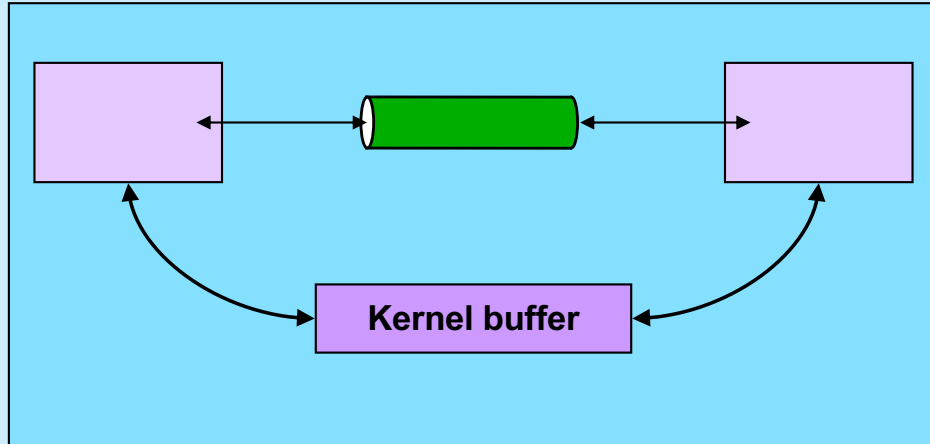
Note that when a process terminates, all its open files are automatically closed.

Interprocess Communication (IPC): Pipes



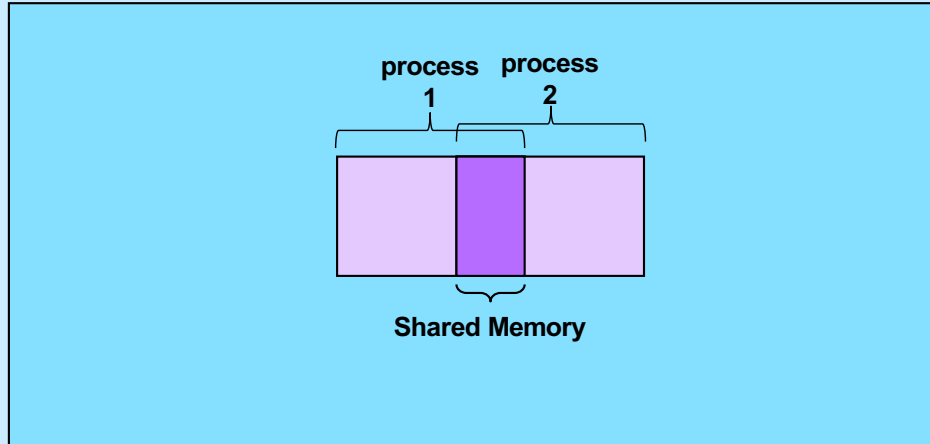
A rather elegant way for different processes to communicate is via a pipe: one process puts data into a pipe, another process reads the data from the pipe.

Interprocess Communication: Same Machine I



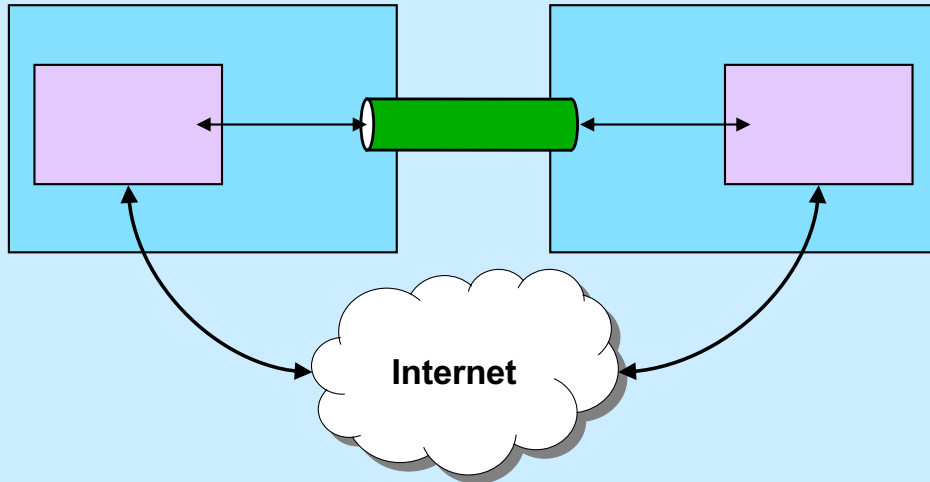
The implementation of a pipe involves the sending process using a write system call to transfer data into a kernel buffer. The receiving process fetches the data from the buffer via a read system call. We'll cover some of the details about how this works when we discuss multithreaded programming later in the semester.

Interprocess Communication: Same Machine II



Another way for processes to communicate is for them to arrange to have some memory in common via which they share information. We discuss this approach later in the semester.

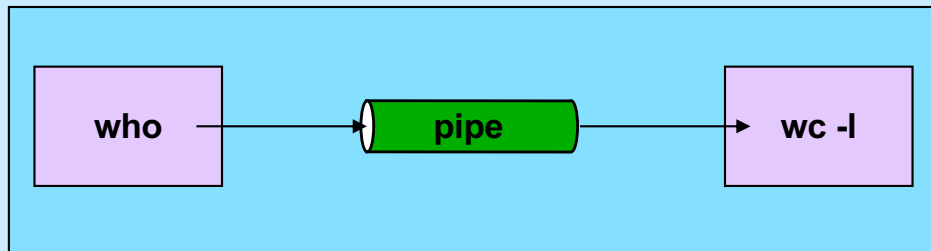
Interprocess Communication: Different Machines



The pipe abstraction can also be made to work between processes on different machines. We discuss this later in the semester.

Pipes

```
$cslab2e who | wc -l
```




The vertical bar (“|”) is the pipe symbol in the shell. The syntax shown above represents creating two processes, one running **who** and the other running **wc**. The standard output of **who** is setup to be the pipe; the standard input of **wc** is setup to be the pipe. Thus, the output of **who** becomes the input of **wc**. The “-l” argument to **wc** tells it to count and print out the number of lines that are input to it. The **who** command writes to standard output the login names of all logged in users. The combination of the two produces the number of users who are currently logged in.

Using Pipes in C

```
$cs1ab2e who | wc -l
```

```
int fd[2];
pipe(fd);
if (fork() == 0) {
    close(fd[0]);
    close(1);
    dup(fd[1]); close(fd[1]);
    execl("/usr/bin/who", "who", 0); // who sends output to pipe
}
if (fork() == 0) {
    close(fd[1]);
    close(0);
    dup(fd[0]); close(fd[0]);
    execl("/usr/bin/wc", "wc", "-l", 0); // wc's input is from pipe
}
close(fd[1]); close(fd[0]);
// ...
```



The **pipe** system call creates a “pipe” in the kernel and sets up two file descriptors. One, in `fd[1]`, is for writing to the pipe; the other, in `fd[0]`, is for reading from the pipe. The input end of the pipe is set up to be **stdout** for the process running **who**, and the output end of the pipe is closed, since it’s not needed. Similarly, the input end of the pipe is set up to be **stdin** for the process running **wc**, and the input end is closed. Since the parent process (running the shell) has no further need for the pipe, it closes both ends. When neither end of the pipe is open by any process, the system deletes it. If a process reads from a pipe for which no process has the input end open, the read returns 0, indicating end of file. If a process writes to a pipe for which no process has the output end open, the write returns -1, indicating an error and **errno** is set to EPIPE; the process also receives the SIGPIPE signal, which we explain in the next lecture.

Shell 1: Artisanal Coding

```
while ((line = get_a_line()) != 0) {
    tokens = parse_line(line);
    for (int i=0; i < ntokens; i++) {
        if (strcmp(tokens[i], ">") == 0) {
            // handle output redirection
        } else if (strcmp(tokens[i], "<") == 0) {
            // handle input redirection
        } else if (strcmp(tokens[i], "&") == 0) {
            // handle "no wait" (done in shell 2)
        } ... else {
            // handle other cases
        }
    }
    if (fork() == 0) {
        // ...
        execv(...);
    }
    // ...
}
```

This is, of course, over simplified. The complete program should be 200 or so lines long.

Note that "handle x" might simply involve taking note of x, then dealing with it later.

Also note that “artisanal” anything is always better than “non-artisanal” anything.

Shell 1: Non-Artisanal Coding (1)

```
while ((line = get_a_line()) != 0) {
    tokens = parse_line(line);
    for (int i=0; i < ntokens; i++) {
        // handle "normal" case
    }
    if (fork() == 0) {
        // ...
        execv(...);
    }
    // ...
}
```

One first writes the code assuming no redirection symbols and no &s. That's perfectly reasonable.

Shell 1: Non-Artisanal Coding (2)

```
next_line: while ((line = get_a_line()) != 0) {
    tokens = parse_line(line);
    for (int i=0; i < ntokens; i++) {
        if (redirection_symbol(token[i])) {
            // ...
            if (fork() == 0) {
                // ...
                execv(...); whoops!
            }
            // ...
            goto next_line;
        }
        // handle "normal" case
    }
    if (fork() == 0) {
        // ... (whoops!)
        execv(...);
    }
    // ...
}
```

The next step is to deal with redirection symbols. Rather than modify the fork/exec code so as to work for both cases, it's copied into the new case and modified there. Thus, we now have two versions of the fork/exec code to maintain. If we find a bug in one, we need to remember to fix it in both.

At this point it's becoming difficult for you to debug your code, and really difficult for TAs to figure out what you're doing so they can help you.

Shell 1: Non-Artisanal Coding (3)

```
next_line: while ((line = get_a_line()) != 0) {
    tokens = parse_line(line);
    for (int i=0; i < ntokens; i++) {
        if (redirection_symbol(token[i])) {
            // ...
            if (fork() == 0) {
                // ...
                execv(...);
            }
            // ... deal with &
            goto next_line;
        }
        // handle "normal" case
    }
    if (fork() == 0) {
        // ...
        execv(...);
    }
    // ... also deal with & here!
}
```

We now have to handle & in multiple places.

If done this way, you could well have a 700-line program (the artisanal code took around 200 lines).

Shell 1: Non-Artisanal Coding (Worse)

```
next_line: while ((line = get_a_line()) != 0) {
tokens = parse_line(line);
for (int i=0; i < ntokens; i++) {
if (redirection_symbol(token[i])) {
// ...
if (fork() == 0) {
// ...
execv(...);
}
// ... deal with &
goto next_line;
}
// handle "normal" case
}
if (fork() == 0) {
// ...
execv(...);
}
// ... also deal with & here!
}
```

If the code is poorly formatted, it's even tougher to understand.

Artisanal Programming

- **Factor your code!**
 - `A; D | B; D | C; D = (A | B | C); D`
- **Format as you write!**
 - don't run the formatter only just before handing it in
 - your code should always be well formatted
- **If you have a tough time understanding your code, you'll have a tougher time debugging it and TAs will have an even tougher time helping you**

It's Your Code

- **Be proud of it!**
 - it not only works; it shows skillful artisanship
- **It's not enough to merely work**
 - others have to understand it
 - » (not to mention you ...)
 - you (and others) have to maintain it
 - » shell 2 is coming soon!

CS 33

Signals Part 1

An Interlude Between Shells

- **Shell 1**
 - it can run programs
 - it can redirect I/O
- **Signals**
 - a mechanism for coping with exceptions and external events
 - the mechanism needed for shell 2
- **Shell 2**
 - it can control running programs

Whoops ...

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



Oh dear...

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

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

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 sent in response to these events can seemingly occur at any moment and are referred to as **asynchronous** signals.

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.

A core dump is the contents of a process's address space, written to a file (called **core**), reflecting what the situation was when it was terminated by a signal. They can be used by gdb to see what happened (e.g., to get a backtrace). Since they're fairly large and rarely looked at, they're normally disabled. We'll look at them further shortly.

Signal Types

SIGABRT	<i>abort</i> 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, the official Unix specification. 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 or ignored.

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
 - type `ctrl-\`
 - » sends signal 3 (SIGQUIT) to current process
 - type `ctrl-z`
 - » sends signal 20 (SIGTSTP) to current process
 - do something bad
 - » bad address, bad arithmetic, etc.

Note that the signals generated by typing control characters on the keyboard are actually sent to the current process group of the terminal, a concept we discuss soon.

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);
```

The **signal** function establishes a new handler for the given signal and returns the address of the previous handler.

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

```
void sigloop() {
    while(1)
        ;
}

int main() {
    void handler(int);
    signal(SIGINT, handler);
    sigloop();
    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.

Digression: Core Dumps

- **Core dumps**
 - files (called “core”) that hold the contents of a process’s address space after termination by a signal
 - they’re large and rarely used, so they’re often disabled by default
 - use the **ulimit** command in bash to enable them

```
ulimit -c unlimited
```

- use **gdb** to examine the process (post-mortem debugging)

```
gdb sig core
```

Don’t forget to delete the core files when you’re finished with them! Note that neither OSX nor Windows supports core dumps.

Some details on the **ulimit** command: it supports both a hard limit (which can’t be modified) and a soft limit (which can later be modified). By default, **ulimit** sets both the hard and soft limits. Thus typing

```
ulimit -c 0
```

sets both the hard and soft limits of core file size to 0, meaning that you can’t increase the limit later (within the execution of the current invocation of this shell).

But if you type

```
ulimit -Sc 0
```

then just the soft limit is modified, allowing you to type

```
ulimit -c unlimited
```

later.

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 myhandler(int);
    sigemptyset(&act.sa_mask); // zeroes the mask
    act.sa_flags = 0;
    act.sa_handler = myhandler;
    sigaction(SIGINT, &act, NULL);
    ...
}
```

The **sigaction** system call is the more general 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 won't get a chance to discuss it, but it's used when more information about the cause of a signal is needed.

When 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 that we describe in upcoming slides.

Example

```
int main() {
    void handler(int);
    struct sigaction act;
    act.sa_handler = handler;
    sigemptyset(&act.sa_mask);
    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.

Quiz 3

```
int main() {  
    void handler(int);  
    struct sigaction act;  
    act.sa_handler = handler;  
    sigemptyset(&act.sa_mask);  
    act.sa_flags = 0;  
    sigaction(SIGINT, &act, NULL);  
  
    while(1)  
        ;  
    return 1;  
}  
  
void handler(int signo) {  
    printf("I received signal %d. "  
        "Whoopee!!\n", signo);  
}
```

You run the example program, then quickly type ctrl-C. What is the most likely explanation if the program then terminates?

- a) this “can’t happen”; thus there’s a problem with the system
- b) you’re really quick or the system is really slow (or both)
- c) what we’ve told you so far isn’t quite correct