CSC209

Software Tools & Systems Programming in UNIX

W. James MacLean
maclean@cdf.toronto.edu
PT290F, 978-6277

About CSC209

General Info

- L0101: Meets Monday & Wednesday, 12pm in SS2118; Tutorials Fridays at 12pm
- L0501: Meets Thursday, 7pm in RW110; Tutorials Thursdays at 6pm
- Website: http://www.cs.toronto.edu/~maclean/csc209/Spring00/
- Newsgroup: ut.cdf.csc209 (or send e-mail to ut.cdf.csc209h@cdf.toronto.edu)

Assignments

- Assignment #1: 5% Fri, Feb 4th
  - Shell Scripts
- Assignment #2: 10% Fri, Feb 18th
  - Utilities, File/Directories
- Assignment #3: 10% Fri, Mar 17th
  - Inter-process communication
- Assignment #4: 15% Fri, Apr 7th
  - Sockets & Threads

- Assignments are due *at the beginning* of tutorials. Late assignments should be submitted to D.L. Pratt Bldg., room 283. Penalties for late submissions:
    - *Time of submission* ***(based on electronic submission time)***:
        - by 5pm on due date - penalty = - 20% (of the maximum mark)
        - by 9am three days after due date - penalty = - 40%
        - after 9am three days after due date - penalty = - 100%
- The late penalties are only waived for a good reason such as a documented medical or other emergency. Discuss such excuses with the instructor, *not* with your tutor.

Plagiarism

- *The work you submit must be your own*, done without participation by others. It is an academic offence to hand in anything written by someone else without acknowledgement.

Tutorials

- TBA
- **Lab Hours:** Announced in advance of each assignment due date, take place in EA107

Contacting Me

- **Office Hours:**
  - Wednesdays, 2–3 pm, D.L. Pratt Bldg. 290F
  - by special appointment
- **Via e-mail:**
  - maclean@cdf.toronto.edu
  - Important! Include "CSC209" in subject line
- **Via phone:**
  - 978-6277 (I check messages only on days I'm in)

Important Dates

- **Last day to enrol:** Jan 14th, 2000
- **Last day to drop:** Mar 3rd, 2000
- **Last class:** Apr 5th/6th, 2000
- **Midterm:** Feb 24th, 2000 (in class)
- **Exam:** TBA (schedule out in March)

Reference Material

- *An introduction to UNIX*, Paul S. Wang
- *UNIX System Programming*, Haviland/Gray/Salama
- *Programming with POSIX Threads*, David Butenhof
- Lecture Notes
- *Advanced Programming in the UNIX Environment*, Richard Stevens

Reference Material (2)

- *The C Programming Language, 2nd Ed.*, Kernighan & Ritchie
- *Managing Projects with Make*, Oram/Talbott
- *UNIX for Programmers and Users, 2nd. Ed.*, Glass/Abels
- *UNIX Network Programming*, Richard Stevens

Course Content

- Why UNIX? History
- UNIX Basics: Processes, Login
- Shells: command processing, running programs, shell programming
- I/O: file descriptors *vs.* streams
- Processes: creating, destroying, monitoring
- System Calls, Standard C Library

- File/directory manipulation
- Signals
- Pipes, Sockets, Inter-process communication
- Multiplexed I/O
- Shared Memory, Semaphores
- POSIX Threads
- Concurrency: Race Conditions, Deadlock, Postponement, Producer-Consumer
- Project Management

Some Self Study Topics

- Standard UNIX Utilities
- Writing MAN pages
- 'C' Refresher
- String Handling in 'C'

Section I

UNIX FOR THE USER

What is UNIX good for?

- Supports many users running many programs at the same time, all sharing (transparently) the same computer system
- Promotes information sharing
- More than just used for running software … geared towards facilitating the job of creating new programs. So UNIX is “expert friendly”
- Got a bad reputation in business because of this aspect

History (Introduction)

- Ken Thompson working at Bell Labs in 1969 wanted a small MULTICS for his DEC PDP-7
- He wrote UNIX which was initially written in assembler and could handle only one user at a time
- Dennis Ritchie and Ken Thompson ported an enhanced UNIX to a PDP-11/20 in 1970
- Ritchie ported the language BCPL to UNIX in 1970, cutting it down to fit and calling the result “B”
- In 1973 Ritchie and Thompson rewrote UNIX in “C” and enhanced it some more
- Since then it has been enhanced and enhanced and enhanced and …
- See Wang, page 1 for a brief discussion of UNIX variations
- POSIX (portable operating system interface) - IEEE, ANSI

Some Terminology

- *Program*: executable file on disk (machine language binary or script)
- *Process*: executing instance of a program
- *Process ID*: unique, non-negative integer identifier (a handle by which to refer to a process)
- *UNIX kernel*: a C program that implements a general interface to a computer to be used for writing programs
- *System call*: well-defined entry point into kernel, to request a service
- *UNIX technique*: for each system call, have a function of same name in the standard C library
  - user process calls this function
  - function invokes appropriate kernel service

Logging in

- Login name, password
- System password file: usually "**`/etc/passwd`**"
- **`/etc/passwd`** has 7 colon-separated fields:

```
maclean:x:132:114:James MacLean:
^^^1^^^ 2 ^3^ ^4^ ^^^^^^5^^^^^^
       /u/maclean:/var/shell/tcsh
       ^^^^^6^^^^ ^^^^^^^7^^^^^^^
```

1: user name
2: password (hidden)
3: uid
4: gid
5: "in real life"
6: $HOME
7: shell

Shells

- Bourne shell, C shell, Korn shell, tcsh
  - command line interpreter that reads user input and executes commands

```
> ls -l /var/shell
  total 6
  lrwxrwxrwx 1 root 12 May 15 1996 csh ->
  /usr/bin/csh
  lrwxrwxrwx 1 root 12 May 15 1996 ksh ->
  /usr/bin/ksh
  lrwxrwxrwx 1 root 17 May 15 1996 newsh ->
  /local/sbin/newsh
  lrwxrwxrwx 1 root 11 May 15 1996 sh -> /usr/bin/sh
  lrwxrwxrwx 1 root 15 May 15 1996 tcsh ->
  /local/bin/tcsh
```

newsh "man page"

```
newsh
   newsh - shell for new users
SYNOPSIS
   newsh
DESCRIPTION
   newsh shows the CDF rules, runs passwd to force the user  to
   change  his  or  her  password,  and runs chsh to change the
   user's shell to the default system shell (/local/bin/tcsh).
FILES
   /etc/passwd
SEE ALSO
   passwd(1), chsh(1)
HISTORY
   Written by John DiMarco at the University of Toronto, CDF
```

Files and Directories

- UNIX filesystem is a hierarchical arrangement of directories & files
- Everything starts in a directory called root whose name is the single character `/`
- Directory: file that contains directory entries
- File name and file attributes
  - type
  - size
  - owner
  - permissions
  - time of last modification

Files: an example

```
> stat /u/maclean

 File: "/u/maclean" -> "/homes/u1/maclean"
  Size: 17   Allocated Blocks: 0
  Filetype: Symbolic Link
  Mode: (0777/lrwxrwxrwx)   Uid: (    0/    root)
  Gid: ( 1/ other)
Device: 0/1  Inode: 221       Links: 1
  Device type: 0/0
Access: Sun Sep 13 18:32:37 1998
Modify: Fri Aug 28 15:42:09 1998
Change: Fri Aug 28 15:42:09 1998
```

Directories and Pathnames

- Command to create a directory: **`mkdir`**
- Two file names automatically created:
  - current directory ("`.`")
  - parent directory ("`..`")

- A pathname is a sequence of 0 or more file names, separated by **`/`**, optionally starting with a **`/`**
  - *absolute* pathnames: begins with a **`/`**
  - *relative* pathnames: otherwise

Working directory

- Current working directory (cwd)
  - directory from which all relative pathnames are interpreted

- Change working directory with the command: **`cd`** or **`chdir`**
- Print the current directory with the command: **`pwd`**

- Home directory: working directory when we log in
  - obtained from field 6 in **`/etc/passwd`**
- Can refer to home directory as **`~maclean`** or **`$HOME`**

"pushd" and "popd"

- Makes directory navigation easier
- `pushd` pushes CWD directory pathname onto a stack maintained by the shell
- `popd` pops a directory pathname from the same stack and makes it the CWD
- **Example:**

```
% pwd
/usr/bin
% pushd  ~/; pwd
~ /usr/bin
% popd; pwd
/usr/bin
```

Permissions

- When a file is created, the UID and GID of the creator are remembered
- Every named file has associated with it a set of permissions in the form of a string of bits:

rwxs	rwxs	rwx
owner	group	others

mode	regular	directory
r	read	list contents
w	write	create and remove
x	execute	search
s	setuid/gid	n/a

- setuid/gid executes program with user/group ID of file's owner
- Use **`chmod`** to change permissions

Input and Output

- File descriptor
  - a small non-negative integer used by kernel to identify a file
- A shell opens 3 descriptors whenever a new program is run:
  - *standard input* (normally connected to terminal)
  - *standard output*
  - *standard error*
- Re-direction:

```
ls >file.list
```

Basic UNIX Tools

```
man  ("man -k", "man man") (1.13)
ls -la ("hidden files")
cd
pwd
du, df
chmod
cp, mv, rm  (in cshrc: "alias rm rm -i" ... )
mkdir, rmdir (rm -rf)
diff
grep
sort
```

More Basic UNIX Tools

```
more, less, cat
head, tail, wc
compress, uncompress,
gzip, gunzip, zcat
lpr, lpq, lprm
quota -v a209xxxx
pquota -v a209xxxx
logout, exit
mail, mh, rn, trn, nn
who, finger
date, passwd
```

Introduction to the C Shell

What is the Shell?

- A command-line interpreter program that is the interface between the user and the Operating System.
- The shell:
  - analyzes each command
  - determines what actions to be performed
  - performs the actions
- Example:

```
wc –l file1 > file2
```

csh Shell Facilities

- Automatic command searching
- Input-output redirection
- Pipelining commands
- Command aliasing
- Job control
- Command history
- Shell script files

I/O Redirection

- **stdin** (fd=0), **stdout** (fd=1), **stderr** (fd=2)

- Redirection examples: (**<, >, >>, >&, >!, >&!**)

```
fmt
fmt < personal_letter
fmt > new_file
fmt < personal_letter > new_file
fmt >> personal letter
fmt < personal_letter >& new_file
fmt >! new_file
fmt >&! new_file
```

Pipes

- Examples:

```
who | wc -l
ls /u/csc209h |& sort -r
```

- For a *pipeline*, the standard output of the first process is connected to the standard input of the second process

Filename Expansion

- Examples:

```
ls *.c
rm file[1-6].?
cd ~/bin
ls ~culhane
```

`*`	Matches any string (including null)
`?`	Matches any single character
`[...]`	Matches any one of the enclosed characters
`[.-.]`	Matches any character lexically between the pair
`[!...]`	Matches any character not enclosed

Command Aliases

- Examples:

```
alias md mkdir
alias lc ls -F
alias rm rm -i
\rm *.o
unalias rm
alias
alias md
alias cd 'cd \!*; pwd'
```

Job Control

- A *job* is a program whose execution has been initiated by the user
- At any moment, a job can be running or stopped (suspended)
- Foreground job:
  - a program which has control of the terminal
- Background job:
  - runs concurrently with the parent shell and does not take control of the keyboard
- Initiate a background job by appending the "**`&`**" metacharacter
- Commands: **`jobs`**, **`fg`**, **`bg`**, **`kill`**, **stop**

Some Examples

`a | b | c`

- connects standard output of one program to standard input of another
- shell runs the entire set of processes in the foreground
- prompt appears after c completes

`a & b & c`

- executes a and b in the background and c in the foreground
- prompt appears after c completes

`a & b & c &`

- executes all three in the background
- prompt appears immediately

`a | b | c &`

- same as first example, except it runs in the background and prompt appears immediately

`nice`, **`nohup`**

- **`nice`** (csh built-in) sets the priority level of a command. The higher the priority number, the slower it will run.
- Usage: **`nice [ + n | - n ] command`**
- Example:

  `nice +20 emacs &`

  `nice -20 importantJob` *only root can give negative value*

- **`nohup`** (csh built-in) makes a process immune to hangup conditions
- Usage: **`nohup command`**
- Example:

  `nohup bigJob &`

- in **`~/.logout`**: **`/usr/bin/kill -HUP -1 >& /dev/null`**

The History Mechanism

- **Example session:**

```
alias grep grep -i
grep a209 /etc/passwd >! ~/list
history
cat ~/list
!!
!2
!-4
!c
!c > newlist
grpe a270 /etc/passed | wc -l
^pe^ep
```

Section II

SHELL SCRIPTING

Core Functionality of Shells

- built-in commands
- variables
- wildcards (file name expansion)
- background processing
- scripts
- redirection
- pipes
- subshells
- command substitution

Executables vs. Built-ins

- Most UNIX commands invoke utility programs that are stored as executable files in the directory hierarchy
- Shells also contains several built-in commands, which it executes internally
- Type **`man shell_builtins`** for a partial listing
- Built-in commands execute as subroutines, and do not spawn a child-shell via `fork()`
  - Expect built-in (*e.g.* cd) to be faster than external (*e.g.* ls)

Built-In:	**Non-Built-In:**
cd, echo, jobs, fg, bg	ls, cp, more

Variables

- Two kinds of variables:
  - local
  - environment
- Both hold data in a string format
- Main difference: when a shell invokes another shell, the child shell gets a copy of its parent's environment variables, but not its local shell variables
- Any local shell variables which have corresponding environment variables (**`term`**, **`path`**, **`user`**, etc.) are automatically inherited by subshells

Variables (cont.)

- Local (shell) variables:
  - Simple variable: holds one value
  - List variable: holds one or more values
  - Use **`set`** and **`unset`** to define, delete, and list values

- Environment variables:
  - Use **`setenv`** and **`printenv`** to set and list values
  - All environment variables are simple (ie: no list variables … compare shell variable **`$path`** to enviroment variable **`$PATH`**)

Shell Scripts

- Shell scripts are just files containing shell commands to be executed
- also make use of shell flow-control structures such as `if ... else`, `foreach`, *etc*.
- like a simple program

Startup Files

- Every time **`csh`** is invoked, **`$HOME/.cshrc`** is read, and contents of the file are executed
- If a given **`csh`** invocation is the login shell, **`$HOME/.login`** will also be read and its contents executed
- **csh -f** starts a shell without reading initialization files
- opening a new **xterm -ls** under X-windows will open a new login shell

Sourcing files

- Assume you create a file called "*my_aliases*"
- Typing **`csh my_aliases`** executes the lines in this file, but it occurs in the forked csh, so it will have no lasting effect on the interactive parent shell
- Correct method is to use the *source* command:

  `source my_aliases`

- Common setup:
  - put all aliases in a file called **`$HOME/.alias`**
  - add the line "source .alias" to the last line of **`$HOME/.cshrc`**

Input Processing

- When a input is typed, it is processed as follows:
  - *history* substitution
  - *alias* substitution
  - *variable* substitution
  - *command* substitution
  - *file name* expansion

Command Substitution

- Can substitute the output from a command into the text string of a command

```
set dir = `pwd`
set name = `pwd`/test.c
set x = `/bin/ls -l
  $file`
```

Line By Line Command Subs

- Examples on previous slides assign each token (separated by "white space") to an array index
- Contrast with:
  set x = "`who`"
- Then `echo $x[1]` prints (for example)
  `bor          pts/46          Jan  2 14:48`

Shell Variables
(setting)

- Examples:

```
set V
set V = abc
set V = (123 def ghi)
set V[2] = xxxx
set
unset V
```

Shell Variables
(referencing and testing)

- Examples:

```
echo $term
echo ${term}
echo $V[1]
echo $V[2-3]
echo $V[2-]
set W = ${V[3]}

set V = (abc def ghi 123)
set N = $#V
echo $?name
echo ${?V}
```

Shell Control Variables

`filec`	a given with tcsh
`prompt`	my favourite: **`set prompt = "%m:%~%#"`**
`ignoreeof`	disables *Ctrl-D* logout
`history`	number of previous commands retained
`mail`	how often to check for new mail
`path`	list of directories where *csh* will look for commands (†)
`noclobber`	protects from accidentally overwriting files in redirection
`noglob`	turns off file name expansion

- *Shell variables* should not to be confused with *Environment variables.*

Variable Expressions

- Examples:

```
set list1 = (abc def)
set list2 = ghi
set m = ($list2 $list1)

@ i = 10     # could be done with “set i = 10”
@ j = $i * 2 + 5
@ i++
```

- comparison operators: `==, !=, <, <=, >, >=, =~, !~`

File-oriented Expressions

Usage:

```
-option filename
```

where 1 (true) is returned if selected option is true, and 0 (false) otherwise

`-r filename`	Test if *filename* can be read
`-e filename`	Test if *filename* exists
`-d filename`	Test if *filename* is a directory
`-w filename`	Test if *filename* can be written to
`-x filename`	Test if *filename* can be executed
`-o filename`	Test if you are the owner of *filename*

- See Wang, table 7.2 (page 199) for more

csh Script Execution

- Several ways to execute a script:
  1) **`/usr/bin/csh script-file`**
  2) **`chmod u+x script-file`**, then:
     - a) make first line a comment, starting with "**`#`**"
       - (this will make your default shell run the script-file)
     - b) make first line "**`#!/usr/bin/csh`**"
       - (this will ensure *csh* runs the script-file, preferred!)

- Useful for debugging your script files:

  "**`#!/usr/bin/csh -x`**" or "**`#!/usr/bin/csh -v`**"

- Another favourite:

  "**`#!/usr/bin/csh -f`**"

if Command

- Syntax:

```
if ( test-expression ) command
```

- Example:

```
if ( -w $file2 ) mv $file1 $file2
```

- Syntax:

```
if ( test-expression ) then
   shell commands
else
   shell commands
endif
```

if Command (cont.)

- Syntax:

```
if ( test-expression ) then
   shell commands
else if ( test-expression ) then
   shell commands
else
   shell commands
endif
```

foreach Command

- Syntax:

```
foreach item ( list-of-items )
   shell commands
end
```

- Example:

```
foreach item ( `ls *.c' )
   cp $item ~/.backup/$item
end
```

- Special statements:

break	causes control to exit the loop
continue	causes control to transfer to the test at the top

while Command

- Syntax:

```
while ( expression )
   shell commands
end
```

- Example:

```
set count = 0
set limit = 7
while ( $count != $limit )
   echo “Hello, ${USER}”
   @ count++
end
```

- **break** and **continue** have same effects as in *foreach*

switch Command

- Syntax:

```
switch ( test-string )
   case pattern1:
      shell commands
      breaksw
   case pattern2:
      shell commands
      breaksw
   default:
      shell commands
      breaksw
end
```

goto Command

- Syntax:

```
goto label
...
other shell commands
...

label:
 shell commands
```

repeat Command

- Syntax:

```
repeat count command
```

- Example:

```
repeat 10 echo “hello”
```

Standard Variables

`$0`	$\Rightarrow$	calling function name
`$N`	$\Rightarrow$	Nth command line argument value
`$argv[N]`	$\Rightarrow$	same as above
`$*`	$\Rightarrow$	all the command line arguments
`$argv`	$\Rightarrow$	same as above
`$#`	$\Rightarrow$	the number of command line arguments
`$<`	$\Rightarrow$	an input line, read from stdin of the shell
`$$`	$\Rightarrow$	process number (PID) of the current process
`$!`	$\Rightarrow$	process number (PID) of the last background process
`$?`	$\Rightarrow$	exit status of the last task

Other Shell Commands

```
source file
shift
shift variable
rehash
```

- Other commands … see Wang, Appendix 7

Example: *ls2*

```
# Usage: ls2
# produces listing that separately lists files and dirs

set dirs = `ls -F | grep '/'`
set files = `ls -F | grep -v '/'`

echo "Directories:"
foreach dir ($dirs)
    echo " " $dir
end

echo "Files:"
foreach file ($files)
   echo " " $file
end
```

Example: *components*

```
#!/usr/bin/csh -f
set test = a/b/c.d
echo "the full string is:" $test
echo "extension (:e) is: " $test:e
echo "head (:h) is: " $test:h
echo "root (:r) is: " $test:r
echo "tail (:t) is: " $test:t

### output:
# the full string is: a/b/c.d
# extension (:e) is:  d
# head (:h) is:  a/b
# root (:r) is:  a/b/c
# tail (:t) is:  c.d
```

Example: *debug*

```
#!/usr/bin/csh -x
while ( $#argv )
   echo $argv[1]
   shift
end
```

```
# while ( 2 )
# echo a
# a
# shift
# end
# while ( 1 )
# echo b
# b
# shift
# end
# while ( 0 )
```

⇒ output of **"debug a b"**

Example: *newcopy*

```
#!/usr/bin/csh -f
### An old exam question:
# Write a csh script "newcopy <dir>" that copies files
# from the directory <dir> to the current directory.
# Only the two most recent files having the name progN.c
# are to be copied, however, where N can be any of 1, 2,
# 3, or 4. The script can be written in 3 to 5 lines:

set currdir = $cwd
cd $argv[1]
set list = (`ls -t -1 prog[1-4].c | head -2 |
             awk '{print $8}'`)
foreach file ($list)
   cp $file $currdir/.
end
```

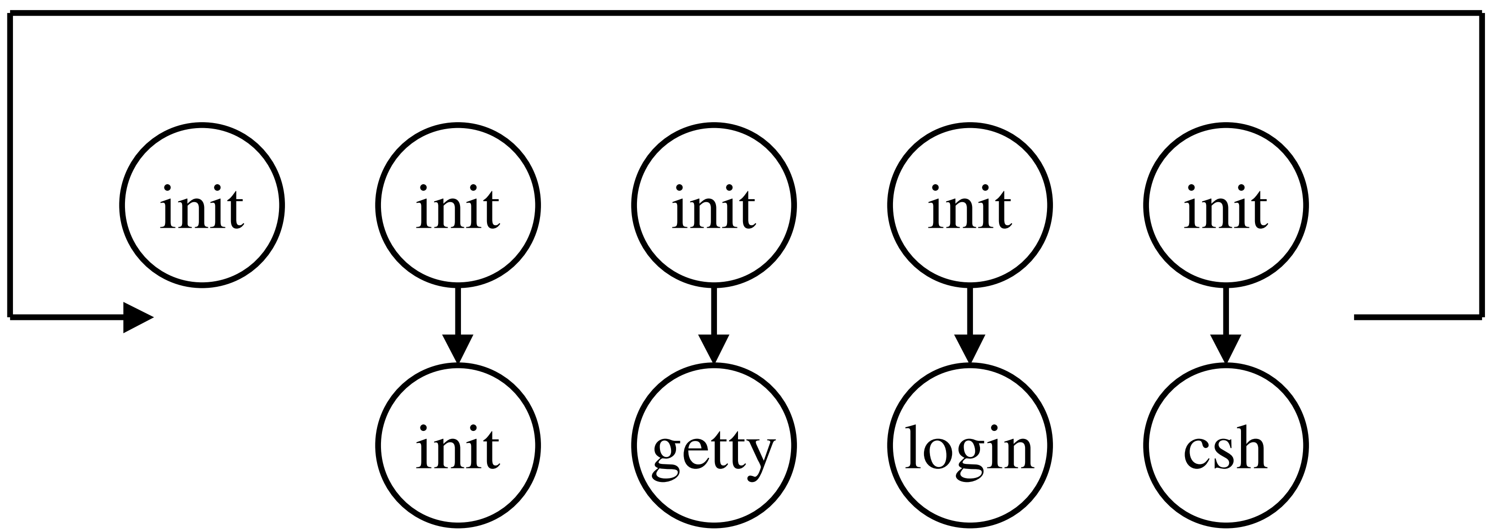
Section III

SYSTEMS PROGRAMMING

Initializing UNIX

- The first UNIX program to be run is called "**/etc/init**"
- It *forks* and then *exec*s one "**/etc/getty**" per terminal
- getty sets up the terminal properly, prompts for a login name, and then *exec*s "**/bin/login**"
- login prompts for a password, encrypts a constant string using the password as the key, and compares the results against the entry in the file "**/etc/passwd**"
- If they match, "**/usr/bin/csh**" (or whatever is specified in the passwd file as being that user's shell) is *exec*'d
- When the user exits from their shell, the process dies. Init finds out about it (*wait* system call), and *forks* another process for that terminal

Initializing UNIX

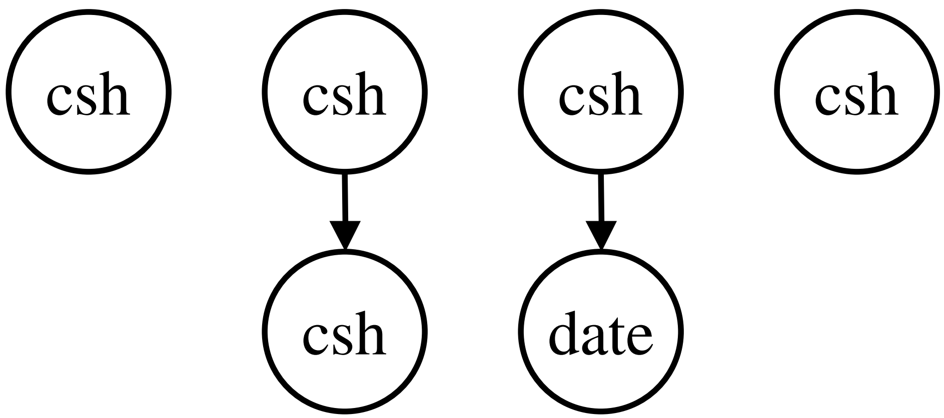


- See “**`top`**”, “**`ps -aux`**”, etc. to see what’s running at any given time
- The only way to create a new process is to duplicate an existing process, therefore the ancestor of ALL processes is **`init`**, with pid=1

How csh runs commands

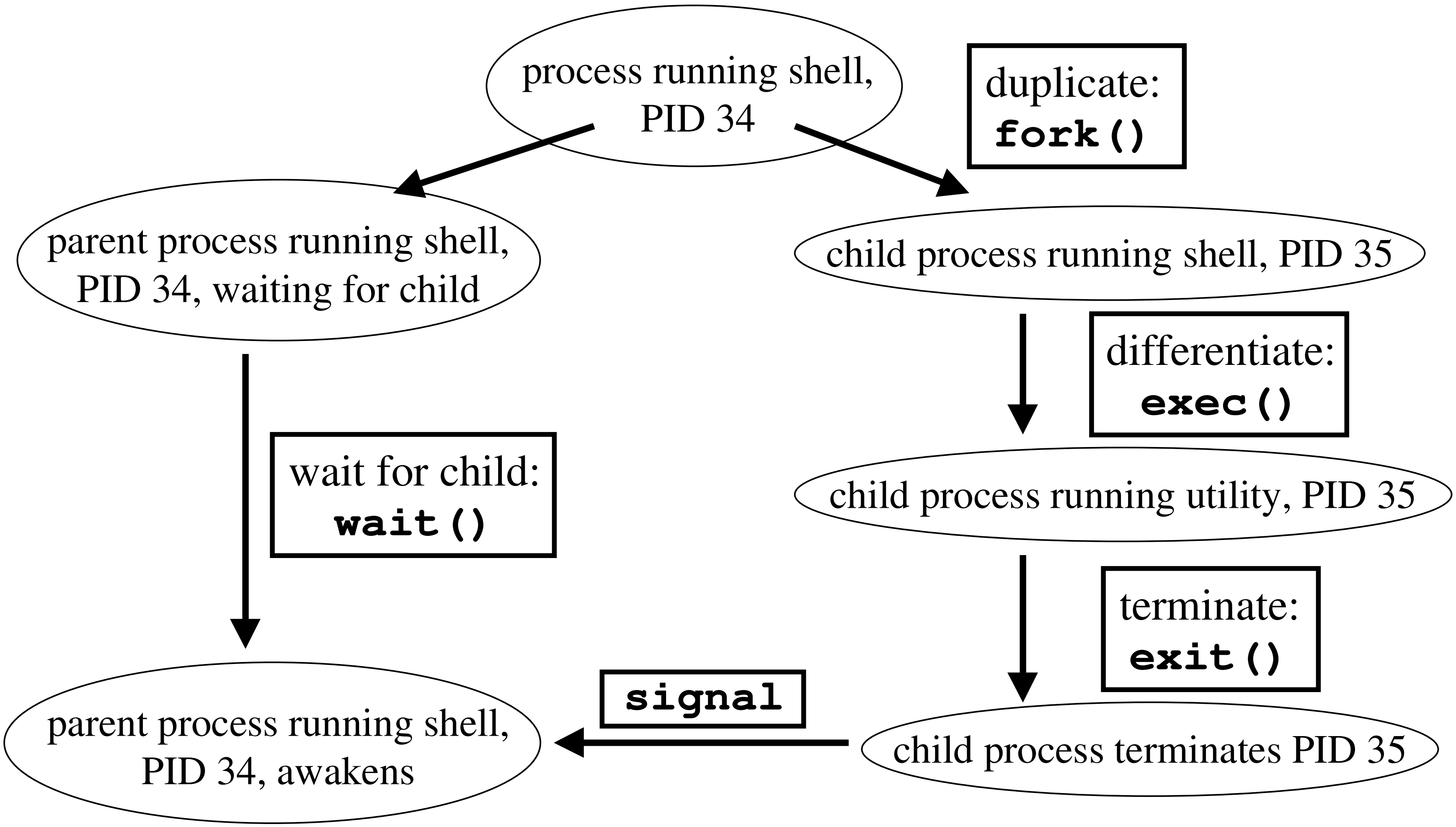
```
> date
Sun May 25 23:11:12 EDT 1997
```

- When a command is typed csh forks and then execs the typed command:

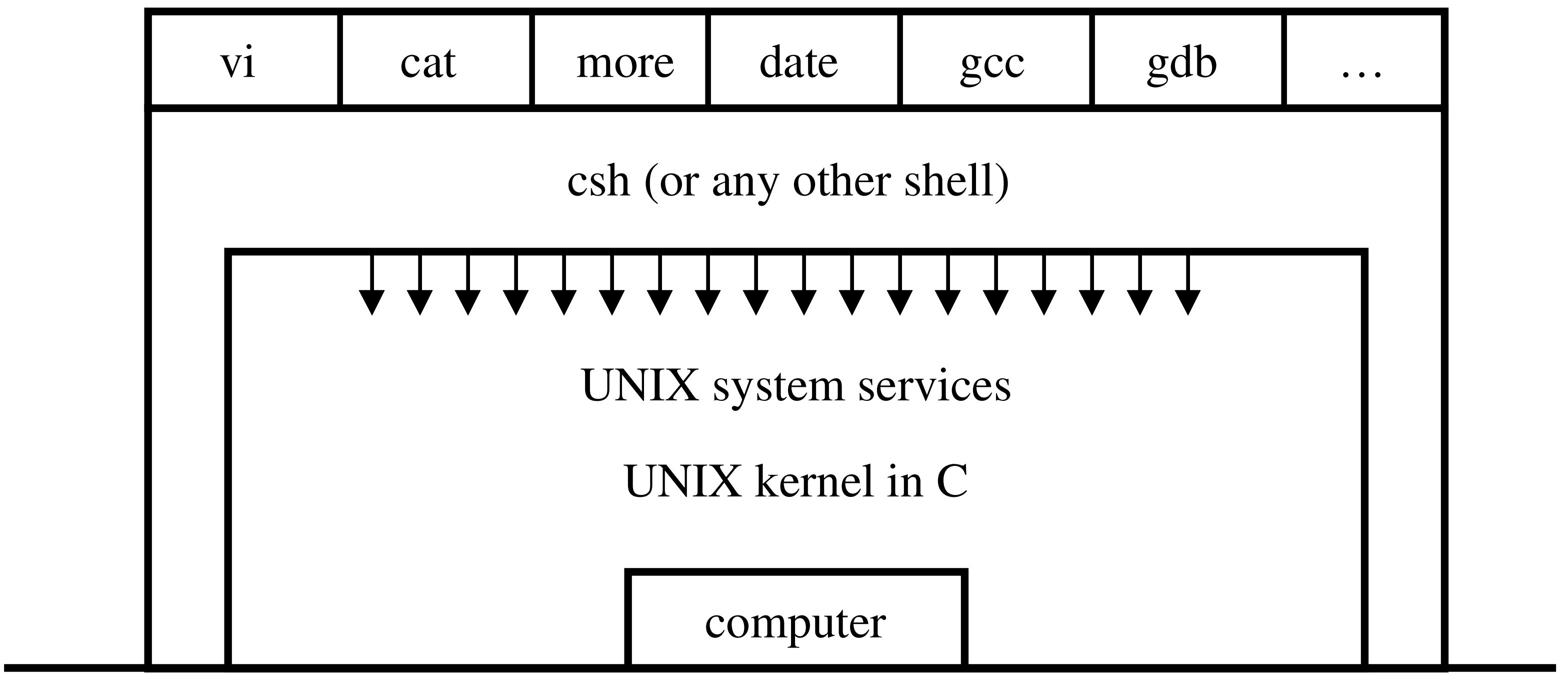


- After the fork and exec, file descriptors 0, 1, and 2 still refer to the standard input, output, and error in the new process
- By UNIX programmer convention, the executed program will use these descriptors appropriately

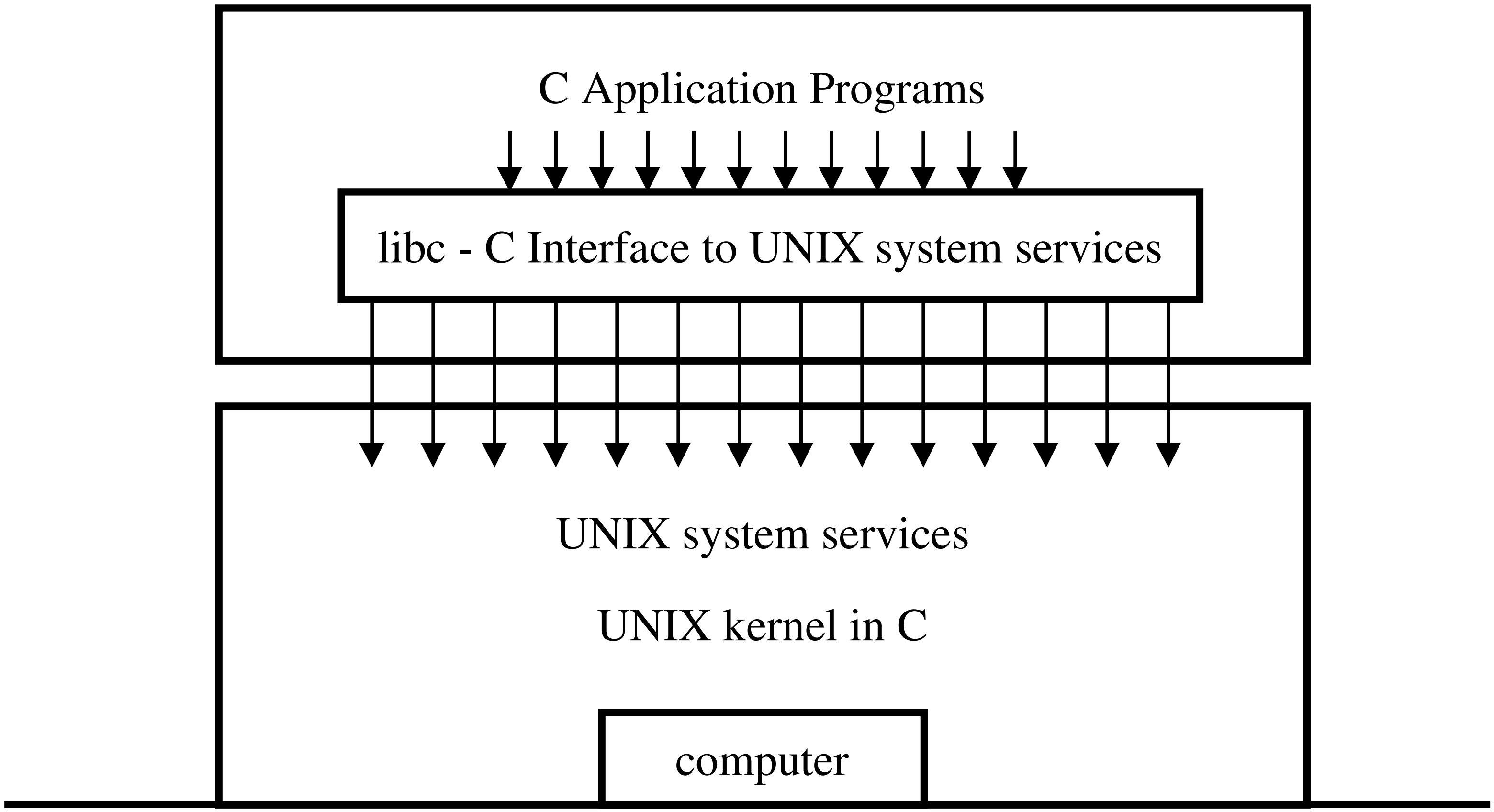
How csh runs (cont.)



Tools and Applications



C and libc


C Application Programs
libc - C Interface to UNIX system services
UNIX system services
UNIX kernel in C
computer

Concurrency

- Most modern developments in computer systems & applications rely on:
  - *communication*: the conveying of info by one entity to another
  - *concurrency*: the sharing of resources in the same time frame

  note: concurrency can exist in a single processor system as well as in a multiprocessor system.

- Managing concurrency is difficult, as execution behaviour (e.g. relative order of execution) is not always reproducible

- More details on this in the last 1/3 or the course

Concurrency Example

Program a:

```
#!/usr/bin/csh -f
@ count = 0
while( $count < 200 )
   @ count++
   echo -n "a"
end
```

Program b:

```
#!/usr/bin/csh -f
@ count = 0
while( $count < 200 )
   @ count++
   echo -n "b"
end
```

- When run *sequentially* (**`a;b`**) output is as expected
- When run *concurrently* (**`a&;b&`**) output is interspersed, and re-running it may produce different output

Race conditions

- A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run
- This is a situation when using forks: if any code after the fork explicitly or implicitly depends on whether or not the parent or child runs first after the fork
- A parent process can call **`wait()`** for a child to terminate (*may block*)
- A child process can wait for the parent to terminate by *polling* it (wasteful)
- Standard solution is to use signals

Example: Race Condition

```
#!/usr/bin/csh -f
set count = 0
while( $count < 50 )
   set sharedData = `cat shareVal`
   @ sharedData++
   echo $sharedData >! shareVal
   @ count++
end
```

- Create two identical copies, “**a**” and “**b**”
- Run as: **./a&; ./b&**

Producer/Consumer Problem

- Simple example:
  `who | wc -l`
- Both the writing process (**`who`**) and the reading process (**`wc`**) of a pipeline execute concurrently
- A pipe is usually implemented as an internal OS buffer
- It is a resource that is concurrently accessed by the reader and by the writer, so it must be managed carefully

Producer/Consumer (cont.)

- *consumer* should be *blocked* when buffer is empty
- *producer* should be *blocked* when buffer is full
- producer and consumer should run independently so far as the buffer capacity and contents permit
- producer and consumer should never both be updating the buffer at the same instant (otherwise, data integrity cannot be guaranteed)
- producer/consumer is a harder problem if there is more than one consumer and/or more than one producer

Intro to Systems Programming

Using argc & argv

- Function main() in C programs declared as
  `int main( int argc, char *argv[])`
- **These parameters are passed by the O/S**
  - `argc`: The number of command-line parameters plus one
  - `argv`: An array of string pointers, each of which points to one parameter

Example: argc/argv

```
#include <stdio.h>
#include <sys/stat.h>
int main( int argc, char *argv[])
{
   if( argc == 2 )
   {
      struct stat buf;
      if( stat( argv[1], &buf ) != -1 )
         printf( "file %s has size %d\n",
                 argv[1], buf.st_size );
   }
   return( 0 );
}
```

System Calls

- System calls:
  - perform a subroutine call directly to the UNIX kernel
- 3 main categories:
  - file management
  - process management
  - error handling

Error Handling

- All system calls return -1 if an error occurs
- **`errno`**:
  - global variable that holds the numeric code of the last system call
- **`perror()`**:
  - a subroutine that describes system call errors
- Every process has **errno** initialized to zero at process creation time
- When a system call error occurs, **`errno`** is set
- See **`/usr/include/sys/errno.h`**
- A successful system call <u>never</u> affects the current value of **`errno`**
- An unsuccessful system call <u>always</u> overwrites the current value of **`errno`**

`perror()`

- Library routine:

  `void perror( char *str )`

- **`perror`** displays **`str`**, then a colon (**`:`**), then an english description of the last system call error, as defined in the header file

  `/usr/include/sys/errno.h`

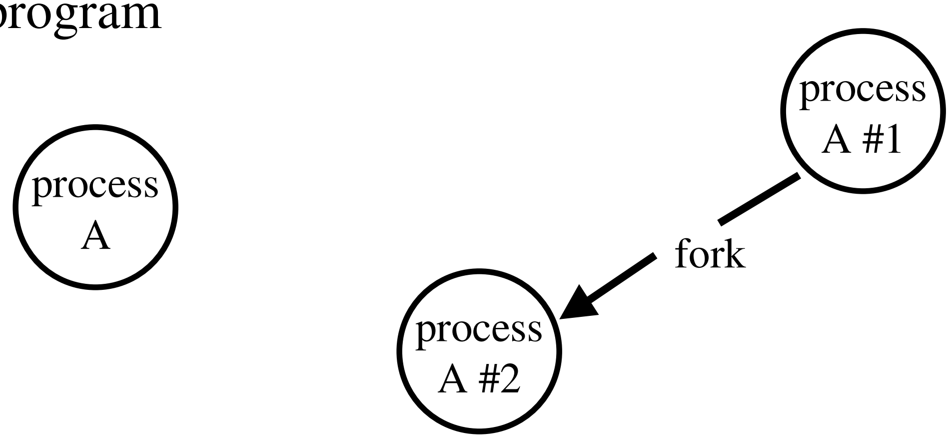
- Protocol:
  - check system calls for a return value of -1
  - call **`perror()`** for an error description during debugging (*see example on next slide*)

`perror()` example

```
#include <stdio.h>
#include <errno.h>

int main( void )
{
   int returnVal;
   printf( "x2 before the execlp, pid=%d\n", getpid()
  );
   returnVal = execlp( "nonexistent_file", (char *)0);
   if( returnVal == -1 )
      perror( "x2 failed" );
   return( 1 );
}
```

Fork

- The *fork* system call is used to create a duplicate of the currently running program



- The duplicate (*child process*) and the original (*parent process*) both proceed from the point of the fork with exactly the same data
- The only difference between the two processes is the *fork* return value, i.e. *(… see next slide)*

Fork example

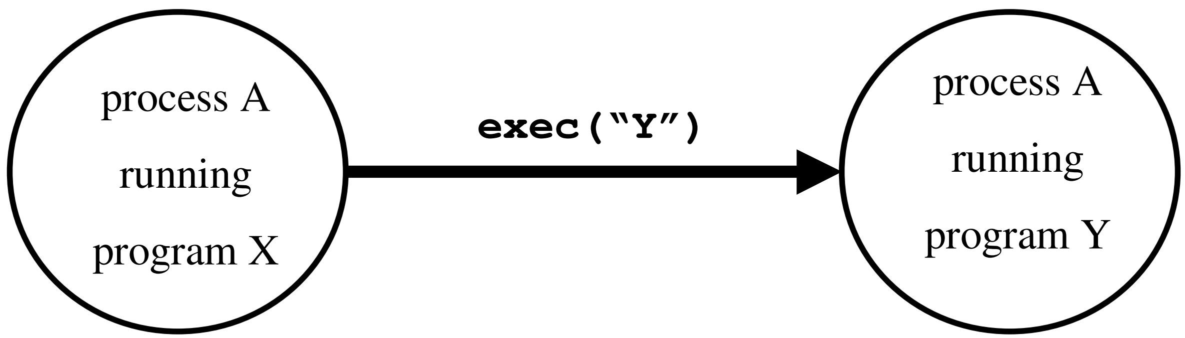
```
int i, pid;
i = 5;
printf( "%d\n", i );
pid = fork();

if ( pid != 0 )
   i = 6; /* only the parent gets to here */
else
   i = 4; /* only the child gets to here */

printf( "%d\n", i );
```

When **`fork()`** Fails

- There is a limit to the max number of processes a user can create
- Once this limit is reached, subsequent calls to `fork()` return -1

Exec

- The *exec* system call replaces the program being run by a process by a different one
- The new program starts executing from its beginning



- Variations on exec: **`execl()`**, **`execv()`**, etc. which will be discussed shortly
- On *success*, exec never returns; on *failure*, exec returns with value -1

Exec example

<u>PROGRAM X</u>

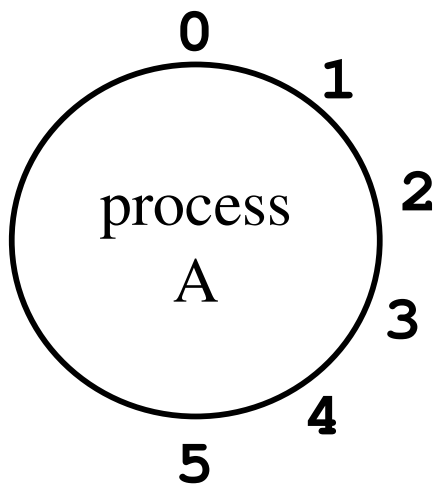
```
int i;
i = 5;
printf( “%d\n”, i );

exec( “Y” );

i = 6;
printf( “%d\n”, i );
```

<u>PROGRAM Y</u>

```
printf( “hello” );
```

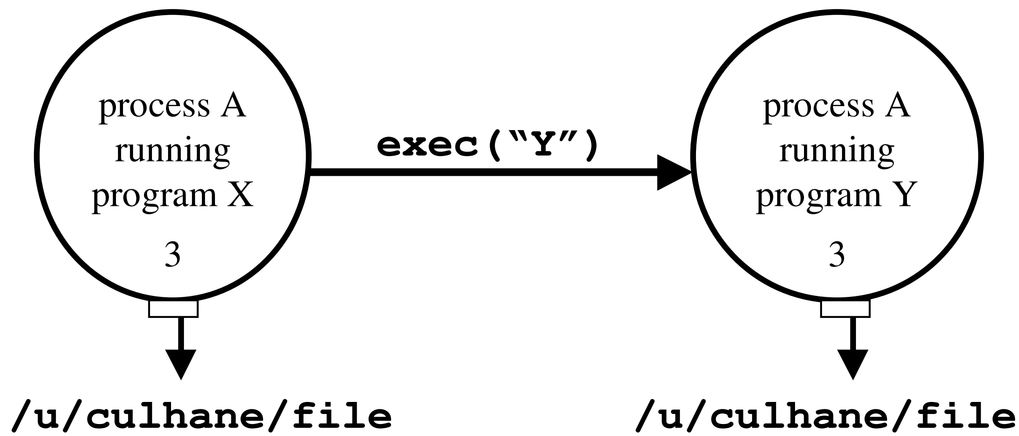
Processes and File Descriptors

- File descriptors (11.1) belong to processes, not programs
- They are a process' link to the outside world



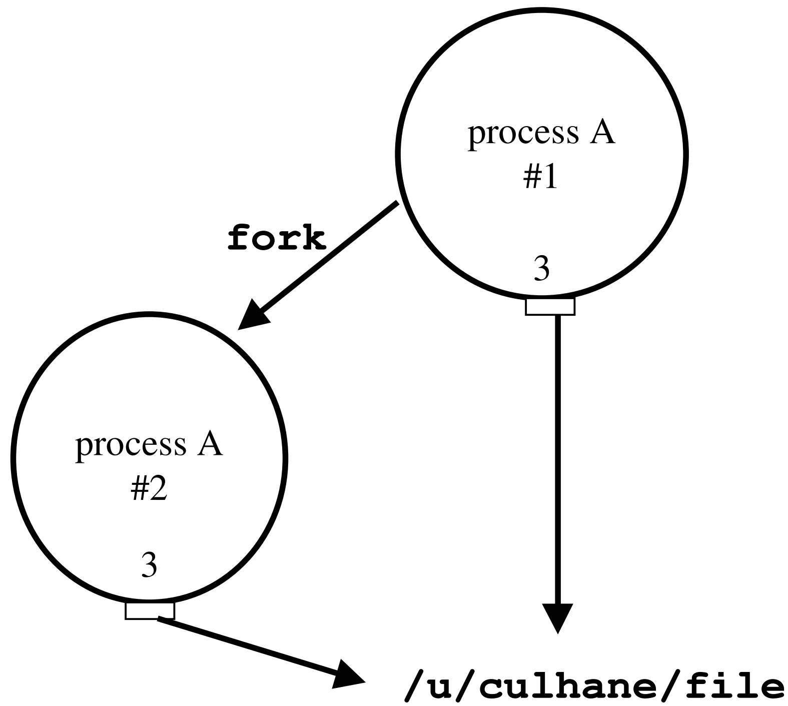
PIDs and FDs across an exec

- File descriptors are maintained across *exec* calls:



PIDs and FDs across a fork

- File descriptors are maintained across *fork* calls:



Fork: PIDs and PPIDs

- System call: **`int fork()`**
- If **`fork()`** succeeds, it returns the child PID to the parent and returns 0 to the child; if it fails, it returns -1 to the parent (no child is created)

- System call: **`int getpid()`**
  `int getppid()`
- **`getpid()`** returns the PID of the current process, and **`getppid()`** returns the PID of the parent process (note: ppid of 1 is 1)
- example (*see next slide …*)

PID/PPID example

```
#include <stdio.h>
int main( void )
{
   int pid;
   printf( "ORIGINAL: PID=%d PPID=%d\n", getpid(), getppid() );
   pid = fork();
   if( pid != 0 )
      printf( "PARENT: PID=%d PPID=%d child=%d\n",
                                        getpid(), getppid(), pid );
   else
      printf( "CHILD:  PID=%d PPID=%d\n", getpid(), getppid() );

   printf( "PID %d terminates.\n\n", getpid() );
   return( 1 );
}
```

Processes Termination

- Orphan process
  - a process whose parent is the init process (pid 1) because its original parent died before it did
- Terminating a process: **`exit()`**
- System call:

```
int exit( int status )
```

- Every normal process is a child of some parent, a terminating process sends its parent a **`SIGCHLD`** signal and waits for its termination code status to be accepted
- The C shell stores the termination code of the last command in the local shell variable **`status`**

`wait()`

- Waiting for a child: system call is

  ```
  int wait( int *status )
  ```

- A process that calls **`wait()`** can:
  - block (if all of its children are still running)
  - return immediately with the termination status of a child (if a child has terminated and is waiting for its termination status to be fetched)
  - return immediately with an error (it it doesn't have any child processes)

- More details in a few weeks, when we cover Chapter 11 of Wang

Zombies

- Zombie process:
  - a process that is “waiting” for its parent to accept its return code
  - a parent accepts a child’s return code by executing **`wait()`**
  - shows up with 'Z' in `ps -a`

- A terminating process may be a (multiple) parent; the kernel ensures all of its children are orphaned and adopted by **`init`**

`wait` and **`waitpid`**

- Recall from a previous slide: **`pid_t wait( int *status )`**
- **`wait()`** can: (a) block; (b) return with status; (c) return with error
- If there is more than one child, **`wait()`** returns on termination of *any* children
- **`waitpid`** can be used to wait for a specific child pid
- **`waitpid`** also has an option to block or not to block

```
pid_t waitpid( pid, &status, option );
    pid    == -1          waits for any child
    option == NOHANG      non-blocking
    option == 0           blocking
waitpid(-1, &status, 0)  equivalent to  wait(&status)
```

example: **wait.c**

```
#include <sys/types.h>
#include <sys/wait.h>
void main( void )
{
   int status;
   if( fork() == 0 ) exit( 7 );      /* normal exit */
   wait( &status ); prExit( status );

   if( fork() == 0 ) abort();        /* generates SIGABRT */
   wait( &status ); prExit( status );

   if( fork() == 0 ) status /= 0;    /* generates SIGFPE */
   wait( &status ); prExit( status );
}
```

prExit.c

```
#include <sys/types.h>
#include <sys/wait.h>
void prExit( int status )
{
   if( WIFEXITED( status ) )
      printf( "normal termination, exit status = %d\n",
                                      WEXITSTATUS( status ));
   else if( WIFSIGNALED( status ) )
      printf( "abnormal termination, signal number = %d\n",
                                      WTERMSIG( status ));
   else if( WIFSTOPPED( status ) )
      printf( "child stopped, signal number = %d\n",
                                      WSTOPSIG( status ));
}
```

exec

- Six versions of exec:

```
execl( char *pathname, char *arg0, ... , (char*) 0 );
execv( char *pathname, char *argv[] );

execle( char *pathname, char *arg0, ..., (char*) 0,
                                 char *envp[] );
execve( char *pathname, char *argv[],
                                 char *envp[] );

execlp( char *filename, char *arg0, ..., (char*) 0 );
execvp( char *filename, char *argv[] );
```

Signals

- Unexpected/unpredictable events:
  - floating point error
  - interval timer expiration (alarm clock)
  - death of a child
  - control-C (termination request)
  - control-Z (suspend request)
- Events are called interrupts
- When the kernel recognizes such an event, it sends the corresponding process a signal
- Normal processes may send other processes a signal, with permission (useful for synchronization)
- Again, we'll cover this in much more detail in a few weeks

System Calls & Pointers

- Consider the following prototype:

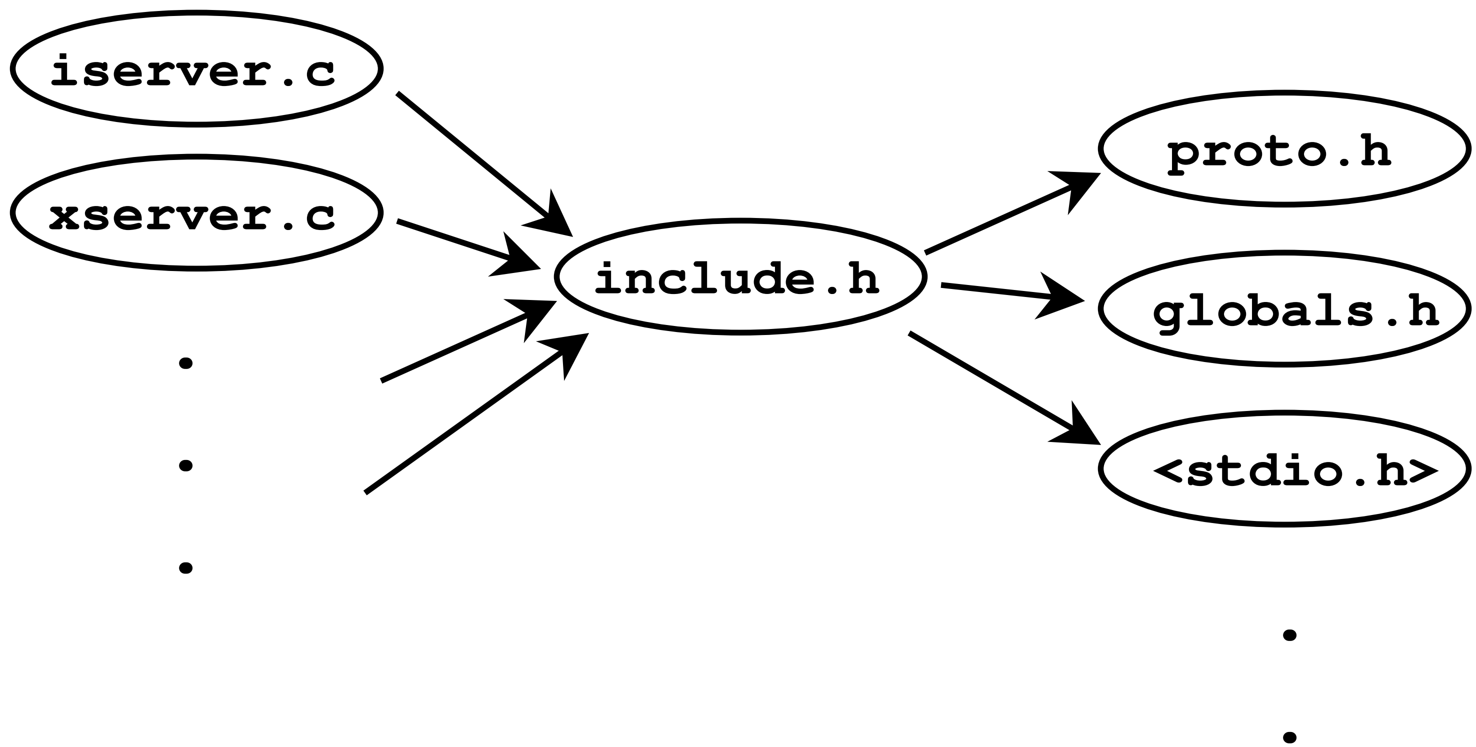
```
int stat(const  char *path ,
             struct stat *buf  );
```

- `buf` is a pointer to a var of type `struct stat`
- many people do this:

```
struct stat *buf ;
int result ;
result = stat("hello.c",buf);
/* seg fault! */
```

Project Management

Dependencies



Makefile

```
OBJS    = iserver.o xserver.o
CC      = gcc
CFLAGS  = -g
.c.o:
        $(CC) $(CFLAGS) -c $<

IServer: $(OBJS)
         $(CC) $(CFLAGS) $(OBJS) -o $@

iserver.o: include.h globals.h proto.h
xserver.o: include.h globals.h proto.h
clean:
        rm -f *.o IServer
```

Makefile Macros

```
<NAME> = <STRING>
  ${<NAME>}
```

- used to simplify makefiles
- example: **CFLAGS = -g -DDEBUG -DANSI**, then can use **${CFLAGS}** in all targets
- can omit **{}** if <NAME> is only one letter
- Special macros:
  - **$@** evaluates to current target
  - **$?** evaluates to a list of prerequisites that are newer than the current target

e.g.
```
libops : interact.o sched.o gen.o
        ar r $@ $?
```

Suffix Rules

- Unix has many "standard" suffixes (**.c .f .o .s .a .so**)
- can specify the same make rule for all files with a given suffix,

```
.SUFFIXES : .o .c .s
  .c.o :
   ${CC} ${CFLAGS} -c $<
  .s.o :
   ${AS} ${ASFLAGS} -o $@ $<
```

- the macro **$<** is just like **$?**, except only for suffix rules
- **$*** evaluates to a filename (without suffix) of the prerequisite

```
cp $< $*.tmp
```

if `main.c` is the prerequisite, then this evaluates to

```
cp main.c main.tmp
```

Suffix Rules (cont'd)

- Only suffixes defined by **`.SUFFIXES`** will be used by make
  - if this line is missing, only "standard suffixes" recognized (.o .c .f .s *etc*.)
  - both suffixes must be present or the suffix rule is ignored
  - order of suffix definition sets "order of precedence", *eg*.
    `.SUFFIXES : .o .c .f .s`
    means that when creating a `.o` file, we first look for a `.c` file of the same name, then a `.f` file, *etc.*
- You can override existing (default) suffix rules
- You can write your own suffix rules (*eg*. **`.prep.o`**)
- To delete default suffix rules, use
  `.SUFFIXES :`

Multiply-defined globals

iserver.c:

```
#include "include.h"

void main( void )
{
   X_ServerPid++;
   PrintPid();
}
```

xserver.c:

```
#include "include.h"
void PrintPid()
{
   printf( "X_ServerPid:%d\n",
            X_ServerPid );
}
```

include.h:

```
#include <stdio.h>
#include "proto.h"
#include "globals.h"
```

proto.h:

```
void PrintPid();
```

globals.h:

```
int X_ServerPid = 14;
```

Two Solutions

for initialized globals:

globals.h:

```
#ifdef _MAIN
   int X_ServerPid = 14;
#else
   extern int X_ServerPid;
#endif
```

iserver.c:

```
#define _MAIN
#include "include.h"
```

for uninitialized globals:

globals.h:

```
#ifdef _MAIN
   #define EXTERN
#else
   #define EXTERN extern
#endif

EXTERN int X_ServerPid;
/* set in Init()*/
```

Libraries & Compilation

Machine Language

- CPU interprets machine language programs:

```
1100101 11111111 11100110 00000000
1010001 00000010 01011101 00000000
1100101 00000000 11111111 00100100
```

- Assembly language instructions bear a one-to-one correspondence with machine language instructions

```
MOVE   FFFFDC00, D0      % b = a * 2
MUL    #2, D0
MOVE   D0, FFFDC04
```

Compilation

- High Level Language (HLL) is a language for expressing algorithms whose meaning is (for the most part) independent of the particular computer system being used
- A *compiler* translates a high-level language into object files (machine language modules).
- A *linker* translates object files into a *machine language* program (an executable)
- Example:
  - create object file "**`fork.o`**" from C program "**`fork.c`**":

```
gcc –c fork.c –o fork.o
```

  - create executable file "**`fork`**" from object file "**`fork.o`**":

```
gcc fork.o –o fork
```

Standard Libraries

- Any system call is *not* part of the C language definition
- Such system calls are defined in *libraries*, identified with the suffix **`.a`**
- Libraries typically contain many **`.o`** object files
- To create your own library archive file:

```
ar crv mylib.a *.o
```

- Disregard "**`ranlib`**" command (no longer needed)
- Look in **`/usr/lib`** and **`/usr/local/lib`** for most system libraries
- Can list all .o files in an archive use "**`ar t /usr/lib/libc.a`**"
- More useful to see all the function names:

```
/usr/ccs/bin/nm /usr/lib/libc.a | grep FUNC
```

Standard Libraries (cont)

- By default, gcc links **`/usr/lib/libc.a`** to all executables
- Typing "**`man 3 intro`**" will give a list of most of the standard library functions
- Any other libraries must be explicitly linked by referring to the absolute pathname of the library, or preferably by using the "**`-l`**" gcc switch:

```
gcc *.o /usr/lib/libm.a -o mathExamples
gcc *.o -lm -o mathExamples
```

- These **`.a`** files are also sometimes referred to as *static libraries*
- Often you will find for each system **`.a`** file a corresponding **`.so`** file, referred to as a *shared object* (not needed for this course)
- Advantage of shared objects: smaller executable files (library functions loaded at run time)

Standard Libraries: Example

```
#include <stdio.h>
/* #include <math.h> */
int main( void )
{
   printf( "Square root of 2 is %f\n", sqrt(2) );
   return( 0 );
}
```

- May get various problems/errors when you compile with:
  1) **`gcc example.c -o example`**
  2) **`gcc example.c -lm -o example`**
  3) **`gcc example.c -lm -o example`** # with math.h included

Advanced Library Functions

String/Character Handling

- All "str" functions require input strings be terminated with a null byte
- Some of the most common ones:

  `strlen`, **`strcpy`**, **`strcmp`**, **`strcat`**
- **`strtok`** used for extracting "tokens" from strings
- **`memcpy`** not just for strings!
- **strncmp** allows limits to be placed on length of strings, other **n** string function
- Some function for testing/converting single characters:

  `isalpha`, **`isdigit`**, **`isspace`**

  `toupper`, **`tolower`**

  `atoi, atol`

Storage Allocation

- Dynamic memory allocation (very important for many C programs):
  malloc, **calloc**, **free**, **realloc**
- An (incomplete) example:

```
#include <stdio.h>
#include <stdlib.h>
struct xx *sp;
sp = (struct xx *) malloc( 5 * sizeof(struct xx) );
if( sp == (struct xx *) NULL )
{
   fprintf( stderr, "out of storage\n" );
   exit( -1 );
}
```

Date and Time Functions

- **`clock_t, clock(), time_t, time()`**
- Most UNIX time functions have evolved from various sources, and are sometimes inconsistent, referring to time as one of:
  - the number of seconds since Jan 1, 1970 (or Jan 1, 1900)
  - the number of clock ticks since Jan 1, 1970 (or Jan 1, 1900)
  - the broken down structure "**`struct tm`**"
    (see **`/usr/include/time.h`**)
  - the broken down structure "**`struct timeval`**"
    (see **`/usr/include/sys/time.h`**)
- Some are intended for time/date, whereas others are intended for measuring elapsed time

Variable Arguments

- An under-used but very powerful feature
- **`printf()`** is an example where the number and types of arguments can differ from invocation to invocation
- **`/usr/include/stdarg.h`** provides definitions of:
  - a special type named **`va_list`**
  - three macros to implement variable arguments:
    - **`va_start`**
    - **`va_end`**
    - **`va_arg`**
- Another useful function is "**`vfprintf`**", as shown in the next slide

Variable Arguments

- A very useful example:

```
#include <stdarg.h>
void Abort( char *fmt, ... )
{
   va_list args;
   va_start( args, fmt );
   fprintf( stderr, "\n\t" );
   vfprintf( stderr, fmt, args );
   fprintf( stderr, "\n\n" );
   va_end( args );
   exit( -1 );
}
```

Environment Interfacing

- Reading environment variables:

```
getenv( "PATH" );
```

- Executing a "**$SHELL**" shell command:

```
fflush( stdout );
system( "ls -atl" );
```

- Can also execute a system call and have its output sent to a pipe instead of stdout: (w*e'll talk more about pipes in chapter 12)*

```
FILE *pipe;
pipe = popen( "ls -atl", "r" );
...
pclose( pipe );
```

UNIX File System

Types of Files

- "regular" files (and links)
  - text, binary executables, images, *etc.*
  - most files are of this type
- directory files
  - a file containing entries for other files: name (file name, not path) plus a pointer into an *i-list* (*i-node* #)
  - an entry in the directory file which specifies an *i-node* is a "hard link"
  - directory tree may have multiple file systems, each with its own *i-list*

- special files
  - I/O devices, such as terminals, printers, tape drives, disk drives
  - two main types:
    - character special files
      - terminal, printer
      - I/O done one character at a time
    - block special files
      - faster devices transfer more than one byte at a time, hence the term "block"
- sockets
  - used for inter-process communications

i-nodes

- Contains file-related info, only one per file
  - Access permissions, size, type, owner UID, owner GID, access/modify/change time (latter changes when permissions change, *etc.*), # of (hard) links
  - Does not store the file's name!
  - Also stores location on disk

Symbolic (Soft) Links

- Hard links do not span file-systems (since hard links contain *i-node* #s, which are only unique within a file system)

  `ln -s file linkname`

- soft links are small files, which contain the (path-) name of the linked file
  - works across file systems
  - has special file type

stat()

- Returns file-name, *i-node* #, and *i-node* contents
- when used on a symbolic link, returns the *i-node* for the linked file, not the link itself!
  - Use **lstat()** in this case
- pre-defined macros for **st_mode**:
  - **S_ISREG(), S_ISDIR(), S_ISCHR(), S_ISBLK(), S_ISFIFO(), S_ISSOCK(), S_ISLNK()**

Files and Directories

- Disk drives divided into partitions
- Each partition contains a filesystem (type **`df`** for a listing of filesystems *mounted* on any given computer)
- Filesystems are mounted onto existing filenames
- Each filesystem has a boot block, a super block, an *ilist* containing ***inodes*** (short for index nodes), directory blocks, and data blocks
- Directories are just lists of inodes (2 files automatically created with mkdir: "`.`" (inode of directory) and "`..`" (inode of parent directory)

Disk Layout

- "Superblock"
  - at start of disk
  - length of *i-node* list
  - list of free *i-nodes*, and its length
  - list of free blocks, and its length
- *i-list*
- blocks
- other details, but beyond this course ...

UNIX Directory Access

`opendir()`

- Used to open a directory for reading:
  `DIR *opendir(const char *filename);`
- returns a null pointer (0x0) on failure: non-existent name, not a directory, **`malloc()`** failure, EACCES
- use **`int closedir(DIR *pDir)`** when done

Reading Directory Entries

`struct dirent *readdir(DIR *pDir)`

- reads directory entries, then moves to next one
- **`struct dirent`** contains:
  - name of file
  - link info (i.e. *i-node* #)
  - other internal info

Positioning Within a Directory

```
long telldir(DIR *dirp);
```

- Where are you currently in directory listing

```
void seekdir(DIR *dirp, long loc);
```

- Move to a specific directory entry

```
void rewinddir(DIR *dirp);
```

- Start at the beginning

UNIX File Access

Streams *vs*. File Descriptors

Streams? File Descriptors?

- UNIX has two main mechanisms for managing file access
  - "streams": high-level, more abstract (and portable)
    - you deal with a pointer to a **`FILE`** structure, which keeps track of info you don't need to know!
  - "file descriptors": each file identified by a small integer, you keep track of integer

Opening and Closing Files

```
FILE *fp;
fp = fopen( fileName, "r" );
fclose( fp );
```

- **`fp`** is of type "**`FILE*`**" (defined in stdio.h)
- **`fopen`** returns a pointer (or NULL if unsuccessful) to the specified *fileName* with the given permissions:
  - "r" read
  - "w" write (create new, or wipe out existing fileName)
  - "a" append (create new, or append to existing fileName)
  - "r+" read and write

Character-by-Character I/O

`fgetc( fp )` # returns next character from files referenced by **fp**

`getc( fp )` # same as fgetc, but implemented as a macro

`getchar()` # same as getc(stdin)

- These return the constant "**`EOF`**" when the end-of-file is reached

`fputc( c, fp )` # outputs character c to file referenced by fp

`putc( c, fp )` # same as fputc, but implemented as a macro

`putchar( c )` # same as putc(c, stdout)

Line-by-Line Input

`fgets( data, size, fp )` # read next line from fp (up to size)

`gets( data )` # read next line from stdin

- **`fgets()`** is preferable to **`gets()`**
- Returns address of **`data`** array (or **`NULL`** if **`EOF`** or other error occurred)
- Example:

```
#define MAX_LENGTH 256
char inputData[MAX_LENGTH];
FILE *fp;
fp = fopen( argv[1], "r" );
fgets( inputData, MAX_LENGTH, fp );
```

Line-by-Line Output

`fputs( data, fp )`	# prints string "data" on stream referenced by fp
`puts( data )`	# same as fputs(data, stdout) except a newline is automatically appended

Formatted Output

```
printf( fmt, args ... )
fprintf( fp, fmt, args ... )
sprintf( string, fmt, args ... )
```

- Examples:

```
fprintf( stderr, "Can't open %s\n", argv[1]
  );
sprintf( fileName, "%s", argv[1] );
```

- **`sprintf`** example above better achieved with "**`strcpy()`**" function
- K&R book or man pages for all the details

Formatted Input

```
scanf( fmt, *args ... )
fscanf( fp, fmt, *args ... )
sscanf( string, fmt, *args ... )
```

- Examples:

```
fscanf( fp, "%s %s", firstName, lastname );
sscanf( argv[1], "%d %d", &int1, &int2 );
```

- Returns number of successful args matched … be careful, scanf should only be used in limited cases where exact format is know in advance
- See K&R book or man pages for all the details

Binary I/O

```
fread( buf, size, numItems, fp )
fwrite( buf, size, numItems, fp )
```

- Examples:

```
fread( readBuf, sizeof( char ), 80, stdin );
fwrite( writeBuf, sizeof(struct utmpx), 1, fp );
```

- Returns number of successful items read or written

- Other functions:

```
rewind(fp); fseek(fp, offset, kind); ftell(fp);
```

`ferror()`

- Error handling is different with stream-I/O functions: most don't return -1 on error
- must check **`ferror()`** periodically

```
int ferror(FILE *stream)
```

  - returns non-zero when a read/write error has occurred

```
int clearerr(FILE *stream)
```

  - clears any error currently set, and also clears EOF indicator

```
int feof(FILE *stream)
```

  - returns non-zero when the end-of-file has been reached

File Buffering

- **`fopen`/`fread`/`fwrite`/`fclose`**, etc. are implemented in terms of low-level *non*-standard i/o functions **`open`/`read`/`write`/`close`**, etc.
- There are 3 types of buffering:
  - fully buffered (or *block buffered*):
    - actual physical i/o takes place only when buffer filled/emptied
  - line buffered:
    - actual i/o takes place when a newline (**`\n`**) is encountered
  - unbuffered:
    - output as soon as possible
- All files are normally block buffered, except *stdout* (line buffered only if it refers to a terminal), and *stderr* (always unbuffered)
- Can use **`fflush()`** to force a buffer to be cleared

File Buffering Control

`int setbuffer(FILE *fp, char *buf, int size)`

- specifies that "**`buf`**" should be used instead of the default system-allocated buffer, and sets the buffer size to "**`size`**"
- if "**`buf`**" is **`NULL`**, i/o will be unbuffered
- used after stream is opened, but before it is read or written

`int setlinebuf( FILE *fp )`

- used to change **`stdout`** or **`stderr`** to line buffered
- can be called anytime

- A stream can be changed from unbuffered or line buffered to block buffered by using **`freopen()`**. A stream can be changed from block buffered or line buffered to unbuffered by using **`freopen()`** followed by **`setbuf()`** with a buffer argument of **`NULL`**.

File Descriptors

- Use functions **`open()`**, **`close()`**, **`read()`**, **`write()`**, **`seek()`**, **`tell()`**
- low-level I/O (e.g. no **`scanf()`** type of functionality, no buffering)
- somewhat implementation dependant (not as portable)
- More on FDs when we need them (when we get to pipes ...)

`fileno()`

`int fileno(FILE *fp)`

- Used to get the file descriptor associated with a the file pointed to by **`fp`**
- handy, since some functionality (*e.g*. file locking) only available via file descriptors

File Locking

- Can "lock" files to prevent unauthorized access
  - "advisory locking": only stops processes/threads that 'ask' permission via the locking functions
- Can use **`fcntl()`** function, see example on web-site.

Signals

Motivation for Signals

- When a program forks into 2 or more processes, rarely do they execute independently of each other
- The processes usually require some form of synchronization, and this is typically handled using signals
- Data usually needs to be passed between processes also, and this is typically handled using pipes and sockets, which we'll discuss in detail in a week or two
- Signals are usually generated by
  - machine interrupts
  - the program itself, other programs, or the user (*e.g.* from the keyboard)

Introduction

- **`<sys/signal.h>`** lists the signal types on cdf. **`signal(5)`** gives a list of some signal types and their default actions
- When a C program receives a signal, control is immediately passed to a function called a signal handler
- The signal handler function can execute some C statements and exit in three different ways:
  - return control to the place in the program which was executing when the signal occurred
  - return control to some other point in the program
  - terminate the program by calling the **`exit`** (or **`_exit)`** function

`signal()`

- A default action is provided for each kind of signal, such as terminate, stop, or ignore
- For nearly all signal types, the default action can be changed using the **`signal()`** function. The exceptions are **`SIGKILL`** and **`SIGSTOP`**
- Usage: **`signal(int sig, void (*disp)(int))`**
- For each process, UNIX maintains a table of actions that should be performed for each kind of signal. The **`signal()`** function changes the table entry for the signal named as the first argument to the value provided as the second argument
- The second argument can be **`SIG_IGN`** (ignore the signal), **`SIG_DFL`** (perform default action), or a pointer to a signal handler function

`signal()` example

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/signal.h>
int i = 0;
void quit( int code ) {
   fprintf( stderr, "\nInterrupt (code=%d, i=%d)\n", code, i );
   exit( 123 );
}
void main( void ) {
  if (signal( SIGINT , quit ) == -1) exit( 1 );
  if (signal( SIGTERM, quit ) == -1) exit( 2 );
  if (signal( SIGQUIT, quit ) == -1) exit( 3 );
  if (signal( SIGKILL, quit ) == -1) printf("Can't touch this!\n");
  for(;;)
     if( i++ % 5000000 == 0 ) putc( '.', stderr );
}
```

Checking the return value

- The data type that **`signal()`** returns is int
- can also use sigset(), returns
  `void (*oldhandler)(int)`
- It is possible for a child process to accept signals that are being ignored by the parent, which more than likely is undesirable
- Thus, another method of installing a new signal handler is:

```
oldhandler = sigset( SIGHUP, SIG_IGN );
if( oldhandler != SIG_IGN )
        sigset( SIGHUP, newhandler );
```

Signalling between processes

- One process can send a signal to another process using the misleadingly named function call

  `kill( int pid, int sig )`

- This call sends the signal "**`sig`**" to the process "**`pid`**"

- Signalling between processes can be used for many purposes:
  - kill errant processes
  - temporarily suspend execution of a process
  - make processes aware of the passage of time
  - synchronize the actions of processes

Timer signals

- Three interval timers are maintained for each process:
  - **`SIGALRM`** (real-time alarm, like a stopwatch)
  - **`SIGVTALRM`** (virtual-time alarm, measuring CPU time)
  - **`SIGPROF`** (used for profilers, which we'll cover later)

- Useful functions to set and get timer info are:
  - **`setitimer(), getitimer()`**
  - **`alarm()`** (simpler version: only sets **`SIGALRM`**)
  - **`pause()`** (suspend until next signal arrives)
  - **`sleep()`** (caused calling process to suspend)
  - **`usleep()`** (like **`sleep()`**, but with finer granularity)

Note: **`sleep()`** and **`usleep()`** are *interruptible* by other signals

Pipes

Inter-Process Communication (IPC)

- Data exchange techniques between processes:
  - message passing: files, pipes, sockets
  - shared-memory model (not the default … but we'll still cover in this, in a few weeks)
- Limitations of <u>files</u> for inter-process data exchange:
  - slow!
- Limitations of <u>pipes</u>:
  - two processes must be running on the same machine
  - two processes communicating must be "related"
- Sockets overcome these limitations (*we'll cover sockets in the next lecture*)

File Descriptors Revisited

- Used by low-level I/O
  - `open()`, `close()`, `read()`, `write()`
- declared as an integer
  `int fd ;`
- Not the same as a "file stream", `FILE *fp`
- streams and file descriptors *are* related (see following slides)

Pipes and File Descriptors

- A fork'd child inherits file descriptors from its parent
- It's possible to alter these using **`fclose()`** and **`fopen()`**:

```
fclose( stdin );
FILE *fp = fopen( "/tmp/junk", "r" );
```

- One could exchange two entries in the fd table by closing and reopening both streams, but there's a more efficient way, using **`dup()`** or **`dup2()`** (…*see next slide*)

`dup()` and **`dup2()`**

```
newFD = dup( oldFD );
if( newFD < 0 ) { perror("dup"); exit(1); }
```

or, to force the newFD to have a specific number:

```
returnCode = dup2( oldFD, newFD );
if(returnCode < 0) { perror("dup2"); exit(1);}
```

- In both cases, **`oldFD`** and **`newFD`** now refer to the same file
- For **`dup2()`**, if **`newFD`** is open, it is first automatically closed
- Note that **`dup()`** and **`dup2()`** refer to fd's and *not* streams
  - A useful system call to convert a stream to a fd is

```
int fileno( FILE *fp );
```

`pipe()`

- The **`pipe()`** system call creates an internal system buffer and two file descriptors: one for reading and one for writing

- With a pipe, typically want the stdout of one process to be connected to the stdin of another process … this is where **`dup2()`** becomes useful (*see next slide for examples*)

- Usage:

```
int fd[2];
pipe( fd );/* fd[0] for reading; fd[1] for writing */
```

`pipe()/dup2()` example

```
/* equivalent to "sort < file1 | uniq" */
int fd[2];
FILE *fp = fopen( "file1", "r" );
dup2( fileno(fp), fileno(stdin) );
fclose( fp );
pipe( fd );
if( fork() == 0 ) {
   dup2( fd[1], fileno(stdout) );
   close( fd[0] );  close( fd[1] );
   execl( "/usr/bin/sort", "sort", (char *) 0 );  exit( 2 );
} else {
   dup2( fd[0], fileno(stdin) );
   close( fd[0] );  close( fd[1] );
   execl( "/usr/bin/uniq", "uniq", (char *) 0 );  exit( 3 );
}
```

`popen()` and **`pclose()`**

- **`popen()`** simplifies the sequence of:
  - generating a pipe
  - forking a child process
  - duplicating file descriptors
  - passing command execution via an exec()

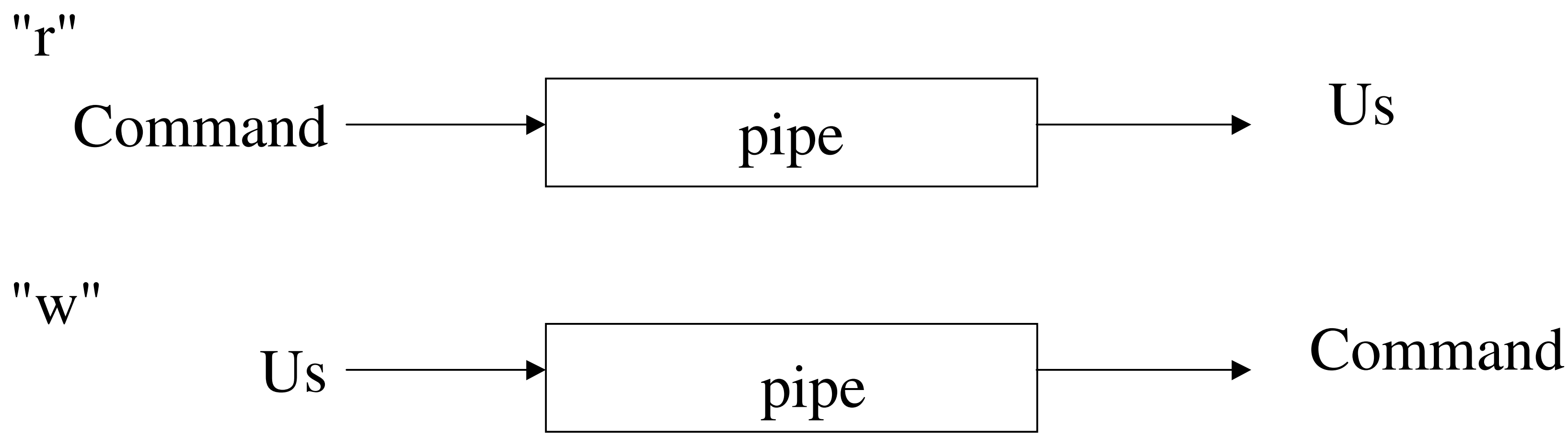
- Usage:

```
FILE *popen( const char *command,
             const char *type );
```

- Example:

```
FILE *pipeFP;
pipeFP = popen( "/usr/bin/ls *.c", "r" );
```

popen()



Sockets

What are sockets?

- Sockets are an extension of pipes, with the advantages that the processes don’t need to be related, or even on the same machine
- A socket is like the end point of a pipe -- in fact, the UNIX kernel implements pipes as a pair of sockets
- Two (or more) sockets must be connected before they can be used to transfer data
- Two main categories of socket types … we’ll talk about both:
  - the <u>UNIX domain</u>: both processes on same machine
  - the <u>INET domain</u>: processes on different machines
- Three main types of sockets: **`SOCK_STREAM`**, **`SOCK_DGRAM`**, and **`SOCK_RAW`** … we’ll only talk about **`SOCK_STREAM`**

Connection-Oriented Paradigm

SERVER		CLIENT
Create a socket **`socket()`**		Create a socket **`socket()`**
Assign a *name* to the socket **`bind()`**		
Establish a queue for connections **`listen()`**		
Extract a connection from the queue **`accept()`**	established	Initiate a connection **`connect()`**
`read()`		**`write()`**
`write()`		**`read()`**

Example: server.c

- *FILE "**server.c**"* … highlights:

```
socket( AF_UNIX, SOCK_STREAM, 0 );
serv_adr.sun_family = AF_UNIX;
strcpy( serv_adr.sun_path, NAME );
bind( orig_sock, &serv_adr, size );
listen( orig_sock, 1 );
accept( orig_sock, &clnt_adr, &clnt_len );

read( new_sock, buf, sizeof(buf) );

close( sd );
unlink( the_file );
```

Example: client.c

- *FILE "**client.c**"* … highlights:

```
socket( AF_UNIX, SOCK_STREAM, 0 );
serv_adr.sun_family = AF_UNIX;
strcpy( serv_adr.sun_path, NAME );

connect(orig_sock, &serv_adr, size );

write( new_sock, buf, sizeof(buf) );

close( sd );
```

- Note: **server.c** and **client.c** need to be linked with the **libsocket.a** library (ie: **gcc -lsocket**)

The INET domain

- The main difference is the **`bind()`** command … in the UNIX domain, the socket name is a *filename*, but in the INET domain, the socket name is a *machine name* and *port number*:

```
static struct sockaddr_in serv_adr;
memset( &serv_adr, 0, sizeof(serv_adr) );
serv_adr.sin_family      = AF_INET;
serv_adr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_adr.sin_port        = htons( 6789 );
```

- Need to open socket with **`AF_INET`** instead of **`AF_UNIX`**
- Also need to include **`<netdb.h>`** and **`<netinet/in.h>`**

The INET domain (cont.)

- The client needs to know the machine name and port of the server

```
struct hostent  *host;
host = gethostbyname( “eddie.cdf” );
```

- Note: need to link with **`libnsl.a`** to resolve **`gethostbyname()`**

- see Wang for:
  - **`server.c, client.c`** UNIX domain example
  - **`iserver.c, iclient.c,`** INET domain example

SIGPIPE

- When you attempt to write/read a socket after the connection has closed (broken), you get SIGPIPE
  - Many socket programs just ignore SIGPIPE

Multiplexed I/O

Motivation

- Consider a process that reads from multiple sources without knowing in advance which source will provide some input first
- Three solutions:
  - alternate non-blocking reads on input sources (wasteful of CPU)
  - fork a process for each input source, and each child can block on one specific input source (can be hard to coordinate/synchronize)
  - use the **`select()`** system call … (*see next slide*)

Background: Number Representations

- All floats/integers stored in binary! (despite what people tell me on exams ☺)
- char: 1 byte, int: 2-4 bytes, long int: 4 bytes, float: 4 bytes, double: 8 bytes
- unsigned int: simple binary
  5 = 0000000000000101
- int: two's complement (invert bits & add 1)
  -5 = 1111111111111011

Comparing Signed & Unsigned

- System calls like `signal()` return pointers (unsigned)
- -1 = 1111111111111111 (signed)
- but for unsigned 1111111111111111 = 65535
- therefore, must cast return value to correct type: `if (signal(...) == (void *)-1)`

Background: Turning Bits On/Off

- Assume unsigned integer representation:
- How to turn on bit `n` of integer `x`?
  `x |= (1 << n)`
- How to turn off bit `n` of integer `x`?
  `x &= ~(1 << n)`
- Used for setting "flags" in status variables
- *E.g.* `open("fred.txt", O_CREAT | O_RDWR)`

`select()`

- Usage:

```
#include <sys/time.h>
#include <sys/types.h>
int select( int nfds,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout );
```

- where the three **`fd_set`** variables are file descriptor *masks*
- **`fd_set`** is defined in **`<sys/select.h>`**, which is included by **`<sys/types.h>`**

Details

- The first argument (**`nfds`**) represents the number of bits in the masks that will be processed. Typically, this is 1 + the value of the highest fd
- The three **`fd_set`** arguments are bit masks … their manipulation is discussed on the next slide
- The last argument specifies the amount of time the select call should wait before completing its action and returning:
  - if **`NULL`**, select will wait (*block*) indefinitely until one of the file descriptors is ready for i/o
  - if **`tv_sec`** and **`tv_usec`** are zero, select will return immediately
  - if timeval members are non-zero, the system will wait the specified time *or* until a file descriptor is ready for i/o
- **`select()`** returns the number or file descriptors ready for i/o

"**FD_**" macros

- Useful macros defined in **<sys/select.h>** to manage the masks:

```
void FD_ZERO( fd_set &fdset );
void FD_SET( int fd, fd_set &fdset );
void FD_CLR( int fd, fd_set &fdset );
int  FD_ISSET( int fd, fd_set &fdset );
```

- Note that each macro is passed the *address* of the file descriptor mask

Example

```
#include <sys/types.h>
fd_set rmask;
int fd;        /* a socket or file descriptor */
FD_ZERO( &rmask );
FD_SET( fd, &rmask ); FD_SET( fileno(stdin), &rmask );
for(;;) {
   select( fd+1, &rmask, NULL, NULL, NULL );
   if( FD_ISSET( fileno(stdin), &rmask )
      /* read from stdin */
   if( FD_ISSET( fd, &rmask ) )
      /* read from descriptor fd */
   FD_SET( fd, &rmask); FD_SET( fileno(stdin), &rmask );
}
```

Shared Memory

Motivation

- Shared memory allows two or more processes to share a given region of memory -- this is the fastest form of IPC because the data does not need to be copied between the client and server

- The only trick in using shared memory is synchronizing access to a given region among multiple processes -- if the server is placing data into a shared memory region, the client shouldn't try to access it until the server is done

- Often, semaphores are used to synchronize shared memory access (… *semaphores will be covered a few lectures from now*)
- not covered in Wang, lookup in Stevens (APUE)

`shmget()`

- **`shmget()`** is used to obtain a shared memory identifier:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget( key_t key, int size, int flag );
```

- **`shmget()`** returns a shared memory ID if OK, -1 on error
- **`key`** is typically the constant "**`IPC_PRIVATE`**", which lets the kernel choose a new key -- keys are non-negative integer identifiers, but unlike fds they are **system-wide**, and their value continually increases to a maximum value, where it then wraps around to zero
- **`size`** is the size of the shared memory segment, in bytes
- **`flag`** can be "**`SHM_R`**", "**`SHM_W`**", or "**`SHM_R|SHM_W`**"

`shmat()`

- Once a shared memory segment has been created, a process attaches it to its address space by calling **`shmat()`**:

```
void *shmat( int shmid, void *addr, int flag );
```

- **`shmat()`** returns pointer to shared memory segment if OK, -1 on error
- The recommended technique is to set **`addr`** and **`flag`** to zero, i.e.:

```
char *buf = (char *) shmat( shmid, 0, 0 );
```

- The UNIX commands "**`ipcs`**" and "**`ipcrm`**" are used to list and remove shared memory segments on the current machine
- The default action is for a shared memory segments to remain in the system even after the process dies -- a better technique is to use **`shmctl()`** to set up a shared memory segment to remove itself once the process dies (… *see next slide*)

`shmctl()`

- **`shmctl()`** performs various shared memory operations:

```
int shmctl( int shmid, int cmd,
            struct shmid_ds *buf );
```

- **`cmd`** can be one of **`IPC_STAT`**, **`IPC_SET`**, or **`IPC_RMID`**:
  - **`IPC_STAT`** fills the **`buf`** data structure (see **`<sys/shm.h>`**)
  - **`IPC_SET`** can change the *uid*, *gid*, and *mode* of the **`shmid`**
  - **`IPC_RMID`** sets up the shared memory segment to be removed from the system once the last process using the segment terminates or detached from it — a process detaches a shared memory segment using **`shmdt( void *addr )`**, which is similar to **`free()`**
- **`shmctl()`** returns 0 if OK, -1 on error

Shared Memory Example

```
char *ShareMalloc( int size )
{
   int  shmId;
   char *returnPtr;

   if( (shmId=shmget( IPC_PRIVATE, size, (SHM_R|SHM_W) )) < 0 )
      Abort( "Failure on shmget {size is %d}\n", size );

   if( (returnPtr=(char*) shmat( shmId, 0, 0 )) == (void*) -1 )
      Abort( "Failure on Shared Mem (shmat)" );

   shmctl( shmId, IPC_RMID, (struct shmid_ds *) NULL );
   return( returnPtr );
}
```

`mmap()`

- An alternative to shared memory is memory mapped i/o, which maps a file on disk into a buffer in memory, so that when bytes are fetched from the buffer the corresponding bytes of the file are read
- One advantage is that the contents of files are non-volatile
- Usage:

```
caddr_t mmap( caddr_t addr, size_t len, int
        prot, int flag, int filedes, off_t off );
```

  - **`addr`** and **`off`** should be set to zero,
  - **`len`** is the number of bytes to allocate
  - **`prot`** is the file protection, typically (**`PROT_READ|PROT_WRITE`**)
  - **`flag`** should be set to **`MAP_SHARED`** to emulate shared memory
  - **`filedes`** is a file descriptor that should be opened previously

Memory Mapped I/O Example

```
char *ShareMalloc( int size )
{
   int  fd;
   char *returnPtr;
   if( (fd = open( "/tmp/mmap", O_CREAT | O_RDWR, 0666 )) < 0 )
      Abort( "Failure on open" );
   if( lseek( fd, size-1, SEEK_SET ) == -1 )
      Abort( "Failure on lseek" );
   if( write( fd, "", 1 ) != 1 )
      Abort( "Failure on write" );
   if( (returnPtr = (char *) mmap(0, size, PROT_READ|PROT_WRITE,
         MAP_SHARED, fd, 0 )) == (caddr_t) -1 )
      Abort( "Failure on mmap" );
   return( returnPtr );
}
```

Semaphores

Motivation

- Programs that manage shared resources must execute portions of code called <u>critical sections</u> in a mutually exclusive manner. A common method of protecting critical sections is to use <u>semaphores</u>

- Code that modifies shared data usually has the following parts:

Entry Section: The code that requests permission to modify the shared data.

Critical Section: The code that modifies the shared variable.

Exit Section: The code that releases access to the shared data.

Remainder Section: The remaining code.

The Critical Section Problem

- The critical section problem refers to the problem of executing critical sections in a fair, symmetric manner. Solutions to the critical section problem must satisfy each of the following:

  Mutual Exclusion: At most one process is in its critical section at any time.

  Progress: If no process is executing its critical section, a process that wishes to enter can get in.

  Bounded Waiting: No process is postponed indefinitely.

- An <u>atomic operation</u> is an operation that, once started, completes in a logical indivisible way. Most solutions to the critical section problem rely on the existence of certain atomic operations

Semaphores

- A semaphore is an integer variable with two atomic operations: wait and signal. Other names for wait are *down*, *P*, and *lock*. Other names for signal are *up*, *V*, *unlock*, and *post*.
- A process that executes a *wait* on a semaphore variable **S** cannot proceed until the value of **S** is positive. It then decrements the value of **S**. The *signal* operation increments the value of the semaphore variable.
- Some (flawed) pseudocode:

```
void wait( int *s )
{
    while( *s <= 0 ) ;
    (*s)--;
}
```

```
void signal( int *s )
{
    (*s)++;
}
```

Semaphores (cont.)

- Three problems with the previous slide's **`wait()`** and **`signal()`**:
  - busy waiting is inefficient
  - doesn't guarantee bounded waiting
  - "**`++`**" and "**`--`**" operations aren't necessarily atomic!

- Solution: use system calls **`semget()`** and **`semop()`** (… *see next slide*)

- The following pseudocode protects a critical section:

```
wait( &s );
/* critical section */
signal( &s );
/* remainder section */
```

- What happens if **S** is initially 0? What happens if **S** is initially 8?

`semget()`

- Usage:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
int semget( key_t key, int nsems, int semflg );
```

- Creates a semaphore set and initializes each element to zero
- Example:

```
int semID = semget( IPC_PRIVATE, 1,
                    S_IRUSR | S_IWUSR );
```

- Like shared memory, **`icps`** and **`ipcrm`** can list and remove semaphores

semop()

- Usage: **`int semop( int semid, struct sembuf *sops,`**
  `int nsops );`
- Increment, decrement, or test semaphores elements for a zero value.
- From **`<sys/sem.h>`**:

  `sops->sem_num, sops->sem_op, sops->sem_flg;`
- If **`sem_op`** is positive, **`semop()`** adds value to semaphore element and awakens processes waiting for the element to increase
- if **`sem_op`** is negative, **`semop()`** adds the value to the semaphore element and if < 0, **`semop()`** sets to 0 and blocks until it increases
- if **`sem_op`** is zero and the semaphore element value is not zero, **`semop()`** blocks the calling process until the value becomes zero
- if **`semop()`** is <u>interrupted by a signal</u>, it returns -1 with **`errno`** = **`EINTR`**

Example

```
struct sembuf semWait[1]   = { 0, -1, 0 },
              semSignal[1] = { 0,  1, 0 };
int semID;

semop( semID, semSignal, 1 ); /* init to 1 */

while( (semop( semID, semWait, 1 ) == -1) &&
       (errno == EINTR) );

{ /* critical section */ }

while( (semop( semID, semSignal, 1 ) == -1) &&
       (errno == EINTR) );
```

Cleaning Up

- Semaphores, like shared memory, are a system-wide resource
- you must delete semaphores when done with them
  - `semctl()` and `IPC_RMID` (like shared memory)
  - `ipcs` and `ipcrm`

Posix Threads

Thread Concepts

- Threads are "lightweight processes"
  - 10 to 100 times faster than `fork()`
- Threads share:
  - process instructions, most data, file descriptors, signal handlers/dispositions, current working directory, user/group Ids
- Each thread has its own:
  - thread ID, set of registers (incl. Program counter and stack pointer), stack (local vars, return addresses), `errno`, signal mask, priority
- Posix threads will (we think) be the new UNIX thread standard

Creating a PThread

```
#include <pthread.h>
int pthread_create(pthread_t *tid, pthread_attr_t *attr,
                   void *(*func)(void *), void *arg)
```

- `tid` is unique within a process, returned by function
- `attr`
  - sets priority, initial stack size, *daemon* status
  - can specify as `NULL`
- `func`
  - function to call to start thread
  - accepts one **`void *`** argument, returns one **`void *`**
- `arg` is the argument to pass to **`func`**

Creating a Pthread [cont'd]

- **`pthread_create()`** returns 0 if successful, a +ve error code if not
- **does not** set `errno`, but returns compatible codes
- can use **`strerror()`** to print error messages

Thread Termination

```
#include <pthread.h>
int pthread_join(pthread_t tid, void **status)
```

- `tid`
  - the thread ID of the thread to wait for
  - cannot wait for any thread (*cf.* `wait()`)

Thread Termination [cont'd]

- `status`, if not **NULL**, returns the **void *** returned by the thread when it terminates
- a thread can terminate by
  - returning from **`func()`**
  - the **`main()`** function exiting
  - **`pthread_exit()`**

```
#include <pthread.h>
void pthread_exit(void *status);
```

- a second way to exit, returns status explicitly
- `status` must not point to an object **local** to thread, as these disappear when the thread terminates

"Detaching" Threads

```
#include <pthread.h>
int pthread_detach(pthread_t tid);
```

- threads are either joinable or detachable
- if a thread is detached, its termination cannot be tracked with **`pthread_join()`** - it becomes a *daemon* thread

```
#include <pthread.h>
pthread_t pthread_self(void);
```

- returns the thread ID of the thread which calls it
- often see **`pthread_detach(pthread_self());`**

Passing Arguments to Threads

```
pthread_t thread_ID;
int fd, result ;

result = pthread_create(&thread_ID,
(pthread_attr_t *)NULL, myThreadFcn, (void *)&fd);
if (result != 0)
  printf("Error: %s\n", strerror(result));
```

- we can pass any variable (including a structure or array) to our thread function; assumes thread function knows what type it is; above example <u>bad</u> if main thread alters **`fd`** later … or if **`void *`** not big enough to hold type **`int`**

Improved Argument Passing

- Use **`malloc()`** to create memory for the var to pass
- init var's value
- pass point to new memory via **`pthread_create()`**
- thread fcn releases memory when done

Argument Passing Example

```
typedef struct myArg {
  int  fd      ;
  char name[25];
} MyArg ;

int result ;
pthread_t threadID ;
MyArg *p ;
p = (MyArg *)malloc(sizeof(MyArg));
p->fd = fd ; /* assume fd is defined */
strcpy(p->name,"CSC209");
result = pthread_create(&threadID, NULL, myThreadFcn,
                         (void *)p);
```

Argument Passing Example [cont'd]

```
void *myThreadFcn(void *p)
{
  MyArg *theArg = (MyArg *)p ;

  write(theArg->fd,theArg->name,
        strlen(theArg->name));
  close(theArg->fd);

  free(theArg);

  return NULL ;
}
```

Thread-Safe Functions

- Not all functions can be called from threads (*e.g.* **`strtok()`**)
  - many use global/static variables
  - new versions of UNIX have *thread-safe* replacements, like **`strtok_r()`**
- **Safe:**
  - **`ctime_r()`, `gmtime_r()`, `localtime_r()`, `rand_r()`, `strtok_r()`**
- **Not Safe:**
  - **`ctime()`, `gmtime()`, `localtime()`, `rand()`, `strtok()`, `gethostXXX()`, `inet_toa()`**
- could use semaphores to protect access

PThread Mutexes (Semaphores)

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *name,
               const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *name);
int pthread_mutex_lock(pthread_mutex_t *name);
int pthread_mutex_trylock(pthread_mutex_t *name);
int pthread_mutex_unlock(pthread_mutex_t *name);
```

- pthread semaphores are easier to use than **`semget()`** and **`semop()`**
- only the thread that locks a mutex can unlock it
- mutexes often declared as globals

PThread Mutexes [cont'd]

```
pthread_mutex_t myMutex ;
int status ;

status = pthread_mutex_init(&myMutex, NULL) ;
if (status != 0)
  printf("Error: %s\n", strerror(status));
pthread_mutex_lock(&myMutex);
/* critical section here */
pthread_mutex_unlock(&myMutex);
status = pthread_mutex_destroy(&myMutex);
if (status != 0)
 printf("Error: %s\n", strerror(status));
```

Mutex Initialization

- The previous slide demonstrated "dynamic initialization"; can also use "static initialization"

```
pthread_mutex_t myMutex =
  PTHREAD_MUTEX_INITIALIZER ;
```

- you needn't explicitly destroy mutexes which are statically initialized
- every mutex <u>must</u> be initialized (only once!) before use ...

Condition Variables

- Used to signal that shared data has changed
  - *E.g.* in producer/consumer, the producer can notify the consumer that the buffer is full, or the consumer can notify the producer it is empty
  - when buffer is full, producer can do no more until it is empty
- Used in conjunction with a mutex variable

Declaring Condition Variables

Static Initialization

```
pthread_cond_t myCond =
  PTHREAD_COND_INITIALIZER ;
```

Dynamic Initialization (non-default attributes)

```
pthread_cond_t myCond ;
pthread_cond_init(&myCond,
             (pthread_condattr_t *)NULL);
```

- Use Dymanic Init when:
  - using non-default attributes
  - **malloc()**-ing structure

Destroying Condition Variables

- Must be initialized before use
- Never make a copy! Pointers to original OK …

Destroying

- When done with it, destroy it (dynamic init only)

`pthread_cond_destroy(&myCond);`

- Make sure no thread attempts to use it after it is destroyed

Using Condition Variables

- Must be associated with a mutex variable
- Use the following:

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_timedwait(
                 pthread_cond_t *cond   ,
                 pthread_mutex_t *mutex ,
                 struct timespec *expires);
```

- don't wait unnecessarily, you may never wake-up!

- The reason you wait is called a "predicate"
- you make be awoken spuriously, so re-check the reason you are waiting (**very rare!**)
- The timed wait returns status ETIMEDOUT if condition not true in time (absolute time)

```
int pthread_cond_signal(
            pthread_cond_t *cond);
int pthread_cond_broadcast(
            pthread_cond_t *cond)
```

- "signal" wakes up a single waiting thread, "broadcast" wakes up all waiting threads
- *E.g.* you may have several worker threads waiting …

Waiting On A Condition

- Associated mutex must be locked before call to **`pthread_cond_wait()`**
- **`pthread_cond_wait()`** will unlock the mutex when it blocks, and relock it before it unblocks

Signalling a Condition

- Associated mutex need not be locked before call to **`pthread_cond_signal()`**
- if in doubt, use "broadcast"

Example

```
int status
status = pthread_mutex_lock(&myMutex);
if (status != 0)
  { perror("Mutex Error!"); pthread_exit(NULL); }
if (buffer == EMPTY){ // test predicate
  status = pthread_cond_wait(&myCond, &myMutex);
  if (status == 0)
    if (buffer == EMPTY) // re-test predicate
      { /* fill buffer ... */ }
if (status != 0) { ... }
}
else perror("Error on myCond!");
status = pthread_mutex_unlock(&myMutex);
```

`pthread_once()`

- Allows you to init data shared by threads, guarantees it is done only once
- initializer function returns void, and takes no parameters
  - all initialized data must therefore be globally accessible

Example

```
pthread_once_t key_once =
                PTHREAD_ONCE_INIT ;

void onceRoutine(void)
{ /* do once only init here */ }

status = pthread_once(&key_once,
                          onceRoutine);
```

Thread-Specific Data

- In a non-threaded program, a function can create private data using "static"
- In a threaded program such data is visible to all threads that use the function
- How to create truly private data? In this case all threads share the variable name, but each has its own value stored in it …
- could use a linked list, where each node stores the data according to a pthread ID, or ...

Thread-Specific Data [cont'd]

- … can require the caller of the function to pass a pointer to private data (if you can specify whatever function interface you wish), or …
- … use the built-in private data mechanism provided by Pthreads! (best solution)

TSD Functions

```
pthread_key_t key;

int pthread_key_create(
    pthread_key_t *key,
    void (*destructor)(void *));

int pthread_key_delete(pthread_key_t key);

int pthread_setspecific(
    pthread_key_t key, const void *value);


void *pthread_getspecific(
      pthread_key_t key);
```

TSD Comments

- Call **`pthread_key_create()`** once, and once only!
- It's difficult to use **`pthread_key_delete()`** properly, and no harm is done by not destroying keys (in almost all cases)
- When a thread terminates, the destructor function is called, and passed the **`void *`** associated with that thread's specific data: can call **`free()`** to cleanup, *etc*.
- could use TSD to fix a non-threadsafe library

Exiting Main Thread

- It ***is*** possible to terminate the main thread without killing the process:
  - simply call **`pthread_exit(void *p)`** from main thread
  - process continues to run until all other threads terminate

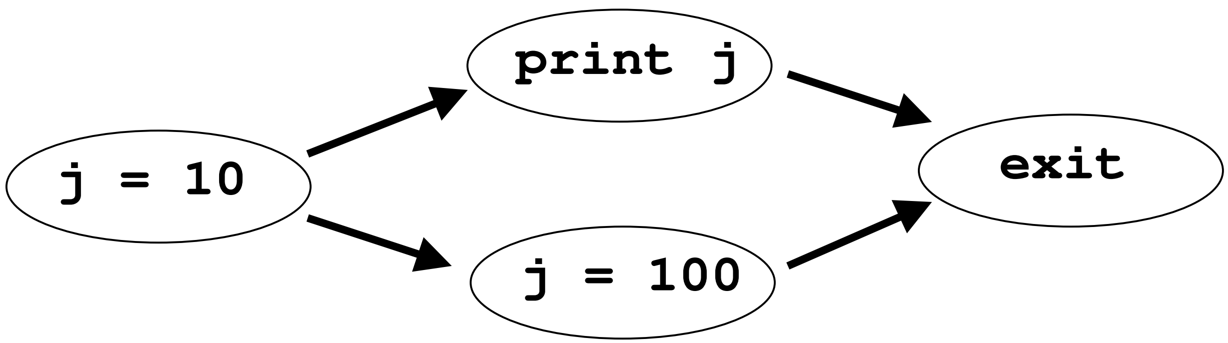
Calling `fork()` from a Thread

- Should be used sparingly, *i.e.* only when about to call `exec()`
- only replicates calling thread (see "man fork" for thread details)
- if the forked child uses mutexes, must be careful about deadlock

Concurrency Concepts

Non-determinism

- A process is deterministic when it always produces the same result when presented with the same data; otherwise a process is called non-deterministic



- Evaluation proceeds non-deterministically in one of two ways, producing an output of 10 or 100
- Race conditions lead to non-determinism, and are generally undesirable

Deadlocks

- A concurrent program is in deadlock if all processes are waiting for some event that will never occur

- Typical deadlock pattern:

  Process 1 is holding resource X, waiting for Y

  Process 2 is holding resource Y, waiting for X

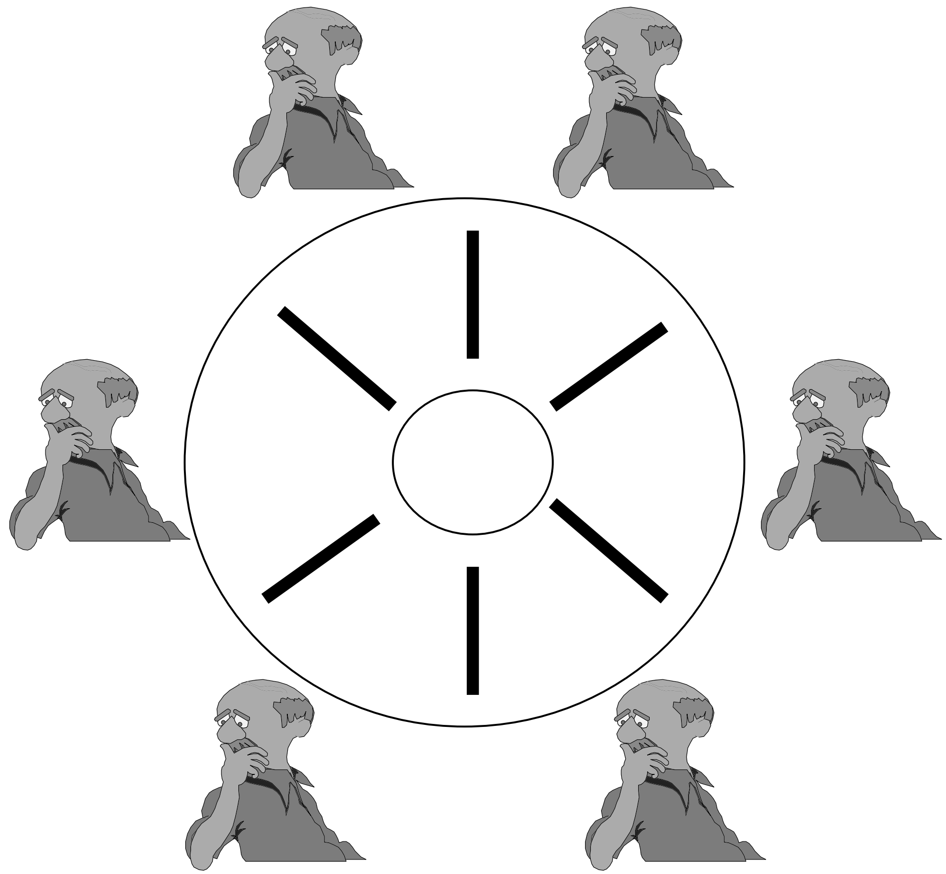
Process 1 will not get Y until Process 2 releases it

Process 2 will not release Y until it gets X,

which Process 1 is holding, waiting for …

Dining Philosophers

- **N** philosophers are seated in a circle, one chopstick between each adjacent pair
- Each philosopher needs two chopsticks to eat, a left chopstick and a right chopstick
- A typical philosopher process alternates between eating and thinking (*see next slide*)



Philosopher Process

```
loop
    <get one chopstick>
    <get other chopstick>

        <eat>

    <release one chopstick>
    <release other chopstick>

        <think>

endloop
```

Deadlock Example

- For **N**=2, call philosophers P1 and P2, and chopsticks C1 and C2
- Deadlocking sequence:

  P1 requests; gets C1

  P2 requests; gets C2

  P1 requests; WAITS for C2

  P2 requests; WAITS for C1

  ** DEADLOCK **

- Can avoid deadlock if the philosopher processes request both chopsticks at once, and then they get both or wait until both are available

Comments on Deadlock

- In practice, deadlocks can arise when waiting for some reusable resources. For example, an operating system may be handling several executing jobs, none of which has enough room to finish (and free up memory for the others)
- Operating systems may detect/avoid deadlocks by:
  - checking continuously on requests for resources
  - refusing to allocate resources if allocation would lead to a deadlock
  - terminating a process that is responsible for deadlock
- One can have a process that sits and watches, and can break a deadlock if necessary. This process may be invoked:
  - on a timed interrupt basis
  - when a process wants to queue for a resource
  - when deadlock is suspected (i.e.: CPU utilization has dropped to 0)

Indefinite Postponement

- Indefinite postponement occurs when a process is blocked waiting for an even that can, but will not occur in some future execution sequence
- This may arise because other processes are “ganging up” on a process to “starve” it
- During indefinite postponement, the overall system does not grind to a halt, but treats some of its processes unfairly
- Indefinite postponement can be avoided by having priority queues which serve concurrent processes on a first-come, first-served basis
- UNIX *semaphores* do this, using a FIFO (first-in, first-out) queue for all requests

Dekker's Algorithm

```
/* other, me are threadID's with values 0, 1 */
int turn ;
int need[2] = { FALSE, FALSE };

void wait()
{
  need(me) = TRUE ;  turn = other ;
  while (need[other] && (turn != me));
}

void signal()
{
  need(me) = FALSE ;
}
```

Self Study Topics

C: Primer and Advanced Topics

Style

- Basics:
  - comments
  - white space
  - modularity
- Naming conventions:
  - variableNames ("*Hungarian Notation*": m_pMyInt, bDone)
  - FunctionNames
  - tTypeDefinitions
  - CONSTANTS

Brace Styles

- K&R:

```
if (total > 0) {
    printf( “Pay up!”
  );
  total = 0;
} else {
  printf( “Goodbye”
  );
}
```

- non-K&R:

```
if (total > 0)
{
  printf("Pay up!");
  total = 0 ;
}
else
{
   printf("Goodbye");
}
```

Variables and Storage

- Syntax:

```
<type> <varName> [= initValue];
```

- Types (incomplete list):
  - char
  - short
  - int
  - long
  - float
  - double
  - all can be: signed (default) or unsigned

Operators

- Arithmetic Operators:

  `*, /, +, -, %`
- Relational Operators:

  `<, <=, >, >=, ==, !=`
- Assignment Operators:

  `=, +=, -=, *=, /=, ++, --`
  - don’t abuse these, ie: `o = --o - o--;`
- Logic Operators:

  `&&, ||, !`
- Bitwise Operators:

  `&, |, ~, >>, <<`

Arrays

- Arrays start at **ZERO!** (a mistake you *will* make often, trust me)
- Arrays of int, float, etc. are pretty intuitive

```
int months[12];
float scores[30];
```

- Strings are arrays of char (C's treatment of strings is not so intuitive)
  - see Wang, Appendix 12 for string handling functions
- Multi-dimensional arrays:

  `int matrix[2][4];` (*not* `matrix[2,4]`)

Decision and Control

```
if( condition )
    statement;
else
    statement;
```

```
while( condition )
    statement
```

```
for( initial; condition; iteration )
    statement;
```

```
do
    statement;
while( condition )
```

- **`break`** and **`continue`** useful inside loops

Decision and Control (cont)

```
switch ( expression )
   case constant1:
         statement;
         break;
   case constant2:
         statement;
         break;
   default:
         statement;
         break;
```

Scope

- Scopes are delimited with curly braces

  “{” <scope> “}”

- New scopes can be added in existing scopes
- Child scopes inherit visibility from parent scope
- Parent scope cannot see into child scopes
- Outermost scopes are all functions
- *These scope rules are all similar to those of Turing and other common programming languages*

Functions

- Definition:

  ```
  <type> <functionName> ( [type paramName], ... )
  ```

- No “procedures” in C … only functions
- Every function should have a prototype
- Example:

```
float area( float width, float height );

float area( float width, float height )
{
   return( width * height );
}
```

Preprocessor

#include (**<file.h>** versus **“file.h”**)

#define (constants as well as macros)

#ifdef (useful for debugging and multi-platform code)

```
#ifdef
    statements
#else
    statements
#endif
```

Structs

```
struct [<structureName>]
{
   <fieldType> <fieldName>;
} [<variableName>];
```

- structureName and variableName are optional, but should always have at least one, otherwise it's useless (can't ever be referenced)
- Example:

```
struct
{
  int quantity;
  char name[80];
} inventoryData;
```

Typedefs and Enumerated Types

`typedef <typeDeclaration>;`

- Example:

```
typedef int tBoolean;
tBoolean flag;
```

`enum <enumName> { tag1, tag2, ... } <variableName>`

- Example:

```
enum days { SUN, MON, TUE, WED, THU, FRI,
  SAT };
enum days today = MON;
```

or

```
typedef enum { SUN, MON, TUE } tDay;
tDay today = MON;
```

Pointers

- A pointer is a type that points to another type in memory
- Pointers are typed: a pointer to an int is different than a pointer to a long
- An asterisk before a variable name in its declaration makes it a pointer
  - *i.e.*: **`int *currPointer;`** (pointer to an integer)
  - *i.e.*: **`char *names[10];`** (an array of char pointers)
- An ampersand (&) gives the address of a pointer
  - *i.e.*: **`currPtr = &value;`** (makes currPtr point to value)
- An asterisk can also be used to de-reference a pointer
  - *i.e.*: **`currValue = *currPtr;`**

Pointers (cont)

- Use brackets to avoid confusion:
  - ie: **`*(currPtr++);`** is *very* different from **`(*currPtr)++;`**
- Using ++ on a pointer will increment the pointer's address by the size of the type pointed to
- You can use pointers as if they were arrays (in fact, arrays are implemented a pointers)

Command Line Arguments

```
int main( int argc, char *argv[] )
{
. . .
```

- **`argc`** is the number of arguments on the command line, including the program name
- The array **`argv`** contains the actual arguments
- Example:

```
    if( argc == 3 )
        printf( "file1:%s file2:%s\n",
                 argv[1], argv[2] );
```

Casting

- You can force one type to be interpreted as another type through casting, ie:

  `someSignedInt = (signed int) someUnsignedInt;`

- Be careful, as C has no type checking, so you can mess things up if you're not careful
- **`NULL`** pointer should always be cast, ie:
  - **`(char *) NULL`**, **`(int *) NULL`**, etc.

UNIX Utilities

Miscellaneous

- **ln** (*symbolic links*)
- **chmod** (*permissions*)
- **man -k fork** and **man 2 fork** (*ie: viewing specific pages*)
- **du** (*disk space usage)*
- **quota -v username** and **pquota -v username**
- **noglob**

gzip, **compress**

- Usage: **`gzip [filename]`**: compress specified filename
  `gunzip [filename]`: uncompress specified filename

- Examples:

`gzip file1`	*creates file1.gz*
**`gunzip <file2.gz	more`**	*leaves file2.gz intact*
**`cat file3	gzip > newFile.gz`**	*leaves file3 intact*

- **`compress`** behaves like **`gzip`**, using a different (less efficient) compression algorithm is used (resulting files have **`.Z`** extension).
- Similarly, **`uncompress`** behaves like **`gunzip`**

`tar`

- Traditionally, tar (short for Tape ARchive) was used for backups to tape drives
- It's also useful to create archive files on disk.

- Example: creating an archive of a directory structure:

  `tar fcvp dir1.tar dir1`
- Example: uncompressing and extracting a tar file:

  `gunzip < dir2.tar.gz | tar fxvp -`
- Example: copying a directory structure:

  `tar fcvp - dir1 | ( cd newloc; tar fxvp - )`
- Advantage over "**`cp -rp`**": preserves symbolic links

`lint`

- **`lint`** is a useful utility that checks programs more thoroughly that **`gcc`** or other compilers
- Usage:

```
lint file1 [file2] ...
```

```
% cat main.c

#include <stdio.h>
void main()
{
   int i;
   printf("Hello\n");
}
```

```
% lint main.c

variable unused in function:
     (5) i in main

function returns value
which is always ignored:
     printf
```

Other System Calls

Named pipes: **`mknod()`**

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main() {
   unlink( “namedPipe” );
   mknod(  “namedPipe”, S_IFIFO, 0 );
   chmod(  “namedPipe”, 0600 );
   if( fork() == 0 ) {
      int fd = open( “namedPipe”, O_WRONLY );
      dup2( fd, fileno(stdout) ); close( fd );
      execlp( "ruptime", "ruptime", (char *) 0 );
   } else {
      int fd = open( “namedPipe”, O_RDONLY );
      dup2( fd, fileno(stdin) ); close( fd );
      execlp( "sort", "sort", "-r", (char *) 0 );
   }
}
```

`vfork()`

- The typical **`fork()`/`exec()`** sequence is inefficient because **`fork()`** creates a copy of the data, heap, and stack area of the original process, which is then immediately discarded when **`exec()`** is called.
- **`vfork()`** is intended to create a new process when the purpose of the new process is to **`exec()`** a new program. **`vfork()`** has the same calling sequence and the same return values as **`fork()`**.
- **`vfork()`** creates the new process, just like **`fork()`**, without fully copying the address space of the parent into the child, since the child won't reference that address space -- the child just calls **`exec()`** right after the **`vfork()`**.
- Another difference between **`vfork()`** and **`fork()`** is that **`vfork()`** guarantees that the child runs first, until the child calls **`exec()`** or **`exit()`**.

`system()`

- It is sometimes convenient to execute a command string from within a program.

- For example, to put a time and date stamp into a certain file, one could:
  - use **`time()`**, and **`ctime()`** to get and format the time, then open a file for writing and write the resulting string.
  - use **`system( "date > file" );`** (*much simpler*)

- **`system()`** is typically implemented by calling **`fork()`**, **`exec()`**, and **`waitpid()`**