

1- Unix Start	2
2- VIM Start-3	38
3-Unix History Files Paths	58
4-UNIX tools	115
5-UNIX Make Tool	142
6-UNIX System Calls	181
7-UNIX env	197
8-UNIX Fork Exec	214
9-UNIX SystemCall File IO	273
10-UNIX fork wait	321
11-UNIX Signal	353
12-UNIX Pipes	406
13-UNIX Shared Memory Msgs	465
14-UNIX Threads	506
C-1 Start	530
C-2 Characters ControlStructures	597
C-3 Loops	654
C-4 Files	678
C-5 Functions	705
C-6 Arrays	768
C-7 Pointers	818
C-8 Structures	901
C-9 Bitwise Operators	941
C-10 Char and Strings	970

OS/Shell Introduction

Getting Started

Computer Accounts

- Each student in the School of Engineering and Computer Science should have an ECS account.
 - Go the site: <http://www.ecs.csus.edu/>
 - Scroll to below the picture.
 - Click on **ECS Quick Links**
 - Choose the option **Get an ECS Account**
 - Follow the directions. You must present a OneCard when you pick up the information.

Process needed to deal with a program (1 of 2):

- Log onto **athena** computer
 - Windows machines: Use PuTTY
 - MAC machines: Open up a terminal/console window (ssh)
- Open an editor. (C programmers use **vim**.)
- Write code, compile, save, etc.
- Get the code to a place where you can open your browser and upload it to Canvas (replacement for SacCT)

Process needed to deal with a program (2 of 2):

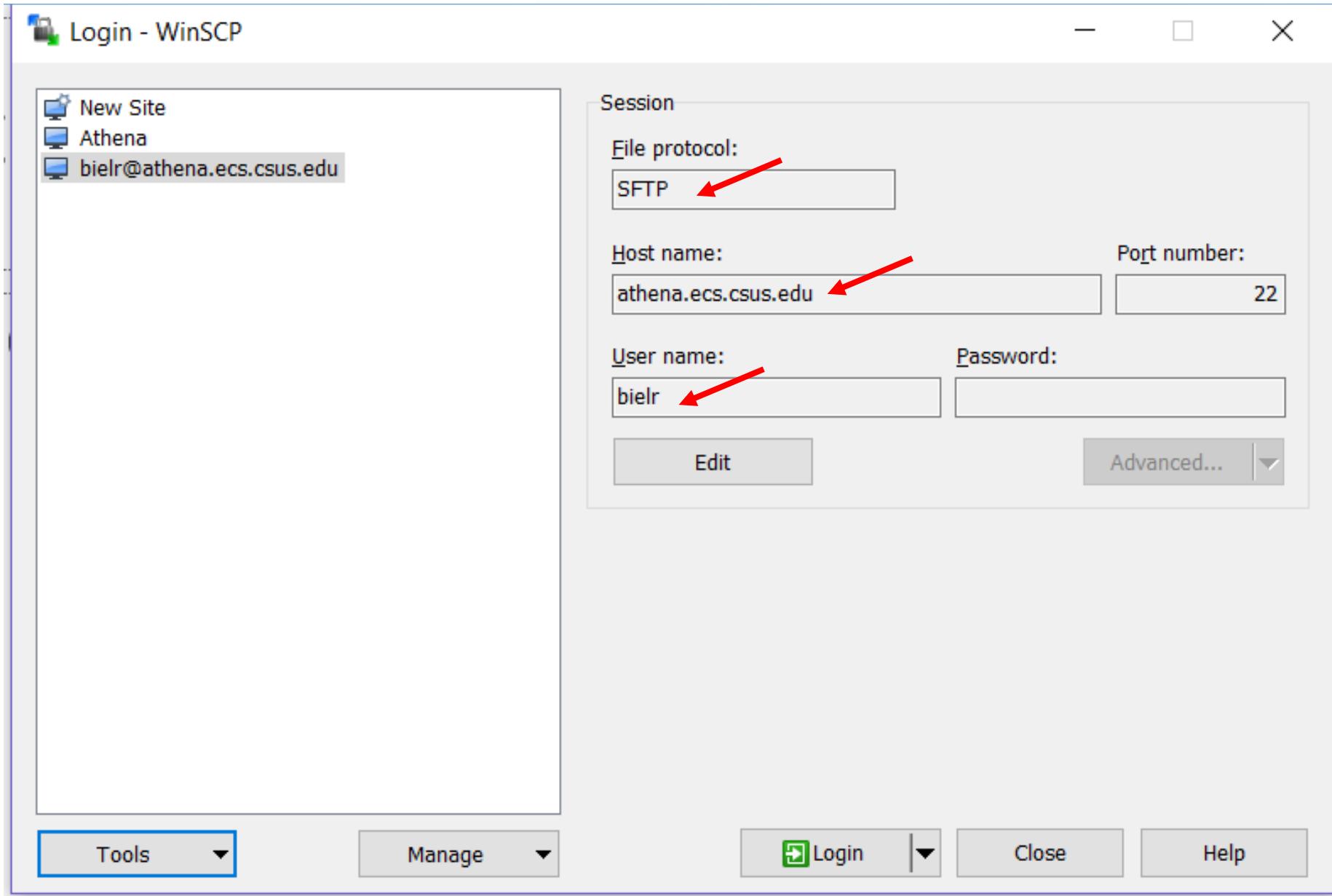
- Get the code to a place where you can open your browser and upload it to Canvas (replacement for SacCT)
 - Use file transfer software to move the code from **athena** to your own computer. (WinSCP, FileZilla, CyberDuck)
 - Email the file from athena to yourself. (pine)
 - Log onto an ECS computer, click on “My Files On **gaia**” to open your file, open a browser, upload it.
 - Mac: Use scp to transfer the file.

Software for moving files
between laptop/home and athena

Optional software for home (Windows only):

- **WinSCP** – free software that allows one to move files from one site to another, from athena to home, and the reverse.
- At the site: <http://winscp.net/>
- Next slide shows startup menu with entries you will need

WinSCP Log-in Screen with settings



WinSCP Sample screen: *left* side is a folder on home computer, *right* side is folder on gaia

The screenshot shows the WinSCP interface comparing two folders:

- Local:** C:\Users\Ruthann\Documents\CSUS Classes\csc 25\Fall 2015\Labs\Lab4
- Remote:** /gaia/home/faculty/bielr/class_files

The left pane displays files from the local folder, and the right pane displays files from the remote gaia folder. Both panes show columns for Name, Size, Type, and Changed.

Local (Left Pane) File List:

Name	Size	Type	Changed
..		Parent directory	10/14/2015 11:12:14 AM
Lab4 solution.c	2 KB	C File	9/20/2015 3:34:27 PM
lab4.c	2 KB	C File	9/21/2015 4:12:06 PM
Lab4_test.c	2 KB	C File	9/20/2015 3:33:42 PM
Lab4.docx	18 KB	Microsoft Word Do...	9/23/2015 5:14:49 PM

Remote (Right Pane) File List:

Name	Size	Changed	Rights	Owner
..		1/21/2016 1:43:26 PM	rwx--x--x	bielr
a.out	6 KB	11/4/2015 11:18:32 AM	rw-r--r--	bielr
lab2a.c	1 KB	9/14/2015 11:02:50 AM	rw-r--r--	bielr
lab2b.c	2 KB	9/14/2015 11:03:05 AM	rw-r--r--	bielr
lab3.c	3 KB	9/16/2015 11:19:26 AM	rw-r--r--	bielr
lab4.c	2 KB	9/21/2015 4:12:06 PM	rw-r--r--	bielr
lab5.c	1 KB	9/21/2015 1:06:31 PM	rw-r--r--	bielr
lab6a.c	2 KB	9/28/2015 4:57:31 PM	rw-r--r--	bielr
lab6b.c	2 KB	9/30/2015 8:56:11 PM	rw-r--r--	bielr
lab7a.c	2 KB	10/5/2015 2:48:12 PM	rw-r--r--	bielr
lab7b.c	2 KB	10/5/2015 2:50:45 PM	rw-r--r--	bielr
lab7c.c	2 KB	10/5/2015 2:52:27 PM	rw-r--r--	bielr
lab8.c	1 KB	10/14/2015 3:21:25 PM	rw-r--r--	bielr
lab11.c	5 KB	10/28/2015 10:22:24 AM	rw-r--r--	bielr
lab12.c	1 KB	11/1/2015 1:20:21 PM	rw-r--r--	bielr
lab12sample.dat	1 KB	3/20/2006 3:38:38 PM	rw-r--r--	bielr
lab13.c	2 KB	11/4/2015 11:18:24 AM	rw-r--r--	bielr
lab14.c	3 KB	11/11/2015 3:54:01 PM	rw-r--r--	bielr
lab14.dat	1 KB	11/11/2015 1:42:09 PM	rw-r--r--	bielr
lab15.c	3 KB	11/15/2015 12:50:32 PM	rw-r--r--	bielr
lab15.dat	1 KB	3/21/2004 9:08:22 PM	rw-r--r--	bielr
lab16.c	4 KB	12/8/2015 12:20:17 PM	rw-r--r--	bielr
lab16a.dat	1 KB	10/31/2005 1:09:48 PM	rw-r--r--	bielr
lab16b.dat	1 KB	10/31/2005 1:11:22 PM	rw-r--r--	bielr

Both panes show 0 B of 22,065 B in 0 of 4 for the local side and 0 B of 43,799 B in 0 of 29 for the remote side.

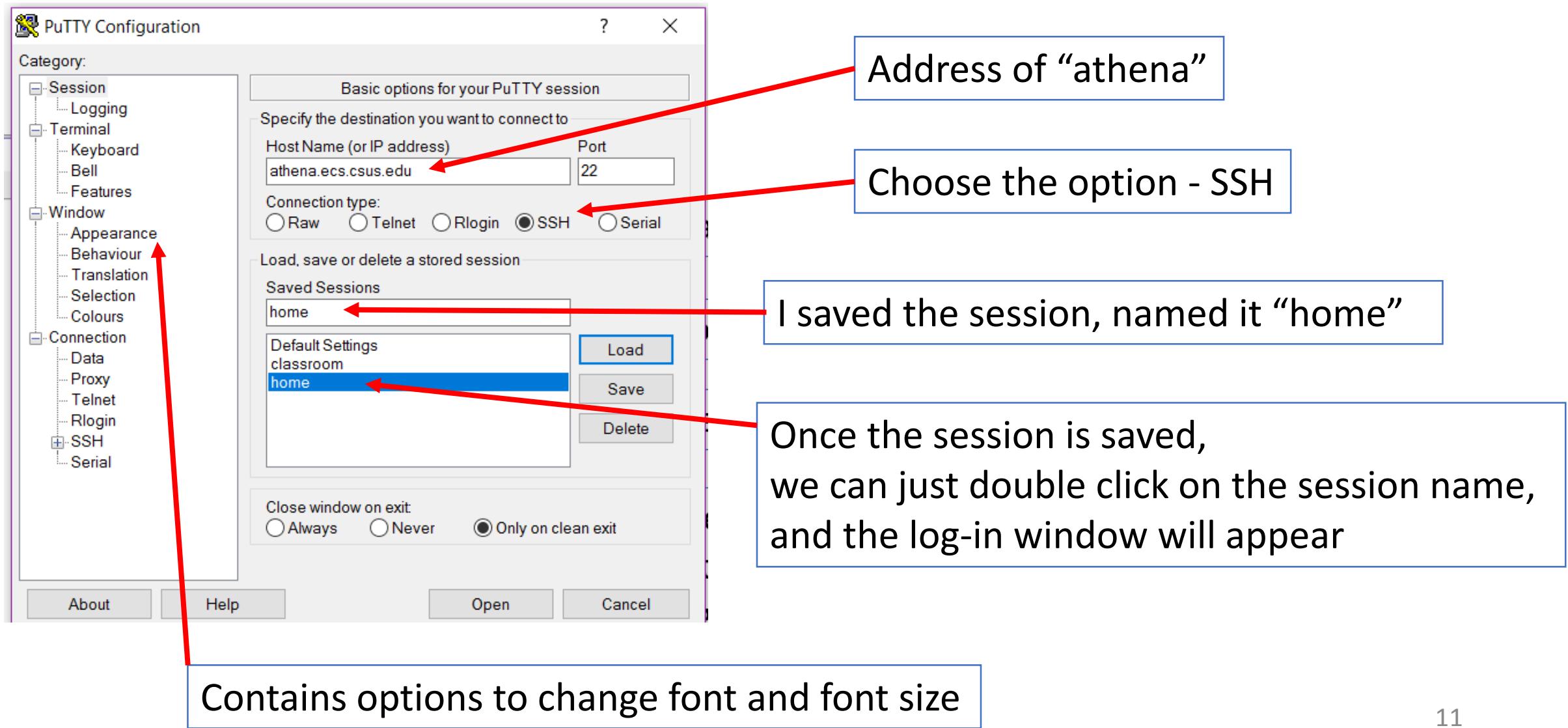
Optional software for home (MAC & Windows):

- Free software that allows one to move files from one site to another, from athena to home, and the reverse.
 - **Filezilla** – <https://filezilla-project.org/>
 - **Cyberduck** - <https://cyberduck.io/?l=en>
- Both software packages work on Windows or Mac.
- A search on “cyberduck vs filezilla” will bring up a couple of comparison articles.

Logging onto a UNIX machine

- Sit at a UNIX machine and log in.
- As you enter your password, nothing will show on the screen.
- Do a remote login using SSH
 - Most of us use PuTTY software to accomplish this.
 - The ECS computers all have PuTTY
 - To get PuTTY at home, download it from:
<http://www.chiark.greenend.org.uk/~sgtatham/putty/>

PuTTY Screen



MAC Users

- Open up a terminal/console window and type the following:

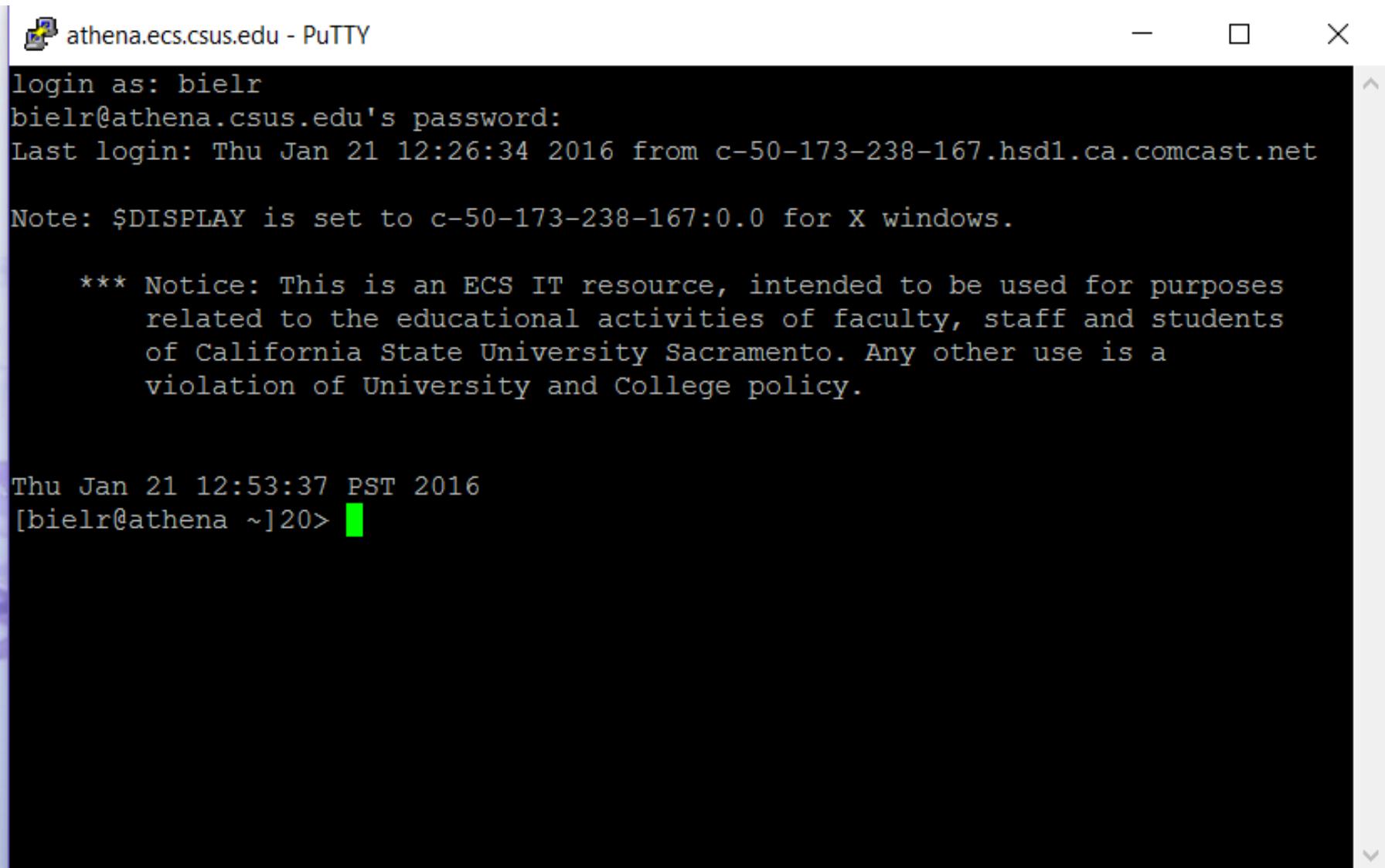
ssh yourECSname@athena.ecs.csus.edu

Press Enter.

When prompted, type “yes” to accept the server’s key.

Then enter your password.

Login Screen



The screenshot shows a PuTTY terminal window titled "athena.ecs.csus.edu - PuTTY". The session has started with the command "login as: bielr". It then prompts for the password of the user "bielr@athena.csus.edu". Below this, it displays the last login information: "Last login: Thu Jan 21 12:26:34 2016 from c-50-173-238-167.hsd1.ca.comcast.net". A note follows, stating that "\$DISPLAY is set to c-50-173-238-167:0.0 for X windows". A prominent notice message is displayed, reading: "*** Notice: This is an ECS IT resource, intended to be used for purposes related to the educational activities of faculty, staff and students of California State University Sacramento. Any other use is a violation of University and College policy." At the bottom of the window, the current date and time are shown as "Thu Jan 21 12:53:37 PST 2016", followed by the prompt "[bielr@athena ~]20>".

Shells:

A shell is an interface between you and the kernel of UNIX/Linux.

Kernel. The center, the core.

Shell. A way to communicate with the Kernel.

The default on our system is **csh**, read aloud as ‘C-shell’.

By doing “**cat /etc/shells**”, I found that athena has:

- sh (Bourne Shell)
- bash (superset, Born Again Shell. LOL)
- nologin
- tcsh
- **csh** (spoken as C-shell) (**Default** on athena)
- dash
- ksh

It is possible to change the default shell,
using the command “**chsh**”.

Shell Verification

To see what Shell you are in, type:

```
> echo $SHELL
```

Getting help:

“Look at the “man” page.” You will hear this.

This means looking at the on-line manual which is extensive.

\$ **man** *command*

Examples: /* will show you... */

 \$ man ls /* all the options for listing */

 \$ man gcc /* options for the compiler */

Maneuvering through a *man* page:

Hit **space bar** to advance thru page.

Hit **Enter** to advance the screen one line.

Hit “**q**” to quit.

Various Commands in UNIX/Linux

Command: ls

Purpose: List files in directory.

Format: ls [options] [file-list]

Some Options:

- a List all files, including hidden ones.
- d List directory names only, not ordinary files.
- g Show group information with listing.
- l Show long listing with extended information.
- r List in reverse order.
- s List in order of increasing size.
- t List in order of time, most recent first.

Example: ls ls -ra
 ls -l ls -ls

Copying and Renaming Files:

Command: `cp`

Purpose: Copy a file.

Format: `cp source-file target-file`

Example: `cp my.file file2`

Result: There are now two identical files with different names.

Moving or renaming files:

Command: mv

Purpose: Move or rename a file.

Format: `mv source-file target-file`

Example: `mv my.file file2`

Result: One file with the target name exists.

Removing files:

Command: rm

Purpose: Remove a file.

Format: rm [option] *file(s)*

Option: -i Ask before deleting.
Often the default.

Example: rm file2

Result: The file is no longer listed or available.

The *cat* command:

Command: cat

Purpose: Display or create files.

Format: cat [*source-file*] [*symbol*] [*target-file*]

Examples:

1. cat this.month
2. cat lab1.c

Result: File displayed on screen, lines echoed on screen

The *pwd* command:

Command: `pwd`

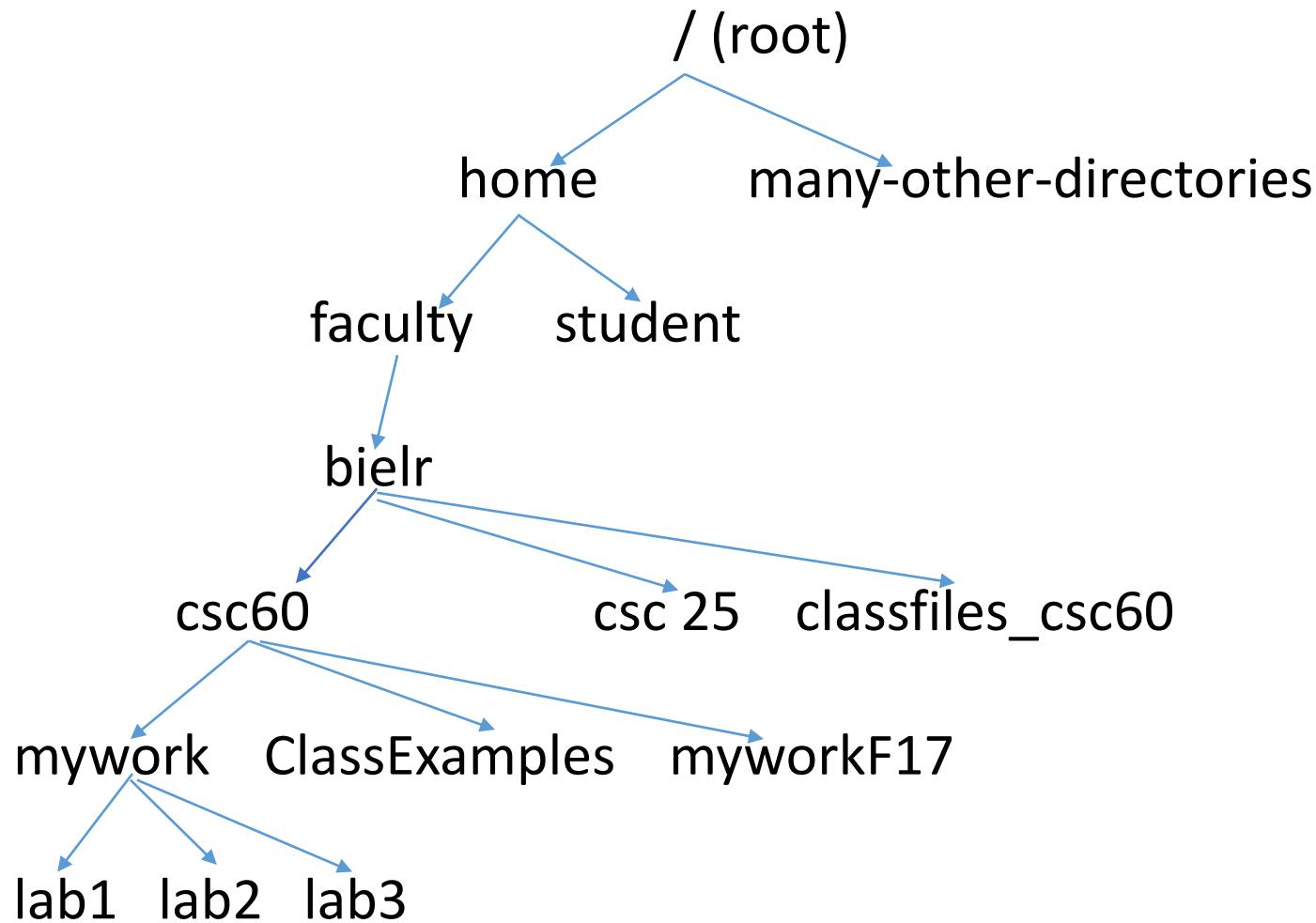
Purpose: print name of current/working directory.

Format: `pwd`

Example & Result:

```
[bielr@athena csc60]> pwd  
/gaia/home/faculty/bielr/csc60  
[bielr@athena csc60]>
```

Starting Directory Structure on Linux



The *cd* command:

Command: `cd`

Purpose: Change directory.

Examples & Results:

1. `cd` Takes you to your home directory
2. `cd ..` Takes you up to the parent directory
3. `cd lab1` Takes you down to a lower directory
4. `cd mywork/lab1` Takes you down to a lower directory

The *mkdir* command:

Command: `mkdir`

Purpose: Make directories.

Examples & Results:

1. `mkdir csc60` Makes a sub-directory named csc60
2. `mkdir lab1` Makes a sub-directory named lab1

Directories can also be moved or renamed (`mv`), and copied (`cp -r`)

The *rmdir* command:

Command: `rmdir`

Purpose: remove empty directories.

Examples & Results:

1. `rmdir csc60` Makes a sub-directory named csc60
2. `rmdir lab1` Makes a sub-directory named lab1

Note:

To delete an empty directory, you must be in the directory above it or you need to type a full path name.

The *clear* command:

Command: clear

Purpose: clear the terminal screen

Example:

```
> clear
```

Other Assorted Commands

- **less, more** – paging utilities
 - Use the “man” pages to find more
- **od – octal dump.** For viewing raw data in octal, hex, control chars, etc.
 - Use the “man” pages to find more
- **ln** – create hard (inode) or soft (symbolic) links to a file
 - [On creating a file, UNIX allocates the file an inode number of 4 bytes, an index value for an array on the disk. So every file has a unique inode number.]

Change The Prompt On athena (csh)

Change the prompt to show the folder/directory that you are in.

You need to type a SPACE after “set” and after the “m”.

Type in the command: `set prompt='[%n%m %~]!>'`

The **%n** will show your name.

The **%m** will show your current folder.

The **%~** will give you the command number.

REDIRECTION:

(1 of 4)

`$ ls` /* lists all files in your directories */

`$ ls | more` /* pipes the output to the *more* program which
gives you a screen-full at a time */

The pipe symbol “|” redirects the standard output of one command to the standard input of another command or process.

REDIRECTION: (2 of 4)

Use `>` to redirect an output to a file.

So `cal` is the calendar command.

```
$ cal 2017 > my_calendar
```

REDIRECTION:

(3 of 4)

Use `>>` to append to a file.

ps means process status.

```
$ ps >> my_file
```

Whatever was in *my_file* will now have the listing from the *ps* command appended to it.

Just to make things clear on Redirection: (4 of 4)

Use of the pipe “|” sends output to a process.

Use of the redirection “>” sends output to a file.

NOTE: The details of Redirection vary from shell to shell.

OS/Shell Introduction

Getting Started

The End

VIM Editor Introduction

Getting Started

VIM – Entering and Exiting

- To enter VIM, at the prompt, type: **vi YourFileName**
or: **vim YourFileName**
- On *athena*, **vi** takes us to **vim**. This is not true on all installations.
- Technically, **vi** is the older version. **vim** is an updated version.

VIM Modes

- **Command Mode.**
 - For entering commands, usually two letters and number.
 - For moving around a file
 - Commands that start with a colon require a Enter/Return key
- **Insert Mode.**
 - To insert, type: **i**
 - Now you can start typing code or other information.
 - To leave Insert Mode, press: **EscapeKey**

VIM – Entering and Exiting

Type: **i** to enter insert mode
escape to exit insert mode
:q! to exit without saving
:wq to exit and save work
ZZ to exit and save work, version 2, & be sure to use caps.
:q to quit
:w YourFileName if you forgot to use a filename at the start,

Basic Cursor Movement

Lowercase Command	Arrow Key equivalent
l	Right arrow
h	Left arrow
j	Down arrow
k	Up arrow

Our version of **vim** allows the use of the arrow keys.

Basic Corrections

- While in Insert Mode:
 - Delete characters by using the back space key
- While in Command Mode:
 - Move Cursor to the first character to be deleted. Press: **x**
 - To delete multiple characters
 - Example: If you want to delete 6 characters in a row, move the cursor on the first one, and type: **6x**

Cursor Movement. 1 of 6

- **Spacebar** – Forward one character position
- **l** - Right (forward) one character position
- **h** - Left (backward) one character position
- **j** - Down to the same position in line below;
moves left to last position
- **k** - Up to the same position in the line above;
moves left to the last position

Cursor Movement. 2 of 6

- **w** - Forward to first letter of next word
- **W** - Forward to first letter of next blank-delimited word
- **b** - Backward to first letter of previous word
- **B** - Backward to first letter of previous blank-delimited word

Cursor Movement. 3 of 6

- **Return** - Forward to beginning of next line
- **0** - Back to beginning of next line (zero)
- **\$** - End of current line

Cursor Movement. 4 of 6

- (- Back to beginning of current sentence
-) - Ahead to beginning of next sentence
- { - Back to beginning of current paragraph
- } - Ahead to beginning of next paragraph

Cursor Movement. 5 of 6

- **H** - Home, or left end of the top line on screen
- **M** - Middle, or left end of middle line on screen
- **L** - Lower, or left end of lowest line on screen
- **G** - Last line in work buffer
- ***n*G** - Indicated relative line *n* in buffer.

Cursor Movement. 6 of 6

- **Ctrl-U** - Up half screen
- **Ctrl-D** - Down half screen
- **Ctrl-F** - Forward (down) almost a full screen
- **Ctrl-B** - Backward (up) almost a full screen

Delete Commands. 1 of 3

- x** - Character at cursor
- X** - Character following cursor
- dw** - To end of word
- dW** - To end of blank-delimited word
- db** - To beginning of word
- dB** - To beginning of blank-delimited word

Delete Commands. 2 of 3

- **d** then **Return** - Two lines; current and following
- **X** - Character following cursor
- **dw** - To end of word
- **dW** - To end of blank-delimited word
- **db** - To beginning of word
- **dB** - To beginning of blank-delimited word

Delete Commands. 3 of 3

- d) - To end of sentence
- d(- To beginning of sentence

- d} - To end of paragraph
- d{ - To beginning of paragraph

Delete Commands. 3 of 3

- d) - To end of sentence
- d(- To beginning of sentence

- d} - To end of paragraph
- d{ - To beginning of paragraph

HINTS: Yank & Put (Copy & Paste)

Copying and pasting in **vim** are accomplished with the commands **yank** and **put**.

<i>Command Syntax</i>	<i>What It Accomplishes</i>
y2w	Yanks two words, starting at the current cursor position, going to the right
4yb	Yanks four words, starting at the current cursor position, going to the left
yy or Y	Yanks the current line
p	Puts the yanked text after the current cursor position (lower case p)
P	Puts the yanked text before the current cursor position (upper case P)
5p	Puts the yanked text in the buffer five times after the current cursor position

Setting up *tabs & line numbers* in vim.

Go to your **home** directory

Type: **vim .vimrc** *which is a settings-for-vim file*

Once in the file, add to the Vim default file (if you have one):

type: **:set shiftwidth=4**

type: **:set smartindent**

type: **:set expandtab**

type: **:set number**

More Help

At the prompt on athena, type **vimtutor** for more instruction.

Go to Google.

Enter “**VIM Tutorial**”

You will find lots of choices, some practical, even one that sells itself as “Zelda meets VIM”.

VIM Editor Introduction

Getting Started

The End

3-Unix

History, Files, Paths



CSC-60

Unix History

A brief history of UNIX OS

- The Unix OS was developed (based on Multics & CTSS operating systems) by Ken Thompson at the AT&T Bell Laboratories in 1969. He wanted to create an multi-user operating system to run “space wars” game.
- Ken’s philosophy was to create an operating system with commands or “utilities” that would do one thing well (i.e. **UNIX**). Pipes could be used combine commands...

History of Unix OS

- The first versions of UNIX were written in “machine-dependent” program (such as PDP-7).
- Ken Thompson approached Dennis Ritchie, developer of C language, and in 1973 they compiled UNIX in C to make operating system “portable” to other computers systems.

History of Unix OS



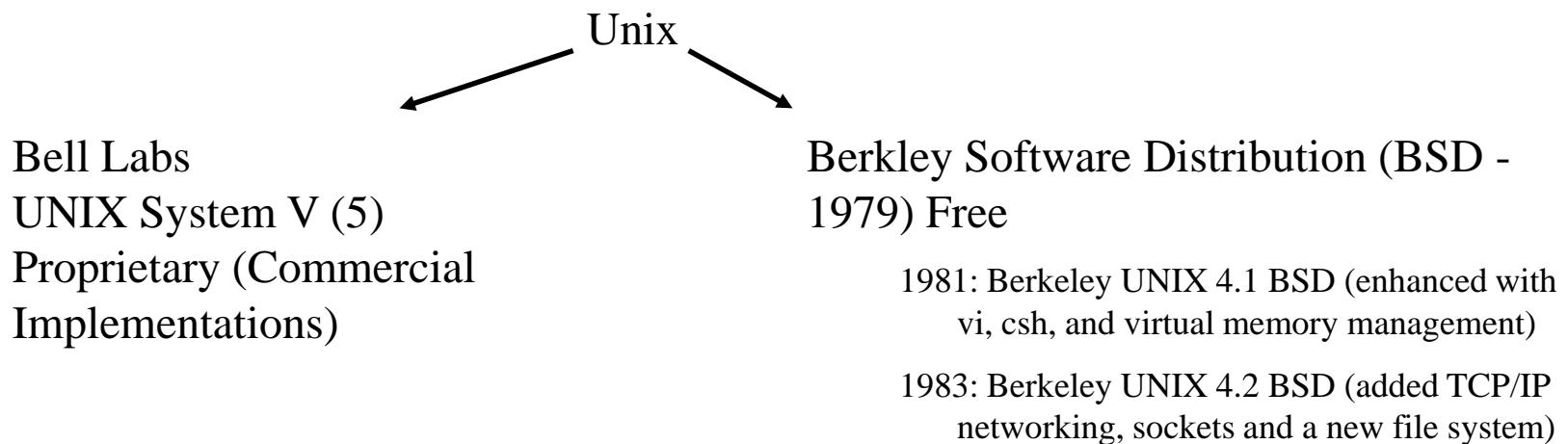
PDP-7 Machine



Ken (seated) and Dennis (standing) at a PDP-11 in 1972.

Development of Unix OS

Students at University of California (in Berkley) further developed the UNIX operating system and introduced the BSD version of Unix



Development of Unix OS

There were versions of UNIX for the Personal Computer (PC), such as XENIX, etc., but they didn't catch on in popularity until Linux was developed in the early 90's.

History of Linux



- Linux operating system developed by programming student Linus Torvalds (1991)
- Linus wanted to develop Unix-like OS just to experiment with new 386 computer at the time...
- Linus invited other programmers to improve the Kernel. Overtime, it was ported to various hardware architectures

GNU (GNU 's NOT UNIX) Project



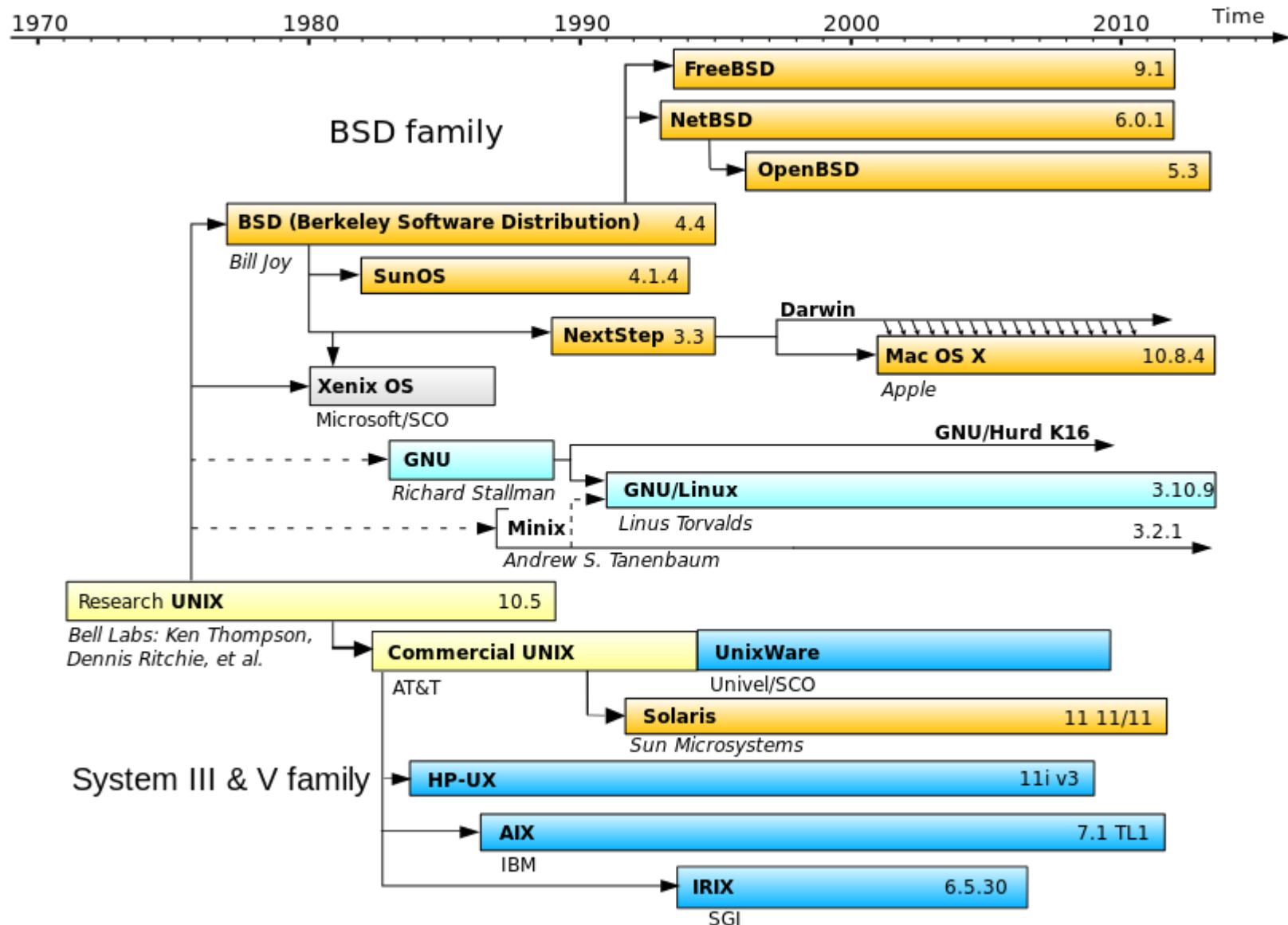
Richard Stallman

- Launched in 1984 to develop a complete UNIX Operating system that is free software:
www.gnu.org (Free Software Foundation - founded by **Richard Stallman**)

Usually C code, but some C++

- GNU has a 72-page document of coding standards – well-written code. It provides useful tools such as Emacs, Gcc, Bash shell, glibc (C library)
- GNU's kernel (Called Hurd) was not working (not stable).

Brief history of Unix-like operating systems



Standardizations (1 of 2)

- POSIX: Portable Operating System Interface
 - An IEEE-standard which describe the behavior of UNIX and UNIX-like OS.
 - POSIX support assures code portability between systems and is increasingly mandated for commercial applications and government contracts.

Standardizations (2 of 2)

- SUSv3: Single UNIX Specification version 3
 - Beginning in 1998, joint working group known as the Austin Group began to develop the combined standard that would be known as the Single UNIX Specification Version 3 and as POSIX:2001. This name serves as referenced points throughout the book.
- Examples from our main textbook:
 - ...This 65-character set, {-._a-zA-Z0-9], is referred to in **SUSv3** as the *portable filename character set*. Page 28.
 - ...And Standard system defined by SUSv3... Page 43.

Operating System

An operating system is a control program for a computer that performs the following operations:

- allocates computer resources
- schedules tasks
- provides a platform to run application software for users to accomplish tasks
- provides an interface between the user & the computer

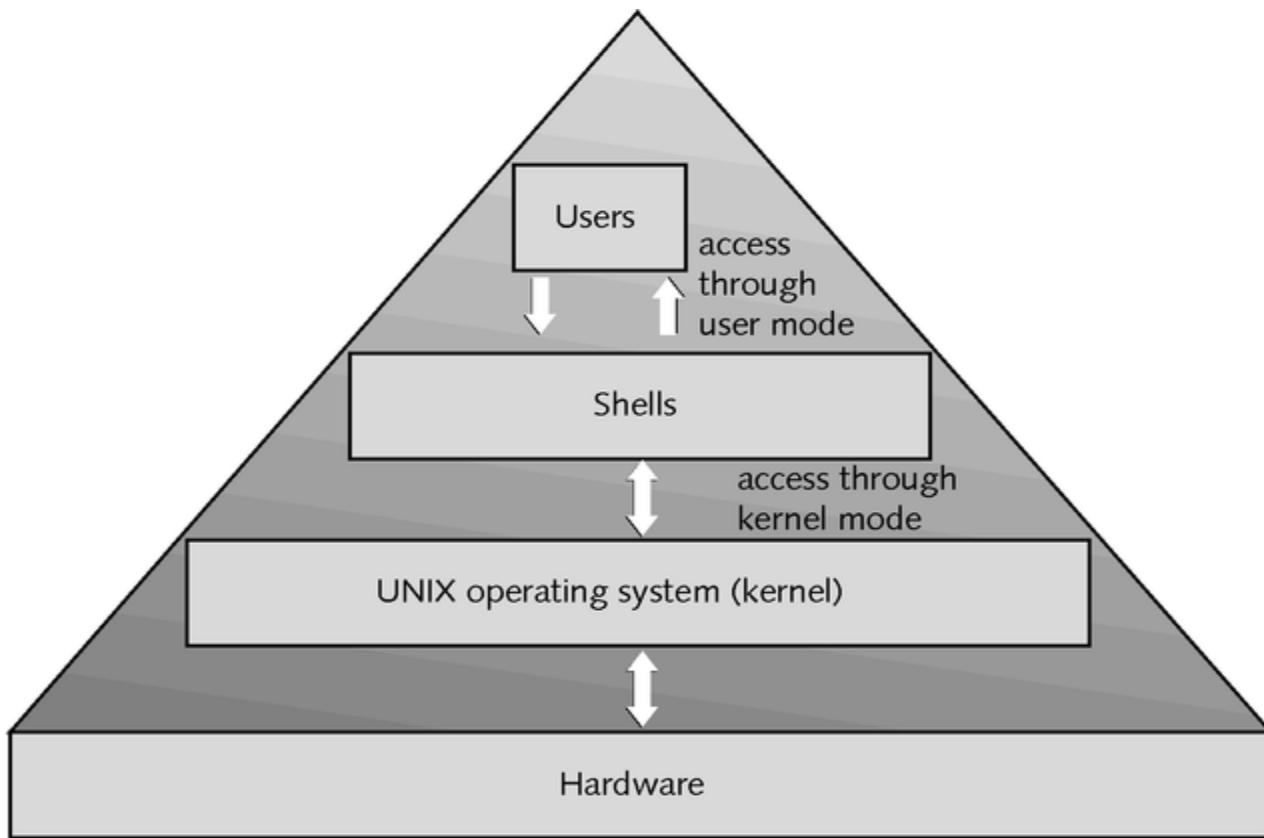


Figure 1-5 Layers of a UNIX system

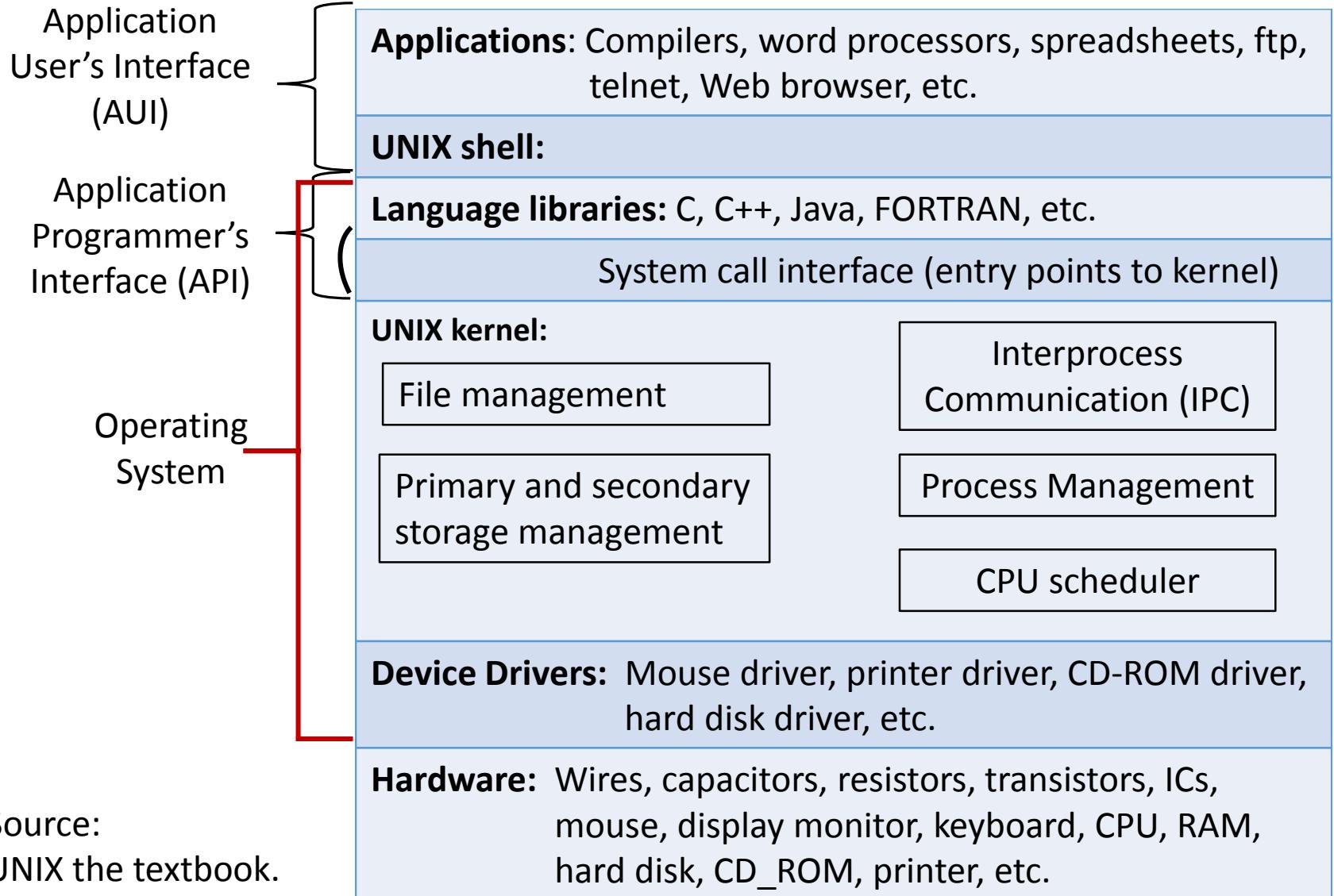
Shell as a user interface

- A shell is a command interpreter, an interface between a human (or another program) and the OS
 - runs a program, perhaps the `ls` program.
 - allows you to edit a *command line*.
 - can establish alternative sources of input and destinations for output for programs.
- `ls`, itself, just another program

Kernel (OS)

- Interacts directly with the hardware through device drivers
- Provides sets of services to programs, insulating these programs from the underlying hardware
- Manages memory, controls access, maintains file system, handles interrupts, allocates resources of the computer
- Programs interact with the kernel through *system calls*

UNIX Software Architecture



Source:
UNIX the textbook.
By Sarwar, Koretsky,
Sarwar.

Linux Files

File Attributes

File attributes

- Every file has some attributes:
 - ❖ Access Times:
 - when the file was created
 - when the file was last changed
 - when the file was last read
 - ❖ Size
 - ❖ Owners (user and group)
 - ❖ Permissions
 - ❖ Type – directory, link, regular file, etc.

File Owners

- Each file is owned by a user.
- You can find out the username of the file's owner with the `-l` or `-o` option to `ls`:

```
athena.ecs.csus.edu - PuTTY
[bielr@athena csc60]> ls -l
total 540
-rw----- 1 bielr faccsc      2 Dec 21 12:58 >
-rwx----- 1 bielr faccsc 6438 Sep 15 13:47 a.out*
drwx----- 2 bielr faccsc 4096 Oct 21 09:34 ClassExamples/
-rw----- 1 bielr faccsc   138 Dec 22 09:39 lsout
drwx----- 5 bielr faccsc 4096 Jan 24 13:08 mywork/
drwx----- 6 bielr faccsc 4096 Dec 18 15:58 myworkf16/
drwx----- 8 bielr faccsc 4096 Dec 16 15:07 myworkS16/
-rwx----- 1 bielr faccsc 6438 Sep 19 09:04 reverse*
-rw----- 1 bielr faccsc   993 Sep 16 13:24 reversel.c
drwx----- 2 bielr faccsc 4096 Jan 15 13:01 student/
-rw----- 1 bielr faccsc   527 Nov 16 08:35 testScript.txt
-rw----- 1 bielr faccsc 235289 Apr 17 2016 tlpi-160401-dist.tar.gz
drwx----- 48 bielr faccsc 4096 Nov 10 09:26 tlpi-dist/
-rw----- 1 bielr faccsc 252898 Sep 21 15:26 trylab1.txt
-rw----- 1 bielr faccsc    12 Dec 22 09:40 wcout
[bielr@athena csc60]>
```

File Attributes shown by **ls -l**

```
drwx----- 2 bielr faccsc 4096 Apr 27 15:43 ClassExamples/
```

Field	Meaning
First letter of first field	File type: - ordinary file b block special file c character special file d directory l link p named pipe (FIFO) s socket

File Attributes shown by **ls -l**

```
drwx----- 2 bielr faccsc 4096 Apr 27 15:43 ClassExamples/
```

Field	Meaning
Remaining letters of first field	Access permissions for owner, group, and others (r w x)
Second field	Number of links

File Attributes shown by **ls -l**

```
drwx----- 2 bielr faccsc 4096 Apr 27 15:43 ClassExamples/
```

Field	Meaning
Third field	Owner's login name
Fourth field	Owner's group name (can also be a number)

File Attributes shown by **ls -1**

```
drwx----- 2 bielr faccsc 4096 Apr 27 15:43 ClassExamples/
```

Field	Meaning
Fifth field	File size in bytes
Sixth, seventh, and eighth field	Date and time of last modification
Ninth field	File name

ls -l

```
$ ls -l foo
```

-rw-rw----	1	bielr	faccsc	13	Jan 10	23:05	foo
permissions		owner		group	size		name
							time

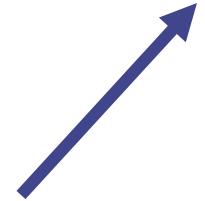
File Permissions

- Each file has a set of permissions that control who can work with the file.
- There are three *types* of permissions:
 - read abbreviated **r**
 - write abbreviated **w**
 - execute abbreviated **x**
- There are 3 *sets* of permission:
 - user
 - group
 - other (the world, everybody else)

`ls -l` and permissions

	-	rwx	rwx	rwx
		User	Group	Others

Type of file:
- – plain file
d – directory
s – symbolic link



rwx

- Files:
 - **r** - allowed to read.
 - **w** - allowed to write
 - **x** - allowed to execute
- Directories:
 - **r** - allowed to see the names of the file.
 - **w** - allowed to add and remove files.
 - **x** - allowed to enter the directory

Changing Permissions

- The `chmod` command changes the permissions associated with a file or directory.
- There are a number of forms of `chmod`, this is the simplest:
 - `chmod mode file`

chmod – numeric modes

- Consider permission for each set of users (user, group, other) as a 3-bit #
 - r – 4
 - w – 2
 - x – 1
- A permission (mode) for all 3 classes is a 3-digit octal #
 - 755 – rwxr-xr-x (user: read/write/execute, group:read/execute, others:read/execute)
 - Example: > chmod 755 lab1.c
 - 644 – rw-r—r— (user: read/write, group:read, others:read)
 - 700 – rwx----- (user: read/write/execute, group: no access, others: no access)

chmod – symbolic modes

- Can be used to set, add, or remove permissions
- Mode has the following form:
 - **[ugoa][+-=][rwx]**
 - u – user g – group o – other a – all
 - + add permission
 - - remove permission
 - = set permission

chmod examples

```
$ ls -al foo
```

```
-rwx rwx --x 1 hollingd grads foo
```

```
$ chmod g-wx foo
```

```
$ ls -al foo
```

```
-rwxr----x 1 hollingd grads foo
```

```
$ chmod u-r .
```

```
$ ls
```

```
ls: ..: Permission denied
```

CSC-60

Unix File System

Home Directory

- The user's personal directory. E.g.,
 - /gaia/class/student/xyz
 - /gaia/class/student/yzx
- Where all your files go (hopefully organized into subdirectories)
- Mounted from a file server – available (seamlessly) on ***any*** department machine you log into
(athena, sp1, sp2, sp3, atoz)

Home Directory

- Your *current directory* when you log in
- cd (by itself) takes you home
- Location of many startup and customization files. E.g.:
 - .vimrc .bashrc .bash_profile .forward
.plan .mozilla/ .elm/ .logout

Directories

- A directory is a special kind of file - Unix uses a directory to hold information about other files and directories.
- We often think of a directory as a container that holds other files (or directories).
- A directory is the same idea as a *folder* on Windows.

More about File Names

- Review: every file has a name (at least one).
- Each file *in the same directory* must have a unique name.
- Files that are in different directories can have the same name.

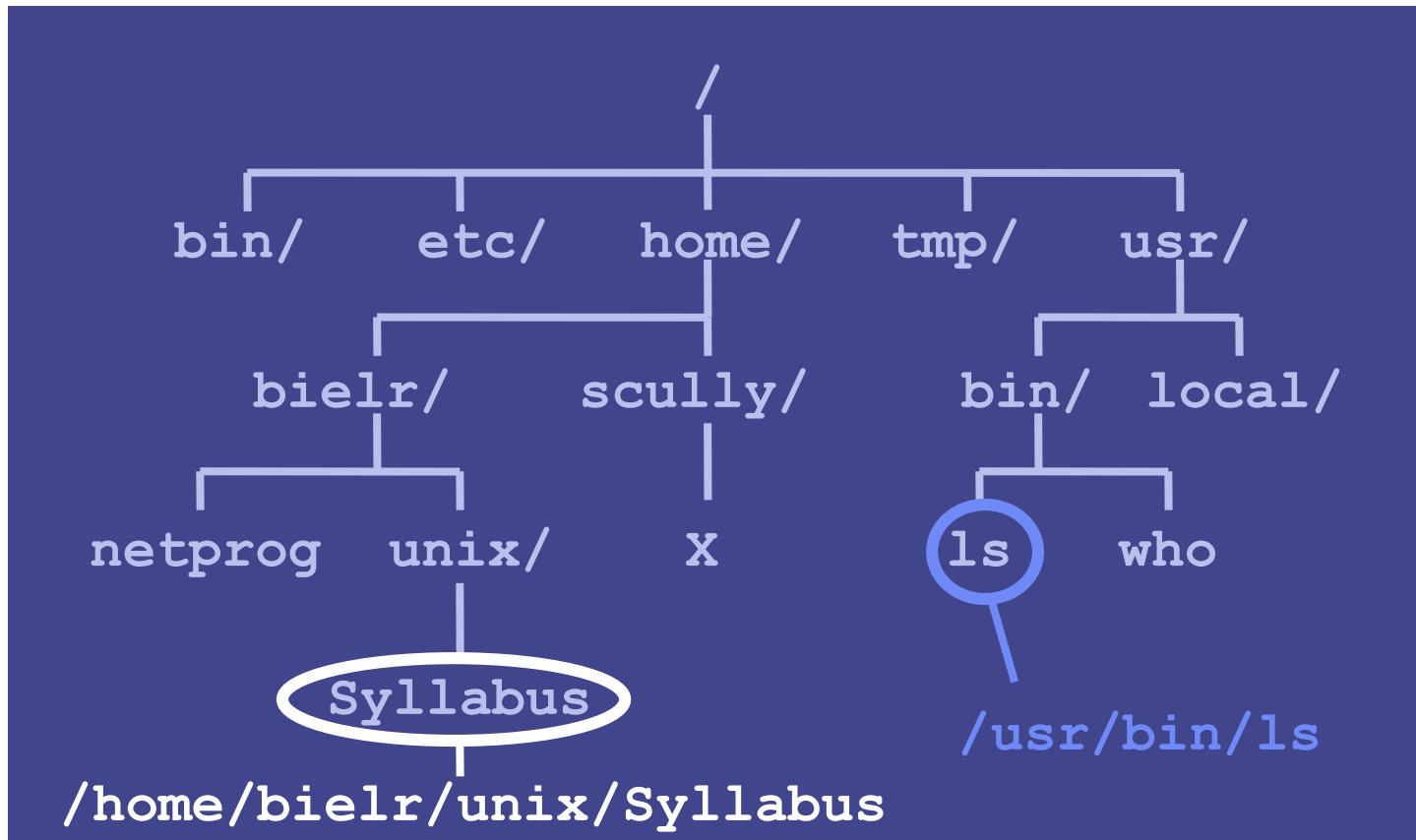
File Time Attributes

- Time Attributes:
 - when the file was last changed **ls -l**
 - sort by modification time **ls -lt**

Unix Filesystem

- The filesystem is a hierarchical system of organizing files and directories.
- The top level in the hierarchy is called the "root" and holds *all* files and directories in the filesystem.
- The **name** of the root directory is **/**

Pathname Examples



/home/bielr/unix/Syllabus is the pathname

Pathnames

- The *pathname* of a file includes the file name and the name of the directory that holds the file, and the name of the directory that holds the directory that holds the file, and the name of the ... up to the root.
- The pathname of every file in a given *filesystem* is unique.

Pathnames (cont.)

- To create a pathname you start at the root (so you start with "/"), then follow the path down the hierarchy (including each directory name) and you end with the filename.
- In between every directory name, we use a delimiter of "/".

Absolute Pathnames

- The pathnames described in the previous slides start at the *root*.
- These pathnames are called "absolute pathnames".

Relative Pathnames

Prefixed w/the current directory, \$PWD
So, **relative** to the current working directory

```
$ cd /home/bielr
```

```
$ pwd
```

/home/bielr *(current working dir)*

```
$ ls unix/Syllabus (relative pathname)
```

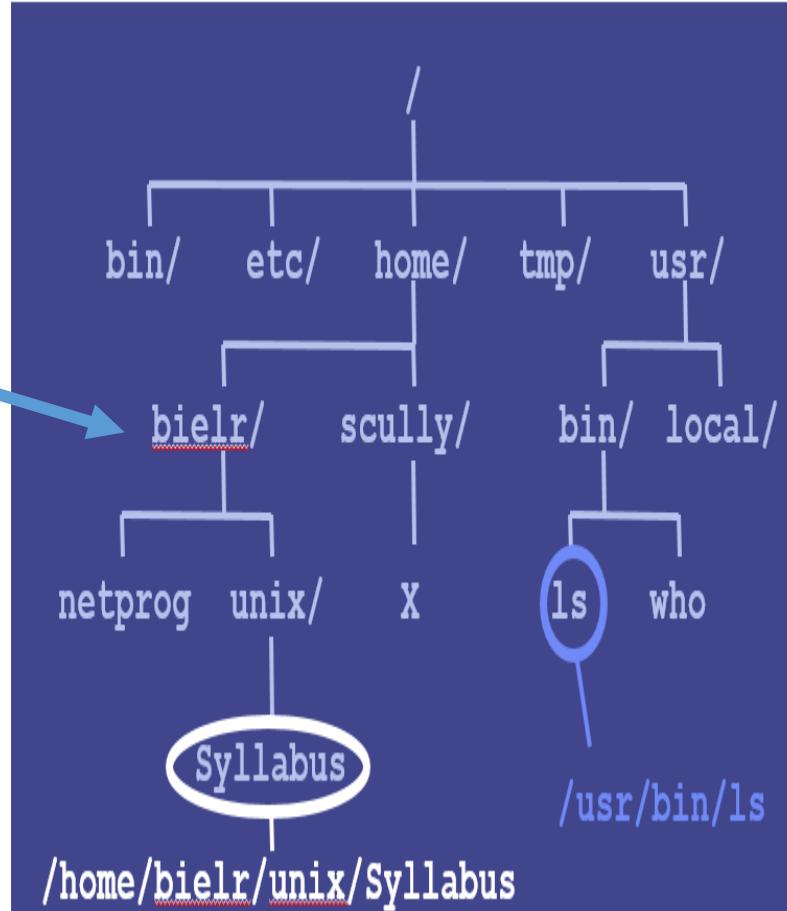
unix/Syllabus

```
$ ls X
```

ls: X: No such file or directory (?)

```
$ ls /home/scully/X
```

/home/scully/X *(found it!)*



Special Relative paths...

- . The current directory
- .. The *parent* directory

\$ **pwd**

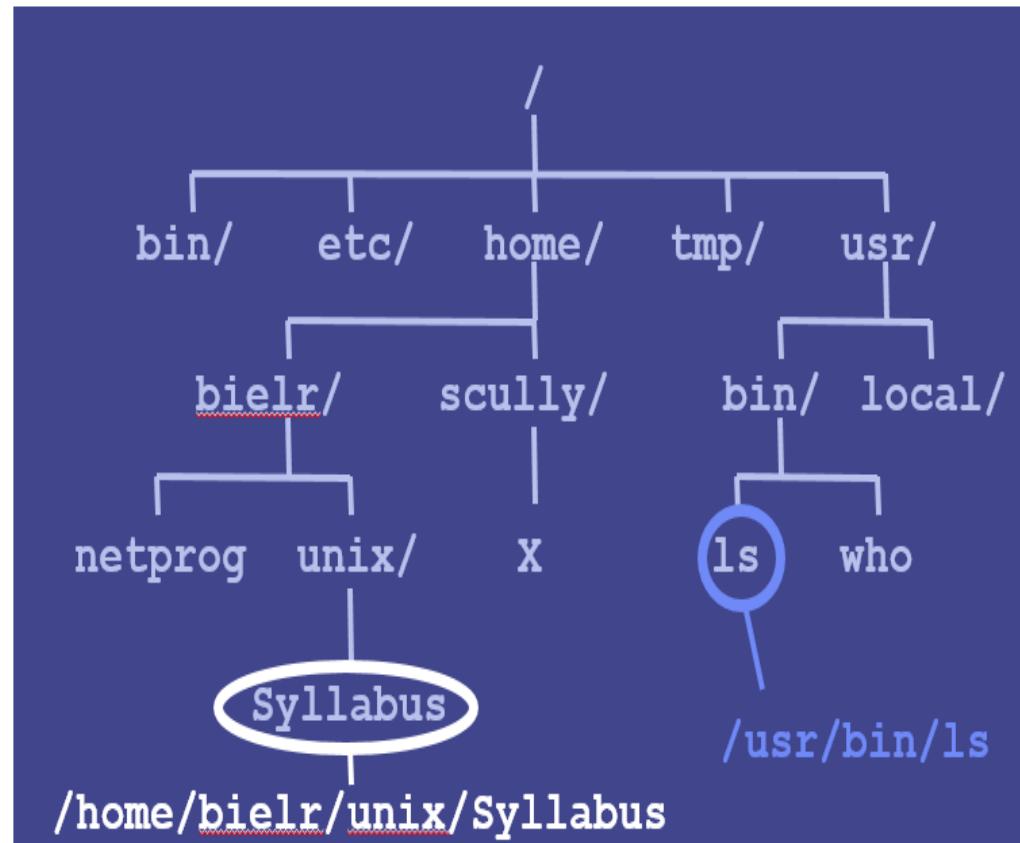
/home/bielr

\$ **ls ./netprog**

./netprog

\$ **ls/scully**

X



Some Standard Directories & Files 1 of 2

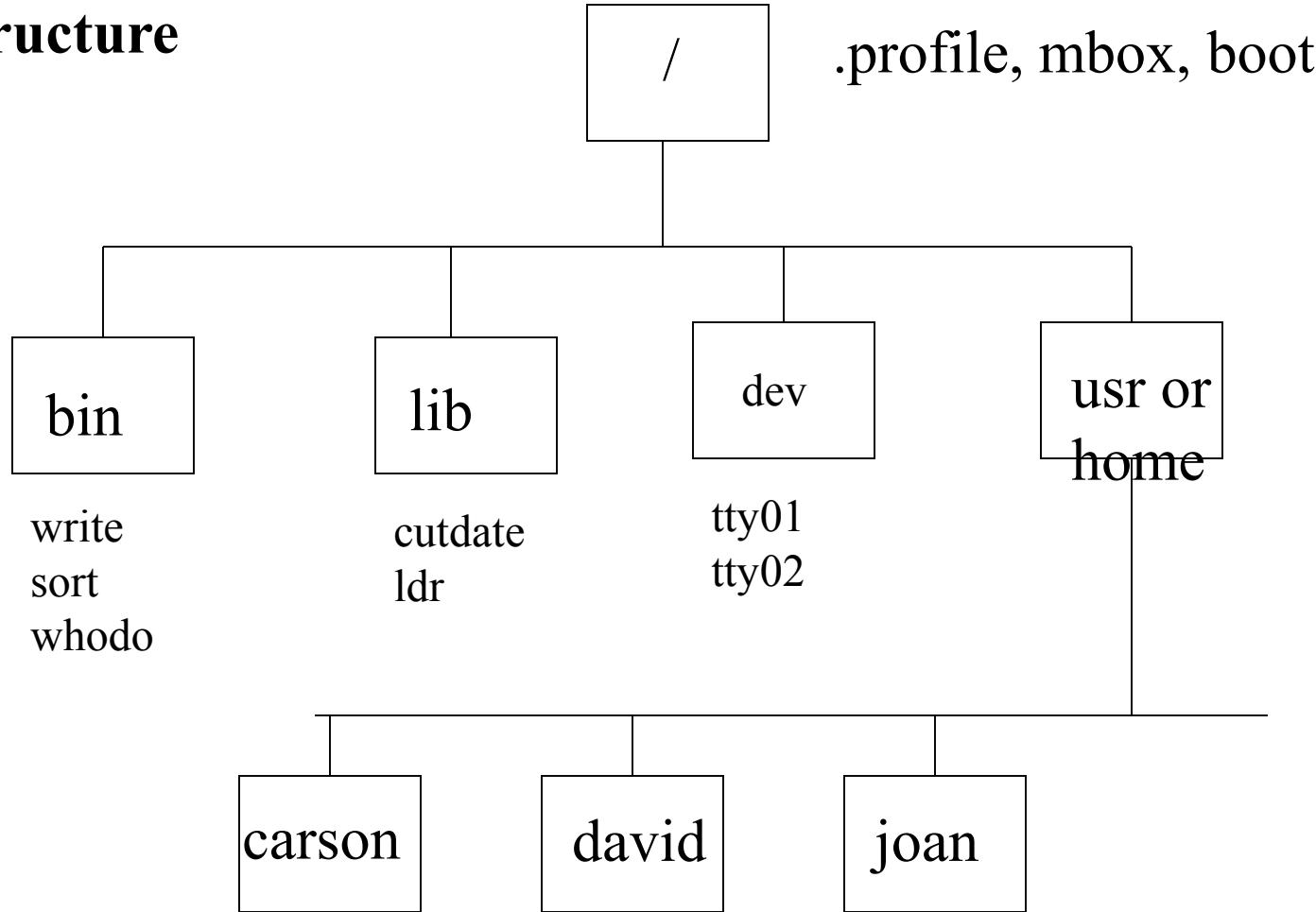
- **Root Directory (/).** The top of the file system
- **/bin.** The binary directory. Contains binary (executable) images of most UNIX/Linux commands.
- **/dev.** The device directory. Has files corresponding to all the devices connected to the computer.
- **/etc.** Contains commands and file for system administration. The typical user is usually not allowed to use these commands and files.
- **/lib.** The library directory. Contains a collection of related files for a given language in a single file called **archive**. Many UNIX/Linux systems contain libraries for C, C++, and FORTRAN.

Some Standard Directories & Files 2 of 2

- **/tmp.** Contains temporary files. The system admin determines the length of their life, usually only a few minutes.
- **/users.** The home directory for all the users on a system.
- **/usr.** UNIX System Resource. Contains subdirectories: utilities, tools, language libraries, manual pages.

Directories & Files

LINUX directory structure



Each of these user then have their own files.

The **root** directory is always named “/”.

Parent directory is the one directly above another directory.

Child directory is a directory directly below another directory.

Sub-directory – another name for a **child** directory.

When you log on, you are put into your **home** directory.

```
>pwd      /* path of the working directory */
```

*Result of a **pwd** in my own directory:*

```
[bielr@athena ~]24> pwd  
/gaia/home/faculty/bielr/csc60
```

Notice the full path name, starting with “/” for the root.

```
[doej@athena/21> pwd  
/gaia/class/student/doej
```

To return to your home directory:

```
> cd /* Change Directory */
```

To go to a sub-directory:

```
> cd directory-name
```

From my home directory, to get to the sub-directory:

```
> cd csc60
```

An alternate to **cd** is **chdir**.

Forming File Names:

All UNIX/Linux systems can handle file names of up to 14 characters. Some can use names as long as 256.

Names are formed from:

A to Z

a to z

0 to 9

_ (underscore)

. (period)

, (comma)

NO slash. Better not to use Space or Dash.

Creating a Directory:

mkdir

Example:

> **mkdir csc60**

Renaming Directories:

You must first be in the parent directory.

>mv *original-name new-name*

Removing Directories:

>rmdir *directory-name*

Ambiguous File Names:

- ? Represents any other character
- * Represents no character or any number of characters.

Examples:

- *a All files with names ending in a
- *[xyz] All files with names ending in x, y, or z.
- *.? All files that contain a period with exactly one character following.
- ?? All files with two-character names.
- *.obj All files with .obj as the last four characters.

3-Unix History, Files, Paths

The End

4-UNIX Tools

grep Command – UNIX

The grep command is used to search text or searches the given file for lines containing a match to the given strings or words.

By default, grep displays the matching lines.

Use grep to search for lines of text that match one or many regular expressions, and outputs only the matching lines.

grep is considered as one of the most useful commands on Linux and Unix-like operating systems.

Simple Syntax: grep “word” filename

Sample Use: grep boo /etc/passwd

chmod Command – UNIX

chmod - change the access permissions
to file system objects like files and directories.

Syntax: chmod *options permissions filename*

Two Examples:

chmod 754 myfile

chmod u-rwx,g-rx,o-r myfile

(The two commands are equivalent.)

diff Command – UNIX

diff - report the difference between 2 text files
compare files line by line

Syntax: **diff *options* filenames**

Example: **diff file1 file2**

find Command - UNIX

find search for files in a directory hierarchy

Syntax: *find options path expressions*

Example: **find *4***

Would find all the filenames in the current directory that contain the character “4”.

finger Command – UNIX

finger displays information about the system users.

Example: **finger -s**

-s displays the user's login name, real name, terminal name and write status (the asterisk before terminal name mean that you don't have write permission with that device), idle time, login time, office location and office phone number.

CSC-60

More about vi

Simple vi editing commands

While in Command mode, type:

- r** replace one character under the cursor
- x** delete 1 character under the cursor.
- 2x** delete 2 characters (3x, etc.)
- u** undo the last change to the file

Cutting text in Vi

d^

Deletes from current cursor position to the beginning of the line

d\$

Deletes from current cursor position to the end of the line

dw

Deletes from current cursor position to the end of the word

dd

Deletes one line from current cursor position.
Specify count to delete many lines.

Cutting & Yanking Text in Vi

While in Command mode, type:

dd Delete (cut) 1 line from current cursor position

2dd Delete (cut) 2 lines (***3dd*** to cut 3 lines, etc.)

p paste lines below current line

Cutting & Yanking Text in Vi

yy yank (copy) a single line

2yy yank (copy) 2 lines (**3yy** to copy 3 lines, etc.)

P paste lines before current line

Vi Editor

To go to a specific line in the file

:linenumber

Examples (in Command Mode):

1. Go to the 3rd line by typing :3
2. Go to the 1st line by typing :1
3. Go to the last line by typing G

Notes:

- (1) **set number** (to set line numbers)
- (2) control-f/b move forward (one page)/backward (one page)

Vi string/search

/[pattern] search forward for the pattern

?[pattern] search backward for the pattern

n search for the next instance of a string

Examples:

1. Search forward for the next line containing the word “printf” by typing **/printf**
2. Search forward for the next instance of **printf** by typing **n**
3. Search backward for the most recent instance of **printf** by typing **?printf**
4. Search backward for the next most recent instance of **printf** by typing **n**

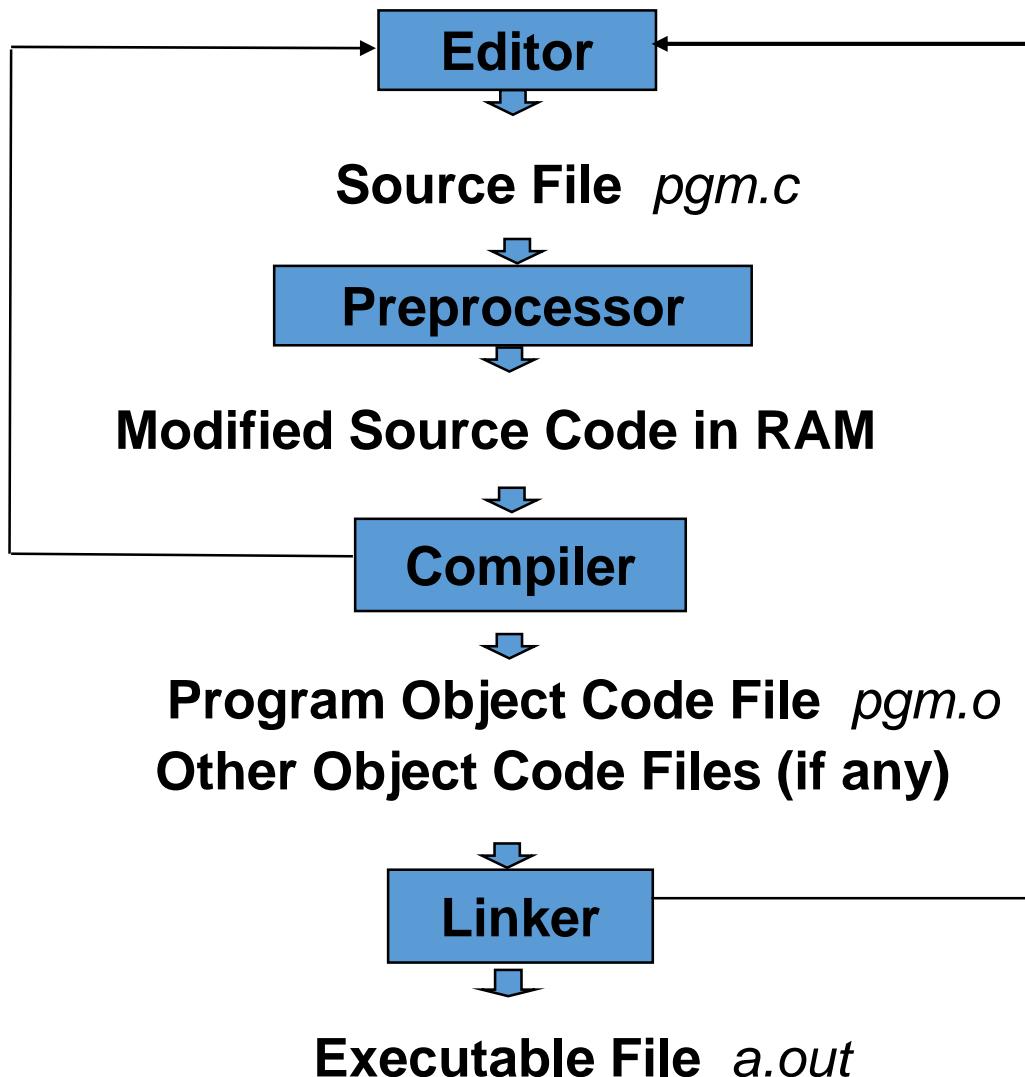
CSC-60

Compilation

What is gcc?

- **gcc** is the GNU Project C compiler
- A command-line program
- **gcc** takes C source files as input
- Outputs an executable: **a.out**
 - You can specify a different output filename
- Note: Although we call this process “compiling a program,” what actually happens is more complicated.

Program Development Using gcc



1 of 3 Stages of Compilation

Stage 1: Preprocessing

- Performed by a program called the **preprocessor**
- Modifies the source code (in RAM) according to **preprocessor directives (preprocessor commands)** embedded in the source code
- Strips comments and white space from the code
- The source code as stored on disk is not modified.

2 of 3 Stages of Compilation

Stage 2: Compilation

- Performed by a program called the **compiler**
- Translates the preprocessor-modified source code into **object code (machine code)**
- Checks for **syntax errors** and **warnings**
- Saves the object code to a disk file, if instructed to do so
- If any compiler errors are received, no object code file will be generated.
 - o An object code file will be generated if only warnings, not errors, are received.

3 of 3 Stages of Compilation

Stage 3: Linking

- Combines the program object code with other object code to produce the executable file.
- The other object code can come from the **Run-Time Library**, other libraries, or object files that you have created.
- Saves the executable code to a disk file. On the Linux system, that file is called **a.out**.
 - If any linker errors are received, no executable file will be generated.

Gcc example:

“hello.c” - the name of the file with the following contents

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

Gcc example:

- To compile simply type: ***gcc -o HelloProg hello.c -g -Wall***
- **-o** - option tells the compiler to name the executable ‘HelloProg’
- **-g** - option adds symbolic information to Hello for debugging
(note on **-g3** option - Level 3 includes extra information, such as all the **macro** definitions)
- **-Wall** - tells it to print out all warnings (very useful!!!)
- To execute the program simply type:
./HelloProg or HelloProg
- It should output “Hello” on the console

CSC-60

Debugging

What is **gdb**?

- **gdb** is the GNU Project debugger
- **gdb** provides some helpful functionality
 - Allows you to stop your program at any given point.
 - You can examine the state of your program when it's stopped.
 - Change things in your program, so you can experiment with correcting the effects of a bug.
- Also a command-line program

Using gdb:

- Compile with the `-g` flag to set up for debugging
- To start gdb with your hello program type:

gdb HelloProg

- When gdb starts, your program is not actually running.
- Before you do try to run, you should place some break points.
- To start execution, you have to use the ***run*** command.
Once you hit a break point, you can examine any variable.

Useful gdb commands

break *place*

place can be the name of a function or a line number

For example: **break main** will stop execution at the first instruction of your program

run *command-line-arguments*

Begin execution of your program with arguments

delete *N*

Removes breakpoints, where *N* is the number of the breakpoint

step

Executes current instruction and stops on the next one

Gdb commands cont.

next

Same as **step** except this doesn't step into functions

print *E*

Prints the value of any variable in your program when you are at a breakpoint, where *E* is the name of the variable you want to print

`print/x var` (i.e `p/x S_IFREG` where `x` is the hex value), other options include: `d` (decimal), `o` (octal), `t`(two - binary), etc.

help *command*

Gives you more information about any command or all if you leave out command

quit

When time to exit gdb

4-UNIX Tools

The End

5_UNIX

The *make* Tool/Command

The reasons for the *make* utility

In a large project, one doesn't want to re-compile *everything*, every time a change is made.

make keeps track of dependencies,
of what needs re-compiling,
of what needs re-linking.

Syntax: `make [-f makefile]`

Most programmers use a standard output name.

Example:

`> make -f Makefile`

Often Used Options:

- f Tells **make** which file to use as its makefile
*Without -f, it looks first for *makefile* and then for *Makefile* by default.*
- n Tells **make** to print out what it would have done without actually doing it.
- k Tells **make** to keep going when an error is found, rather than stopping as soon as the first problem is detected.

A simple code file & its make file

```
/* power.c      */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main(void)
{
    float x, y;

    printf("\nThe program takes x and y from stdin and displays x^y.\n");

    printf("Enter integer x: ");
    scanf ("%f", &x);
    printf("Enter integer y: ");
    scanf ("%f", &y);

    printf("x^y is: %6.3f\n", pow((double)x,(double)y));
    return EXIT_SUCCESS;
}
/* The RUN is on the next slide */
```

```
[bielr@athena ~]30> gcc power.c -o power  
/tmp/ccEzY1AX.o(.text+0x83): In function `main':  
: undefined reference to `pow'  
collect2: ld returned 1 exit status  
[bielr@athena ~]31>
```

Forgot to add... -lm ...to link to the math library.

```
[bielr@athena ~]38> gcc power.c -lm -o power  
[bielr@athena ~]39> power
```

The program takes x and y from stdin and displays x^y .

Enter integer x: 9.82

Enter integer y: 2.3

x^y is: 191.362

```
[bielr@athena ~]40>
```

A makefile for the power program:

>vim makefile

```
# Sample makefile for the power program
# Remember: each command line starts with a TAB

power: power.c
    gcc power.c -o power -lm
```

Contents of the file
named *makefile*

>

```
[bielr@athena ~]40> make
```

```
make: `power' is up to date.
```

```
[bielr@athena ~]41>
```

Use touch to alter the dates to force recompilation

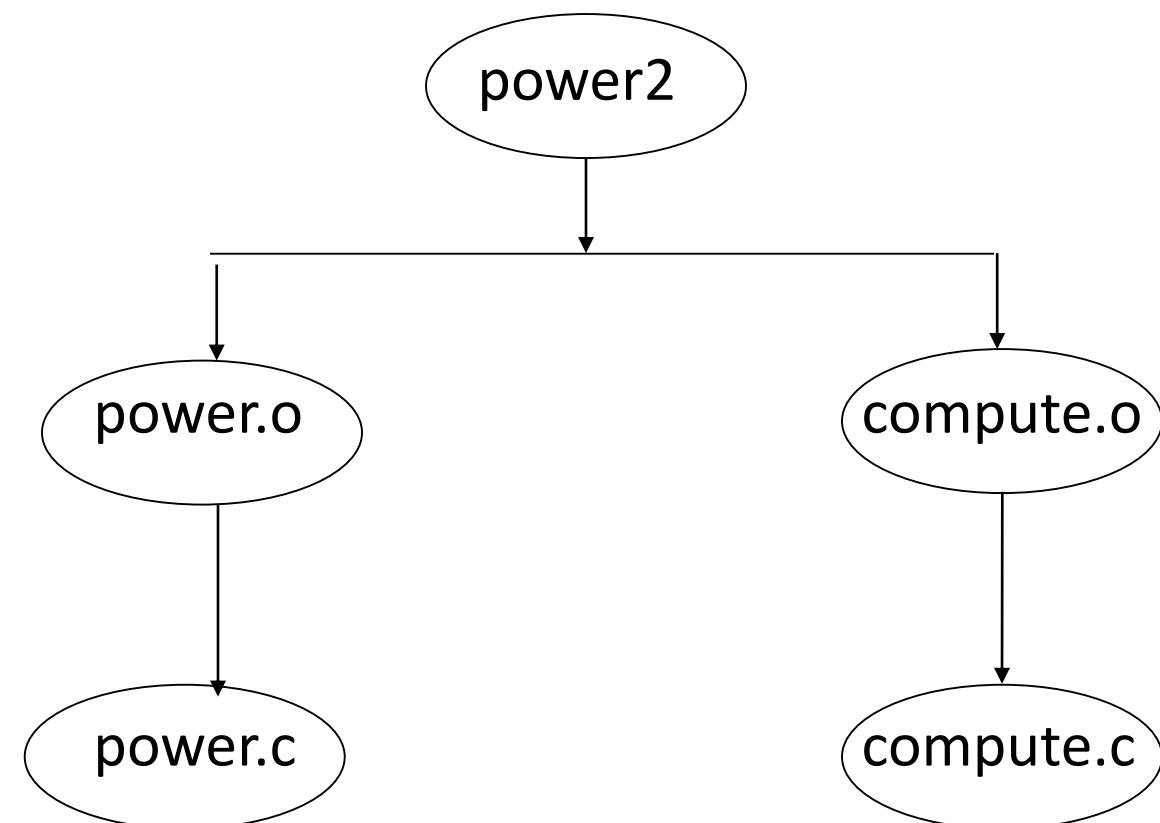
```
[bielr@athena ~]44> touch power.c
```

```
[bielr@athena ~]45> make
```

```
gcc power.c -o power -lm
```

```
[bielr@athena ~]46>
```

Dependency Chart: Alter ***power*** so it is in two functions, two files.



/* power2.c

*Alter power so it is in two functions. */
/* page 1 of 2 */*

```
#include <stdio.h>
#include <stdlib.h>
double compute(double x, double y);
int main(void)
{
    float x, y;

    printf("\nThe program takes x and y from stdin and displays x^y.\n");
    printf("Enter integer x: ");
    scanf ("%f", &x);

    printf("Enter integer y: ");
    scanf ("%f", &y);

    printf("\nx^y is: %6.3f\n\n", compute(x,y));
    return EXIT_SUCCESS;
}
```

```
/* The new function  page 2 of 2 */  
/* compute.c */
```

```
#include <math.h>  
double compute(double x, double y)  
{  
    return (pow(x, y));  
}
```

First pass at a makefile:

```
>cat makefile
power: power.o compute.o
        gcc power.o compute.o -o power2 -lm
>
```

```
[bielr@athena ~]60> make
make: *** No rule to make target `power.o', needed by `power'. Stop.
[bielr@athena ~]61>
```

```
/* Second pass at a makefile: */  
/* Look at its contents. We have no p2.h but it is included in light italics  
to show where it would be placed. */
```

```
>cat makefile
```

```
power2: power2.o compute.o p2.h  
        gcc power2.o compute.o -o power2 -lm  
power2.o: power2.c p2.h  
        gcc -c power2.c  
compute.o: compute.c p2.h  
        gcc -c compute.c
```

```
/* Run make using our new makefile */
```

```
[bielr@athena ~]/csc60]68> make  
gcc -c power2.c  
gcc -c compute.c  
gcc power2.o compute.o -o power2 -lm  
[bielr@athena ~]/csc60]69>
```

```
# Third and last pass at a makefile:  
#   power2.h is not needed but included in light italics to  
#   show where it would be located if it was needed
```

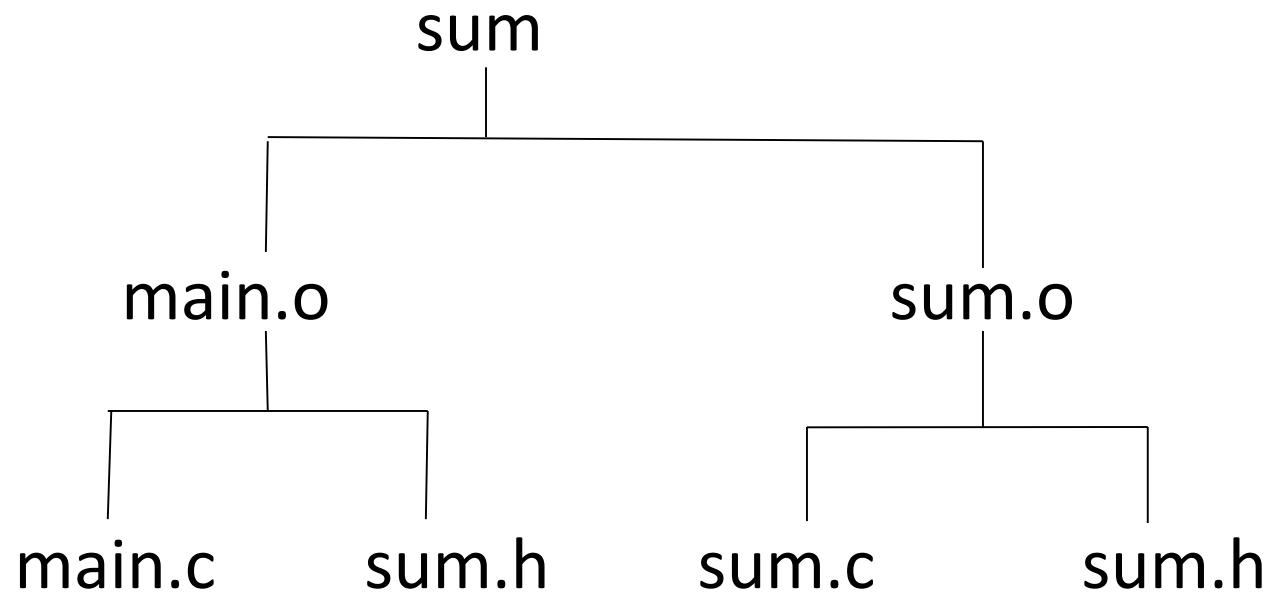
```
# >cat makefile  
power2: power2.o compute.o power2.h  
        gcc power2.o compute.o -o power2 -lm  
  
power2.o: power2.h  
  
compute.o: power2.h  
>
```

/ Helpful Comments */*

- Start by opening *vim*, and typing in the commands to a file named *makefile*.
Close *vim* and then at the prompt, type: *make*
- When you enter **vim**, type: **:set list**
This will show the non-printable characters:
 $\wedge I$ = tab
 $\$$ = end of line
- To create a tab on *athena*, you may have to hit the tab key **twice** in a row.

Another Example

Example with two functions and a *.h file:



Makefile Contents:

sum: main.o sum.o	<i>dependency</i>
gcc -o sum main.o sum.o	<i>action</i>
main.o: main.c sum.h	<i>dependency</i>
gcc -c main.c	<i>action</i>
sum.o: sum.c sum.h	<i>dependency</i>
gcc -c sum.c	<i>action</i>

Dependency lines start in column 1.

Action lines must begin with a **tab**.

If anything on a dependency line has changed, then the associated action(s) take place.

A dependency and its actions together are called a **rule**.

An alternate example of this makefile:

sum: main.o sum.o	<i>dependency</i>
gcc -o sum main.o sum.o	<i>action</i>
main.o: sum.h	<i>dependency</i>
gcc -c main.c	<i>action</i>
sum.o: sum.h	<i>dependency</i>
gcc -c sum.c	<i>action</i>

A makefile with a macro:

```
sum: main.o sum.o          dependency  
      gcc -o sum main.o sum.o    action
```

```
main.o sum.o: sum.h        dependency  
      gcc -c $*.c             action
```

*The second rule states that the two .o files depend on sum.h.
If we edit sum.h, both main.o and sum.o must be remade.*

The macro \$.c expands first to main.c and then to sum.c*

Macros in a Makefile

The *make* utility supports simple macros that allow simple text substitution.

The macro must be defined before use, and is usually placed at start of the file.

Syntax: Macro_name = text

These are also called Macro Variables.

A more complicated example

The Compare_Sorts makefile in its entirety:

(Dissection of each line follows on the next slides.)

Makefile for compare_sorts

#After excution, use prof to get a profile.

BASE = /c/c/blufox

CC = gcc

CFLAGS = -p

EFILE = \$(BASE)/bin/compare_sorts

INCLS = -I\$(BASE)/include

LIBS = \$(BASE)/lib/g_lib.a

OBJS = main.o chk_arrays.o compare.o \
prn_arrays.o slow_sort.o

\$(EFILE): \$(OBJS)

 @echo “linking.”

 @\$ \$(CC) \$(CFLAGS) \$(INCLS) -c \$*.c

\$(OBJS): compare_sorts.h

 \$(CC) \$(CFLAGS) \$(INCLS) -c \$*.c

```
# Makefile for compare_sorts  
#After execution, use prof to get a profile.
```

Comments start with # and go till end of line.

`BASE = /c/c/blufox`

a macro definition.

Syntax: `macro_name = replacement_string`

BASE represents our base of operations on the local computer.

Doesn't have to be home directory.

CC = gcc

The CC macro specifies the C compiler we are using.

CFLAGS = -p

*The CFLAGS macro specifies the options, if any, that will be used with the **gcc** command.*

-p = Generate extra code to write profile information.
You must use this option when compiling the source files you want data about, and you must also use it when linking.

What is profiling?

Profiling is an important aspect of software programming. Through profiling one can determine the parts in program code that are time consuming and need to be re-written. This helps make your program execution faster which is always desired.

In very large projects, profiling can save your day by not only determining the parts in your program which are slower in execution than expected but also can help you find many other statistics through which many potential bugs can be spotted and sorted out.

EFILE = \$(BASE)/bin/compare_sorts

Specifies the executable file.

INCLS = -I\$(BASE)/include

*Specifies a directory for include files proceeded by the **-I** option.*

-I dir, --include-dir=dir

*Specifies a directory dir to search for included makefiles. If several **-I** options are used to specify several directories, the directories are searched in the order specified. Unlike the arguments to other flags of make, directories given with **-I** flags may come directly after the flag: **-Idir** is allowed, as well as **-I dir**.*

*This syntax is allowed for compatibility with the C preprocessor's **-I** flag.*

LIBS = \$(BASE)/lib/g_lib.a

Tells the compiler where to find our programmer-constructed header file.

```
OBJS      = main.o chk_arrays.o compare.o \
             prn_arrays.o slow_sort.o
```

In this macro definition the replacement string is the list of object files that occurs on the right side of the equal sign.

(Used backslash \ to continue the line.)

Order is unimportant.

```
$(EFILE): $(OBJS)
    @echo "linking. . . ."
    @$(CC) $(CFLAGS) $(INCLS) -c $*.c
```

The first line is a dependency line, and the other two specify the actions to be taken.

The @ symbol means that the action line itself is not to be echoed on the screen.

Macro invocation has form:

`$(macro_name)`

So \$(EFILE) is replace by \$(BASE)/bin/compare_sorts

which then becomes /c/c/blufox/bin/compare_sorts

```
$(OBJS): compare_sorts.h  
        $(CC) $(CFLAGS) $(INCLS) -c $*.c
```

\$(OBJS) will be replace by the list of object files.

The second action line is expanded to:

```
@gcc -p -o /c/c/blufox/bin/compare_sorts main.o \
    chk_arrays.o compare.o prn_array.o slow_sort.o \
    /c/c/blufox/lib/g_lib.a
```

Because of the backslash \, the line acts as one line.

-p causes the compiler to generate extra code suitable for the profiler.

More details about ***make*** (these expand just before use):

\$? List of prerequisites changed more recently than the current target

\$@ Name of the current target

\$< Name of the current prerequisite

\$* Name of the current prerequisite, without any suffix

- Tells ***make*** to ignore any errors.

@ Tells ***make*** not to print the command to standard output before executing it.

The ***touch*** command:

Syntax: `touch [options] files`

Changes two timestamps associated with a file:

- its *modification time* (when the file's data was last changed)
- its *access time* (when the file was last read)

If a given file doesn't exist, *touch* creates it as an empty file.

If we had done a *touch* on *compare_sorts.h*,
we would have forced all the other files to be recompiled.

Other Useful Tools:

diff Prints the lines that differ in two files.

wc Word Count. Counts lines, words, and characters in one or more files.

awk A pattern scanning and processing language.
On Linux, it is ***gawk***, a superset of *awk*.

lex Generates C code for lexical analysis.
On Linux, it is ***flex***, a superset of *lex*.

sed A stream editor that takes its commands from a file.

yacc A parser generator.

bison GNU Project parser generator (yacc replacement)

5_UNIX

The *make* Tool/Command

The End

6-UNIX System Calls & Start of mini-shell

(The Linux Program Interface
our text **Chapter 3**)

System Calls

Operating systems offer processes running in User Mode *a set of interfaces* to interact with *hardware devices* such as the **CPU**, disks, and printers.

Unix systems implement most interfaces between *User Mode processes* and *hardware devices* by means of *system calls* issued to the kernel.

A *system call* is an explicit request to the kernel made via a *software interrupt*.

System Calls

You can see the whole list of system calls by typing:

```
> man systemcalls
```

From a programming point of view, invoking a system call looks much like calling a C function.

However behind the scenes, many steps occur!

System Call Examples

- Create a new process
- Perform I/O
- Create a pipe for interprocess communications
- Send/Receive signals

General Points about System Calls

- It changes the processor state from user mode to kernel mode, so that the CPU can access protected kernel memory
- The set of system calls is fixed
 - Each call has a unique number
 - Programmer identifies the call by name
- It may have a set of arguments that specify information to be transferred from user space to kernel and vice versa

What is a wrapper function?

- A **wrapper function** is a subroutine in a software library or a computer program
- Main purpose is to call a second subroutine or a system call with little or no additional computation
- It provides an interface (prototype) to the outside.
- It provides a mapping from the interface to the actual system call on the inside.

From a Wrapper Routine to a System Call

- Unix systems include several *libraries of functions* that provide APIs to programmers.
 - API = Application Program Interface
- Some of the APIs defined by the **libc** standard C library refer to *wrapper routines* (routines whose only purpose is to issue a *system call*).
- Usually, each system call has a corresponding wrapper routine, which defines the API that application programs should employ.

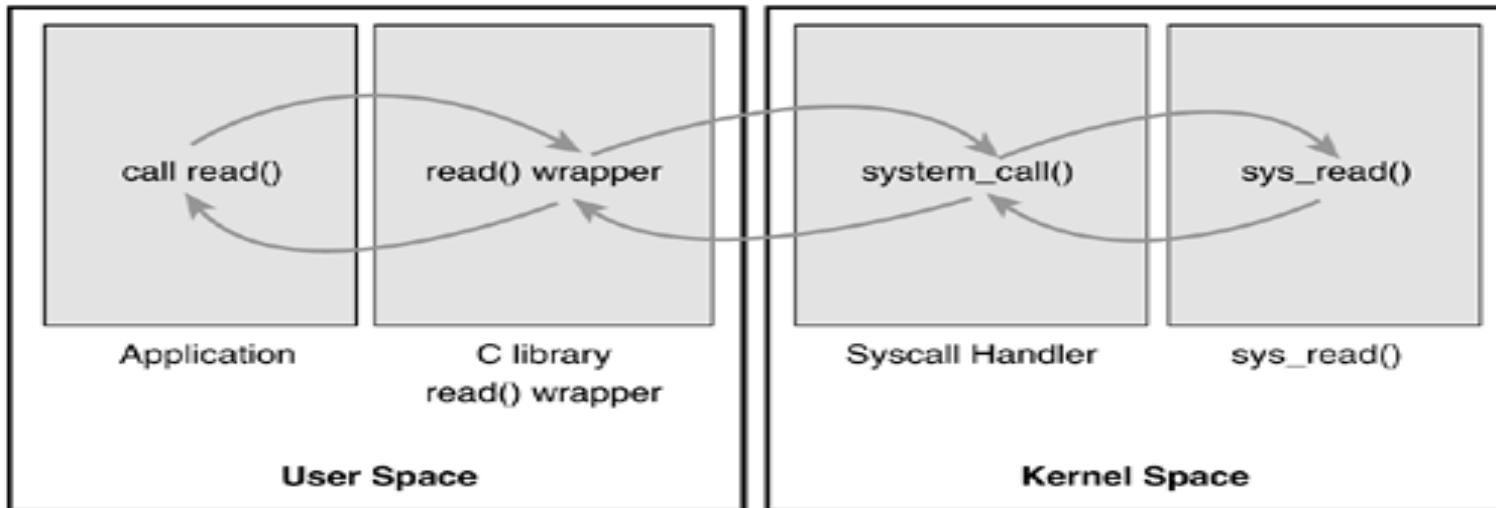
The Return Value of a Wrapper Routine

- Most wrapper routines return an integer value, whose meaning depends on the corresponding system call.
- A return value of **-1** usually indicates that the ***kernel*** was unable to satisfy the process request.
- A failure in the ***system call handler*** may be caused by
 - invalid parameters
 - a lack of available resources
 - hardware problems, and so on.
- The specific ***error code*** is contained in the ***errno*** variable, which is defined in the ***libc*** library.

Execution Flow of a System Call

- When a **User Mode process** invokes a **system call**, the **CPU** switches to **Kernel Mode** and starts the execution of a **kernel function**.

It is a jump to an assembly language function called the **system call handler**.



System Call Number

- Because the kernel implements many different system calls, the User Mode process must pass a parameter called the ***system call number*** to identify the required system call.
- The **eax** register is used by Linux for this purpose.
 - Additional parameters are usually passed when invoking a system call.
 - See: On Athena, `/usr/include/asm/unistd_32.h` for listing of system calls (32 bits)

The Return Value of a System Call

- All system calls return an integer value.
- The conventions for these return values are different from those for wrapper routines.
 - In the kernel
 - positive or 0 values denote a successful termination of the system call
 - negative values denote an error condition
 - In the latter case, the value is the negation of the error code that must be returned to the application program in the **errno** variable.
 - The **errno** variable is not set or used by the kernel. Instead, the wrapper routines handle the task of setting this variable after a return from a system call.

Operations Performed by a System Call Handler

- The ***system call handler*** performs the following operations:
 - Saves the contents of most registers in the Kernel Mode stack.
 - This operation is common to all system calls and is coded in assembly language.
 - Handles the system call by invoking a corresponding C function called the ***system call service routine***.
 - Exits from the handler:
 - The registers are loaded with the values saved in the Kernel Mode stack
 - The **CPU** is switched back from Kernel Mode to User Mode.
 - This operation is common to all system calls and is coded in assembly language.

Tracing System Calls – **strace** command (See Appendix A in LPI, page 1401)

\$ **strace** *command arg...*

This runs *command*, with the given command-line arguments, producing a trace of the system calls it makes. By default, *strace* writes its output to *stderr*, but we can change this using the *-o filename* option.

Examples of the type of output produced by *strace* include the following (taken from the output of the command *strace date*):

```
execve("/bin/date", ["date"], /* 114 vars */) = 0
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)       = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=111059, ...}) = 0
mmap2(NULL, 111059, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7f38000
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)          = 3
fstat64(3, {st_mode=S_IFREG|0755, st_size=1491141, ...}) = 0
close(3)                                = 0
write(1, "Mon Jan 17 12:14:24 CET 2011\n", 29) = 29
exit_group(0)                            = ?
```

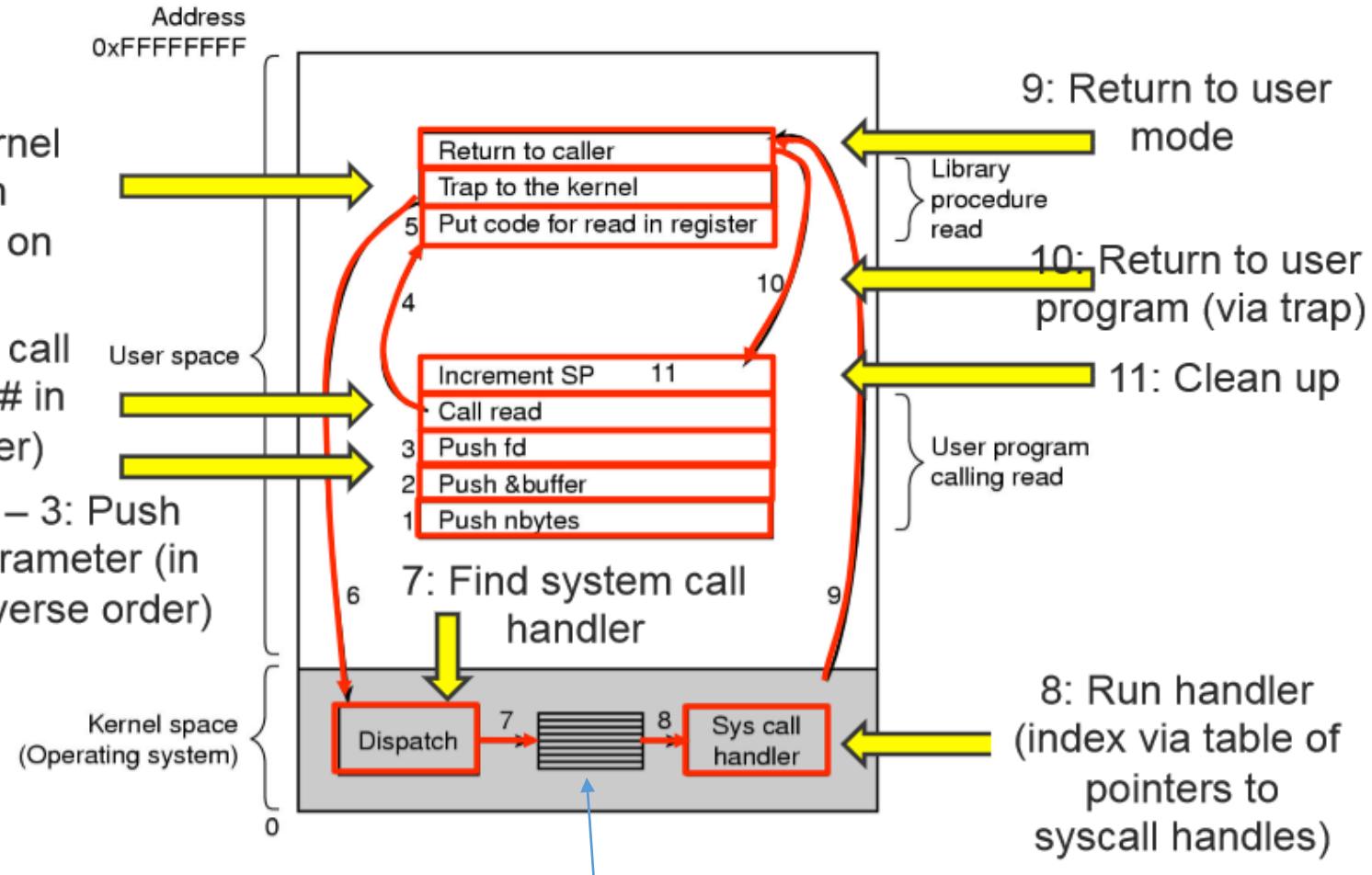
Steps for Making a System Call (Example: read call)

6: Switch to kernel mode (return address saved on stack)

4 – 5: Library call (puts syscall # in CPU register)

1 – 3: Push parameter (in reverse order)

`count = read(fd, buffer, nbytes);`



Wrapper:

- Move parameters from the user stack to processor registers
- Switch to kernel mode and jump to the system call handler
- Post-process the return value and compute errno

6: Switch to kernel mode (return address saved on stack)

4 – 5: Library call (puts syscall # in CPU register)

1 – 3: Push parameter (in reverse order)

C program:

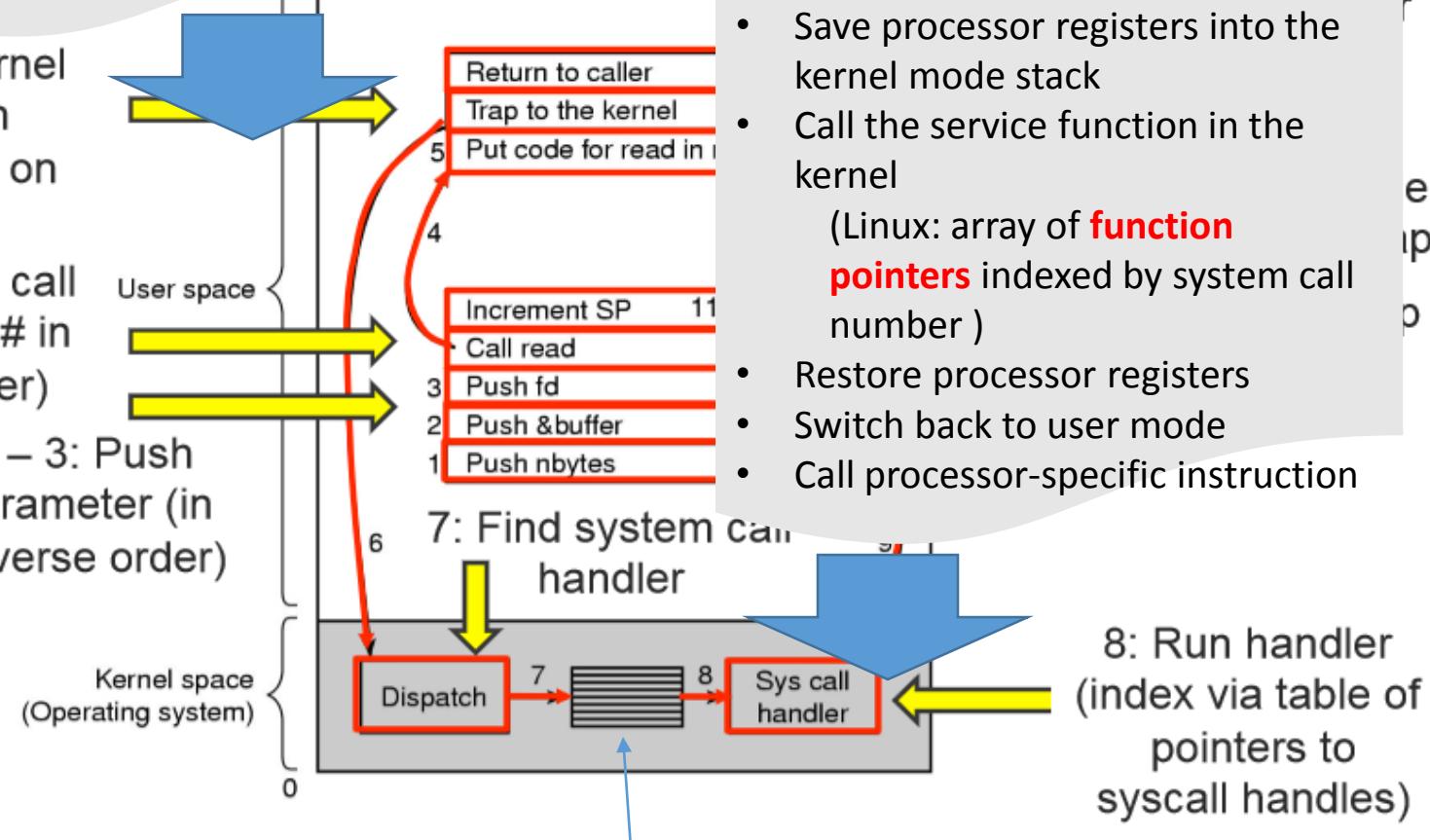
• `count = read(fd, buffer, nbytes);`

e: read(fd, buffer, nbytes)

`(fd, buffer, nbytes);`

System Call Handler:

- Save processor registers into the kernel mode stack
- Call the service function in the kernel
(Linux: array of **function pointers** indexed by system call number)
- Restore processor registers
- Switch back to user mode
- Call processor-specific instruction



6-UNIX System Calls & Start of mini-shell

The End

(The Linux Program Interface
our text **Chapter 3**)

7-UNIX

Environment



User and Group ID

Material from Chapter 8, Users and Groups

Material from Chapter 9, Process Credentials

Material from Chapter 15, File Attributes

User and Group ID

GID – Real Group ID.

Users can belong to one or more groups.

UID – Real User ID.

Identifies the user who is responsible for the running process.

EGID – Effective Group ID. The ID that matters.

EUID - Effective User ID. The ID that matters.

A user with effective user ID of zero has all the privileges of the superuser.

It is called a *privileged process*.

Effective User ID - *euid*

Used to assign ownership of newly created files, to check file access permissions, and to check permission to send signals to processes.

To change *euid*:

- executes a setuid-program that has the set-uid bit set
- or invokes the setuid() system call.

The **setuid(uid)** system call:

if euid is not superuser,

- uid must be the real uid
- or the saved uid (the kernel also resets euid to uid).

Real and effective uid: inherit (fork), maintain (exec).

Real ID Functions

`pid_t getuid(void);`

Returns the real user ID of the current process

`pid_t geteuid(void);`

Returns the effective user ID of the current process

`gid_t getgid(void);`

Returns the real group ID of the current process

`gid_t getegid(void);`

Returns the effective group ID of the current process

Change UID and GID (1)

```
#include <unistd.h>
#include <sys/types.h>
int setuid( uid_t uid )
int setgid( gid_t gid )
```

Sets the effective user ID of the current process.

Superuser process resets the real effective user IDs to *uid*.

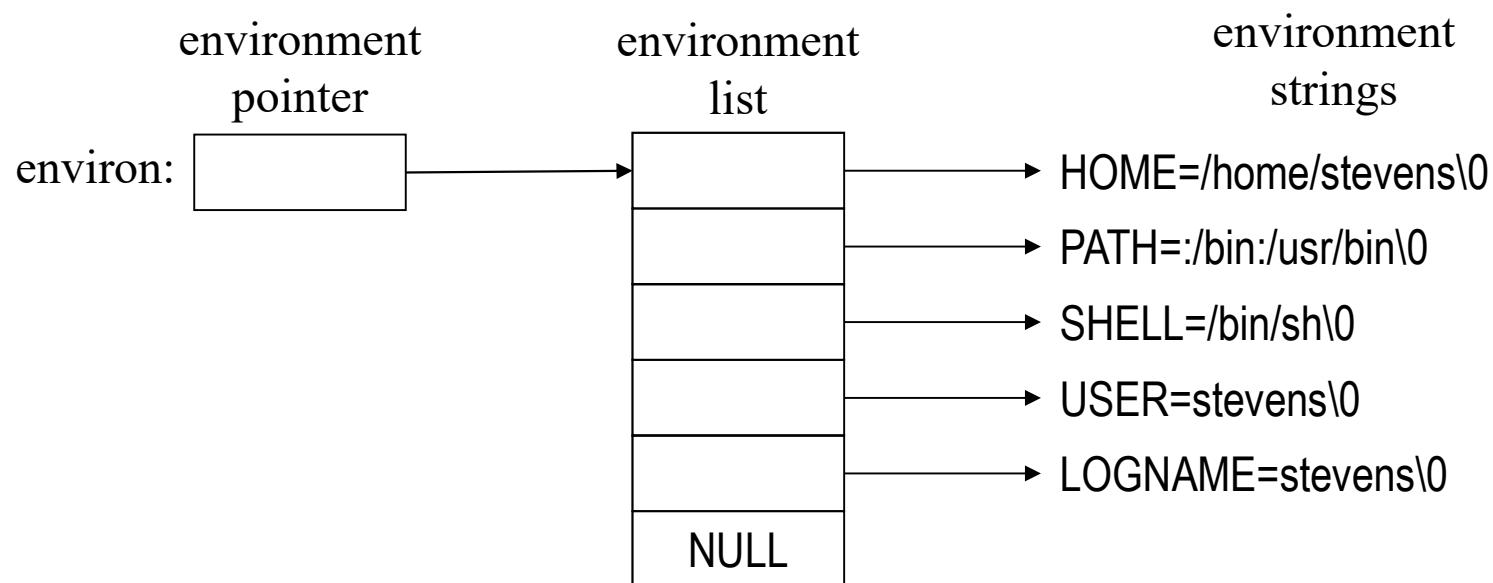
Non-superuser process can set effective user ID to *uid*,
only when *uid* equals real user ID or the saved set-user ID
(set by executing a setuid-program in exec).

In any other cases, **setuid** returns error.

Environment

```
extern char **environ;
```

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



Example: environ

```
#include <stdio.h>

void main( int argc, char *argv[], char *envp[] )
{
    int i;
    extern char **environ;

    printf( "from argument envp\n" );

    for( i = 0; envp[i]; i++ )
        puts( envp[i] );

    printf("\nFrom global variable environ\n");

    for( i = 0; environ[i]; i++ )
        puts(environ[i]);
}
```

getenv

(Page 127)

```
#include <stdlib.h>  
char *getenv(const char *name);
```

Retrieves individual values from the process environment.

Searches the environment list for a string that matches the string pointed to by *name*.

Returns a pointer to the value in the environment, or NULL if there is no match.

putenv

(page 128)

```
#include <stdlib.h>
```

```
int putenv(const char *string);
```

- Adds or changes the value of the calling process's environment variables.
- The argument *string* is of the form name=value.
- If name does not already exist in the environment, then string is added to the environment.
- If name does exist, then the value of name in the environment is changed to value.
- Returns zero on success, or -1 if an error occurs.

Example : getenv, putenv

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

void main(void)
{
    printf("Home directory is %s\n", getenv("HOME"));

    putenv("HOME=/");

    printf("New home directory is %s\n", getenv("HOME"));
}
```

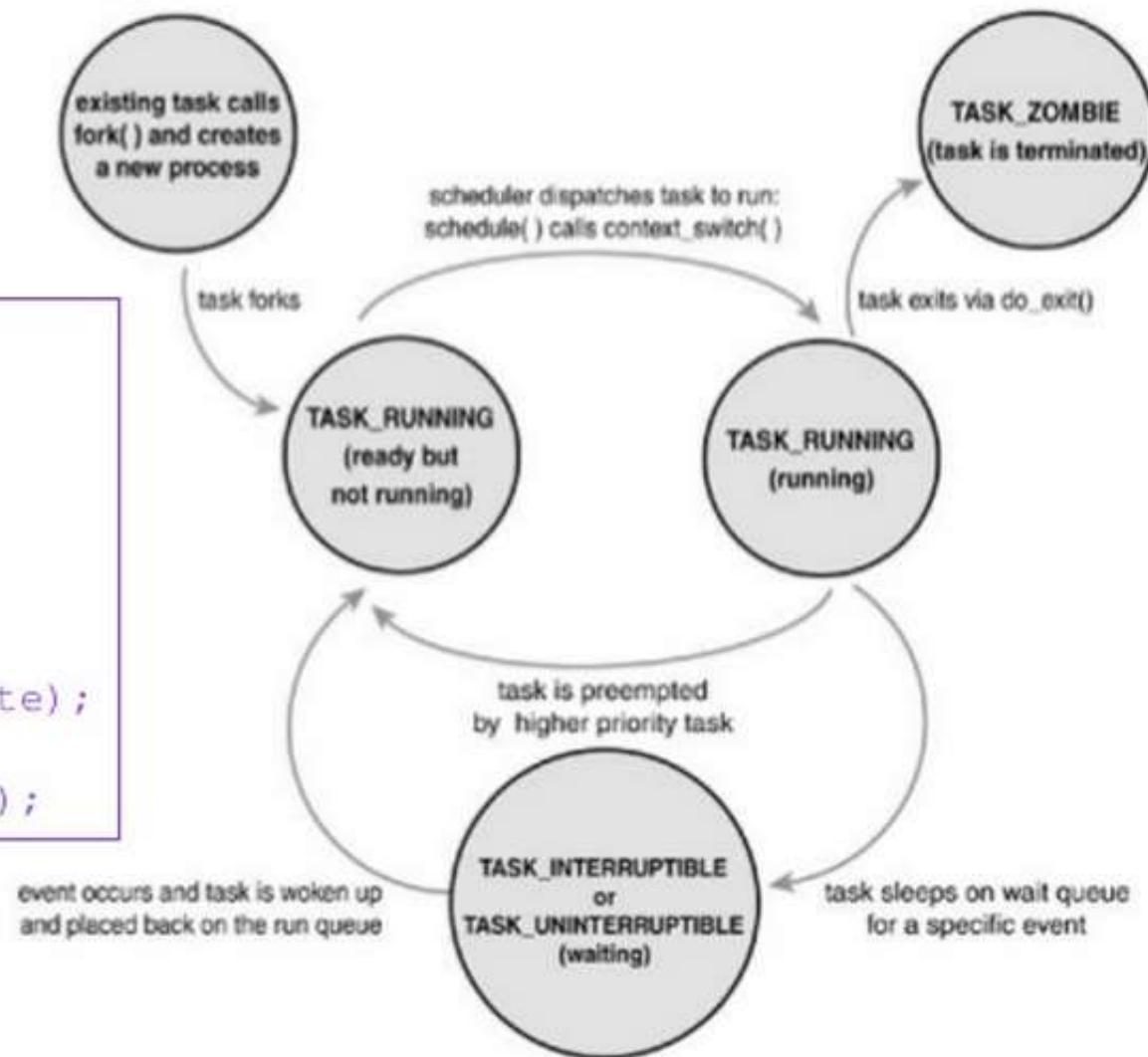
Linux Scheduling

Re-typed on next two slides.

[include/linux/sched.h]

```
TASK_RUNNING  
TASK_INTERRUPTIBLE  
TASK_UNINTERRUPTIBLE  
TASK_STOPPED  
EXIT_ZOMBIE  
EXIT_DEAD
```

```
set_task_state(task, state);  
task_state = state;  
set_current_state( state);
```



This slide shows the contents of the box on the previous slide.

(include/linux/sched.h)

TASK_RUNNING

TASK_INTERRUPTIBLE

TASK_UNINTERRUPTIBLE

TASK_STOPPED

EXIT_ZOMBIE

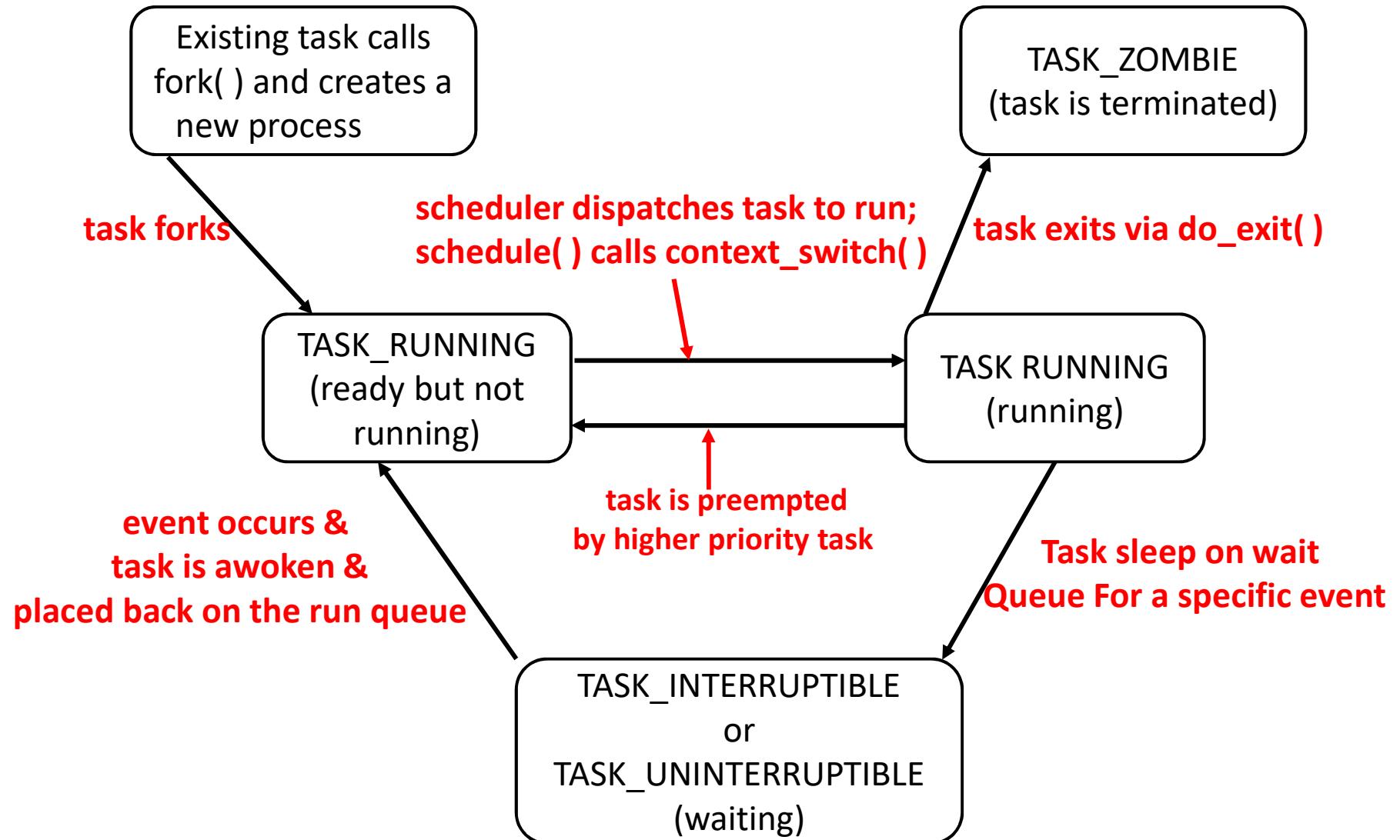
EXIT_DEAD

set_task_state(task, state);

task_state = state;

set_current_state (state);

Linux Scheduling



Process State Codes (from *ps* command)

PROCESS STATE CODES

Here are the different values that the s, stat and state output specifiers (header "STAT" or "S") will display to describe the state of a process.

- D Uninterruptible sleep (usually IO)
- R Running or runnable (on run queue)
- S Interruptible sleep (waiting for an event to complete)
- T Stopped, either by a job control signal or because it is being traced.
- W paging (not valid since the 2.6.xx kernel)
- X dead (should never be seen)
- Z Defunct ("zombie") process, terminated but not reaped by its parent.

Process State Codes (from *ps* command)

For BSD formats and when the stat keyword is used, additional characters may be displayed:

- < high-priority (not nice to other users)
- N low-priority (nice to other users)
- L has pages locked into memory (for real-time and custom IO)
- s is a session leader
- I is multi-threaded (using CLONE_THREAD,
like NPTL pthreads do)
- + is in the foreground process group



7-UNIX

Environment

The End



8-UNIX

exec & fork

System Calls

exec Commands

- `exec()` = a generic name
- `execve(pathname, argv, envp)` (LPI p.514)
 - loads a new program and environment into the process's memory
- `execvp(filename, argv[])`
- `execlp(filename, arg, ...)`
- `execl(filename, arg, ...)`
- `execv(filename, argv[])`
- `execle(pathname, arg, ...)` (LPI p.567)

Note: None of the above returns on success; all return -1 on error.

The naming convention: exec*

- 'l' indicates a list arrangement (a series of null terminated arguments)
- 'v' indicate the array or vector arrangement (like the argv structure).
- 'e' indicates the programmer will construct (in the array/vector format) and pass their own environment variable list
- 'p' indicates the current PATH string should be used when the system searches for executable files.

NOTE:

In the four system calls where the PATH string is not used (execl, execv, execle, and execve) the path to the program to be executed must be fully specified.

exec System Call Functionality

Library Call Name	Argument List	Pass Current Environment Variables	Search PATH automatic?
execl	list	yes	no
execv	array	yes	no
execle	list	no	no
execve	array	no	no
execlp	list	yes	yes
execvp	array	yes	yes

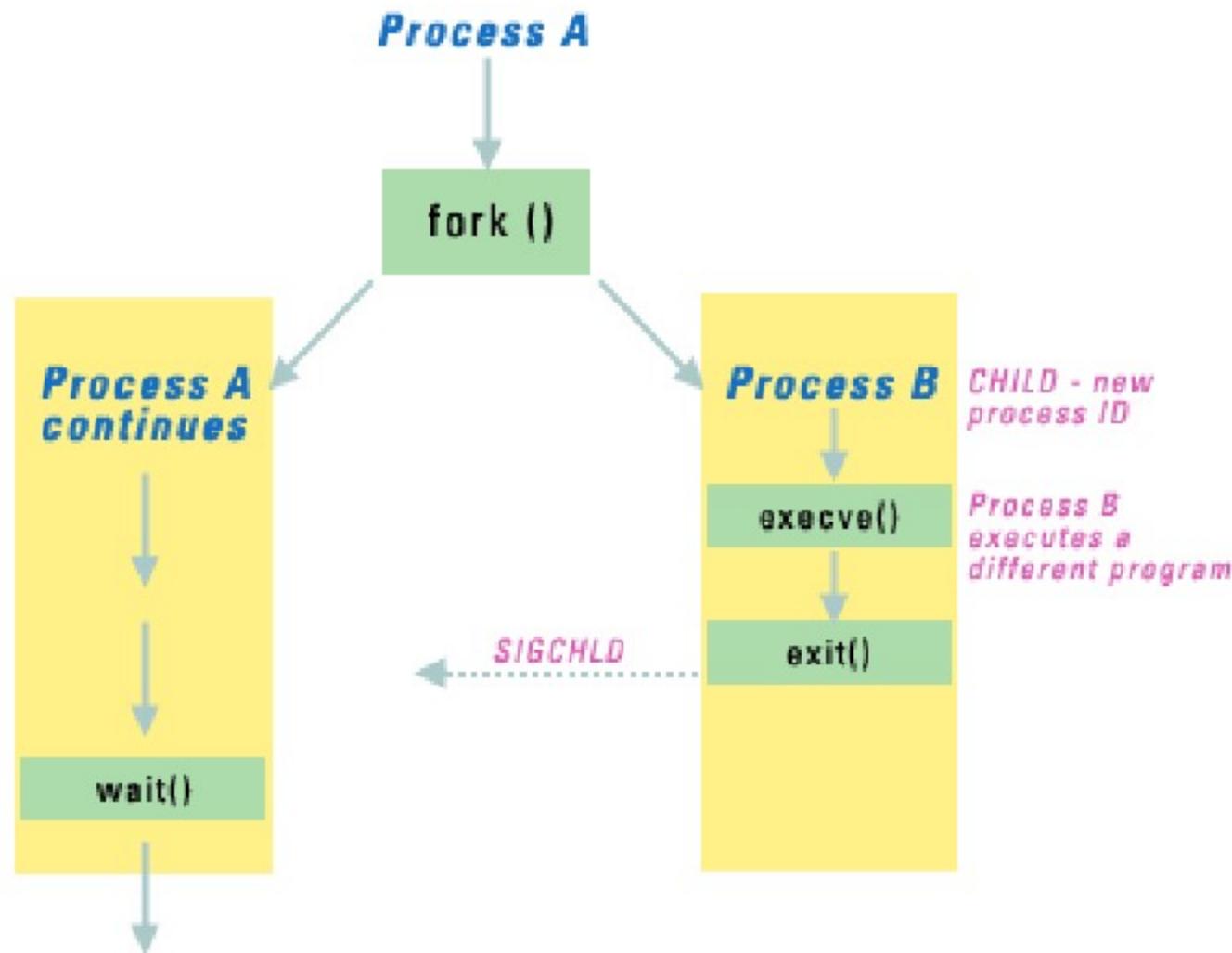
<http://www.cs.uregina.ca/Links/class-info/330/Fork/fork.html#exec>

Things to remember about exec*:

- This system call simply replaces the current process with a new program -- the pid does not change
- The exec() is issued by the calling process and what is exec'ed is referred to as the new program -- not the new process since no new process is created
- It is important to realize that control is not passed back to the calling process unless an error occurred with the exec() call
- In the case of an error, the exec() returns a value back to the calling process
- If no error occurs, the calling process is lost

<http://www.cs.uregina.ca/Links/class-info/330/Fork/fork.html#exec>

fork() as a diagram – what we ‘ve known



What does *fork()* return?

- On success, the PID of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution
- On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

fork() – Output

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    pid_t pid; /* could be int */
    printf("Hello\n");
    pid = fork();
    printf("Goodbye\n");
}
```

fork1.c

Program's output:

Which will it do?

A) Hello
Goodbye

B) Hello
Goodbye
Goodbye

C) Hello
Goodbye
Goodbye
Goodbye

.....

fork() – Who prints what ?

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    pid_t pid;      /* could be int */
    printf("Hello\n");
    pid = fork();
    if (pid == 0)
        printf("Child: Goodbye\n");
    else
        printf("Parent: Goodbye\n");
}
```

fork2.c

Program's Output:
Which will it do?

A)
Hello
Parent: Goodbye
Child: Goodbye

B)
Hello
Parent: Goodbye

C)
Hello
Child: Goodbye

fork() – parent waits for child ?

```
.....  
printf("Hello\n");  
pid = fork();  
if (pid == 0) {  
    printf("Child: Goodbye\n");  
    sleep(2); // sleep 2 seconds  
}  
else {  
    if (wait(&status) == -1)  
        perror("Shell Program error");  
    else  
        printf("Child returned: %d\n",  
               WEXITSTATUS(status));  
}
```

Program's Output:

Hello
Child: Goodbye
Child returned: 0

What does the parent process do?

What does the child process do?

fork3.c

Execute a command via argument passing from argv

```
/* Program to execute a command using an argument from argv */
#include <stdio.h>
#include <unistd.h>
int main (int argc, char *argv[])
{
    if (argc != 2) {
        printf("Usage: input a command name with no option i.e. ls \n");
        return 0;
    }
    printf("About to run: %s \n", argv[1]);
    char *cmd[] = {argv[1], 0 }; /* make sure place a 0 at the end of
                                array of pointer cmd */
    execvp(cmd[0], cmd);
    return 0;
}
```

argv.c

The output of argv.c

```
[bielr@sp1 ClassExamples]> argv ls
```

About to run: ls

argv	fork1	globvar.out	strspn_example
argv.c	fork1.c	infloop	strspn_example.c
count_chars	fork2	infloop.c	Text1.dat
count_chars.c	fork2.c	ouch	thread
count_words	fork3	ouch.c	thread.c
count_words.c	fork3.c	pause	tlpi_hdr.h
data-file	getenv_putenv	pause.c	waitpid
environ	getenv_putenv.c	shfile	waitpid.c
environ.c	globex	shfile.c	word_count.c
error_functions.h	globex.c	shfile.out	word_counts.c

```
[bielr@sp1 ClassExamples]>
```

Review I/O Redirection



First, review a program from C7
named *sincos*

Example Trigonometric Functions

```
// prints tables showing the values of cos,sin  
#include <math.h>  
#include <stdio.h>  
#include <stdlib.h>  
void tabulate(double (*f)(double), double first, double last, double incr);  
int main(void) {  
    double final, increment, initial;  
    printf ("Enter initial value: ");  
    scanf ("%lf", &initial);  
    printf ("Enter final value: ");  
    scanf (%lf", &final);  
    printf ("Enter increment : ");  
    scanf (%lf", &increment);  
    Printf("\n  x  cos(x) \n"  
          " ----- ----- \n");  
    tabulate(cos, initial,final,increment);  
    Printf("\n  x  sin (x) \n"  
          " ----- ----- \n");  
    tabulate(sin, initial,final,increment);  
    return (EXIT_SUCCESS);  
}
```

The **main** function in little print.
Bigger print used in following slides.

Example Trigonometric Functions (1 of 4)

```
// prints tables showing the values of cos, sin  
  
#include <math.h>  
#include <stdio.h>  
#include <stdlib.h>  
  
void tabulate(double (*f)(double), double first,  
              double last, double incr);  
  
int main(void)
```

Example Trigonometric Functions (2 of 4)

```
int main(void)
{
    double final, increment, initial;
    // Enter the data at the keyboard
    printf ("Enter initial value: ");
    scanf ("%lf", &initial);
    printf ("Enter final value: ");
    scanf (%lf", &final);
    printf ("Enter increment : ");
    scanf (%lf", &increment);
```

Example Trigonometric Functions (3 of 4)

```
// Print the headers and call tabulate
printf("\n  x  cos(x) \n"
      " ----- ----- \n");
tabulate(cos, initial,final,increment);

printf("\n  x  sin (x) \n"
      " ----- ----- \n");
tabulate(sin, initial,final,increment);
return (EXIT_SUCCESS);

}
```

Trigonometric Functions (4 of 4)

```
// when passed a pointer f, the function prints a table  
// showing the value of f  
  
void tabulate(double (*f) (double), double first,  
                double last, double incr)  
{  
    double x;  
    int i, num_intervals;  
    num_intervals = ceil ( (last -first) /incr );  
    for (i=0; i<=num_intervals; i++) {  
        x= first +i * incr;  
        printf("%10.5f %10.5f\n", x , (*f) (x));  
    }  
}
```

Output of the Example

Enter initial value: 0

Enter final value: .5

Enter increment: .1

X	cos(x)
0.00000	1.00000
0.10000	0.99500
0.20000	0.98007
0.30000	0.95534
0.40000	0.92106
0.50000	0.87758

X	sin(x)
0.00000	0.00000
0.10000	0.09983
0.20000	0.19867
0.30000	0.29552
0.40000	0.38942
0.50000	0.47943

Two redirection examples:

- 1) ls > out.txt
- 2) sincos < input.txt

In input.txt:

5.0 - initial value
10.0 – final value
1.0 - increment value

Redirection of standard output (redir.c)

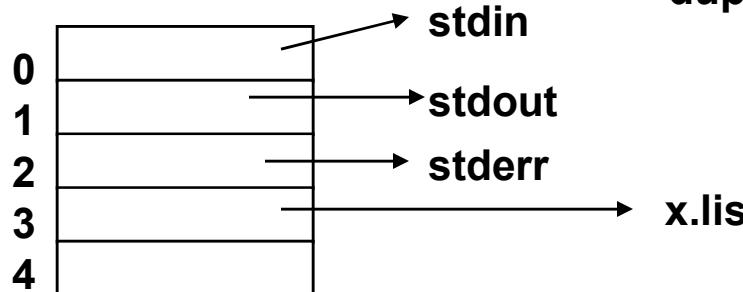
Example: implement shell to do: **ls > x.ls**

What happens **inside**:

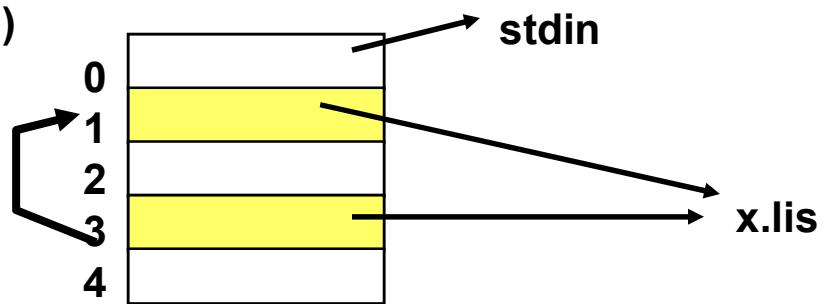
- (1) Open a new file *x.ls*
- (2) Redirect standard output to *x.ls* using **dup2** command
Everything sent to standard output ends in *x.ls*
- (3) execute ls in the process

dup2(int fin, int fout) - copies **fin** to in the file table

File descriptor table



dup2(3,1)



Note about the next program

It uses the CREAT call...to create a file that does not exist.

It is becoming antiquated.

```
fileID = creat( "x.lis",0640 ); // 0640 is the mode
```

To do the call with OPEN, we would type:

```
fileID = open ( "x.lis", O_WRONLY | O_CREAT | O_TRUNC,  
                S_IRUSR | S_IWUSR | S_IRGRP);
```

0640 = 0400 (User_read) + 0200 (User-write) + 040 (Group-read)

The mode for file permission bits is explained in a chart on
LPI page 295.

Constants for file permission bits

Constant	Octal value	Permission bit
S_ISUID	04000	Set-user-ID
S_ISGID	02000	Set-group-ID
S_ISVTX	01000	Sticky
S_IRUSR	0400	User-read
S_IWUSR	0200	User-write
S_IXUSR	0100	User-execute
S_IRGRP	040	Group-read
S_IWGRP	020	Group-write
S_IXGRP	010	Group-execute
S_IROTH	04	Other-read
S_IWOTH	02	Other-write
S_IXOTH	01	Other-execute

Code Example - implement ls > x.lis

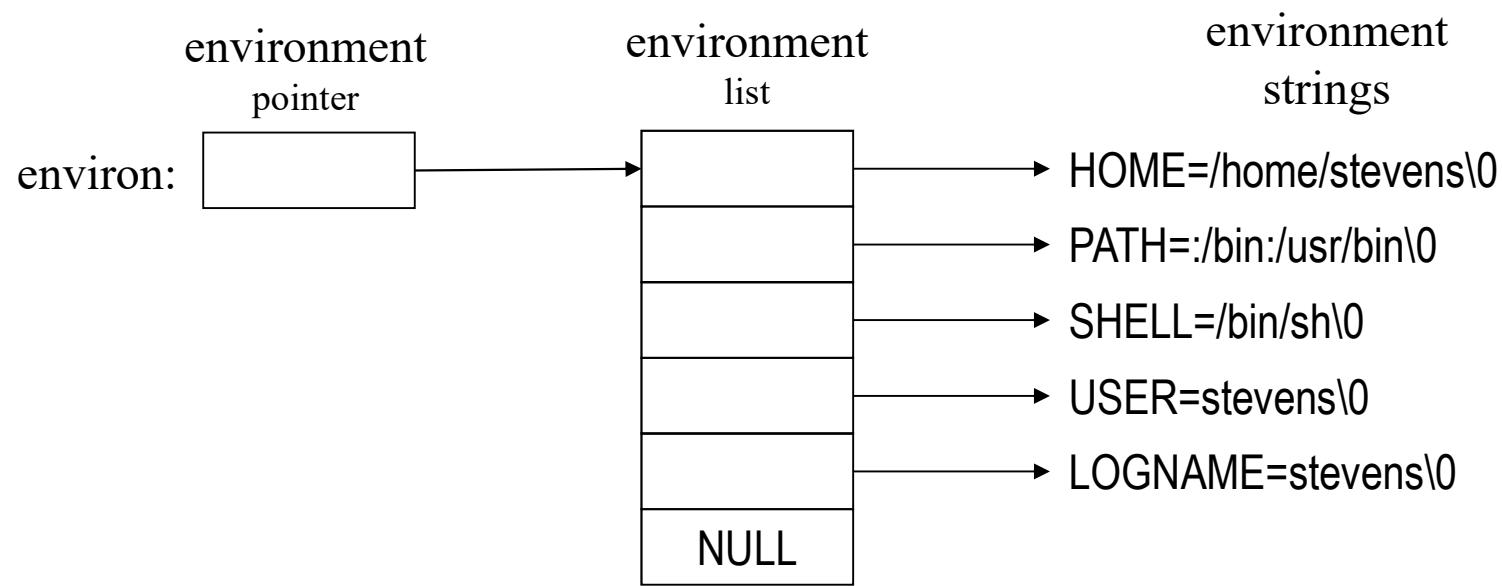
```
/* fileID,  code file = redir.c */
#include <unistd.h>
int main (void)
{
    int fileID;
    fileID = creat( "x.lis",0640 ); // 0640 is the mode
    if( fileID < 0 ) {
        printf( "error creating x.lis\n" );
        exit (1);
    }
    dup2( fileID, stdout ); /* copy fileID to stdout */
    close( fileID );
    execl( "/bin/ls", "ls", 0 ); return EXIT_SUCCESS;
}
```



Environment

Environment

```
extern char **environ;  
int main( int argc, char *argv[ ], char *envp[ ] )
```



Example: environ

```
#include <stdio.h>
void main( int argc, char *argv[], char *envp[] )
{
    int i;
    extern char **environ;
    printf( "from argument envp\n" );
    for( i = 0; envp[i]; i++ )
        puts( envp[i] );
    printf("\nFrom global variable environ\n");
    for( i = 0; environ[i]; i++ )
        puts(environ[i]);
}
```

Sample output

```
MANPATH=/usr/man:/usr/local/man:/usr/share/man:/usr/local/share:man  
rvm_bin_path=/usr/local/rvm/bin  
HOSTNAME=athena.ecs.csus.edu  
GEM_HOME=/usr/local/rvm/gems/ruby-2.0.0-p0  
TERM=xterm  
SHELL=/bin/bash  
HISTSIZE=1000  
SSH_CLIENT=130.86.204.182 56016 22  
MAIL=/var/spool/mail/nguyendh  
PATH=/bin:/usr/bin:/sbin:/usr/sbin:/usr/local/bin:/etc:/usr/local/etc:/usr/bin/X  
11:/opt/bin:/usr/X11R6/bin:/usr/pkg/fm/bin:/usr/pkg/mv/bin:/usr/pkg/pts/bin:/usr  
/pkg/sml/bin:/usr/pkg/syn/bin:/usr/pkg/synmc/bin:/usr/pkg/vcs/bin:/usr/pkg/virsi  
mrm/bin:  
HOME=.....
```

getenv

```
#include <stdlib.h>
char *getenv(const char *name);
```

Searches the environment list for a string that matches the string pointed to by *name*. Returns a pointer to the value in the environment, or NULL if there is no match.

For example:

```
myhome = getenv("HOME");
```

putenv

- **#include <stdlib.h>**

int putenv(const char **string*);

- Adds or changes the value of environment variables.
- The argument *string* is of the form name=value.
- If name does not already exist in the environment, then string is added to the environment.
- If name does exist, then the value of name in the environment is changed to value.
- Returns zero on success, or -1 if an error occurs.
- For example:
 - `putenv("HOME=/");`

setenv

(1 of 2)

```
#include <stdlib.h>
int setenv(const char *name, const char
          *value, int overwrite);
```

The `setenv()` function adds the variable name to the environment with the value `value`, if `name` does not already exist.

If `name` does exist in the environment, then its value is changed to `value` if `overwrite` is nonzero if `overwrite` is zero, then the value of `name` is not changed (and `setenv()` returns a success status).

setenv

(2 of 2)

This function makes copies of the strings pointed to by name and value (by contrast with

`putenv(3)`).

Example: `setenv("PWD", tempbuf, 1);`

Example : getenv, putenv (Example 1)

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

void main(void)
{
    printf("Home directory is %s\n", getenv("HOME"));

    putenv("HOME=/");

    printf("New home directory is %s\n", getenv("HOME"));
}
```

Example : getenv, setenv (Example 2)

```
// User types cd command
#ifndef PATH_MAX
#define PATH_MAX 255
#endif

.....
temp = getenv("HOME");
chdir(temp); /* CHDIR(2) */
/* update PWD environment variable with the new directory */
getcwd(tempbuf,PATH_MAX);
setenv("PWD", tempbuf,1);
```

User and Group ID

User and Group ID

(1 of 2)

Each process has three IDs

(1) Real user ID (RUID)

used to determine which user started the process

Same as the user ID of parent (unless changed)

(2) Effective user ID (EUID)

- Used to assign ownership of newly created files
- to check file access permissions
- to check permission to send signals to processes.

Two ways to change: from set user ID bit on the file being executed, or system call

(3) Saved user ID (SUID)

So previous EUID can be restored

User and Group ID

(2 of 2)

Notes:

Real and effective uid: inherit (fork), maintain (exec).

Normally, effective user and effective group user ids have the same value as real user ids except when we change it (in two ways)

Real **group** ID, effective **group** ID, used similarly .

Type of Users in Linux System (optional)

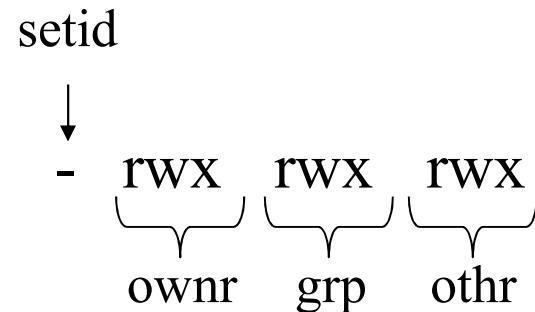
- * Super User (or Root) – uid 0
- * System User – uid 1-499
- * Regular User – uid: 500 < 6000
- * Network User – uid > 6000
- * Pseudo User – Replica of Super User

```
[nguyendh@athena ~]> id  
uid=5206(nguyendh) gid=4104(othcsc) groups=4104(othcsc)  
[nguyendh@athena ~]>
```

User changes password ? i.e use passwd

Recall: Unix file security

- Each file has owner and group
- Permissions set by owner
 - Read, write, execute
 - Owner, group, other
 - Represented by vector of four octal values
- Only owner, root can change permissions
 - This privilege cannot be delegated or shared



Recall: Special note on File type and permissions

File type and permissions

The `st_mode` field is a bit mask serving the dual purpose of identifying the file type and specifying the file permissions. The bits of this field are laid out as shown in Figure 15-1.

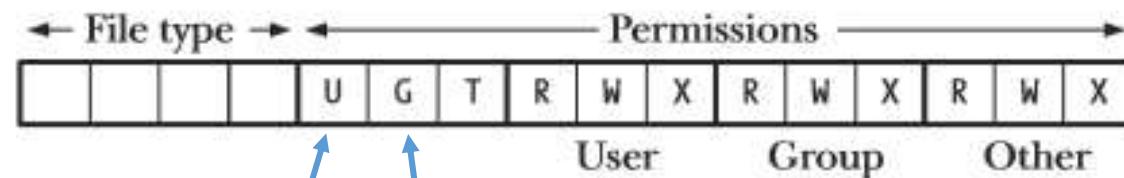


Figure 15-1: Layout of `st_mode` bit mask

The file type can be extracted from this field by ANDing (&) with the constant `S_IFMT`. (On Linux, 4 bits are used for the file-type component of the `st_mode` field.)

Set-UID Set-GID

LPI (Page 281)

Side Note.

After the Set-UID and Set-GID bits on the previous slide, there is a bit labeled “T”;

That stands for Sticky Bit.

A **sticky bit** is a permission **bit** that is set on a directory that allows only the owner of the file within that directory or the root user to delete or rename the file. No other user has the needed privileges to delete the file created by some other user.

First Option: Set User ID Bit

Owner sets it with chmod

u – owner; g – group; o – all other user

chmod u+x executable_file

chmod u+s executable_file

-rws----- executable_file

Process Operations and IDs

Root

ID=0 for superuser root; can access any file

Fork and Exec

Inherit three IDs

ID	Set-user-ID bit off	Set-user-ID bit on
RUID	Unchanged	Unchanged
EUID	Unchanged	Set from user ID of program file
SUID	Copied from EUID	Copied from EUID

Process Operations and IDs (1 of 2)

When a set-user-ID program is run (i.e. loaded into a process's memory by an `exec()`), the kernel sets the effective user ID of the process to be the same as the user ID of the executable file. Running a set-group-ID program has an analogous effect for the effective group ID of the process. Changing the effective user or group ID in this way gives a process (in other words, the user executing the program) privileges it would not normally have. For example, if an executable file is owned by root (superuser) and has the set-user-ID permission bit enabled, then the process gains superuser privileges when that program is run.

(LPI – P169)

Process Operations and IDs – Example (2 OF 2)

```
$ su
```

Password:

```
# chown root check_password
```

Make this program owned by root.

```
#chmod u+s check_password
```

With the set-user-ID bit enabled

```
# ls -l check_password
```

```
-rwsr-xr-x 1 root users 18150 Oct 28 10:49 check_password
```

```
# exit
```

```
$ whoami
```

This is an unprivileged login.

```
Mtk
```

```
$ ./check_password
```

But we can now access the shadow password file using this program

Username: avr

Password:

Successfully authenticated: UID=1001

LPI page 169-70

Second Option: setuid(*uid*) system calls

ID	Super user	Unprivileged user
RUID	Set to <i>uid</i>	Unchanged
EUID	Set to <i>uid</i>	Set to <i>uid</i>
SUID	Set to <i>uid</i>	unchanged

System call: Read IDs

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

```
pid_t getuid(void);
```

Returns the real user ID of the current process

```
pid_t geteuid(void);
```

Returns the effective user ID of the current process

```
gid_t getgid(void);
```

Returns the real group ID of the current process

```
gid_t getegid(void);
```

Returns the effective group ID of the current process

System call: Change UID and GID

```
#include <unistd.h>
int setuid( uid_t uid )
int setgid( gid_t gid )
```

Sets the effective user ID of the current process.

- Superuser process resets the real effective user IDs to *uid*.
- Non-superuser process can set effective user ID to *uid*, only when *uid* equals real user ID or the saved set-user ID (set by executing a setuid-program in exec).
- In any other cases, setuid returns error.

Difference between:

Real User ID

Effective User ID

Saved User ID

<http://stackoverflow.com/questions/32455684/difference-between-real-user-id-effective-user-id-and-saved-user-id>

or

<http://tinyurl.com/z3eakft>

The contents of the link are pasted into the next slides.

Page 1 of 2.

The distinction between a real and an effective user id is made because you may have the need to temporarily take another user's identity (most of the time, that would be root, but it could be any user). If you only had one user id, then there would be no way of changing back to your original user id afterwards (other than taking your word for granted, and in case you are root, using root's privileges to change to any user).

So, the real user id is who you really are (the one who owns the process), and the effective user id is what the operating system looks at to make a decision whether or not you are allowed to do something (most of the time, there are some exceptions).

When you log in, the login shell sets both the real and effective user id to the same value (your real user id) as supplied by the password file.

Page 2 of 2.

Now, it also happens that you execute a setuid program, and besides running as another user (e.g. root) the setuid program is also supposed to do something on your behalf. How does this work? After executing the setuid program, it will have your real id (since you're the process owner) and the effective user id of the file owner (for example root) since it is setuid.

The program does whatever magic it needs to do with superuser privileges and then wants to do something on your behalf. That means, attempting to do something that you shouldn't be able to do should fail. How does it do that? Well, obviously by changing its effective user id to the real user id!

Now that setuid program has no way of switching back since all the kernel knows is your id and... your id. Bang, you're dead.

This is what the saved set-user id is for.

Some Applications of fork

Definitions

- **getty** , short for "get tty", is a Unix program running on a host computer that manages physical or virtual terminals (TTYs). When it detects a connection, it prompts for a username and runs the 'login' program to authenticate the user.

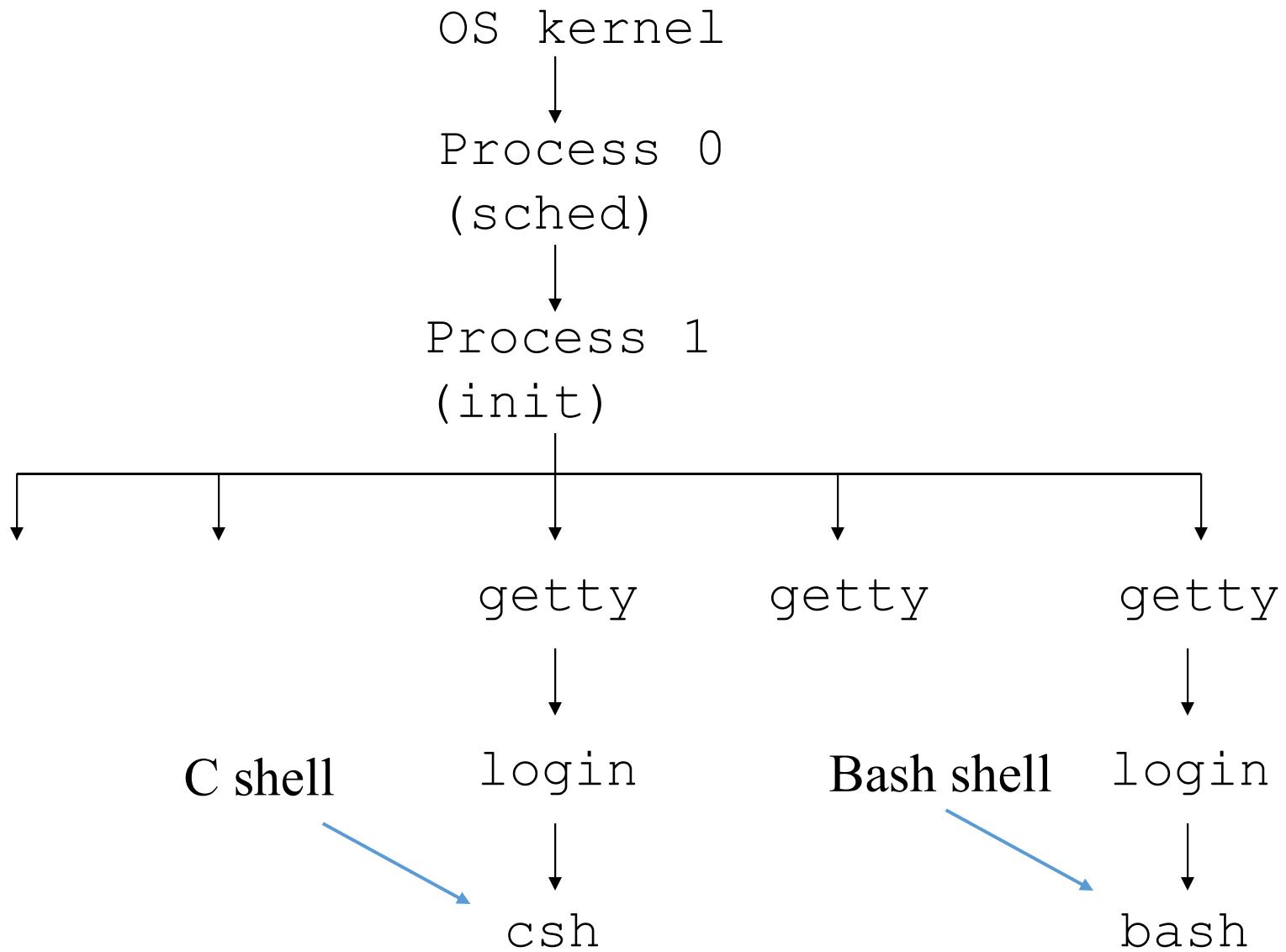
Definition of init

- **init** (short for *initialization*) is the first process started during booting of the computer system.
- **init** is a daemon or background process that continues running until the system is shut down.
- It is the direct or indirect ancestor of all other processes and automatically adopts all orphaned processes.
- **init** is started by the kernel using a hard-coded filename; a kernel panic will occur if the kernel is unable to start it.
- **init** is typically assigned process identifier 1.

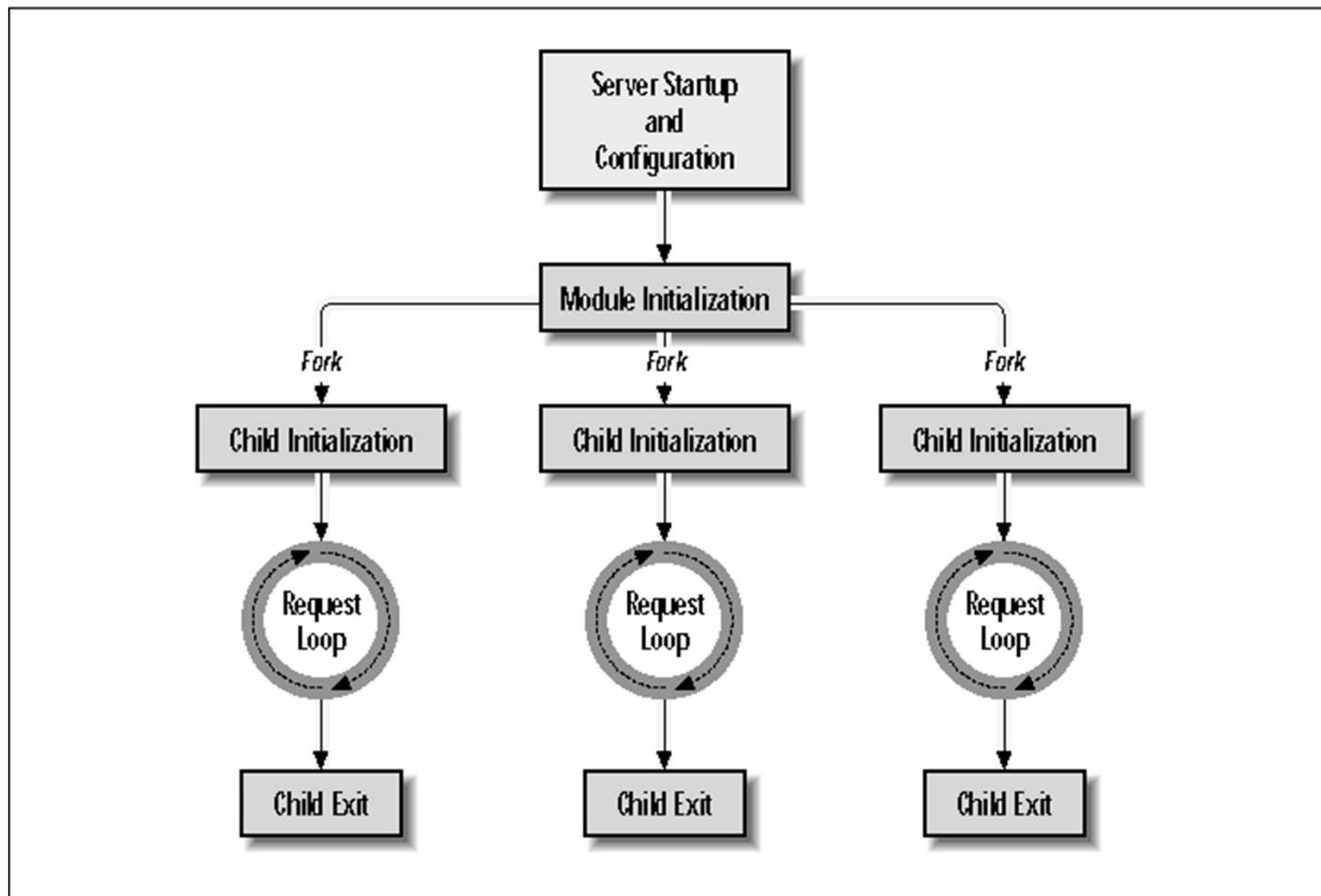
Definitions

- **login** - is used when signing onto a system. If no argument is given, **login** prompts for the username

Unix Start Up Processes Diagram



Apache Web Server



Source: http://docstore.mik.ua/orelly/apache_mod/24.htm⁵⁸



8-UNIX exec & fork System Calls

The End

9-UNIX File I/O Calls

Linux Program Interface
Chapter 3-4

Unix I/O API

- Some of the most common Unix I/O API functions used by applications are:
 - `open()`
 - `close()`
 - `read()`
 - `write()`
 - `lseek()`
- API = Application Program Interface
- More on page 70 of LPI

File Descriptor (fd)

- Usually a small nonnegative number
- Returned by **open** and used by the other I/O commands
- Used by all system calls for performing I/O to open files
- Some types of files:
 - Pipes
 - FIFOs (used in inter-process communication)
 - Terminals
 - Devices
 - Regular files

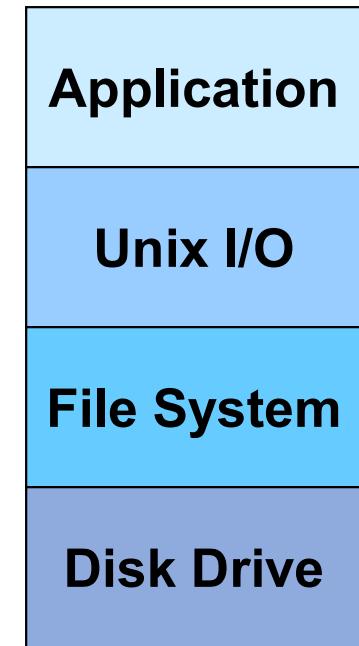
Three Standard File Descriptors

- These are inherited by a program on opening, from the shell's file descriptors

File Descriptors	Purpose	POSIX name	<i>stdio stream *</i>
0	Standard input	STDIN_FILENO	<i>stdin</i>
1	Standard output	STDOUT_FILENO	<i>stdout</i>
2	Standard error	STDERR_FILENO	<i>stderr</i>

The Role of Unix File Input/Output

- I/O : the process that copies data between memory and external devices
- Applications work at the byte level
- File system works at the block level
- Unix I/O converts the byte level access to block level operations



File System
Layering

open call (1 of 5)

- Opening a file informs the kernel that an application wants to access a file
- Allows the kernel to set aside resources
- Returns file descriptor on success, or -1 on error

open call (2 of 5)

Call:

```
#include <sys/stat.h>
#include <fcntl.h>
```

```
int open (const char *pathname, int flags, ... /* mode_t mode */);
```

Example:

```
/* Open new or existing file for reading and writing, truncating
   to zero bytes; file permissions read+write for owner, nothing
   for all others */
```

```
fd = open("myfile", O_RDWR | O_CREAT |
            O_TRUNC, S_IRUSR | S_IWUSR);
if (fd == -1)
    errExit("open");
```

Note: a const char * pathname means that the program can't change the data that pathname points to through the pathname pointer!

open call (3 of 5)

- Flags indicating access type:
 - O_RDONLY : read only
 - O_WRONLY : write only
 - O_RDWR: read/write
 - O_CREAT: create the file if doesn't exist
 - O_APPEND: write at end
 - O_TRUNC: Truncate exist file to zero length
 - etc.
- Can also bitwise – inclusive – or them
 - i.e. O_WRONLY | O_APPEND
 - See: **Table 4-3 (in LPI, page 74)**

open call (4 of 5)

- Different mode values (file permissions)
- S_IRUSR: read permission, owner
- S_IWUSR: write permission, owner
- S_IROTH: read permission, others
- S_IWOTH: write permission, others
- etc

Note: for more information, please do a
man 2 open to get all modes values.

open call (5 of 5)

- Open returns a small integer called a ***file descriptor (fd)***
- Application passes this value back to the kernel in subsequent requests to work with a file
- Each process created starts with three open files:
 - 0: **standard input (stdin)**
 - 1: **standard output (stdout)**
 - 2: **standard error (stderr)**

<inistd.h> contains constants

STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO for them

close call

- Call: **int close(int fd)**
- Closing a file tells the kernel it may free resources associated with managing the file
- close returns 0 if OK, -1 if error
- The Call:

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

```
int close (int fd);
```

Returns 0 on success,
-1 on error

Example:

```
if (close(fd) == -1)  
    errExit("close");
```

read call (1 of 2)

- Call:

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

```
ssize_t read(int fd, void *buffer, size_t count);
```

Returns number of bytes read, 0 on EOF, or -1 on error.

- Each open file has a notion of a current position in the stream of bytes
- `read()` copies at most **count** bytes from the current file position to **buffer** and updates the file position
- `read()` returns the number of bytes read
 - returns <0 if error
 - returns 0 if end-of-file (EOF) occurs
- `read` may return fewer bytes than requested (short reads)

read call - example (2 of 2)

```
#define MAX_READ 20
char buffer[MAX_READ];

if(read(STDIN_FILENO, buffer, MAX_READ) == -1)
    errExit("read");
printf("The input data was: %s\n", buffer);
```

write call (1 of 2)

- Call:

```
#include <unistd.h>

ssize_t write (int fd, void *buffer, size_t count);
```

Returns number of bytes written, or -1 on error.

- `write()` copies at most **count** bytes from **buffer** to the file position and updates position
- Returns the number of bytes written
 - returns <0 if error
 - It is possible that fewer bytes were written than requested (short writes) this is not an error, but certainly a challenge to deal with

write Example (2 of 2)

```
/* Transfer data until we encounter end of input or an error */

while ((numRead = read(inputFd, buf, BUF_SIZE)) > 0)
    if (write(outputFd, buf, numRead) != numRead)
        fatal("couldn't write whole buffer");
    if (numRead == -1)
        errExit("read");
```

Iseek call (1 of 2)

- Call

```
#include <unistd.h>

Off_t Iseek(int fd, off_t offset, int whence);
```

Returns new file offset if successful, or -1 on error
- Causes the logical position in the file to change
 - i.e. where the next read or write will commence from
 - Also referred to as Changing the File Offset
- *whence* determines how position will change:
 - SEEK_SET : pointer is set to offset bytes.
 - SEEK_CUR: pointer is set to its current location plus offset.
 - SEEK_END: pointer is set to the size of the file plus offset.
- Note: file offset or size – signed integer
 - See table 3-1 (System Data Type)(page 64-65)

`lseek` call – examples (2 of 2)

```
lseek(fd, 0, SEEK_SET); /* Start of file */  
lseek(fd, 0, SEEK_END); /* Next byte after the end of the file */  
lseek(fd, -1, SEEK_END); /* Last byte of file */  
lseek(fd, -10, SEEK_CUR); /* Ten bytes prior to current location */  
lseek(fd, 10000, SEEK_END); /* 10001 bytes past last byte of file */
```

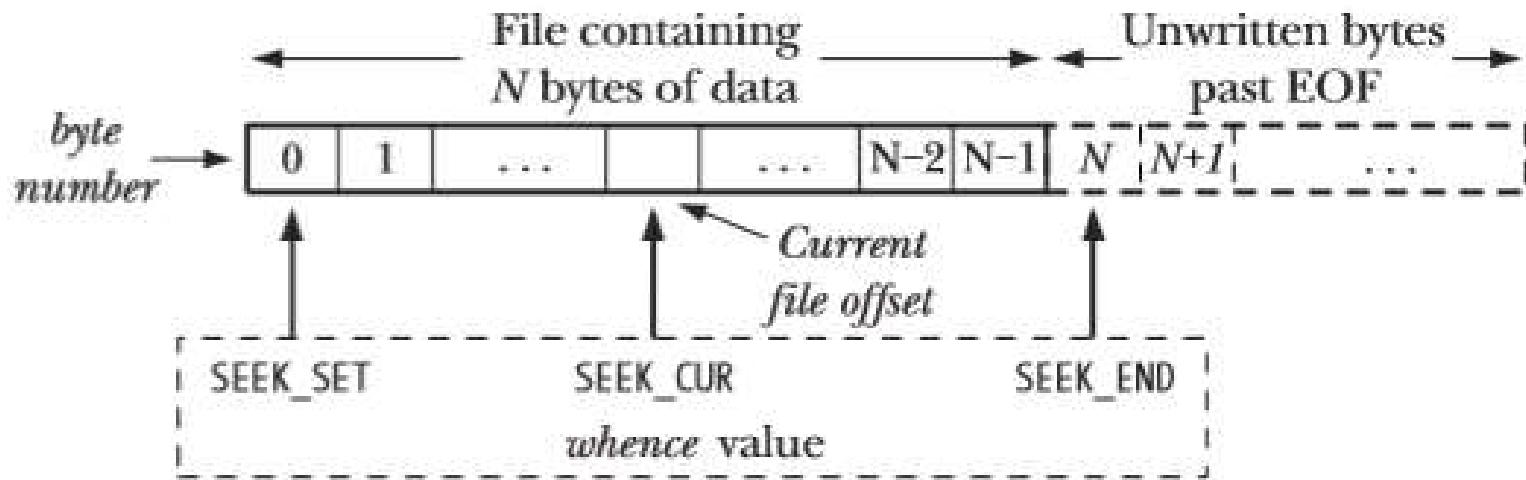


Figure 4-1: Interpreting the *whence* argument of `lseek()`

Restrictions on *Iseek*

- It cannot be applied to:
 - pipe - inter-process communication
 - FIFO - a list or queue
 - socket - inter-process communication
 - terminal

Unix I/O Example

- Simple program that copies contents of file named by argument 1 to file named by argument 2 (i.e. the cp command)

```
cs060copy fname1 fname2
```

Pseudo Code

open argument 1 for input

open argument 2 for output

If there is error

 exit

copy data until we reach end of input or an error

From the textbook:

The following three slides show code from the textbook that implements a “copy” command.

The code uses functions that are **exclusive** to the textbook and its environment.

These functions are **not** available on athena.

Code that will work on athena will follow.

Example: Unix Copy Command (1 of 3)

From Textbook

```
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"
#ifndef BUF_SIZE    /* Allow "cc -D" to override definition */
#define BUF_SIZE 1024
#endif
int main(int argc, char *argv[])
{
    int inputFd, outputFd, openFlags;
    mode_t filePerms;
    ssize_t numRead;
    char buf[BUF_SIZE];
    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s old-file new-file\n", argv[0]);
```

Example (cont') (2 of 3) Textbook

```
/* Open input and output files */
inputFd = open(argv[1], O_RDONLY);
if (inputFd == -1)
    errExit("opening file %s", argv[1]);

openFlags = O_CREAT | O_WRONLY | O_TRUNC;
filePerms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
            S_IROTH | S_IWOTH; /* rw-rw-rw- */
outputFd = open(argv[2], openFlags, filePerms);
if (outputFd == -1)
    errExit("opening file %s", argv[2]);
```

Example (cont') (3 of 3) Textbook

```
/* Transfer data until we encounter end of input or an error */

while ((numRead = read(inputFd, buf, BUF_SIZE)) > 0)
    if (write(outputFd, buf, numRead) != numRead)
        fatal("couldn't write whole buffer");
    if (numRead == -1)
        errExit("read");

    if (close(inputFd) == -1)
        errExit("close input");
    if (close(outputFd) == -1)
        errExit("close output");

    exit(EXIT_SUCCESS);
}
```

Note: **errExit** displays error message including 'errno' diagnostic, and terminates the process by calling `_exit()`.

Code for athena

(1 of 4)

```
#include <sys/stat.h>
#include <fcntl.h>      //needed for open
#include <sys/stat.h>  //needed for open
#include <unistd.h>    //needed for close, read, write
#include <stdio.h>
#include <stdlib.h>
#ifndef BUF_SIZE      /* Allow "cc -D" to override definition */
#define BUF_SIZE 1024
#endif
```

Code for athena

(2 of 4)

```
int main(int argc, char *argv[])
{
    int inputFd, outputFd, openFlags;
    mode_t filePerms;
    ssize_t numRead;
    char buf[BUF_SIZE];

    if (argc != 3 || strcmp(argv[1], "--help") == 0) {
        fprintf(stderr,"%s old-file new-file\n", argv[0]);
        exit(EXIT_FAILURE);
    }
```

Code for athena

(3 of 4)

```
/* Open input and output files */

inputFd = open(argv[1], O_RDONLY);
if (inputFd == -1)
    perror("opening file argv[1]");

openFlags = O_CREAT | O_WRONLY | O_TRUNC;
filePerms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
            S_IROTH | S_IWOTH; /* rw-rw-rw- */

outputFd = open(argv[2], openFlags, filePerms);
if (outputFd == -1)
    perror("opening file argv[2]");
```

Code for athena

(4 of 4)

```
/* Transfer data until we encounter end of input or an error */
while ((numRead = read(inputFd, buf, BUF_SIZE)) > 0) {
    if (write(outputFd, buf, numRead) != numRead)
        perror("couldn't write whole buffer");
}
if (numRead == -1)
    perror("read error");
if (close(inputFd) == -1)
    perror("close input");
if (close(outputFd) == -1)
    perror("close output");
exit(EXIT_SUCCESS);
}
```

Unix I/O

- By making everything appear to be a file, the kernel can provide a single simple interface for performing I/O to a variety of devices
- Recall the basic operations are:
 - Opening and closing files
 - `open()` and `close()`
 - Changing the current file position
 - `lseek()`
 - Reading and writing files
 - `read()` and `write()`

Important Note: A File is a File is a File

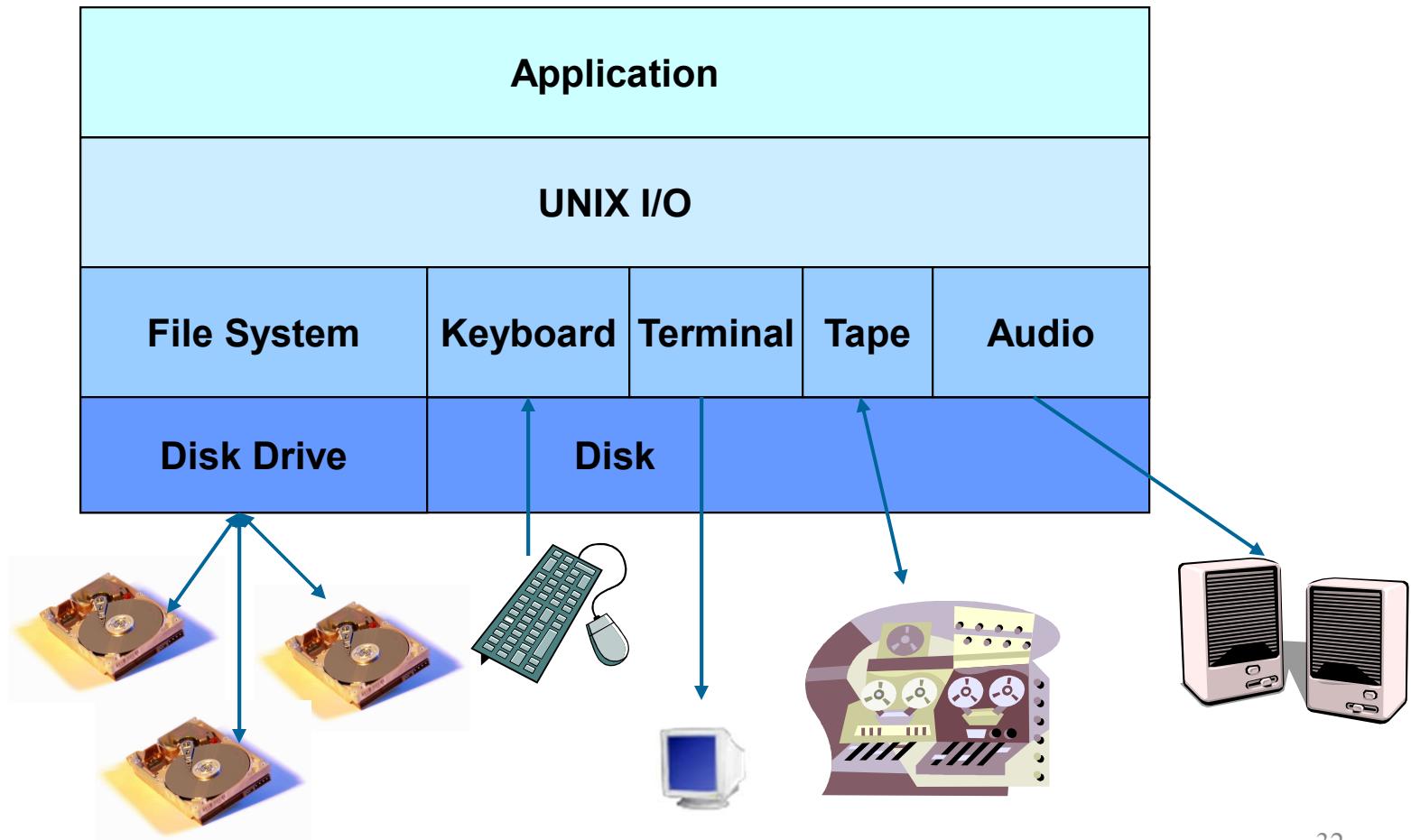
- Remember, “Everything in Unix is a File”
- This means that all low-level I/O is done by reading and writing file handles, regardless of what particular peripheral device is being accessed—a tape, a socket, even your terminal, they are all *files*.
- Low level I/O is performed by making *system calls*

Adding Other Devices

- Most devices tend to be producers or consumers of streams of data and fit UNIX I/O API model described

Mouse	➡	producer
Joystick	➡	producer
Keyboard	➡	producer
Display	⬅	Consumer
Audio device	⬅	consumer
Tape	↔	both

New Devices

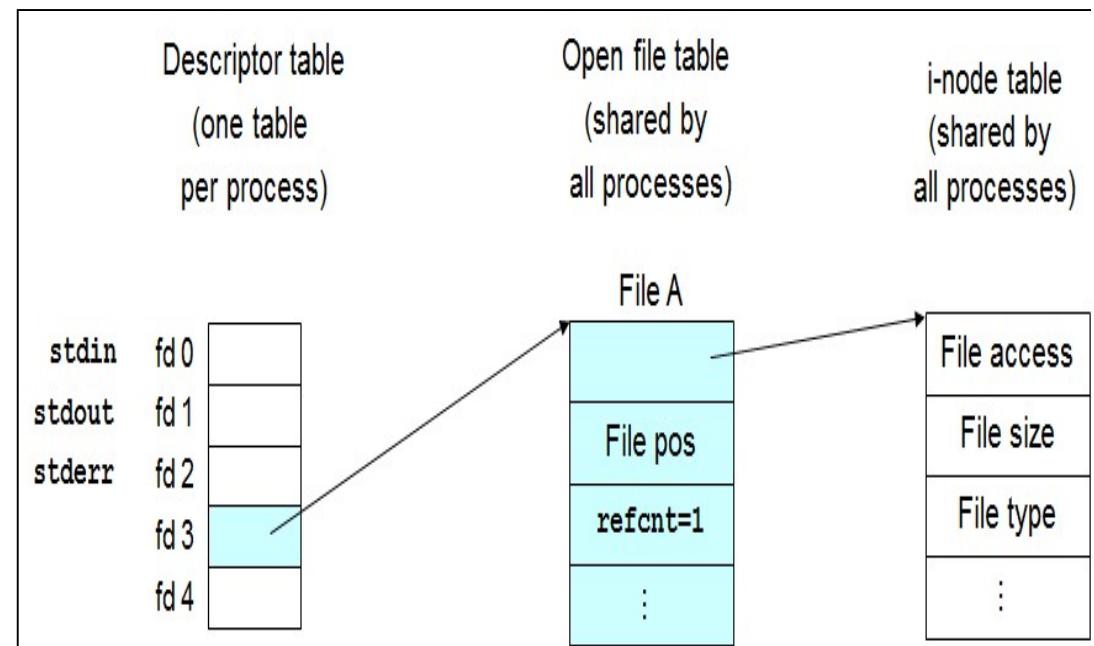


File Descriptors

- Calls to routines like *open()*, *socket()*, *accept()* and *pipe()* return file descriptors
- A file descriptor is just a small integer
- When this “integer” is passed back to the kernel via calls like *read()* or *write()* the kernel manipulates the opened “file” the descriptor corresponds to

The Kernel's View of a File Descriptor

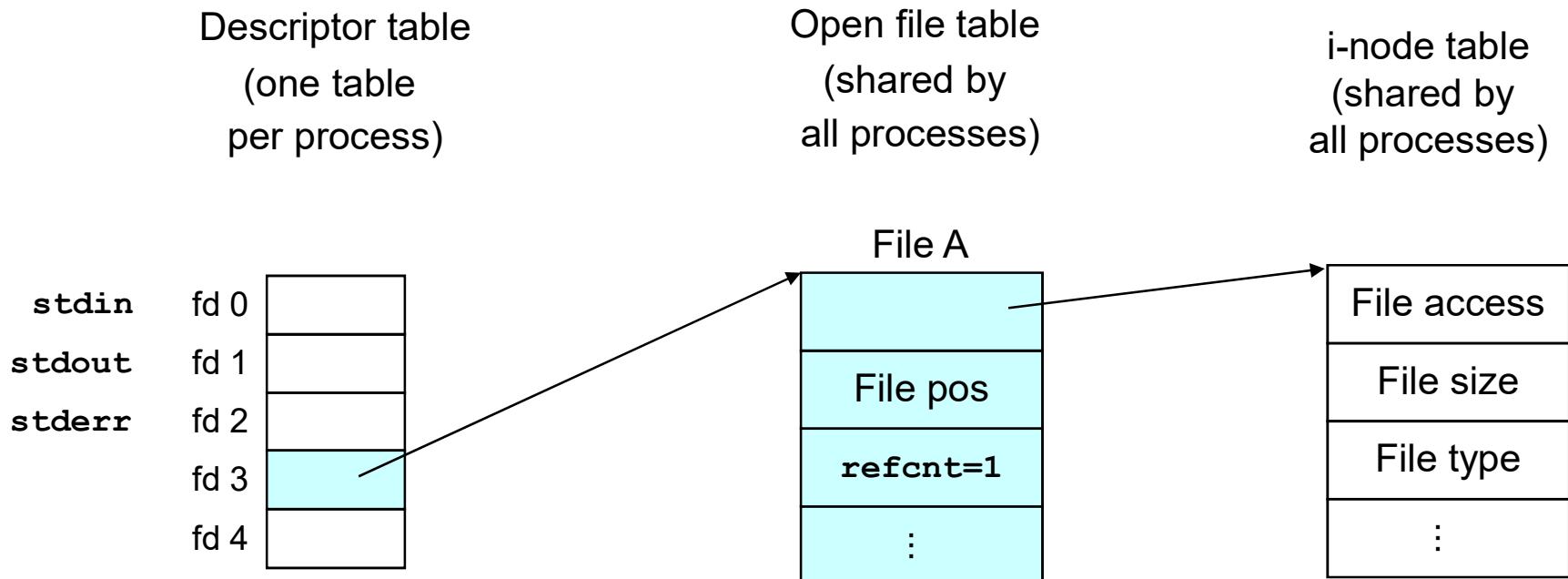
- Each process has associated with it a fixed size **file descriptor table**
- The file descriptor is just the index into this table!
- Each active entry in the table identifies an entry in a shared system wide **open file table**
- Entries are created in the open file table each time **open()** succeeds



Open File Table

- Shared by all processes
- Each entry in the open file table contains:
 - a pointer to an entry in the **i-node table** that corresponds to that file current position, and reference count of its usage
- `close()` decrements count
 - contains the info in the file's stat structure
 - may contain pointers to buffers/caches for the file/device
 - identifies legal operations on a file/device

The Kernel View



inode details

- Every file is associated with a potentially unique inode.
- The inode contains information about the file and the inode itself, like:
 - File type (regular, link, directory, etc.)
 - Number of Hard Links to the inode
 - Associated file byte stream length in bytes
 - Device ID where the file is located (/dev/hda1)
 - Inode number of this file
 - File owner's useuid and groupid
 - mtime, atime, and ctime
 - permissions (rwx)
- ls -i (ls -lif)
- The stat command
- inode table part of the file system

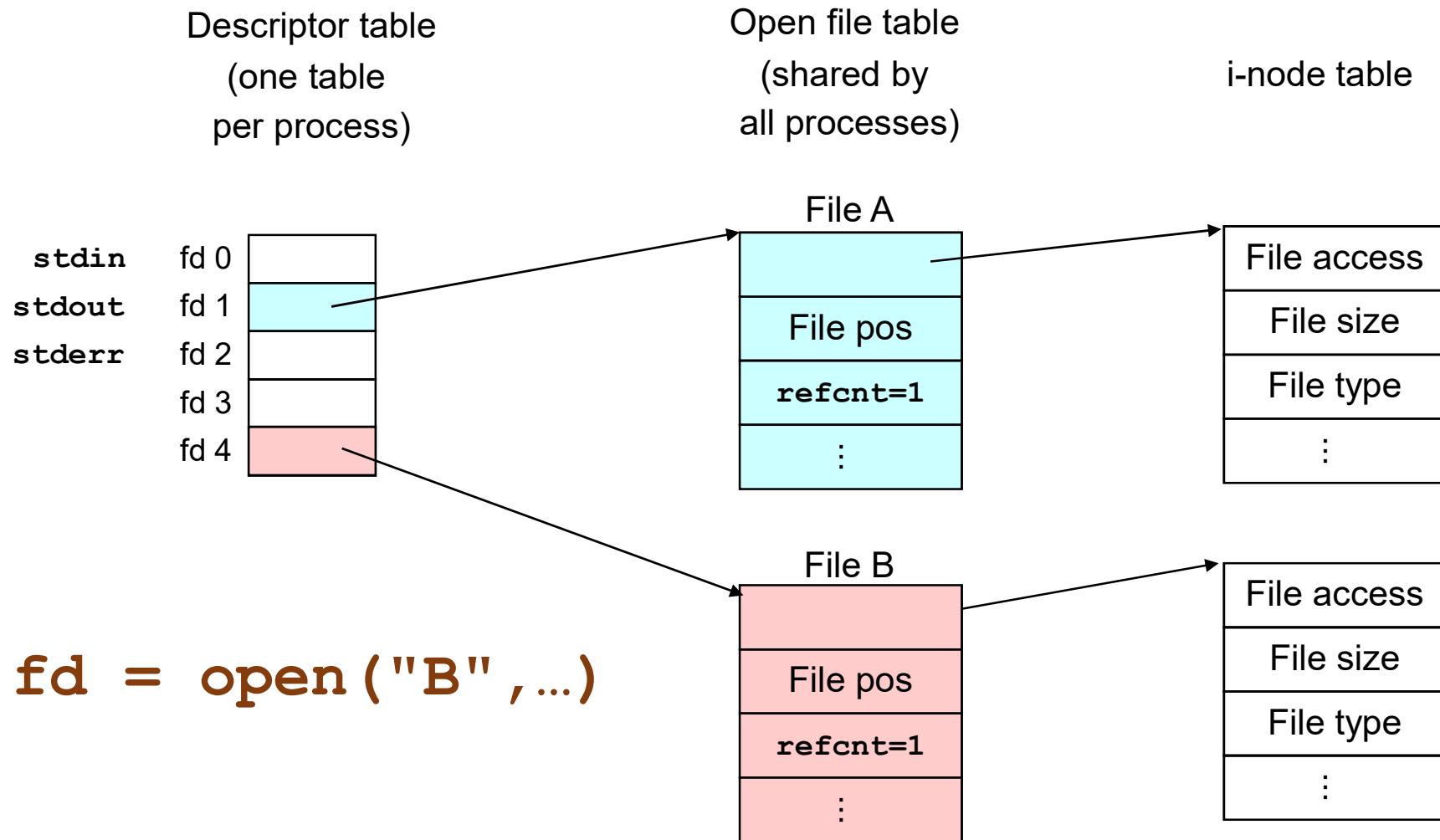
To the Device

- Unix I/O uses the open file table and i-node table to determine the “device” specific code for the standard operations (**open, close read, write...**)
- These routines use buffers identified by the i-node table
- Buffers are caches of disk blocks
- Changes to buffers result in writes being scheduled

Sharing Files

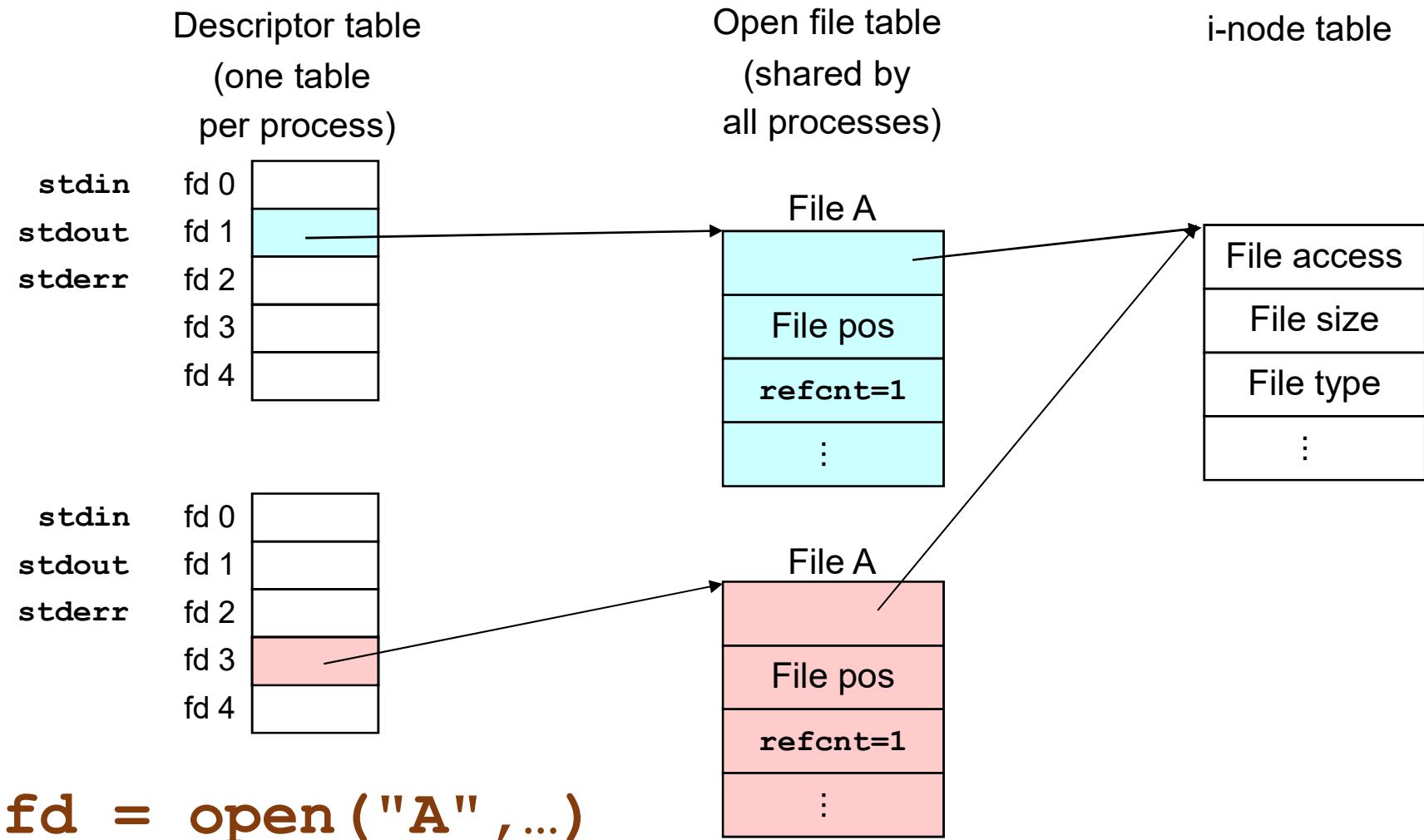
- At this point we have
 - File descriptors table
 - The open file table
 - i-nodes table
- It is relatively easy to explain what happens when file sharing results from:
 - Open's in the same process
 - Open's in different processes
 - Fork's

Actions on `open()`



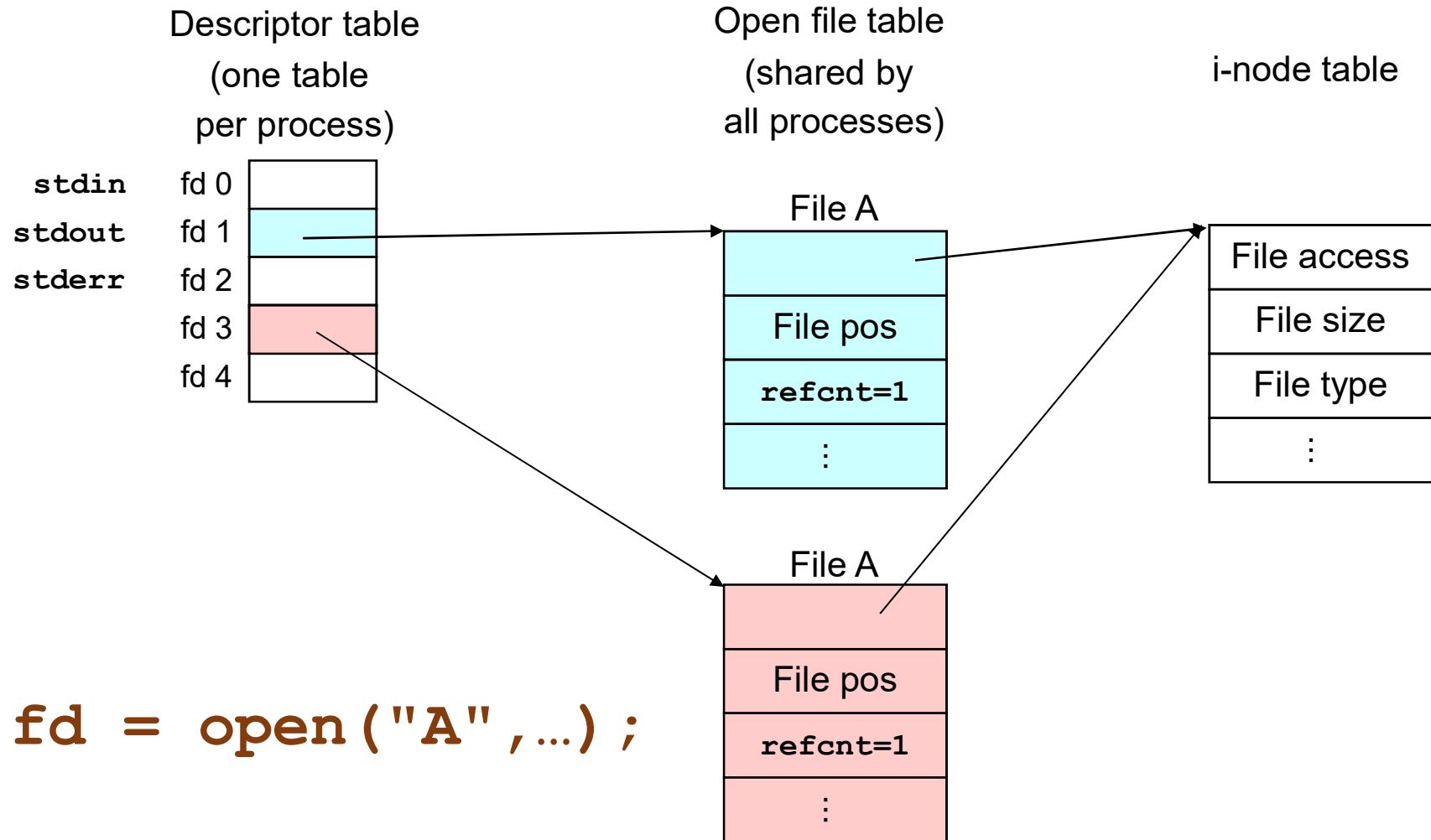
Adapted from: *Computer Systems: A Programmer's Perspective*

Same File Different Process



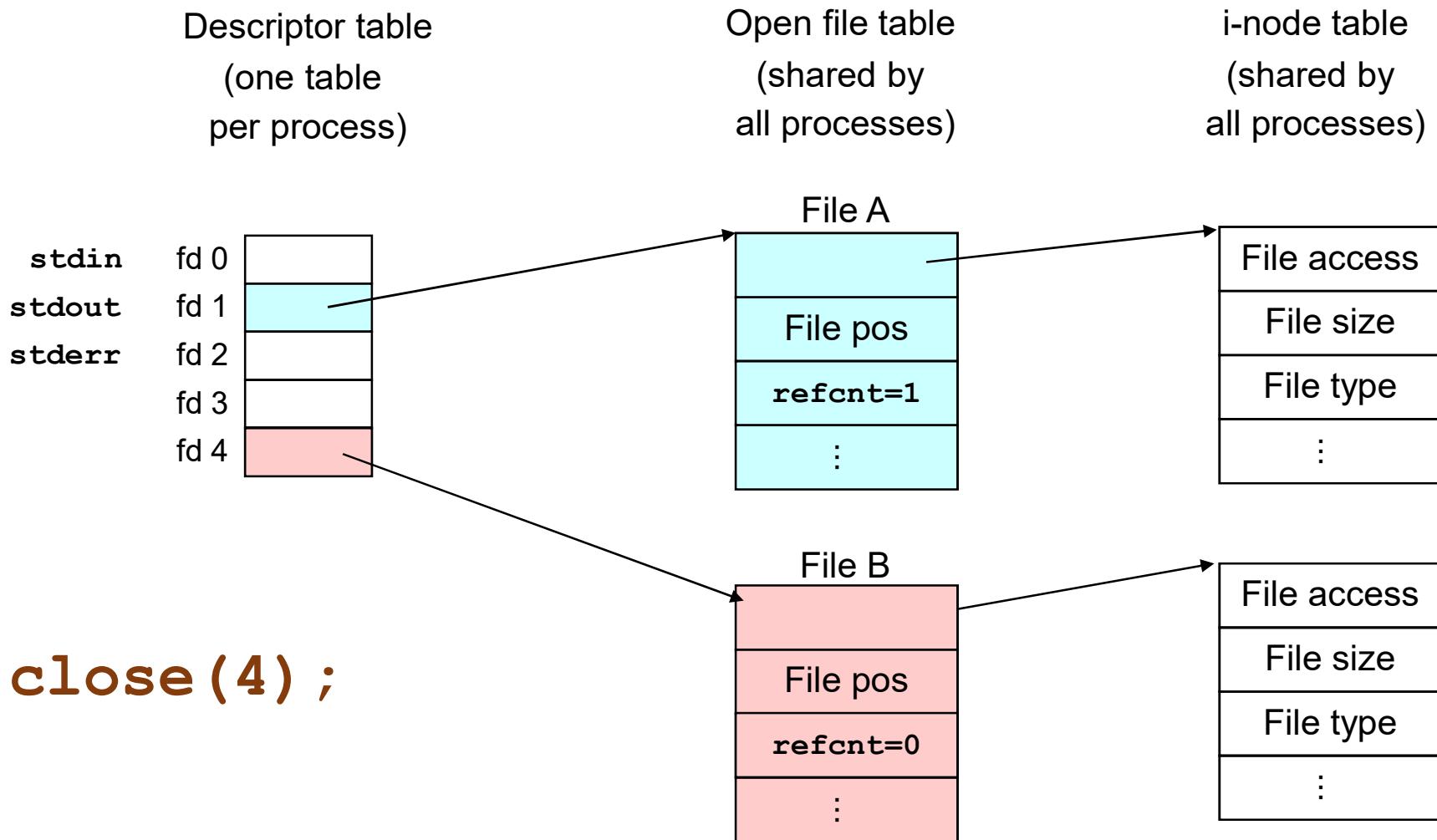
Adapted from: *Computer Systems: A Programmer's Perspective*

Same File Same Process



Adapted from: *Computer Systems: A Programmer's Perspective*

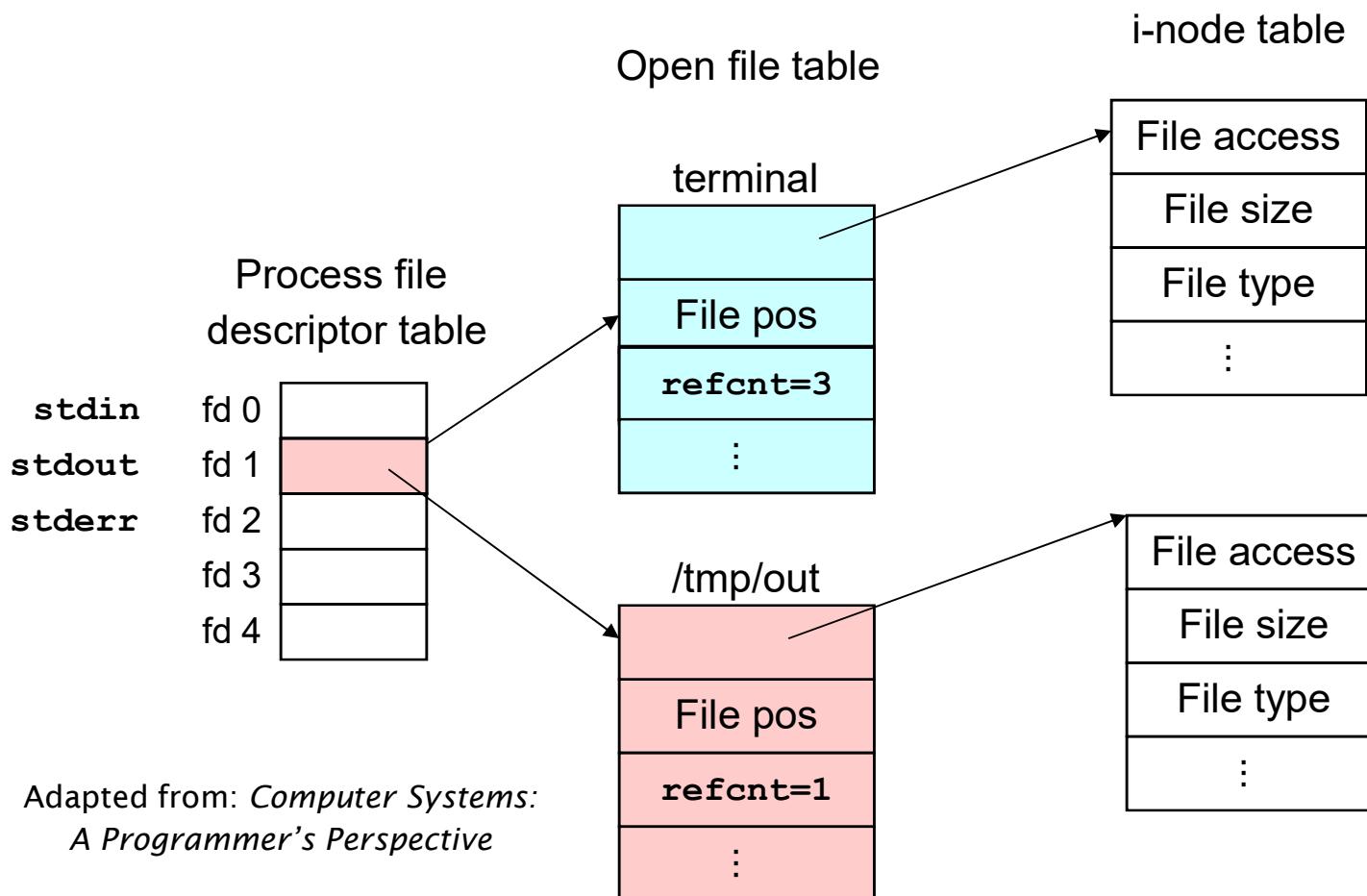
Close()



I/O Redirection

COMOX (114) : `ls > /tmp/out`

- The above causes standard output (file descriptor 1) to be set to /tmp/out



dup2

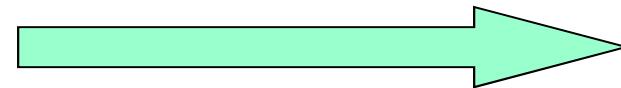
- The Unix system call dup2, which has the form:

dup2 (fd, newfd)

copies fd to newfd in the descriptor table.

fd 0	
fd 1	a
fd 2	
fd 3	
fd 4	b

dup2 (4, 1)

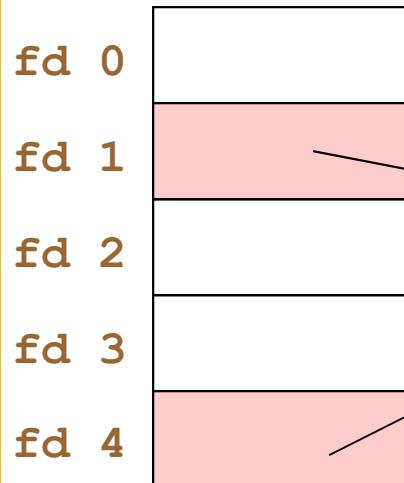


fd 0	
fd 1	b
fd 2	
fd 3	
fd 4	b

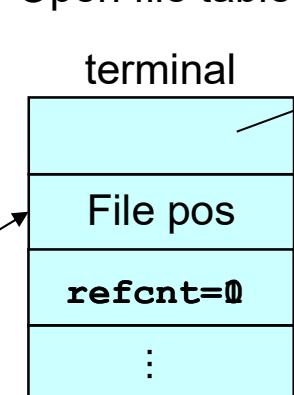
dup2 example

```
open ("/tmp/out", ...);  
dup2 (4, 1);  
close (4);
```

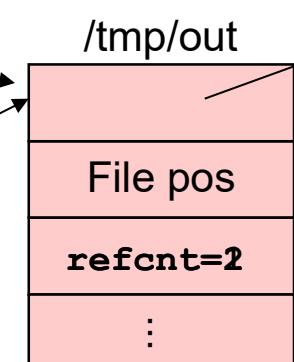
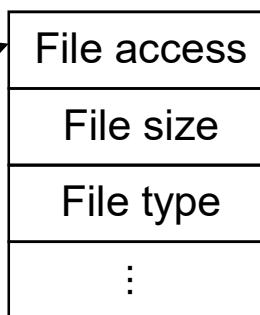
Process file descriptor table



Open file table



i-node table



Sharing Files

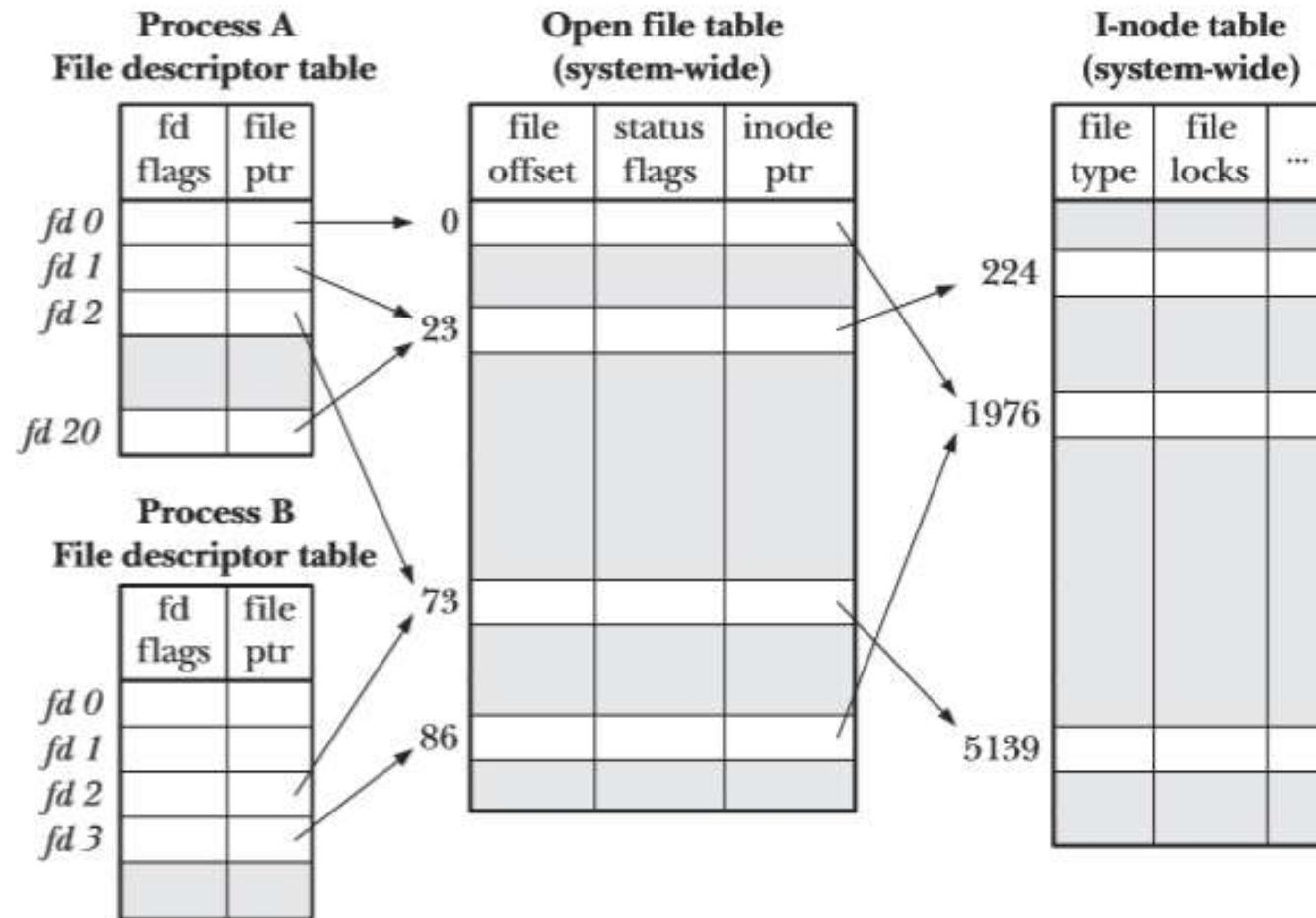


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Process A (PA): fd1 & fd20 share same file description (entry 23). This is due to dup/dup2 or fcntl calls.

Process A (PA) & Process B (PB): fd2 (PA) and fd3 (PB) share the same file description (entry 73). This is due to a fork.

Process A (PA) & Process B (PB): fd0 (PA) and fd3 share the same i-node (1976). This is due that the two processes independently call open with the same filename.

9-UNIX File I/O Calls

Linux Program Interface
Chapter 3-4



10-UNIX fork & wait



Process Creation

Chapter 24

The *fork()* System Call

- *fork()* creates a new process, the *child*, which is an almost exact duplicate of the calling process, the *parent*.
 - The *child* has its own process ID.
 - The *child* inherits the same stack, data and heap.
 - After the *fork()*, both processes can make changes independently.

The *fork()* System Call

Call: `#include <unistd.h>`
`pid_t fork (void);`

In **parent**: returns process ID of child on success,
or -1 on error;

If successfully created, **child** always returns a 0.



wait System Calls

- wait
- waitpid
- waitid
- They all wait for a process to Change State

wait Calls

A state change is considered to be:

- the child was stopped by a signal
- the child was resumed by a signal
- the child terminated
 - In the case of a terminated child, performing a wait allows the system to release the resources associated with the child
 - if a wait is not performed, then the terminated child remains in a "zombie" state

wait() System Call

It suspends execution of the calling process until one of its children terminates. A simple call without much flexibility.

```
Call: #include <sys/wait.h>  
      pid_t wait( int *status);
```

Returns process ID of terminated child,
or -1 on error

waitpid() System Call (1 of 2)

- It suspends execution of the calling process until a child specified by *pid* argument has a **changed state**.
- Has more flexibility than *wait()* but also more complexity.

waitpid() function

(2 of 2)

```
Call: #include <sys/types.h>
      #include <sys/wait.h>
      pid_t waitpid( pid_t pid, int *status, int opts )
```

The value of PID can be:

Tag	Description
< -1	meaning wait for any child process whose process group ID is equal to the absolute value of <i>pid</i> .
-1	meaning wait for any child process.
0	meaning wait for any child process whose process group ID is equal to that of the calling process.
> 0	meaning wait for the child whose process ID is equal to the value of <i>pid</i> .

waitpid() does not block

wait vs waitpid

wait

- Can block the caller until a child process terminates
- If the caller has multiple children, **wait** returns when one terminates

waitpid

- Has an option that prevents it from blocking.
- If the caller has multiple children, **waitpid** has a number of options that control which process it waits for

Options for *waitpid()*

Zero or more of the following constants can be OR-ed.

WNOHANG

Return immediately if no child has exited.

(Demands status information immediately!)

WUNTRACED

Also return if a child has stopped. Status for traced children which have stopped is provided even if this option is not specified.

(Reports on stopped child processes as well as terminated ones!)

Return value

The process ID of the child which exited:

-1 on error;

0 if WNOHANG was used and no child was available.

The Wait Status Field

wait() and ***waitpid()*** return a pointer to the status of the ‘child’ which terminated:

- The child terminated by calling `_exit()` (or `exit()`) specifying an integer *exit status*
- The child was terminated by the delivery of an unhandled signal
- The child was stopped by a signal, and `waitpid()` was called with the **WUNTRACED** flag
- The child was resumed by a **SIGCONT** signal, and `waitpid()` was called with the **WCONTINUED** flag.

The Wait Status Field

Status can be tested using one of three POSIX macros:

WIFEXITED(status): true if normal exit

WEXITSTATUS(status) fetches exit/return value

WIFSIGNALED(status): true if abnormally exited

WTERMSIG(status) fetches signal number

WIFSTOPPED(status): true if stopped

WSTOPSIG(status) fetches signal number

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    pid_t pid;
    int status;

    if( (pid = fork() ) == 0 )
    { /* child */
        printf("I am a child with pid = %d\n", getpid());
        sleep(10);
        printf("Child. Child terminates\n");
        _exit(EXIT_SUCCESS);
    }
}
```

Example: waitpid (1 of 2)

(2 of 2)

```
else
{ /* parent */
    while (1) {
        waitpid( pid, &status, WUNTRACED );
        if( WIFSTOPPED(status) ){
            printf("Parent. Child stopped, signal(%d)\n",
                   WSTOPSIG(status));
            continue;
        }
        else if( WIFEXITED(status) ){
            printf("Parent: Normal termination with
                   status(%d)\n",
                   WEXITSTATUS(status));
            break;
        }
        else if (WIFSIGNALED(status)) {
            printf("Parent: Abnormal termination,
                   signal(%d)\n",
                   WTERMSIG(status));
            exit(EXIT_SUCCESS);
        }
    } /* while */
} /* parent */    return EXIT_SUCCESS;
} /* main */
```





Process Data



Process Data

- Since a child process is a **copy** of the parent, it has copies of the parent's data.
- A change to a variable in the child will **not** change that variable in the parent.

Example (globex.c) (1 of 3)

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int globvar = 6;
char buf[] = "stdout write\n";

int main(void)
{
    int w = 88;
    pid_t pid;
```

continued

```
write( 1, buf, sizeof(buf)-1 ) ;           (2 of 3)
printf( "Before fork()\n" ) ;

if( (pid = fork()) == 0 ) {
    /* child */
    globvar++;
    w++;
}
else if( pid > 0 )                  /* parent */
    sleep(2);
else
    perror( "fork error" );

printf( "pid = %d, globvar = %d, w = %d\n",
        getpid(), globvar, w );
return EXIT_SUCCESS;
} /* end main */
```

Output (3 of 3)

```
$ globex
stdout write      /* write not buffered */

Before fork()
pid = 430, globvar = 7, w = 89 /*child chg*/
pid = 429, globvar = 6, w = 88 /* parent no
                                chg */
-----
$ globex > temp.out
$ cat temp.out
stdout write
Before fork()
pid = 430, globvar = 7, w = 89
Before fork() /* fully buffered */
pid = 429, globvar = 6, w = 88
```

Process File Descriptors

Process File Descriptors

A child and parent have copies of the file descriptors, but the R-W pointer is maintained by the system:

the R-W pointer is shared

This means that a `read()` or `write()` in one process will affect the other process since the R-W pointer is changed.

Example: File used across processes (1 of 5) (shfile.c)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>

void printpos(char *msg, int fd);
void fatal(char *msg);

int main(void)
{ int fd; /* file descriptor */
  pid_t pid;
  char buf[10]; /* for file data */
  :
```

Example: File used across processes (2 of 5)

```
if ((fd=open("data-file", O_RDONLY)) < 0)
{ perror("error on open");
}
    read(fd, buf, 10); /* move R-W ptr */

    printpos( "Before fork", fd );

    if( (pid = fork()) == 0 )
    { /* child */
        printpos( "Child before read", fd );
        read( fd, buf, 10 );
        printpos( " Child after read", fd );
    }
```

continued

24

Example: File used across processes (3 of 5)

```
else if( pid > 0 )
    { /* parent */
        wait((int *)0);
        printpos( "Parent after wait", fd );
    }
else
    perror( "fork" );
}
```

Example: File used across processes (4 of 5)

```
void printpos( char *msg, int fd )
/* Print position in file */
{
    long int pos;
    if( (pos = lseek(fd, 0L, SEEK_CUR) ) < 0L )
        perror("lseek");
    printf( "%s: %ld\n", msg, pos );
}

/* 0L = Long constant.
   Changes the file position by zero bytes.
   Then return the new Current Position. */
```

Output (5 of 5)

```
$ shfile
```

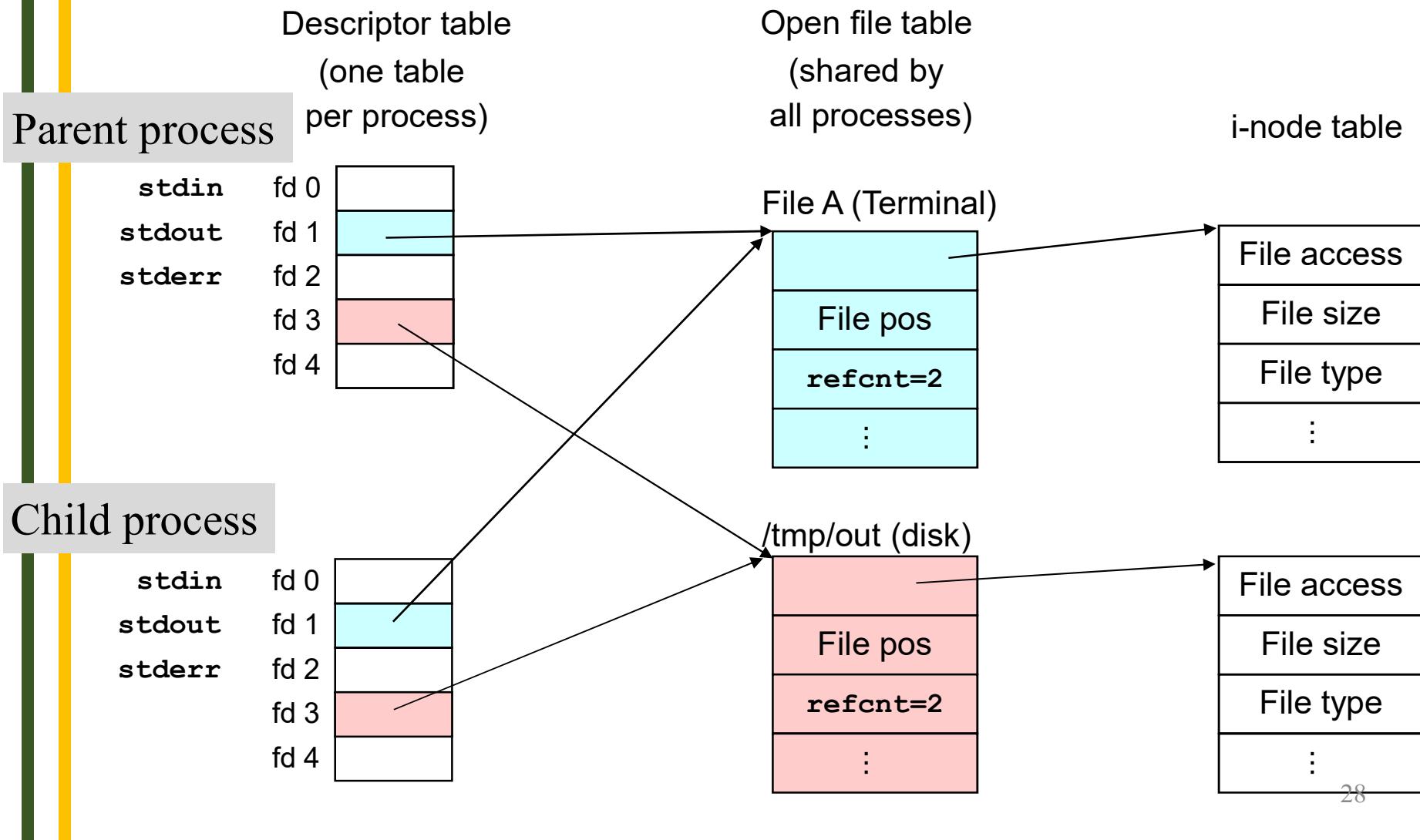
```
Before fork: 10  
Child before read: 10  
Child after read: 20  
Parent after wait: 20
```

What's happened?

Parent and child both use the same file descriptor,
so the current position of *fd* changed in both.

Child process inherits its parent's open files

After fork() call: Child's descriptor table is the same as parent's,
and +1 to each refcnt



Parent-Child relationship with the file.

- The parent opens the file with the fd.
- The child inherits the open file from the parent.
- The child can close the file, but it will still be open by the parent.



Special Exit Cases

Special Exit Cases

Two special cases:

- 1) A child exits when its parent is not currently executing `wait()`
the child becomes a ***zombie***
status data about the child is stored until the parent does a
`wait()`

- 2) A parent exits when 1 or more children are still running
children are adopted by the system's initialization process
(/etc/init)
It can then monitor/kill them



10-UNIX fork & wait

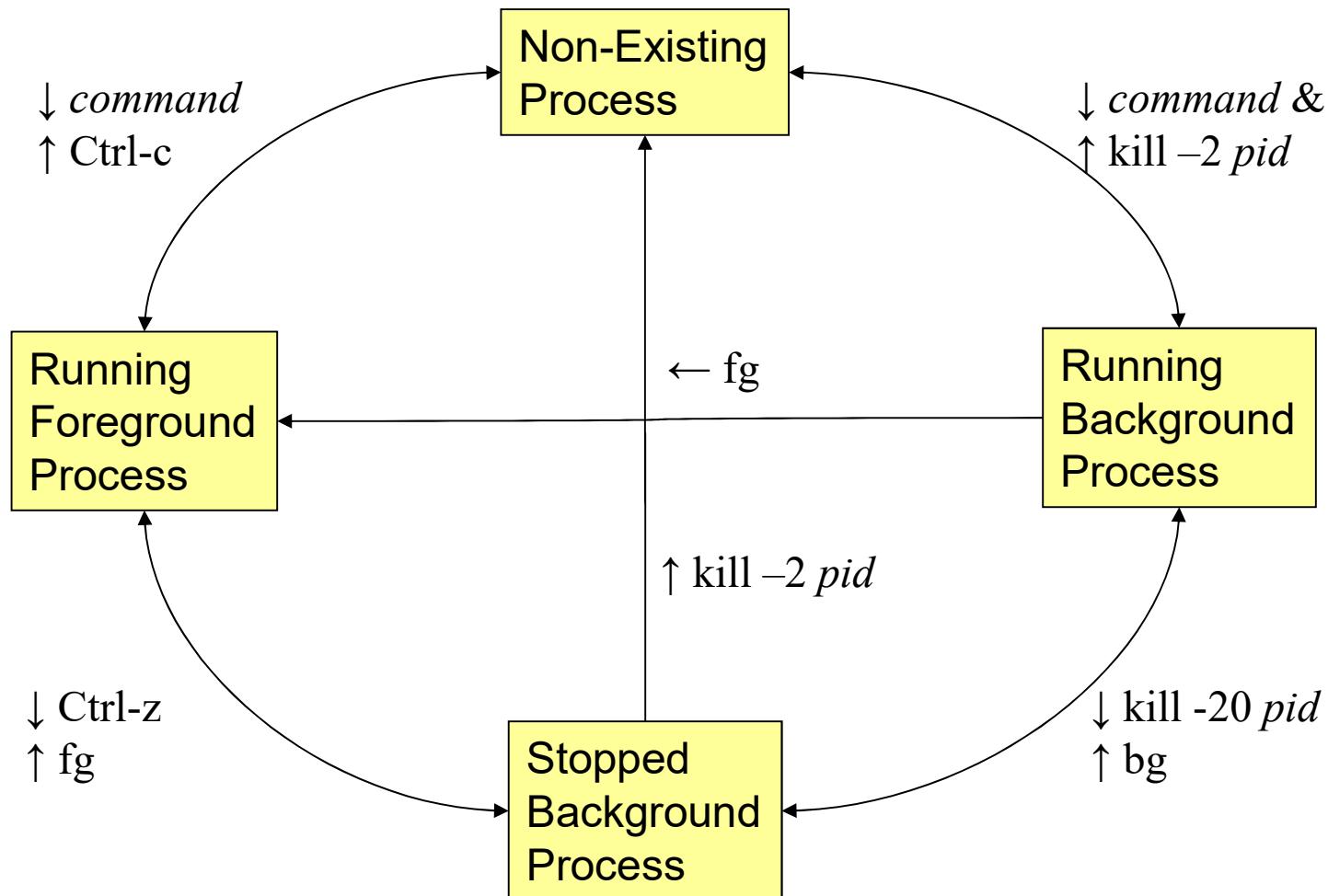
The End

11-UNIX

The **signal** System Call

Chapter 20,24,25,26

UNIX Process Control



sleep – system command

Definition - delay for a specified amount of time

Form - sleep NUMBER[SUFFIX]...
sleep OPTION

Description - Pause for NUMBER seconds.

UNIX Process Control

[Demo of UNIX process control using **infloop.c**]

```
*****  
/*                      infloop.c                      */  
*****  
  
#include <stdio.h>  
#include <stdlib.h>  
  
int main(void)  
  
/* print doing something, rest, and repeat again */  
{  
    for (;;) {  
        printf("doing something ...\\n") ;  
        sleep(2);  
    }  
    return EXIT_SUCCESS;  
}
```

Running *infloop*

```
[bielr@athena ClassExamples]> infloop
```

```
doing something ...
```

```
^C
```

```
[bielr@athena ClassExamples]>
```

Signals

(LPI page 388)

A *signal* is a notification to a process that an event has occurred.

Signals can come from another process or the kernel. A process can also send a signal to itself.

Signals from the Kernel

- Types of events that cause the kernel to generate a signal:
 - Hardware exceptions
 - Problem with a machine instruction
 - Divide by zero
 - A reference to inaccessible memory
 - User-typed special characters
 - CTRL-C or CTRL-Z
 - Software event
 - Timer went off
 - Child of this process terminated
 - Terminal window was resized

Signals

Standard signals in Linux have:

- names. Ex. SIGINT
- numbers. Ex. SIGINT has the number 2
- Linux has 31 standard signals
- Standard signals are sometimes called *Traditional* signals
- *Realtime signals* (LPI section 22.8)
 - Extension set to Standard signals
 - Allow signals to be queued
 - Allow data to accompany the signal

Signal Types (31 in POSIX) Table 20-1

Name	Description	Default Action
SIGINT	Interrupt character typed	terminate process
SIGQUIT	Quit character typed (^\\)	create core image
SIGKILL	Sure kill	terminate process
SIGSEGV	Invalid memory reference	create core image
SIGPIPE	Write on pipe but no reader	terminate process
SIGALRM alarm()	clock 'rings'	terminate process
SIGUSR1	user-defined signal type	terminate process
SIGUSR2	user-defined signal type	terminate process
SIGCHLD	user-defined signal type	ignore

See man 7 signal

For more, see table 20-1 in our LPI book, page 396.

Process Control Implementation (1 of 2)

Exactly what happens when you:

Type **Ctrl-c**?

- Keyboard sends hardware interrupt
- Hardware interrupt is handled by OS
- OS sends a 2/SIGINT signal

Type **Ctrl-z**?

- Keyboard sends hardware interrupt
- Hardware interrupt is handled by OS
- OS sends a 20/SIGTSTP signal

Process Control Implementation (2 of 2)

Exactly what happens when you:

Issue a “**kill –sig pid**” command?

- OS sends a **sig signal** to the process whose id is *pid*

Issue a “**fg**” or “**bg**” command?

- OS sends a **18/SIGCONT signal**
- fg – foreground process, a shell command
- bg – background process, a shell command

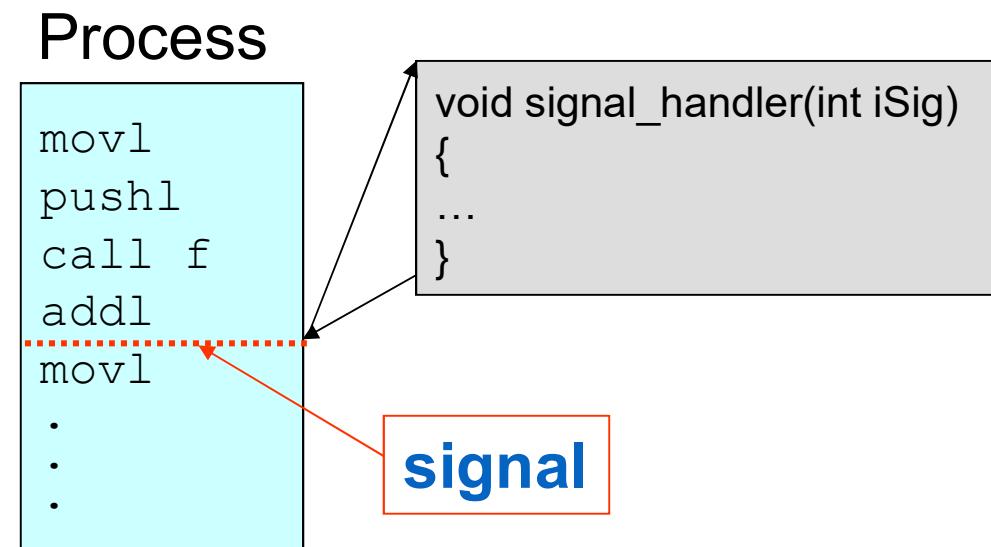
Definition of Signal

Signal: A signal is an *asynchronous* event which is delivered to a process.

Event gains attention of the OS

- OS stops the application process immediately, sending it a signal
- **Signal handler** executes to completion
- Application process resumes where it left off

e.g. user types
ctrl-C
at anytime.



Examples of Signals (1 of 2)

User types Ctrl-c



- Event gains attention of OS
- OS stops the application process immediately, sending it a 2/SIGINT signal
- Signal handler for 2/SIGINT signal executes to completion
Default signal handler for 2/SIGINT signal exits process

Examples of Signals (2 of 2)

Process makes illegal memory reference

- Event gains attention of OS
- OS stops application process immediately, sending it a 11/SIGSEGV signal
- Signal handler for 11/SIGSEGV signal executes to completion

```
int *ptr;  
*ptr = 20;
```

*Default signal handler for 11/SIGSEGV signal prints
“segmentation fault” and exits process*

PS: Since **ptr** is uninitialized, we would get a SEGFAULT

If we said “ptr = 20”, it would not get fault until you tried to access *ptr

Example of Signal:

*fork,
exit,
wait,
& execve*

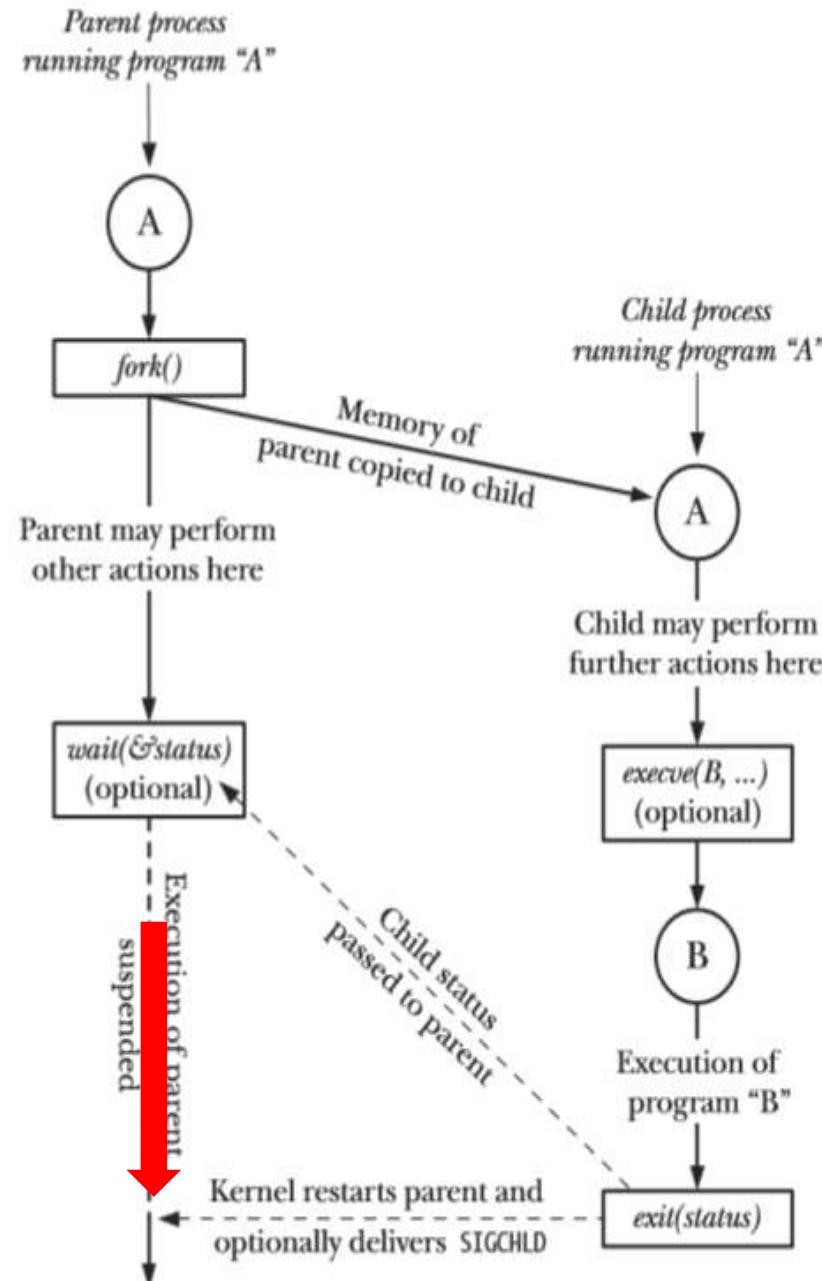


Figure 24-1: Overview of the use of `fork()`, `exit()`, `wait()`, and `execve()`

Sending Signals via Keystrokes

Three signals can be sent from keyboard:

Ctrl-c → 2/SIGINT signal

Default handler exits process

Ctrl-z → 20/SIGTSTP signal

Default handler suspends process

Ctrl- → 3/SIGQUIT signal

Default handler exits process

Sending Signals via Commands

`kill -signal pid`

*Send a signal of type **signal** to the process with id **pid***

Can specify either signal type name (-SIGINT) or number (-2)

No signal type name or number specified => sends 15/SIGTERM signal

Default 15/SIGTERM handler exits process

Examples:

`kill -2 1234`

`kill -SIGINT 1234`

Same as pressing Ctrl-c if process 1234 is running in foreground

signal call

```
#include <signal.h>

void ( signal (int sig, void (*handler)(int)) ) (int);
```

Returns previous signal disposition on success,
or **SIG_ERR** on error

- *sig* – the signal whose disposition we wish to change
- *handler* – the address of the function that should be called when this signal is delivered

Form:

```
void handler (int sig)
{
    /* code for the handler */
}
```

Sending Signals via Function Call

raise()

```
int raise(int iSig);
```

- Commands OS to send a signal of type **iSig** to current process, **itself**.
- Returns 0 to indicate success, non-0 to indicate failure

Example

```
int ret = raise(SIGINT);
/* Process commits termination. */
/* where SIGINT is the number    */
/* of the signal to send          */
```

Sending Signals via Function Call

kill()

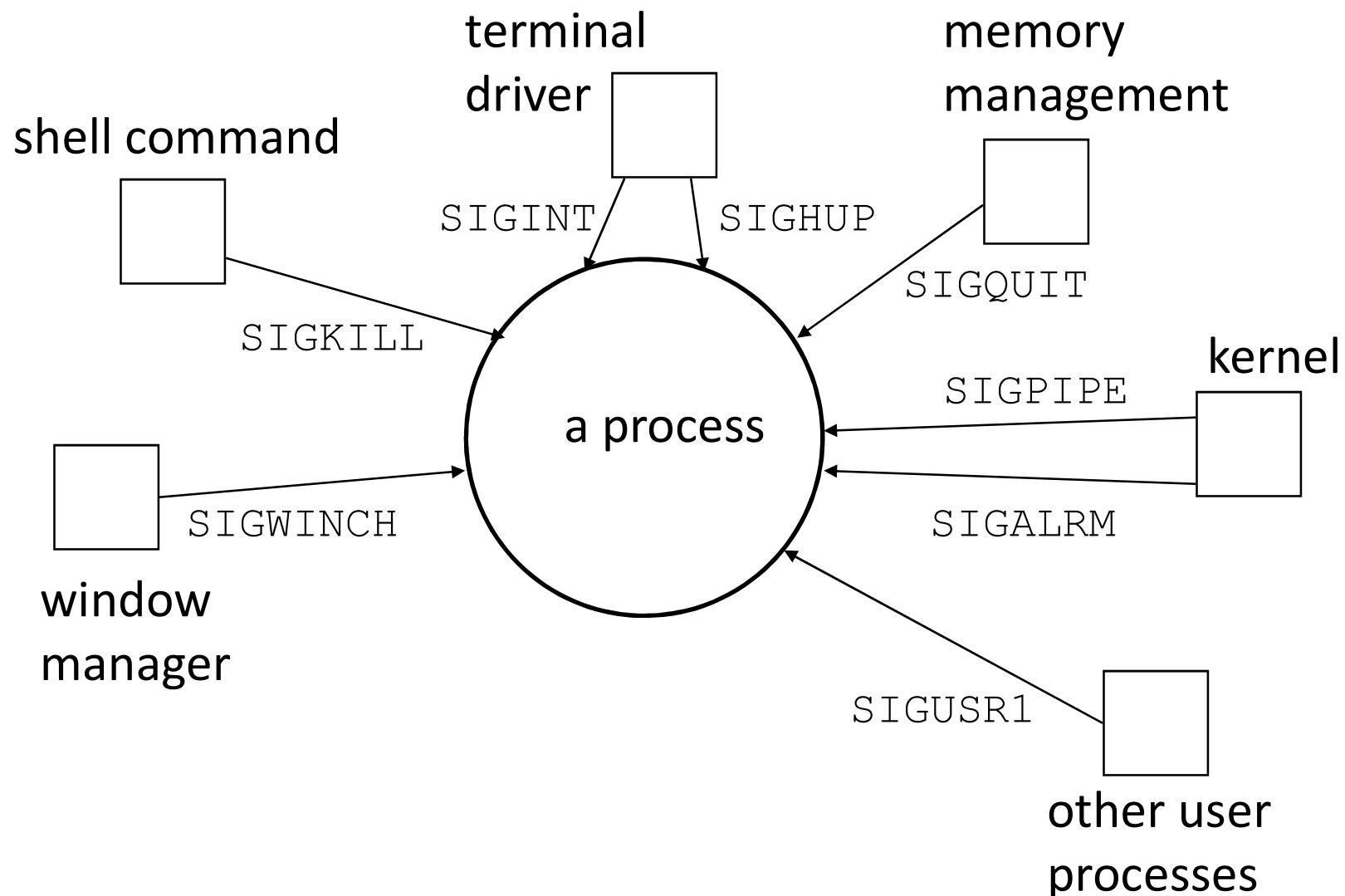
```
int kill(pid_t iPid, int iSig);
```

- Sends a **iSig** signal to the process whose id is **iPid**
- Equivalent to **raise(iSig)** when **iPid** is the id of current process

Example:

```
pid_t iPid = getpid(); /* Process gets its id.*/
kill(iPid, SIGINT);    /* Process sends itself a
                           SIGINT signal */
```

Signal Sources



Responding to a Signal

A process can:

- ignore/discard the signal
 - not possible with SIGKILL or SIGSTOP
- execute a **signal handler** function, and then possibly resume execution or terminate
- carry out the default action for that signal

The **choice** is called the process' *signal disposition*

Signal delivery & handler execution

(LPI Page 399)

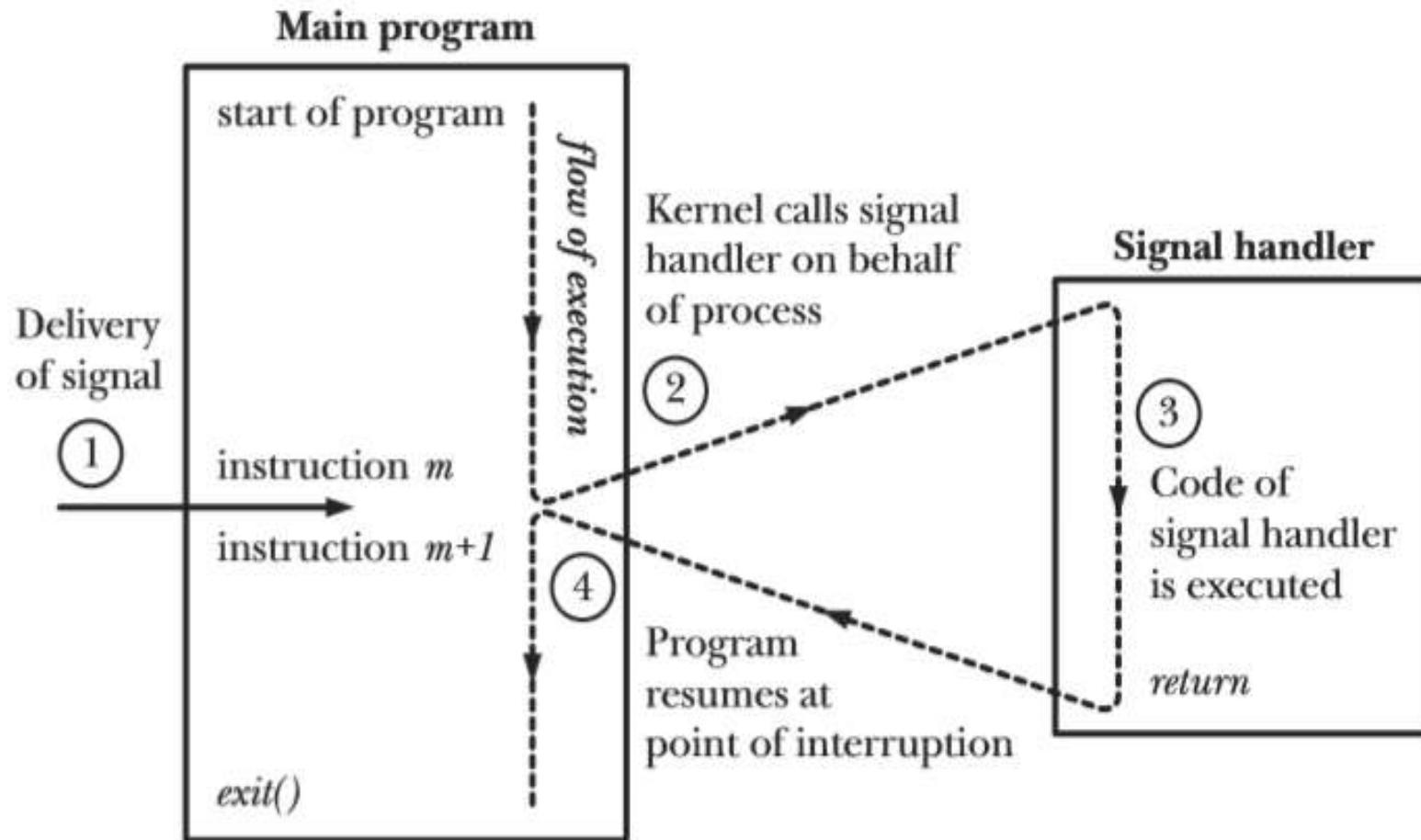


Figure 20-1: Signal delivery and handler execution

Example: ouch.c

(LPI page 399)

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

static void sigHandler(int sig) {
    printf("Ouch!\n"); /* UNSAFE (see Section 21.1.2) */
}

int main(int argc, char *argv[]) {
    int j;
    if (signal(SIGINT, sigHandler) == SIG_ERR)
        fprintf(stderr, "signal");
    for (j = 0; ; j++) {
        printf("%d\n", j);
        sleep(3); /* Loop slowly... */
    }
    return EXIT_SUCCESS;
}
```

Running the *ouch* program

```
[bielr@athena ClassExamples]> ouch
```

```
0
```

```
1
```

```
^C Ouch!
```

```
2
```

```
3
```

```
^C Ouch!
```

```
4
```

```
5
```

```
6
```

```
^C Ouch!
```

```
7
```

```
8
```

```
^\Quit (core dumped)
```

```
[bielr@athena ClassExamples]>
```



Chapter 21. Signals: Signal Handlers

Async-Signal-Safe Function (1 of 2)

An Async-Signal-Safe function is one in which the implementation guarantees to be safe when calling from the signal handler.

A function is Async-Signal-Safe because it is not interruptible by a signal handler.

Note: /* UNSAFE (see Section 21.1.2)
Page 422 */

Async-Signal-Safe Function (2 of 2)

For example:

Suppose a program is in the middle of a call to `printf(3)` and a signal occurs whose handler itself calls `printf()`.

In this case, the output of the two `printf()` statements would be intertwined. To avoid this, the handler should not call `printf()` itself when `printf()` might be interrupted by a signal.

Question: Then, what function(s) would we use instead?

Async Signal Safe Functions

(Table 21-1/Page 426, **UPPER** half of table)

Table 21-1: Functions required to be **async-signal-safe** by POSIX.1-1990, SUSv2, and SUSv3

<code>_Exit() (v3)</code>	<code>getpid()</code>	<code>sigdelset()</code>
<code>_exit()</code>	<code>getppid()</code>	<code>sigemptyset()</code>
<code>abort() (v3)</code>	<code>getsockname() (v3)</code>	<code>sigfillset()</code>
<code>accept() (v3)</code>	<code>getsockopt() (v3)</code>	<code>sigismember()</code>
<code>access()</code>	<code>getuid()</code>	<code>signal() (v2)</code>
<code>aio_error() (v2)</code>	<code>kill()</code>	<code>sigpause() (v2)</code>
<code>aio_return() (v2)</code>	<code>link()</code>	<code>sigpending()</code>
<code>aio_suspend() (v2)</code>	<code>listen() (v3)</code>	<code>sigprocmask()</code>
<code>alarm()</code>	<code>lseek()</code>	<code>sigqueue() (v2)</code>
<code>bind() (v3)</code>	<code>lstat() (v3)</code>	<code>sigset() (v2)</code>
<code>cgetispeed()</code>	<code>mkdir()</code>	<code>sigsuspend()</code>
<code>cgetospeed()</code>	<code>mkfifo()</code>	<code>sleep()</code>
<code>csetispeed()</code>	<code>open()</code>	<code>socket() (v3)</code>
<code>csetospeed()</code>	<code>pathconf()</code>	<code>socketmark() (v3)</code>
<code>chdir()</code>	<code>pause()</code>	<code>socketpair() (v3)</code>
<code>chmod()</code>	<code>pipe()</code>	<code>stat()</code>
<code>chown()</code>	<code>poll() (v3)</code>	<code>symlink() (v3)</code>
<code>clock_gettime() (v2)</code>	<code>posix_trace_event() (v3)</code>	<code>sysconf()</code>
<code>close()</code>	<code>pselect() (v3)</code>	<code>tcdrain()</code>

Special Sigfunc* Values used in signal() function

<u>Value</u>	<u>Meaning</u>
SIG_IGN	Ignore / discard the signal. <i>Example:</i> To ignore a ctrl-c command from the command line.
SIG_DFL	Use default action to handle signal. <i>Example:</i> To reset system so that SIGINT causes a termination at any place in our program.
SIG_ERR	Returned by signal () as an error.

Handling Multiple Signals

If many signals of the *same* type are waiting to be handled (e.g. two SIGINTs), then most UNIXs will only deliver *one* of them. (Signals are not queue). The others are thrown away.

If many signals of *different* types are waiting to be handled (e.g. a SIGINT, SIGSEGV, SIGUSR1), they are not delivered in any fixed order.

pause()

Suspend the calling process until a signal is caught.

```
#include <errno.h>
#include <unistd.h>
int pause(void);
```

Returns -1 with `errno` assigned `EINTR`.
(Linux assigns it `ERESTARTNOHAND`).

`pause()` only returns after a signal handler has finished.

pause() Example

(1 of 3)

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void sig_usr( int signo ); /* handles two signals */
int main() {
    int i = 0;
    if( signal( SIGUSR1,sig_usr ) == SIG_ERR )
        printf( "Cannot catch SIGUSR1\n" );
    if( signal( SIGUSR2,sig_usr ) == SIG_ERR )
        printf("Cannot catch SIGUSR2\n");
    while(1) {
        printf("%2d\n", i );
        pause();
        /* pause until signal handler has processed signal */
        i++;
    }      /* end of while loop */
    return 0;
}      /* end of main */
```

pause() – Example

(2 of 3)

```
/* argument is signal number */
void sig_usr( int signo )
{
    if( signo == SIGUSR1 )
        printf("Received SIGUSR1\n");
    else if( signo == SIGUSR2 )
        printf("Received SIGUSR2\n");
    return;
}
```

Note: Executing a program with an “&” puts the program in the background.

pause() – Example

(3 of 3)

```
[bielr@sp2 ClassExamples]> gcc -o pause pause.c
[bielr@sp2 ClassExamples]> pause &
[1] 10885
[bielr@sp2 ClassExamples]> 0

[bielr@sp2 ClassExamples]> ps
  PID TTY          TIME CMD
10819 pts/1    00:00:00 bash
10885 pts/1    00:00:00 pause
10889 pts/1    00:00:00 ps
[bielr@sp2 ClassExamples]> kill -USR1 10885
Received SIGUSR1
 1
[bielr@sp2 ClassExamples]> kill -USR2 10885
Received SIGUSR2
 2
[bielr@sp2 ClassExamples]> kill -USR1 10885
Received SIGUSR1
 3
[bielr@sp2 ClassExamples]> kill -USR2 10885
Received SIGUSR2
 4
[bielr@sp2 ClassExamples]> ps
  PID TTY          TIME CMD
10819 pts/1    00:00:00 bash
10885 pts/1    00:00:00 pause
10929 pts/1    00:00:00 ps
[bielr@sp2 ClassExamples]> kill -SIGKILL 10885
[bielr@sp2 ClassExamples]> ps
  PID TTY          TIME CMD
10819 pts/1    00:00:00 bash
10966 pts/1    00:00:00 ps
[1]+  Killed                  pause
[bielr@sp2 ClassExamples]>
```

kill and raise functions

```
int kill(pid_t pid, int sig);
```

```
int raise(int sig);
```

kill sends a signal to a process or group of processes

pid > 0 – send to process with PID = pid

pid = 0 – send to all processes with PGID = PGID of caller

pid < 0 – send to all processes with PGID = |pid| (to a group)

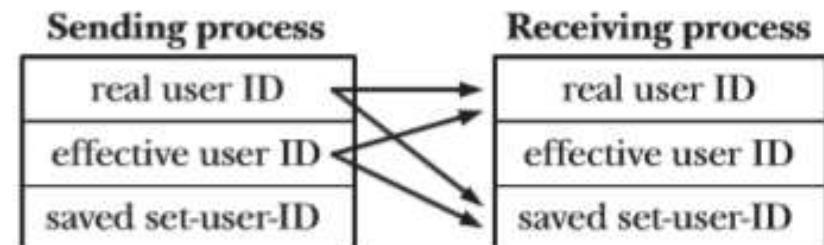
pid = -1 – send to all processes which it has permission to send a signal

To send signals to other processes,

Real user ID/EUID (Effective User Id)

must match

**raise allows a process to
send a signal to itself**



→ indicates that if IDs match,
then sender has permission
to send a signal to receiver

Signal Sets

Multiple signals are represented using a data structure called a *signal set*, provided by the system data type *sigset_t*.

The signal set stores collections of signal types.

Sets are used by signal functions to define which signal types are to be processed.

POSIX contains several functions for creating, changing and examining signal sets.

Prototypes

```
#include <signal.h>
```

```
/* Initialize signal set to contain no member */  
int sigemptyset( sigset_t *set );
```

```
/* Initialize signal set to contain all signal */  
int sigfillset( sigset_t *set );
```

```
/* Add individual signal */  
int sigaddset( sigset_t *set, int signo );
```

```
/* Remove individual signal */  
int sigdelset( sigset_t *set, int signo );
```

```
/* Check to see if a signal is a member of a set */  
int sigismember( const sigset_t *set, int signo );
```

The Signal Mask

For each process, the kernel maintains a *signal mask* – a set of signals whose delivery to the process is currently blocked.

If a signal that is blocked is sent to a process, delivery of that signal is delayed until it is unblocked by being removed from the process signal mask

sigprocmask()

A process uses a signal set to create a mask which defines the signals it is **blocking** from delivery. – good for critical sections where you want to block certain signals.

```
#include <signal.h>
```

```
int sigprocmask( int how,  
                 const sigset_t *set,  
                 sigset_t *oldset);
```

```
/* how – indicates how mask is modified */
```

Note: SIGKILL and SIGSTOP cannot be blocked by sigprocmask.

“how” Meanings

Value	Meaning
SIG_BLOCK	set signals are added to mask
SIG_UNBLOCK	set signals are removed from mask
SIG_SETMASK	set becomes new mask

A Critical Code Region

```
sigset_t newmask, oldmask;  
  
sigemptyset( &newmask );  
sigaddset( &newmask, SIGINT );  
  
/* block SIGINT; save old mask */  
sigprocmask(SIG_BLOCK, &newmask, &oldmask );  
  
/* critical region of code */  
/* where a signal would interfere */  
/* with task */  
  
/* reset mask which unblocks SIGINT */  
sigprocmask( SIG_SETMASK, &oldmask, NULL );
```

Signal - Review

- A signal is a notification that some kind of event has occurred.
- Send to a process by kernel, another process, or by itself.
- Different kind of signals. Each has unique id and purpose.
- Signal is typically asynchronous (unpredictable).
- Signal can be ignored, terminate a process, stop a process, or restart of stopped process. Also, see table 20-1.
- Signal can be also be ignored and handled by a programmer's handler (catcher) function.
 - Later: Recommend to use sigaction() – more flexible/portable

sigaction() system call

- An alternative to *signal()*
- Used to change the action taken by a process on receipt of a specific signal.
- More complex than *signal()* but offers greater flexibility

(See *signal(7)* for an overview of signals.)

sigaction() System Call

```
#include <signal.h>

int sigaction(int signo,
              const struct sigaction *act,
              struct sigaction *oldact );
```

The arguments are explained on the next slide.

Note: why there is no const in front of struct sigaction *oldact ?
Because the call is going to modify it, so no constant!

sigaction() arguments

- *sig* – identifies the signal whose disposition we want to retrieve or change.
- *act* – pointer to a structure specifying a new disposition for the signal
- *oact* – pointer to a structure of the same type & is used to return information about the signal's previous disposition

sigaction Structure

```
struct sigaction
{
    void (*sa_handler)( int );           /* address of handler */

    sigset_t     sa_mask;                /* signal blocked during invocation */

    int  sa_flags;                     /* flags control invocation */
                                         /* (see page 417) */

    void (*sa_sigaction)( int, siginfo_t *, void * );
                                         /* not for application use */

}
```

sa_flags – (typically has a 0 value)

SIG_DFL reset handler to default upon return

SA_SIGINFO denotes extra information is passed to handler

(.i.e. specifies the use of the “second” handler in the structure.

sigaction() Behavior

- A signo signal causes the sa_handler signal handler to be called.
- While sa_handler executes, the signals in sa_mask are blocked.
- sa_handler remains installed until it is changed by another sigaction() call.

Signal Raising

```
int main() {  
    struct sigaction act;  
    act.sa_handler = ouch;  
    sigemptyset( &act.sa_mask );  
    act.sa_flags = 0;  
    sigaction( SIGINT, &act, 0 );  
    while(1) {  
        printf("Hello World!\n");  
        sleep(1);  
    }  
}
```

```
struct sigaction  
{  
    void (*) (int) sa_handler  
    sigset_t sa_mask  
    int sa_flags  
}
```

Set the signal handler to be the function ouch

No flags are needed here.
Possible flags include:
SA_NOCLDSTOP
SA_RESETHAND
SA_RESTART
SA_NODEFER

We can manipulate sets of signals...

This call sets the signal handler for the SIGINT (ctrl-C) signal

```
#include <signal.h>      //the code (that works)
#include <stdlib.h>
#include <stdio.h>

static void ouch(int sig) {
    printf("Ouch!\n"); /* UNSAFE (see Section 21.1.2) */
}

int main(void)
{
    struct sigaction act;
    act.sa_handler = ouch;
    sigemptyset( &act.sa_mask );
    act.sa_flags = 0;
    sigaction( SIGINT, &act, 0 );
    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```

Signals - Ignoring signals

Other than SIGKILL and SIGSTOP, signals can be ignored:

Instead of in the previous program:

```
act.sa_handler = ouch; /* or whatever */
```

We use:

```
act.sa_handler = SIG_IGN;
```

The ^C key will be ignored

Restoring previous action

The third parameter to `sigaction`, **oact**, can be used:

```
/* save old action */  
sigaction( SIGTERM, NULL, &oact );
```

```
/* set new action */  
act.sa_handler = SIG_IGN;
```

```
sigaction( SIGTERM, &act, NULL );
```

```
/* restore old action */  
sigaction( SIGTERM, &oact, NULL );
```

11-UNIX

The **signal** System Call

The End

12-UNIX

Inter-Process Communication (IPC)
Pipe

Chapter 43-44

Overview of IPC in Linux

(Chapter 43)

- communication
 - data transfer
 - shared memory
- signal
 - standard signal
 - realtime signal
- synchronization
 - semaphore
 - file lock
 - mutex (threads)
 - condition variable (threads)

Based on chart
LPI page 878.

Communication expanded

- data transfer
 - byte stream
 - pipe
 - FIFO
 - stream socket
 - pseudoterminal
 - message
 - System V message queue
 - POSIX message queue
 - datagram socket
- shared memory
 - System V shared memory
 - POSIX shared memory
 - memory mapping
 - anonymous mapping
 - mapped file

Synchronization Expanded

- Semaphore
 - System V semaphore
 - POSIX semaphore
 - Names
 - unnamed
- file lock
 - “record” lock (*fctl()*)
 - File lock (*flock()*)
- mutex (threads)
- condition variable (threads)

We know:

- What is a process?
- How a process is created?
- How a process can replace its image by running a new program ?
- Parent Process waits for a Child Process.

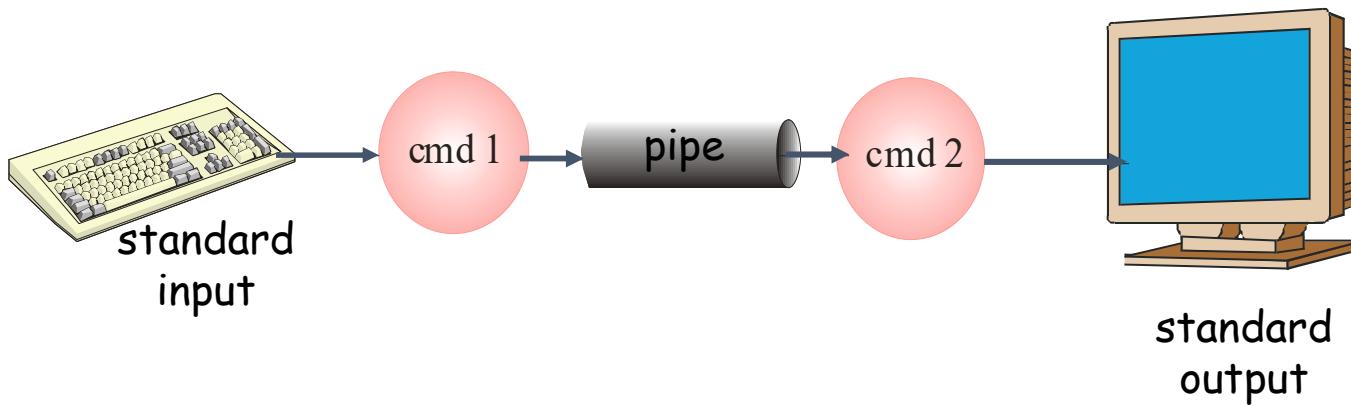
Next:

How do we pass useful information between processes?

How do we synchronize processes?

Idea of Pipe

cmd1 | cmd2



Pipe connects a data flow
from one process to another

Pipelines: ls -l | sort -k5 -n

The starting simple *ls*

```
athena.ecs.csus.edu - PuTTY
[bielr@athena csc60]> ls -l
total 544
-rw----- 1 bielr faccsc      2 Dec 21 12:58 >
-rwx----- 1 bielr faccsc 5074 Mar 16 14:08 a.out*
drwx----- 2 bielr faccsc 4096 Apr 19 13:07 ClassExamples/
-rw----- 1 bielr faccsc 162 Apr 11 18:16 lsls
-rw----- 1 bielr faccsc 138 Dec 22 09:39 lsout
drwx----- 16 bielr faccsc 4096 Apr 24 10:31 mywork/
drwx----- 6 bielr faccsc 4096 Dec 18 15:58 myworkf16/
drwx----- 8 bielr faccsc 4096 Dec 16 15:07 myworkS16/
-rwx----- 1 bielr faccsc 6438 Sep 19 2016 reverse*
-rw----- 1 bielr faccsc 993 Sep 16 2016 reverse1.c
drwx----- 4 bielr faccsc 4096 Apr 23 10:36 student/
-rw----- 1 bielr faccsc 527 Nov 16 08:35 testScript.txt
-rw----- 1 bielr faccsc 235289 Apr 17 2016 tlpi-160401-dist.tar.gz
drwx----- 48 bielr faccsc 4096 Nov 10 09:26 tlpi-dist/
-rw----- 1 bielr faccsc 252898 Sep 21 2016 trylab1.txt
-rw----- 1 bielr faccsc 12 Dec 22 09:40 wcout
[bielr@athena csc60]>
```

Note: -k5 means sorting column 5; -n means sorting by numerical value

Pipelines: ls -l | sort -k5 -n

The result of the sort

```
[bielr@athena csc60]> ls -l | sort -k5 -n
total 544
-rw----- 1 bielr faccsc      2 Dec 21 12:58 >
-rw----- 1 bielr faccsc     12 Dec 22 09:40 wcout
-rw----- 1 bielr faccsc    138 Dec 22 09:39 lsout
-rw----- 1 bielr faccsc   162 Apr 11 18:16 lsls
-rw----- 1 bielr faccsc  527 Nov 16 08:35 testScript.txt
-rw----- 1 bielr faccsc  993 Sep 16 2016 reversel.c
drwx----- 16 bielr faccsc 4096 Apr 24 10:31 mywork/
drwx-----  2 bielr faccsc 4096 Apr 19 13:07 ClassExamples/
drwx----- 48 bielr faccsc 4096 Nov 10 09:26 tlpi-dist/
drwx-----  4 bielr faccsc 4096 Apr 23 10:36 student/
drwx-----  6 bielr faccsc 4096 Dec 18 15:58 myworkf16/
drwx-----  8 bielr faccsc 4096 Dec 16 15:07 myworkS16/
-rwx-----  1 bielr faccsc 5074 Mar 16 14:08 a.out*
-rwx-----  1 bielr faccsc 6438 Sep 19 2016 reverse*
-rw-----  1 bielr faccsc 235289 Apr 17 2016 tlpi-160401-dist.tar.gz
-rw-----  1 bielr faccsc 252898 Sep 21 2016 trylab1.txt
[bielr@athena csc60]>
```

Note: -k5 means sorting column 5; -n means sorting by numerical value

Process Pipes (Formally)

A **pipe** is a mechanism for interprocess communication; data written to the pipe by one process can be read by another process.

The data is handled in a first-in, first-out (FIFO) order.

The pipe has no name, so it can only be used by the process that created it **and by descendants that inherit the file descriptors on fork().**

Process Pipes

A pipe has to be open at both ends simultaneously.

If you read from a pipe file that doesn't have any processes writing to it (perhaps because they have all closed the file, or exited), the read returns end-of-file (**EOF**).

Using normal blocking (**BLOCK**) reads however, the read will block if the pipe is empty.

Writing to a pipe that doesn't have a reading process is treated as an error (**ERROR**) condition; it generates a SIGPIPE signal, and fails with error code EPIPE if the signal is handled or blocked.

Process Pipes

Pipes do not allow file positioning (i.e. lseek). Both reading and writing operations happen sequentially; reading from the beginning of the file and writing at the end. System keeps track of last read/write location.

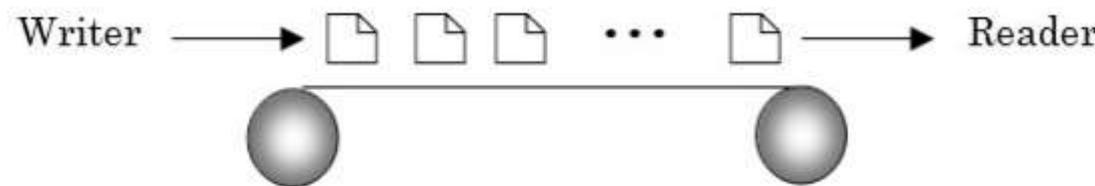


Figure 5.1 Conceptual data access using a pipe.

A common use of pipes is to send data to or receive data from a program being run as a sub-process.

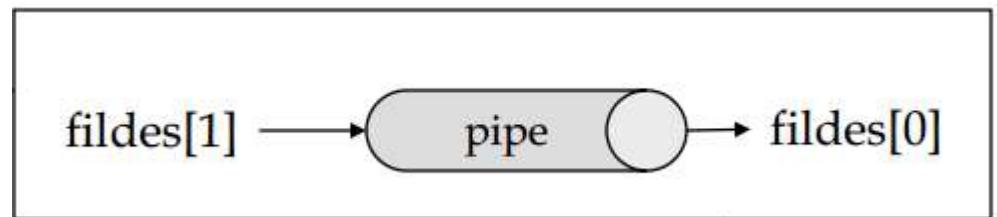
The Pipes system call (2)

The pipe call is a **system call** (man 2 pipe)

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

```
int pipe(int fildes[2]);
```

returns
0 if successful
-1 if the call fails



The pipe call fills in two file descriptors
fildes[0] is the file descriptor for reading from the pipe
fildes[1] is the file descriptor for writing to the pipe

It is easy to remember which comes first if you
remember that 0 is standard in and 1 is standard out.

You read from fildes[0] (think STDIN==0)
You write to fildes[1] (think STDOUT=1)

The Pipe - Declaration

The following code fragment creates an un-named pipe:

```
int fd[2];
```

then pass them to the
pipe command. Declare the file
descriptors

```
if (pipe(fd) == -1)  
    perror("Failed to create pipe...");
```

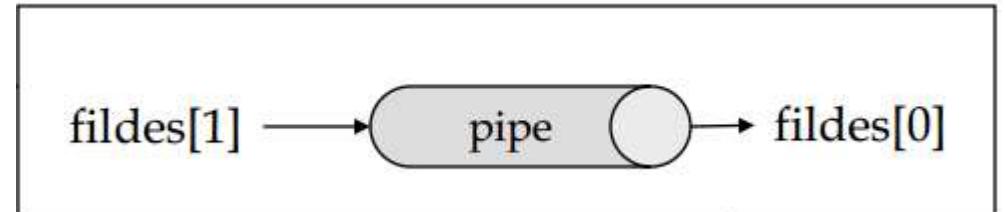
Using Pipe

In typical use, a process creates a pipe just before it forks one or more child processes.

The pipe is then used for communication either between the parent or child processes, or between two sibling processes.

Pipe Example

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#define MAX_LENGTH 100
int main(int argc, char ** argv) {
    int fildes[2];
    char result[] = "";
    pipe(fildes);
    if (fork())
        write(fildes[1], "I am writing into the pipe", MAX_LENGTH);
    else {
        read(fildes[0], result, MAX_LENGTH);
        printf("I read <<%s>> from the pipe.\n", result);
    }
}
```



Output: I read <<I am writing into the pipe>> from the pipe.

Pipe and Fork (LPI – Page 893)

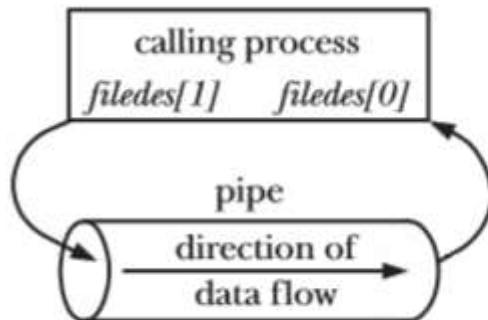


Figure 44-2: Process file descriptors after creating a pipe

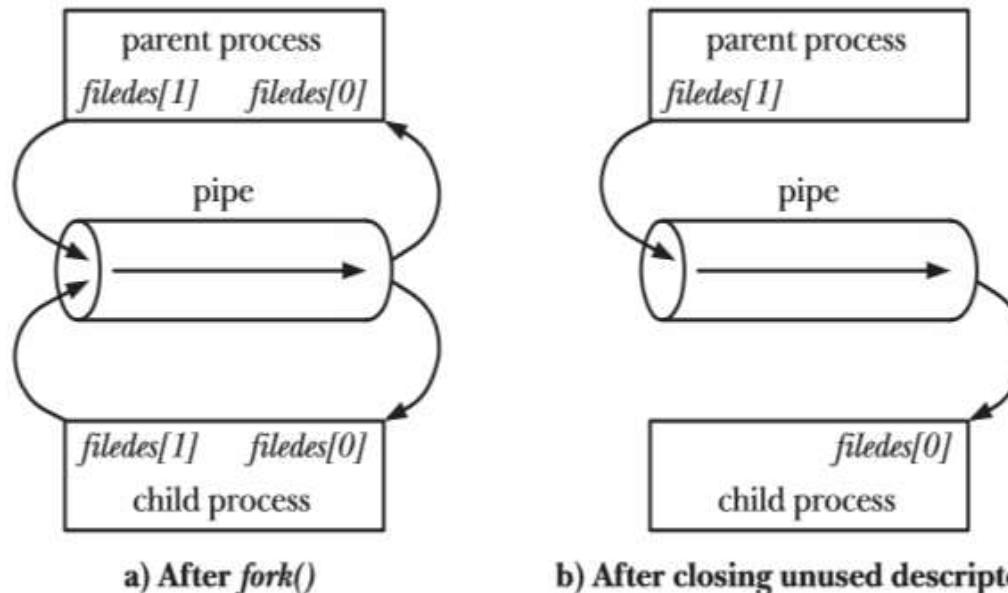


Figure 44-3: Setting up a pipe to transfer data from a parent to a child

Some good reasons for closing unused file descriptors (See LPI - page 894-895)

- The reading process closes write descriptor in order that it can see “end-of-file” status (if not, instead it sees “block waiting” for data – due to kernel’s indication that there some write descriptor is still opened)
- If the writing process does not close read descriptor, even after the read process closes the read descriptor, it can still write to the pipe’s until it is full. Once the pipe is full, it will block the write process indefinitely.
- Free resources to be used by other processes.

Race Condition (formally)

An unanticipated execution ordering of concurrent flows that results in undesired behavior is called a race condition—a software defect and frequent source of vulnerabilities.

Race Condition - Example

```
char c;
pid_t pid;
int fd = open(filename, O_RDWR);
if (fd == -1) {
    /* Handle error */
}
read(fd, &c, 1);
printf("root process:%c\n",c);

pid = fork();
if (pid == -1) {
    /* Handle error */
}

if (pid == 0) { /*child*/
    read(fd, &c, 1);
    printf("child:%c\n",c);
}
else { /*parent*/
    read(fd, &c, 1);
    printf("parent:%c\n",c);
}
```

Filename = “text.txt”
(Contents = “abc”)

Possible Output:

(1)
root process: a
parent: b
child: c

Or

(2)
root process: a
child: b
parent: c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main (void){
    char c;
    pid_t pid;

    int fd = open("text.txt", O_RDWR);
    if (fd == -1) {
        perror("Error on open");
    }
    read(fd, &c, 1);
    printf("root process:%c\n",c);

    pid = fork();
    if (pid == -1) {
        perror("Error on Fork");
    }
```

```
if (pid == 0) { /*child*/
    read(fd, &c, 1);
    printf("child:%c\n",c);
}
else { /*parent*/
    read(fd, &c, 1);
    printf("parent:%c\n",c);
}

return(EXIT_SUCCESS);
}
```

```
[bielr@athena ClassExamples]> race
root process:a
parent:b
child:c
```

Code that really works

Using pipe as a method for synchronization (1 of 2)

Listing 44-3: Using a pipe to synchronize multiple processes

```
pipes/pipe_sync.c
----- /* Declaration of currTime() */

#include "curr_time.h"
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int pfd[2];                                /* Process synchronization pipe */
    int j, dummy;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sleep-time...\n", argv[0]);

    setbuf(stdout, NULL);                      /* Make stdout unbuffered, since we
                                                 terminate child with _exit() */
    printf("%s Parent started\n", currTime("%T"));

    ①   if (pipe(pfd) == -1)
        errExit("pipe");

    ②   for (j = 1; j < argc; j++) {
        switch (fork()) {
        case -1:
            errExit("fork %d", j);

        case 0: /* Child */
            if (close(pfd[0]) == -1)          /* Read end is unused */
                errExit("close");

            /* Child does some work, and lets parent know it's done */

            sleep(getInt(argv[j], GN_NONNEG, "sleep-time"));
                                         /* Simulate processing */
            printf("%s Child %d (PID=%ld) closing pipe\n",
                   currTime("%T"), j, (long) getpid());
            if (close(pfd[1]) == -1)
                errExit("close");

            /* Child now carries on to do other things... */

            _exit(EXIT_SUCCESS);
        }
    }
}
```

Enlarged version following

Using pipe as a method for synchronization (2 of 2)

```
        default: /* Parent loops to create next child */
            break;
    }

/* Parent comes here; close write end of pipe so we can see EOF */

④ if (close(pfd[1]) == -1)                  /* Write end is unused */
    errExit("close");

/* Parent may do other work, then synchronizes with children */

⑤ if (read(pfd[0], &dummy, 1) != 0)
    fatal("parent didn't get EOF");
printf("%s Parent ready to go\n", currTime("%T"));

/* Parent can now carry on to do other things... */

exit(EXIT_SUCCESS);
}
```

Using pipe as a method for synchronization

(1 of 4)

```
/* LPI page 897, Listing 44-3 */

#include "curr_time.h"
#include "tlpi_hdr.h"

int main(int argc, char *argv[ ]) {
    int pfd[2];                  /* Process synchronization pipe */
    int j, dummy;

    if (argc < 2 || strcmp(argv[1] == "--help") == 0)
        usageErr("%s sleep-time...\n", argv[0]);
    setbuf(stdout, NULL);      /* Make stdout unbuffered, since we
                                terminate child with _exit() */
    printf("%s Parent started\n", currTimec));

    if (pipe(pfd) == -1)        /* build the pipe before creating child process */
        errExit("pipe");       /* errExit is a textbook function, not a system function
                                */
}
```

Using pipe as a method for synchronization

(2 of 4)

```
for (j = 1; j < argc; j++) {  
    switch (fork()) { /* Create the child process */  
        case -1:  
            errExit("fork &d", j);  
  
        case 0:  
            if (close(pfd[0]) == -1 /* Read end is unused */)  
                errExit("close"); /* errExit is a textbook function, not a system  
                           function */  
  
            /* Child does some work, and lets parent know it is done */  
  
            sleep(getInt(argv[j], GN_NONNEG, "sleep-time")); /* Simulate  
                           processing */  
  
            printf("%s Child %d (PID=%ld) closing pipe\n",  
                  currTime("%T"), j, (long) getpid());
```

Using pipe as a method for synchronization

(3 of 4)

```
if (close(pfd[1]) == -1) /* Each child inherits a fd for the write end of  
                           pipe and closes this fd once it has  
                           completed its action */  
  
    errExit("close");      /* errExit is a textbook function, not a  
                           system function */  
  
/* Child now carries on to do other things... */  
  
_exit(EXIT_SUCCESS);  
  
default:      /* parent loops to create next child */  
break;  
}  
     /* end of the switch */  
}  
     /* end of the for loop */
```

Using pipe as a method for synchronization (4/4)

```
/* Parent comes here; close write end of pipe so we can see EOF */  
/* Note that closing the unused write end of the pipe in the parent is  
   essential to the correct operation of this technique; otherwise, the  
   parent would block forever when trying to read from the pipe. */  
if (close(pfd[1]) == -1)          /* Write end is unused */  
    errExit("close");  
/* Parent may do other work, then synchronizes with children */  
/* After all the children have closed their file descriptors for the write end  
   of the pipe, the parent's read() from the pipe will complete, returning  
   end-of-file (0). */  
if (read(pfd[0], &dummy, 1) != 0)  
    fatal("parent didn't get EOF");  
printf("%s Parent ready to go\n", currTime("%T"));  
/* Parent can now carry onto do other things... */  
exit(EXIT_SUCCESS);  
}
```

Important Notes:

Communications buffers such as pipes can be empty if all of the information previously written has been read.

The empty buffer is not an end-of-file condition.

Rather, it reflects the asynchronous nature of inter-process communication.

A read call will normally block, waiting for data to become available.

However, a read on a pipe that has the other end closed for writing will **not** block, but will return a zero. This allows the reading process to detect the pipe equivalent to an end-of-file condition and react accordingly.

exit Function

- **exit()**
 - The exit() function causes normal process termination and the value of status & 0377 is returned to the parent.
 - All open stdio(3) streams are **flushed and closed**.
(C standard library - from man 3 exit)
(informally)
clean shutdown, flush streams, close files, etc

_exit Function

- _exit()

- The function _exit() terminates the calling process "immediately".
- Any open file descriptors belonging to the process are closed; any children of the process are inherited by process 1, init, and the process's parent is sent a **SIGCHLD** signal.
- (System call - from man 2 _exit)
(informally) drop out, files are closed but streams are not flushed

exit vs. *_exit* Functions

The two functions terminate normally short of return:

Note:

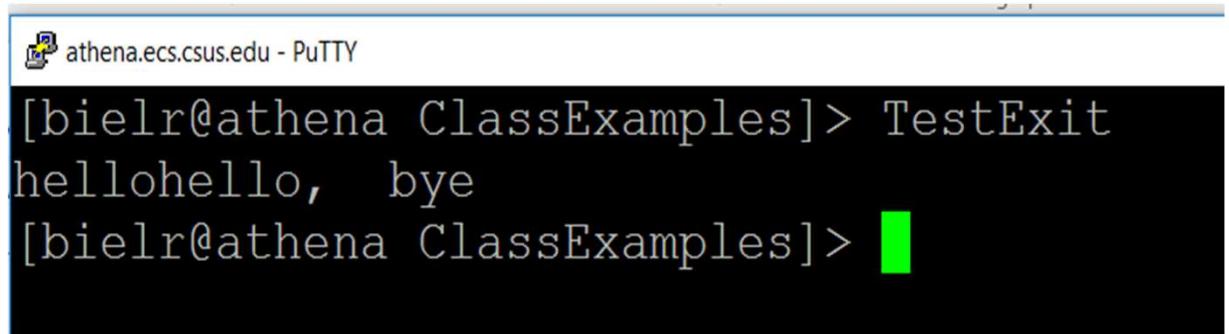
Child and parent could have buffers with a copy of the unflushed data.

If both call `exit()`, the pending stdio buffers to be **flushed twice**.

Thus, child should call `_exit()` instead.

Example 1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main () {
    int status;
    printf("hello");
    pid_t pid = fork();
    if (pid == 0) {
        sleep (2);
        exit(0);
    } else {
        wait(&status);
        printf(", bye\n");
    }
    exit(0);
}
```



The screenshot shows a terminal window titled "athena.ecs.csus.edu - PuTTY". The command "TestExit" was run, which prints "hellohello, bye". The terminal window has a dark background with white text and a blue header.

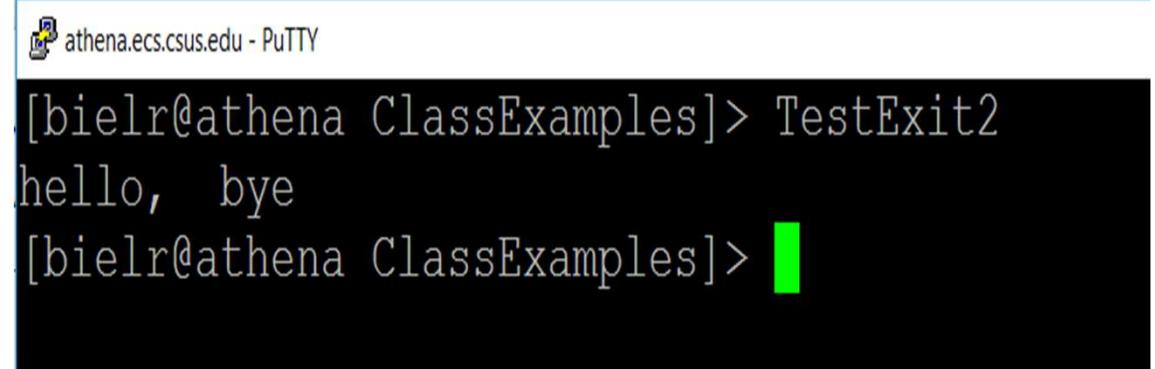
What is going on here? Why there are two hellos displayed ?



Example 2

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main () {
    int status;
    printf("hello");
    pid_t pid = fork();
    if (pid == 0) {
        sleep (2);
        _exit(0);
    } else {
        wait(&status);
        printf(", bye\n");
    }
    exit(0);
}
```



```
[bielr@athena ClassExamples]> TestExit2
hello, bye
[bielr@athena ClassExamples]>
```

Make more sense!



Pipelines ...

Getting the idea (1 of 2)

What do these commands do?

```
$ ls -l > myfile.txt  
$ sort -k5 -n < myfile.txt
```

Redirect standard output
of the ls command to
the file myfile.txt

Sort is a filter. It normally takes
input from stdin and outputs to stdout.
In this case we redirected its standard
input to come from myfile.txt

Getting the ideas (2 of 2)

We can achieve the same effect by using a pipe.
This eliminates the intermediate file *myfile.txt*

```
ls -l | sort -k5 -n
```

Pipelines: ls -l | sort -k5 -n

The starting simple *ls*

```
athena.ecs.csus.edu - PuTTY
[bielr@athena csc60]> ls -l
total 544
-rw----- 1 bielr faccsc      2 Dec 21 12:58 >
-rwx----- 1 bielr faccsc 5074 Mar 16 14:08 a.out*
drwx----- 2 bielr faccsc 4096 Apr 19 13:07 ClassExamples/
-rw----- 1 bielr faccsc 162 Apr 11 18:16 lsls
-rw----- 1 bielr faccsc 138 Dec 22 09:39 lsout
drwx----- 16 bielr faccsc 4096 Apr 24 10:31 mywork/
drwx----- 6 bielr faccsc 4096 Dec 18 15:58 myworkf16/
drwx----- 8 bielr faccsc 4096 Dec 16 15:07 myworkS16/
-rwx----- 1 bielr faccsc 6438 Sep 19 2016 reverse*
-rw----- 1 bielr faccsc 993 Sep 16 2016 reverse1.c
drwx----- 4 bielr faccsc 4096 Apr 23 10:36 student/
-rw----- 1 bielr faccsc 527 Nov 16 08:35 testScript.txt
-rw----- 1 bielr faccsc 235289 Apr 17 2016 tlpi-160401-dist.tar.gz
drwx----- 48 bielr faccsc 4096 Nov 10 09:26 tlpi-dist/
-rw----- 1 bielr faccsc 252898 Sep 21 2016 trylab1.txt
-rw----- 1 bielr faccsc 12 Dec 22 09:40 wcout
[bielr@athena csc60]>
```

Note: -k5 means sorting column 5; -n means sorting by numerical value

Pipelines: ls -l | sort -k5 -n

The result of the sort

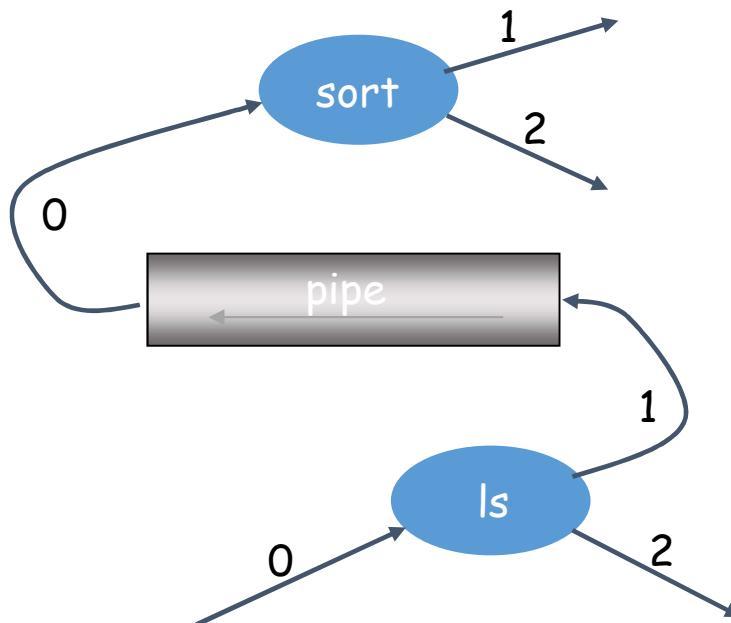
```
[bielr@athena csc60]> ls -l | sort -k5 -n
total 544
-rw----- 1 bielr faccsc      2 Dec 21 12:58 >
-rw----- 1 bielr faccsc     12 Dec 22 09:40 wcout
-rw----- 1 bielr faccsc    138 Dec 22 09:39 lsout
-rw----- 1 bielr faccsc   162 Apr 11 18:16 lsls
-rw----- 1 bielr faccsc  527 Nov 16 08:35 testScript.txt
-rw----- 1 bielr faccsc  993 Sep 16 2016 reversel.c
drwx----- 16 bielr faccsc 4096 Apr 24 10:31 mywork/
drwx-----  2 bielr faccsc 4096 Apr 19 13:07 ClassExamples/
drwx----- 48 bielr faccsc 4096 Nov 10 09:26 tlpi-dist/
drwx-----  4 bielr faccsc 4096 Apr 23 10:36 student/
drwx-----  6 bielr faccsc 4096 Dec 18 15:58 myworkf16/
drwx-----  8 bielr faccsc 4096 Dec 16 15:07 myworkS16/
-rwx-----  1 bielr faccsc 5074 Mar 16 14:08 a.out*
-rwx-----  1 bielr faccsc 6438 Sep 19 2016 reverse*
-rw-----  1 bielr faccsc 235289 Apr 17 2016 tlpi-160401-dist.tar.gz
-rw-----  1 bielr faccsc 252898 Sep 21 2016 trylab1.txt
[bielr@athena csc60]>
```

Note: -k5 means sorting column 5; -n means sorting by numerical value

Using pipe

We can achieve the same effect by using a pipe.
This eliminates the intermediate file myfile.txt

```
ls -l | sort -k5 -n
```



sort file descriptor table

0	pipe read
1	standard out
2	standard err

ls file descriptor table

0	standard in
1	pipe write
2	standard err

Recall - dup2

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

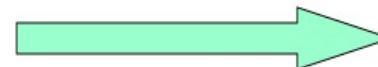
```
int dup2(int fd1, int fd2);
```

Duplicates this file descriptor on $fd2$.
If the file on $fd2$ is open, it is closed first and then the duplicate is made.

copies fd to $newfd$ in the descriptor table.

fd 0	a
fd 1	a
fd 2	
fd 3	
fd 4	b

dup2 (4,1)

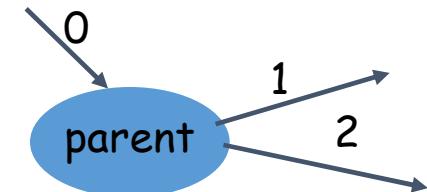


fd 0	
fd 1	b
fd 2	
fd 3	
fd 4	b

In a program ... Beginning

```
int main ( )
{
    pid_t childpid;
    int fd[2];

    if ((pipe(fd) == -1) ||
        ((childpid = fork( )) == -1))
    {
        perror("Failed to set up pipeline...");
        return (1);
    }
}
```



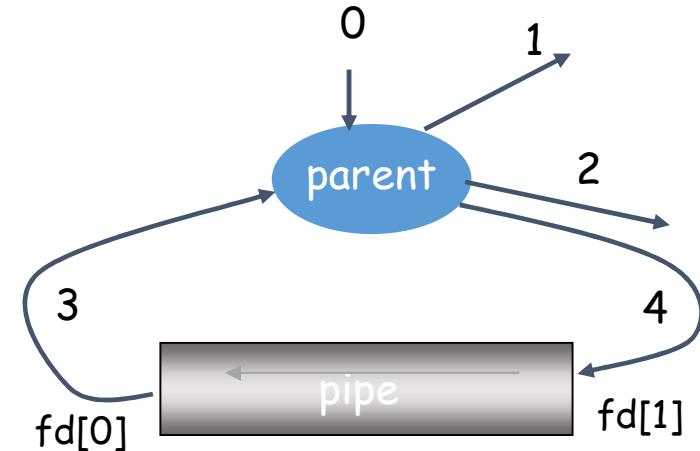
parent file descriptor table

0	standard in
1	standard out
2	standard err

In a program ... with pipe execution

```
int main ( )
{
    pid_t childpid;
    int fd[2];

    if ((pipe(fd) == -1) || ←
        ((childpid = fork( )) == -1))
    {
        perror("Failed to set up pipeline...");
        return (1);
    }
}
```



parent file descriptor table

0	standard in	
1	standard out	
2	standard err	
3	pipe read	fd[0]
4	pipe write	fd[1]

In a program ... with fork execution

```
int main ( )
{
    pid_t childpid;
    int fd[2];

    if ((pipe(fd) == -1) ||
        ((childpid = fork( )) == -1))
    {
        perror("Failed to set up pipeline...");
        return (1);
    }
}
```

parent file descriptor table

0	standard in	
1	standard out	
2	standard err	
3	pipe read	fd[0]
4	pipe write	fd[1]

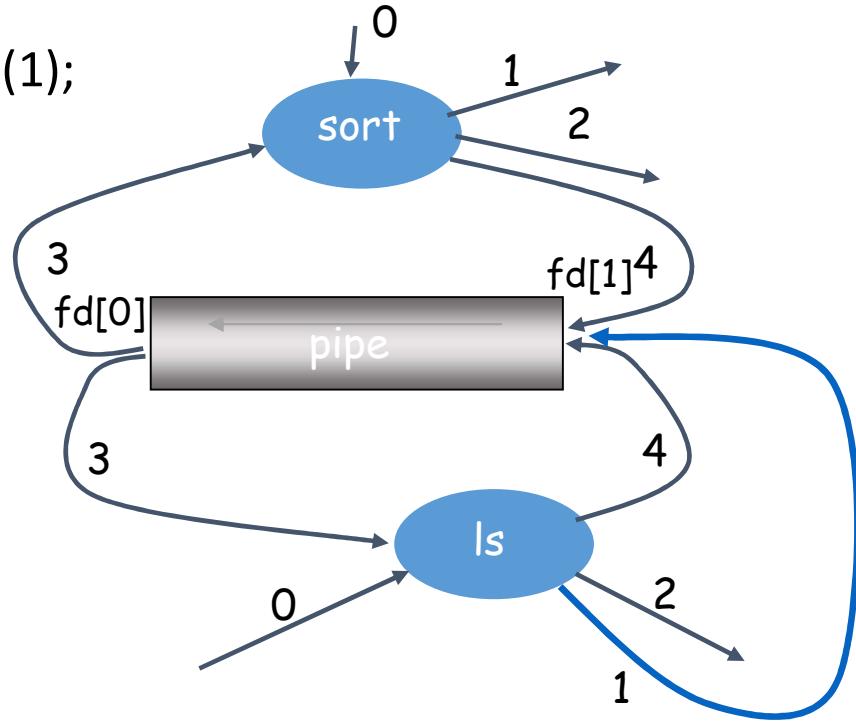


child file descriptor table

0	standard in	
1	standard out	
2	standard err	
3	pipe read	fd[0]
4	pipe write	fd[1]

Child calls dup2

```
if (childpid == 0)
{
    if (dup2(fd[1], STDOUT_FILENO) == -1) ←
        perror ("Failed to redirect stdout of ls");
    else if ((close(fd[0]) == -1) || (close(fd[1]) == -1))
        perror ("Failed to close extra file descriptors");
    else
    {
        execl("/bin/ls","ls", "-l", NULL);
        perror("Failed to exec ls ...");
    }
    return (1);
}
```



Standard out is first closed, then file descriptor fd[1] is duplicated on the file descriptor for stdout.

child file descriptor table
after dup2 call

	standard in	
0	standard in	
1	pipe write	
2	standard err	
3	pipe read	fd[0]
4	pipe write	fd[1]

```
if (childpid == 0)
{
```

```
    if (dup2(fd[1], STDOUT_FILENO) == -1)
        perror ("Failed to redirect stdout of ls");
    else if ((close(fd[0]) == -1) || (close(fd[1]) == -1))
        perror ("Failed to close extra file descriptors");
```

```
else
```

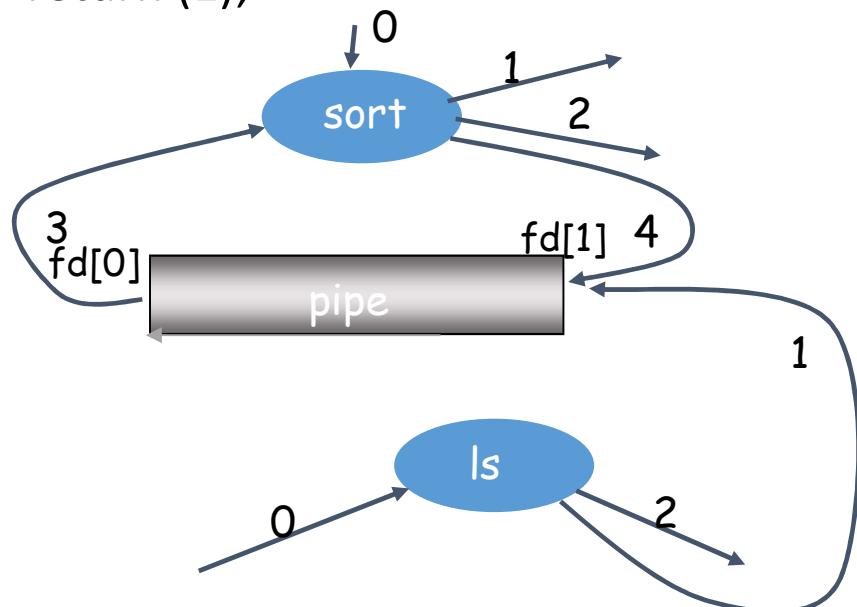
```
{
```

```
    execl("/bin/ls","ls", "-l", NULL);
    perror("Failed to exec ls ...");
```

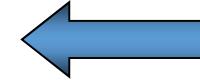
```
}
```

```
return (1);
```

```
}
```



Child closes file descriptors

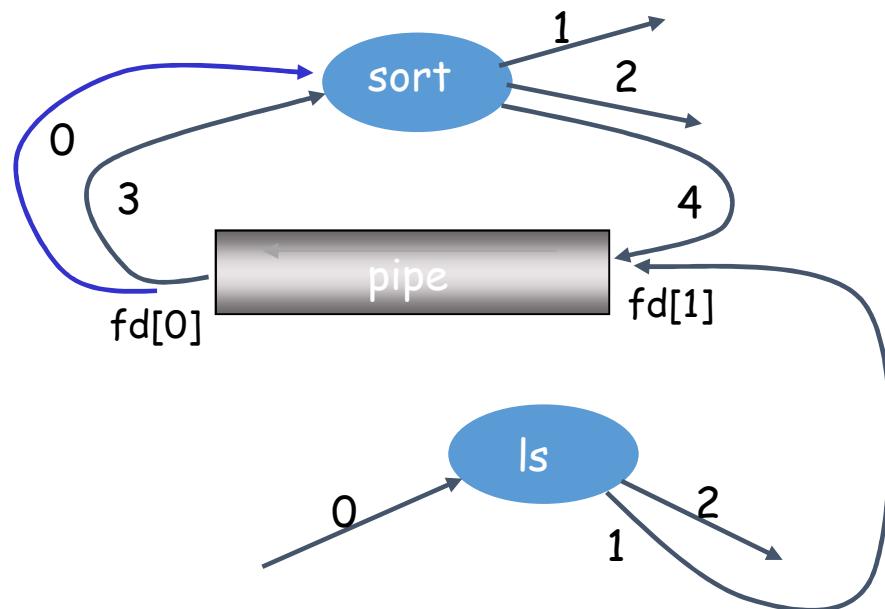


child file descriptor table
after dup2 and close(s) calls

0	standard in
1	pipe write
2	standard err

Parent calls dup2

```
if(dup2(fd[0], STDIN_FILENO) == -1) /* Parent executes sort */
    perror("Failed to redirect stdin of sort...");
else if ((close(fd[0]) == -1) || close(fd[1]) == -1)
    perror("Failed to close extra file descriptors");
else
{
    execl("/usr/bin/sort", "sort", "-k5", "-n", NULL);
    perror("Failed to exec sort");
}
return 1;
}
```

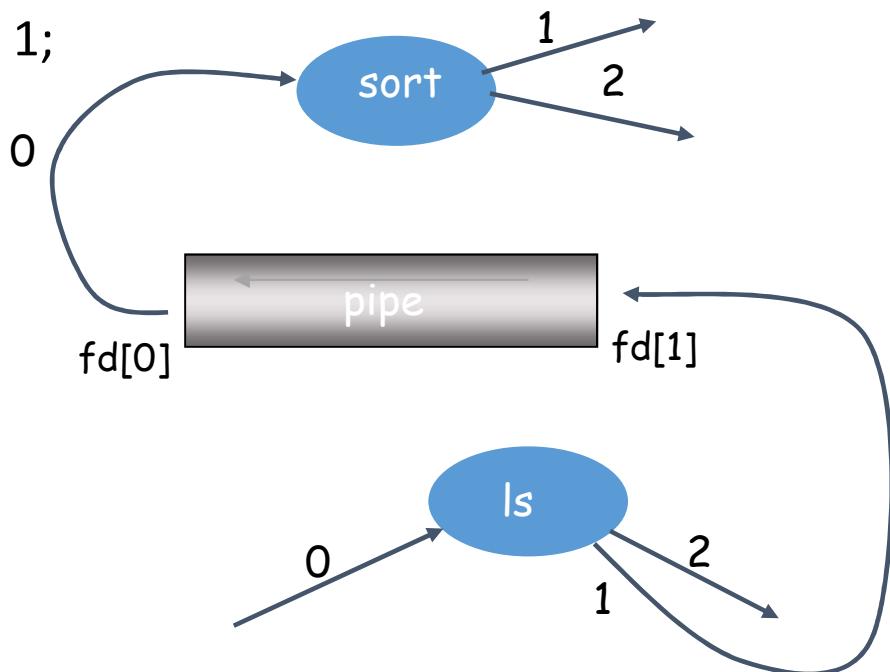


parent file descriptor table
after dup2 call

0	pipe read	
1	standard out	
2	standard err	
3	pipe read	fd[0]
4	pipe write	fd[1]

Parent closes file descriptors

```
if(dup2(fd[0], STDIN_FILENO) == -1)
    perror("Failed to redirect stdin of sort...");
else if ((close(fd[0]) == -1) || close(fd[1]) == -1)) ←
    perror("Failed to close extra file descriptors");
else
{
    execl("/usr/bin/sort", "sort", "-k5", "-n", NULL);
    perror("Failed to exec sort");
}
return 1;
}
```

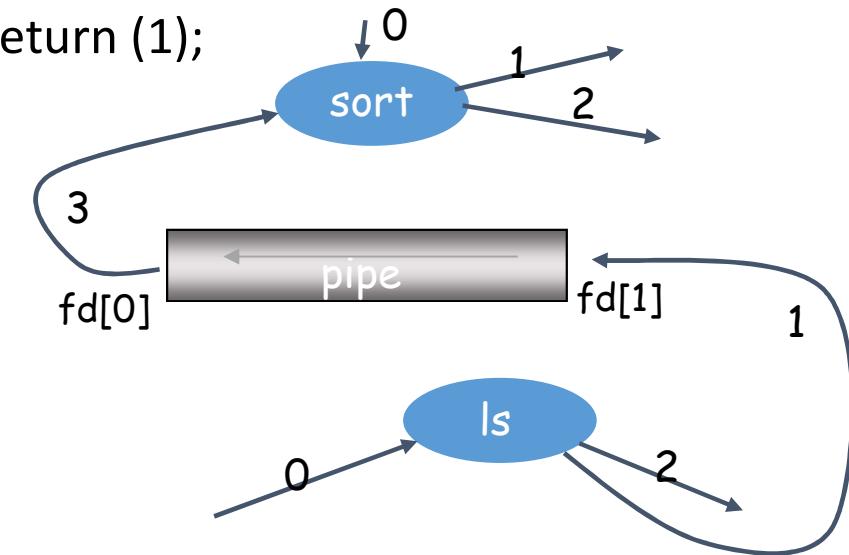


parent file descriptor table
After dup2 & close(s) calls

0	standard in
1	pipe write
2	standard err

Child executes ls

```
if (childpid == 0)
{
    if (dup2(fd[1], STDOUT_FILENO) == -1)
        perror ("Failed to redirect stdout of ls");
    else if ((close(fd[0]) == -1) || (close(fd[1]) == -1))
        perror ("Failed to close extra file descriptors");
    else
    {
        execl("/bin/ls","ls", "-l", NULL); ←
        perror("Failed to exec ls ...");
    }
    return (1);
}
```



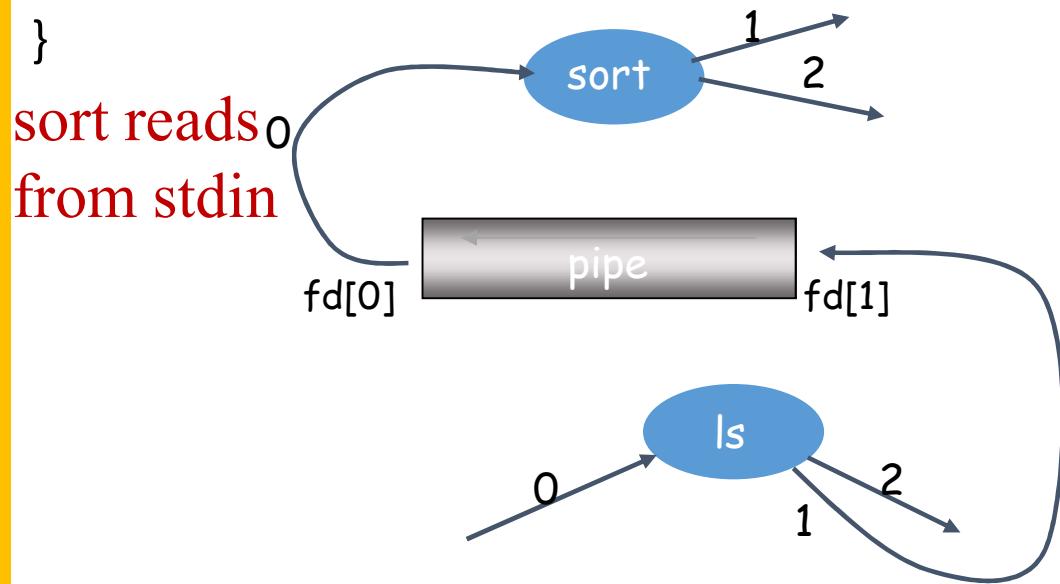
child file descriptor table
After dup2 & close(s) calls

0	standard in
1	pipe write
2	standard err

ls writes to stdout

Parent executes sort

```
if(dup2(fd[0], STDIN_FILENO) == -1)
    perror("Failed to redirect stdin of sort...");
else if ((close(fd[0]) == -1) || close(fd[1]) == -1))
    perror("Failed to close extra file descriptors");
else
{
    execl("/usr/bin/sort", "sort", "-k5", "-n", NULL);
    perror("Failed to exec sort");
}
return 1;
}
```

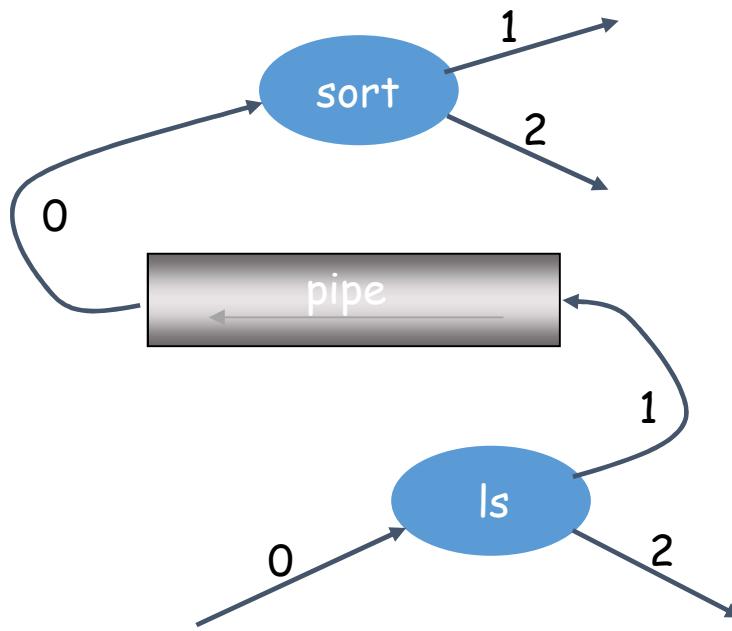


parent file descriptor table
after dup2 & close(s) calls

0	pipe read
1	standard out
2	standard err

Finally: At the end – we have

```
ls -l | sort -k5 -n
```



sort file descriptor table

0	pipe read
1	standard out
2	standard err

ls file descriptor table

0	standard in
1	pipe write
2	standard err

GDB debugger with *fork* (1 of 2)

GDB Commands using with fork	Description
<p>(gdb) set follow-fork-mode (child or parent)</p> <p>Example: To follow the fork, type: follow-fork-child</p>	<p>Set debugger response to a program call of fork. follow-fork-mode can be:</p> <p> parent - the original process is debugged after a fork child - the new process is debugged after a fork</p> <p>The unfollowed process will continue to run. By default, the debugger will follow the parent process.</p>
(gdb) set detach-on-fork (on or off)	Specifies whether GDB should debug both parent and child process after a call to fork() - Default is on: The child process (or parent process, depending on the value of follow-fork-mode) will be detached and allowed to run independently. This is the default.

GDB debugger with *fork* (2 of 2)

GDB Commands using with fork	Description
(gdb) catch fork	Catch calls to fork.
(gdb) info inferiors	Display IDs of currently known inferiors.
(gdb) inferior N	Use this command to switch between inferiors. The new inferior ID must be currently known (See above command).

Demo # 1: set follow-fork-mode

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#define MAX_LENGTH 100
int main(int argc, char ** argv) {
    int fildes[2];
    char result[] = "";
    pipe(fildes);
    int pid = fork(); // Test: set follow-fork-mode : parent or child
    if (pid > 0) {
        write(fildes[1], "I am writing into the pipe", MAX_LENGTH); }
    else {
        read(fildes[0], result, MAX_LENGTH);
        printf("I read <<%s>> from the pipe.\n", result);
        _exit(0);
    }
    exit(EXIT_SUCCESS);
}
```

Demo # 1: set follow-fork-mode OUTPUT

```
athena.ecs.csus.edu - PuTTY  
[bielr@athena ClassExamples]>  
[bielr@athena ClassExamples]> pipeFork  
I read <<I am writing into the pipe>> from the pipe.  
[bielr@athena ClassExamples]> |
```

Demo # 2: set detach-on-fork off (Process Synchronization)

Listing 44-3: Using a pipe to synchronize multiple processes

pipes/pipe_sync.c

```
#include "curr_time.h"                                /* Declaration of currTime() */
#include "tlpi_hdr.h"

int
```

```
main(int argc, char *argv[])
{
    int pfd[2];                                     /* Process synchronization pipe */
    int j, dummy;
```

```
    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sleep-time...\n", argv[0]);
```

```
    setbuf(stdout, NULL);                            /* Make stdout unbuffered, since we
                                                       terminate child with _exit() */
    printf("%s Parent started\n", currTime("%T"));
```

① if (pipe(pfd) == -1)
 errExit("pipe");

② for (j = 1; j < argc; j++) {
 switch (fork()) {
 case -1:
 errExit("fork %d", j);

```
      case 0: /* Child */
          if (close(pfd[0]) == -1)           /* Read end is unused */
              errExit("close");
```

```
          /* Child does some work, and lets parent know it's done */

          sleep(getInt(argv[j], GN_NONNEG, "sleep-time"));
              /* Simulate processing */
```

```
          printf("%s Child %d (PID=%ld) closing pipe\n",
                 currTime("%T"), j, (long) getpid());
```

③ if (close(pfd[1]) == -1)
 errExit("close");

```
      /* Child now carries on to do other things... */

      _exit(EXIT_SUCCESS);
```

Enlarged version following

(1 of 3): set detach-on-fork off (Process Synchronization)

```
#include <sys/wait.h>
#include "tlpi_hdr.h"
#define BUF_SIZE 10

int main (int argc, char *argv[ ]) {
    int pfd[2];                      /* Pipe file descriptors */
    char buf[BUF_SIZE];
    ssize_t numRead;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s string\n". argv[0]);
    if (pipe(pfd) == -1)             /* Create the pipe */
        errExit("pipe");
    switch (fork()) {                /* Call fork to create child */
        ...continued....
```

(2 of 3): set detach-on-fork off (Process Synchronization)

```
switch (fork()) {                                /* Call fork to create child */
    case -1:
        errExit("fork");
    case 0:                                     /* Child -reads from pipe */
        if(close(pfd[1]) == -1)                  /* Write end is unused */
            errExit("close - child");
        for (;;) {                               /* Read data from pipe, echo on stdout */
            numRead = read(pfd[0], buf, BUF_SIZE); /* read the data */
            if (numRead == -1)
                errExit("read");
            if (numRead == 0)
                break;                            /* encounters End-of-file */
            if (write(STDOUT_FILENO, buf, numRead) != numRead)
                fatal("child - partial/failed write");
        } /* end of for loop */
```

(3 of 3): set detach-on-fork off (Process Synchronization)

```
    write(STDOUT_FILENO, "\n", 1); /* exit loop */

    if (close(pdf[0]) == -1)
        errExit("close"); /*closed fd on read end of pipe*/
    _exit(EXIT_SUCCESS); /* terminate */

default: /* Parent - writes to pipe */
    if (close(pdf[0]) == -1) /* Read end is unused */
        errExit("close - parent");
    if (write(pdf[1], argv[1], strlen(argv[1])) != strlen(argv[1]))
        fatal("parent - partial/failed write");
    if (close(pdf[1]) == -1) /* Child will see EOF */
        errExit("close");
    wait(NULL); /* Wait for child to finish */
    exit(EXIT_SUCCESS);

} /* end of switch */
}
```

Program *pipe_sync* Output

```
[bielr@athena ClassExamples]> pipe_sync 4 2 6
11:37:44 Parent started
11:37:46 Child 2 (PID=27091) closing pipe
11:37:48 Child 1 (PID=27090) closing pipe
11:37:50 Child 3 (PID=27092) closing pipe
11:37:50 Parent ready to go
[bielr@athena ClassExamples]>
```

12-UNIX

Inter-Process Communication - Pipe

The End

13-UNIX

Shared Memory

Shared Memory, Message Queues

Chapter 45-46-47-48

Memory Organization for an Executed Program

- When a program is loaded into memory, it is organized into three areas of memory, called segments:
 - **text** segment (or **code** segment) is where the compiled code of the program itself resides.
 - **stack** segment is where memory is allocated for automatic variable within functions.
 - **heap** segment provides more stable storage of data for a program since memory allocated in the heap remains in existence for the duration of a program.

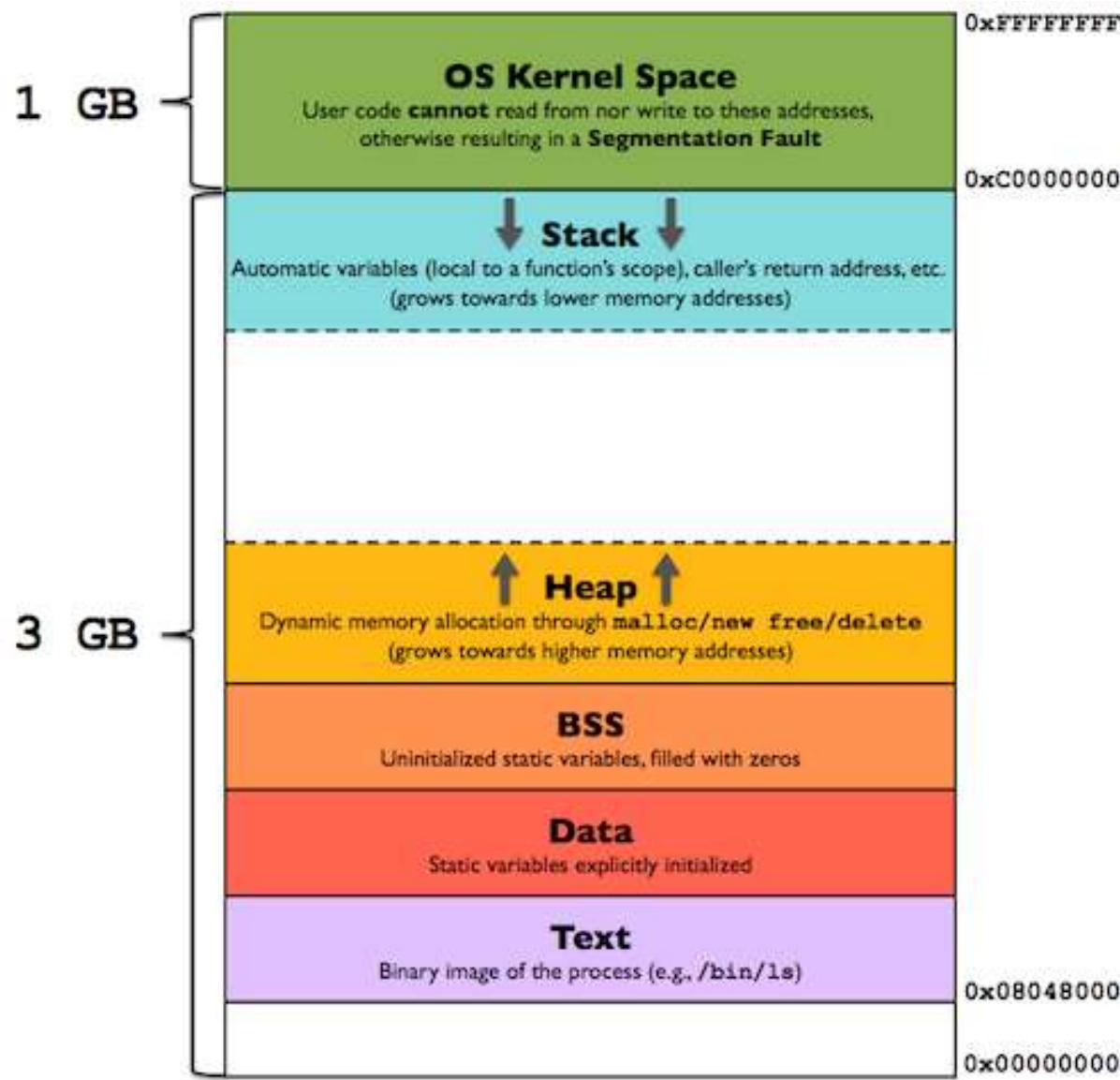
Stack

- Local variables (variables declared inside a function) are put on the stack - unless they are declared as 'static' or 'register'
- Function parameters are allocated on the stack
- Local variables that are stored in the stack are **not** automatically initialized by the system
- Variables on the stack disappear when the function exits

Heap

- Global, static, register variables are stored on the heap before program execution begins
- They exist the entire life of the program (even if scope prevents access to them - they still exist)
- They are initialized to zero
 - Global variables are on the heap
 - Static local variables are on the heap (this is how they keep their value between function calls)
- Memory allocated by new, malloc, calloc, etc., are on the heap

A typical Memory Layout on Linux X86/32

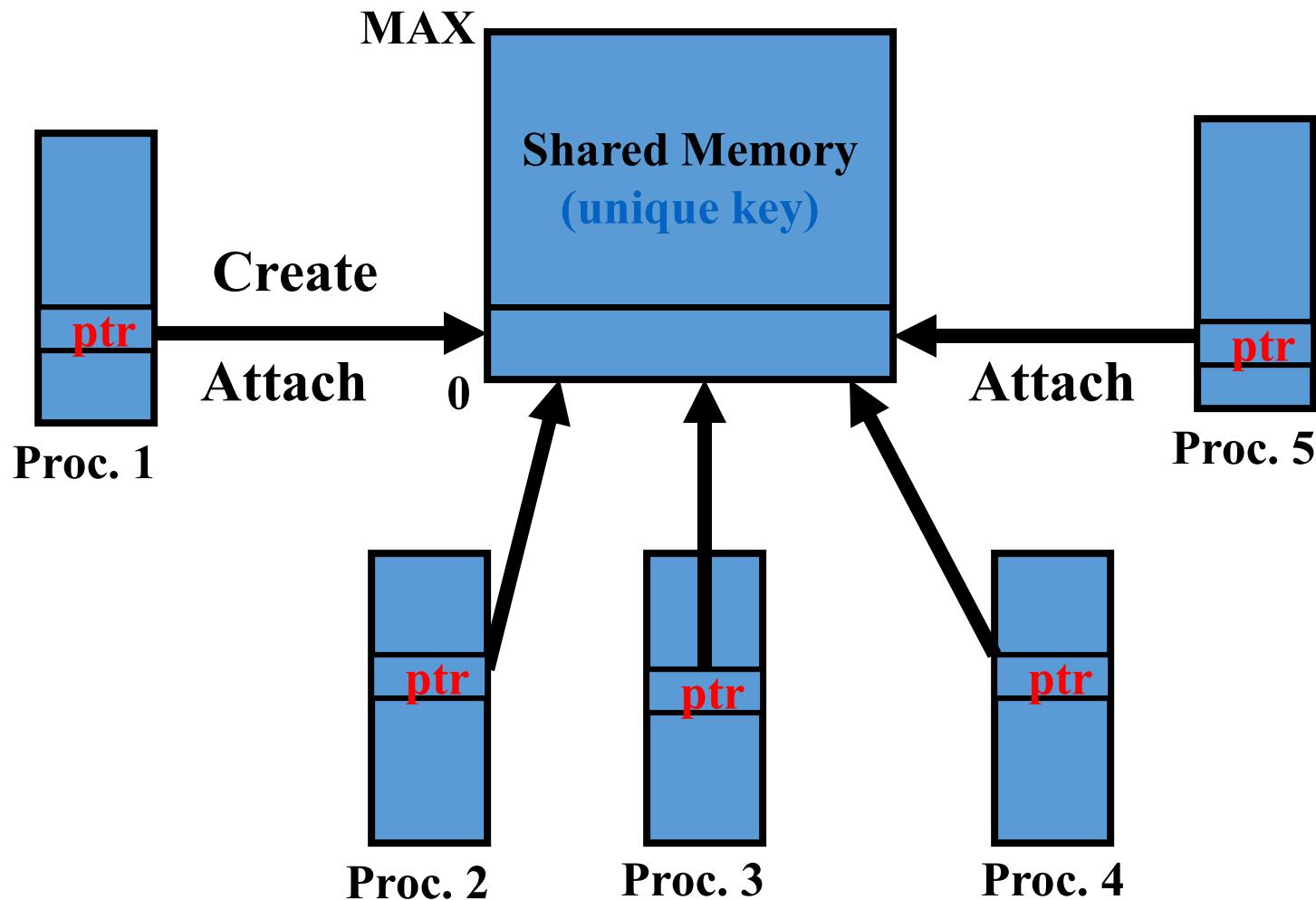


Shared Memory

- Normally, the Unix kernel prohibits one process from accessing (reading, writing) memory belonging to another process.
- Sometimes, however, this restriction is inconvenient.
- At such times, System V IPC Shared Memory can be created to specifically allow one process to read and/or write to memory created by another process

Shared Memory

Common chunk of read/write memory
among processes



Advantages of Shared Memory

- **Random Access**
 - you can update a small piece in the middle of a data structure, rather than the entire structure.
 - Very fast: accessed just like regular memory
- **Efficiency**
 - unlike message queues and pipes, which copy data from the process *into* memory within the kernel, shared memory is directly accessed.
 - Shared memory resides in the user process memory, and is then shared among other processes

Disadvantages of Shared Memory

- No automatic synchronization as in pipes or message queues (you have to provide any synchronization). Synchronize with *semaphores* or signals.
- You must remember that pointers are only valid within a given process. Thus, pointer offsets cannot be assumed to be valid across inter-process boundaries. This complicates the sharing of linked lists or binary trees.
- Processes must be on same machine.

Shared Memory: Overview of the system calls (1 of 2)

- Create/Access Shared Memory
 - $i = \text{shmget}(\text{KEY}, \text{Size}, \text{IPC_CREAT} | \text{PERM})$
- Deleting Shared Memory
 - $i = \text{shmctl}(i, \text{IPC_RMID}, 0)$
 - Or use ipcrm

Shared Memory: Overview of the system calls (2 of 2)

- Accessing Shared Memory
 - $\text{memaddr} = \text{shmat}(\text{id}, 0, 0)$
 - $\text{memaddr} = \text{shmat}(\text{id}, \text{addr}, 0)$
 - $\text{memaddr} = \text{shmat}(\text{id}, 0, \text{SHM_READONLY})$
 - System will decide address to place the memory at
 - $\text{shmdt}(\text{memaddr})$
 - Detach from shared memory

shmget()

This function is used to create a shared memory segment

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

returns a shared memory identifier or -1

the amount of memory required, in bytes

permission bits and IPC_CREAT

a programmer defined key
i.e. #define SHM_KEY 0x1234 /* Key for shared memory segment */

shmat()

This function attaches the shared memory segment to the process's address space.

```
#include <sys/shm.h>
```

```
void *shmat(int shm_id, const void *shm_addr, int shmflg);
```

Returns a pointer to the shared memory, or `(void *) -1` on error.

the shared memory identifier gotten from `shmget()`

if non-zero, the segment is attached for read-only. If 0, it is attached read/write.

address in the process's address space where the shared memory is to be attached. Normally this is a NULL pointer, allowing the system to determine where.
This is the recommended form.

shmdt()

This function detaches the shared memory segment

```
#include <sys/shm.h>
```

```
int shmdt(const void *shm_addr);
```

returns 0 if successful
or -1 if not

pointer to the shared memory
segment to be detached.

shmctl()

This call performs a range of control operations on the shared memory segment identified by *shmid*.

```
#include <sys/shm.h>
```

```
int shmctl(int shm_id, int command, struct shmid_ds *buf);
```

returns 0 if successful
or -1 if fails

the shared memory
segment's identifier

```
struct shmid_ds {  
    uid_t shm_perm.uid;  
    uid_t shm_perm.gid;  
    mode_t shm_perm.mode;  
}
```

IPC_STAT Sets the data in the shmid structure to reflect
the values associated with the shared memory.

IPC_SET Sets the values associated with the shared memory
to those provided in the data structure.

IPC_RMID Delete the shared memory

Implementing a binary semaphore protocol

The following code and examples are from the textbook, written by the author of the book. They are NOT part of the Linux operating system.

If you ever need to create semaphores, you could copy (and possibly alter) the functions to your project's requirements.

Implementing a binary semaphore protocol (1 of 2)

A binary semaphore can have two values:
available (free), reserved (in use)

Reserve: Attempt to reserve this semaphore for executive use. If semaphore is being reserved by another process, then block.

```
int reserveSem(int semId, int semNum);
```

Release: Free a current reserved semaphore, so that it can be reserved by another process.

```
int releaseSem(int semId, int semNum);
```

Source:

http://man7.org/tlpi/code/online/book/svsem/binary_sems.h.html (Functions from the text book)

Implementing a binary semaphore protocol

(2 of 2)

int **initSemAvailable**(int semId, int semNum);

+ arg.val = 1;

int **initSemInUse**(int semId, int semNum);

+ arg.val = 0;

int **reserveSem**(int semId, int semNum);

int **releaseSem**(int semId, int semNum);

semNum is used for identifying the semaphore within the set.

semId is for semaphore identification.

(Functions from the text book)

Example of shared memory

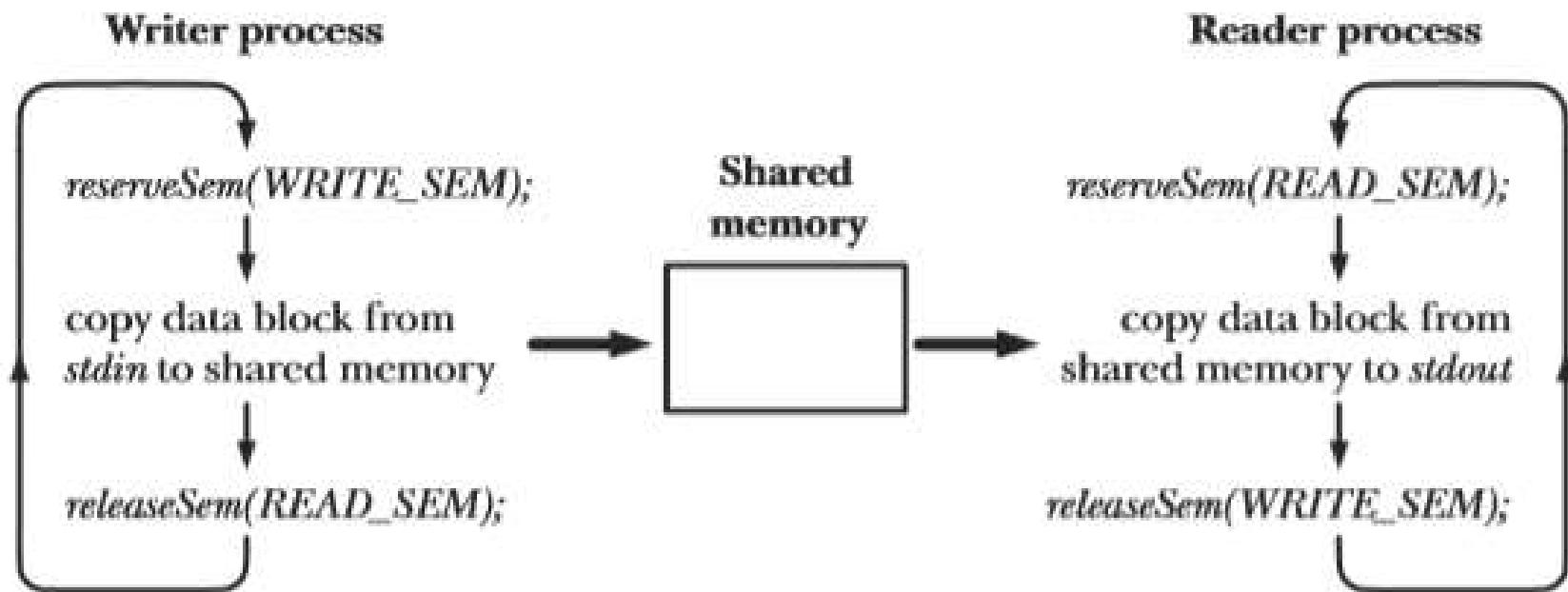


Figure 48-1: Using semaphores to ensure exclusive, alternating access to shared memory

Example of shared memory – writer (P 1003 in LPI book)

```
main(int argc, char *argv[])
{
    int semid, shmid, bytes, xfrs;
    struct shmseg *shmp;
    union semun dummy;

    /* Create set containing two semaphores; initialize so that
       writer has first access to shared memory. */

    semid = semget(SEM_KEY, 2, IPC_CREAT | OBJ_PERMS);
    if (semid == -1)
        errExit("semget");

    if (initSemAvailable(semid, WRITE_SEM) == -1)
        errExit("initSemAvailable");
    if (initSemInUse(semid, READ_SEM) == -1)
        errExit("initSemInUse");

    /* Create shared memory; attach at address chosen by system */

    shmid = shmget(SHM_KEY, sizeof(struct shmseg), IPC_CREAT | OBJ_PERMS);
    if (shmid == -1)
        errExit("shmget");

    shmp = shmat(shmid, NULL, 0);
    if (shmp == (void *) -1)
        errExit("shmat");

    /* Transfer blocks of data from stdin to shared memory */

    for (xfrs = 0, bytes = 0; ; xfrs++, bytes += shmp->cnt) {
        if (reserveSem(semid, WRITE_SEM) == -1)           /* Wait for our turn */
            errExit("reserveSem");

        shmp->cnt = read(STDIN_FILENO, shmp->buf, BUF_SIZE);
        if (shmp->cnt == -1)
            errExit("read");

        if (releaseSem(semid, READ_SEM) == -1)             /* Give reader a turn */
            errExit("releaseSem");

        /* Have we reached EOF? We test this after giving the reader
           a turn so that it can see the 0 value in shmp->cnt. */

        if (shmp->cnt == 0)
            break;
    }

    /* Wait until reader has let us have one more turn. We then know
       reader has finished, and so we can delete the IPC objects. */

    if (reserveSem(semid, WRITE_SEM) == -1)
        errExit("reserveSem");

    if (semctl(semid, 0, IPC_RMID, dummy) == -1)
        errExit("semctl");
    if (shmctl(shmid, IPC_RMID, 0) == -1)
        errExit("shmctl");

    fprintf(stderr, "Sent %d bytes (%d xfrs)\n", bytes, xfrs);
    exit(EXIT_SUCCESS);
}
```

Declare shared

memory segment

Create semaphore
array of 2 elements

Initialize semaphores

Create and
attach shared
memory

Wait for
Writer turn
Give Reader
a turn

Side note on following code: (1 of 2)

It uses “union” which is like a structure.

A **structure** is an object consisting of a sequence of named members of various types.

A **union** is an object that contains, at different times, any one of several members of various types.

The following code is from: #include "semun.h"

```
union semun {          /* Used in calls to semctl() */
    int           val;
    struct semid_ds * buf;
    unsigned short * array;
#if defined(__linux__)
    struct seminfo * __buf;
#endif
};
```

Side note on following code: (2 of 2)

I found more information with a Google search.

“In c, the difference between struct and union”

Here are links to two articles with more information.

<http://cs-fundamentals.com/tech-interview/c/difference-between-structure-and-union-in-c-language.php>

<http://www.thecrazyprogrammer.com/2015/03/difference-between-structure-and-union.html#union-in-c-language.php>

Example of shared memory – Writer

Transfer blocks of data from *stdin* to a system V
shared memory segment

(LPI, P. 1003) (1 OF 6)

```
/* TLPI page 1003, Listing 48-2 */
```

```
#include "semun.h"
```

```
#include "svshm_xfr.h"
```

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

```
    int semid, shmid, bytes, xfrs;
```

```
    struct shmseg *shmp;
```

```
    union semun dummy;
```

Declare a shared memory segment

```
/* Create a set containing the two semaphores that are used by the writer and
   reader program to ensure that they alternate in accessing the shared
   memory segment. The semaphores are initialized so that the writer has
   first access to the shared memory segment. Since the writer creates the
   semaphore set, it must be started before the reader. */
```

Example of shared memory – Writer (2 OF 6)

```
semid = semget(SEM_KEY, 2, IPC_CREAT | OBJ_PERMS); ← Create semaphore array of 2 elements
if (semid == -1)
    errExit("semget");
if (initSemAvailable(semid, WRITE_SEM) == -1) ← Initialize semaphores
    errExit("initSemAvailable");
if (initSemInUse(semid, READ_SEM) == -1)
    errExit("initSemInUse");

/* Create the shared memory segment and attach it to the writer's virtual
   address space at an address chosen by the system. */
shmid = shmget(SHM_KEY, sizeof(strut shmseg), IPC_CREAT | OBJ_PERMS);
if (shmid == -1)          /* build the “pipe” before creating child process */
    errExit("shmget");
shmp = shmat(shmid, NULL, 0); ← Create and attach shared memory
if (shmp == (void *) -1)
    errExit("shmat");
```

Example of shared memory – **Writer** (3 OF 6)

```
/* Transfer blocks of data from stdin to shared memory */  
  
/* Enter a loop that transfers data from standard input to the shared  
   memory segment. */  
/* The following steps are performed in each loop iteration:  
   Reserve (decrement) the writer semaphore.  
   Read data from standard input into the shared memory segment.  
   Release (increment) the reader semaphore. */
```

Example of shared memory – Writer (4 OF 6)

```
for (xfrs = 0, bytes = 0; ; xfrs++, bytes += shmp->cnt) {  
    if (reserveSem(semid, WRITE_SEM) == -1)  
        errExit("reserveSem");
```

Wait for Writer turn

```
    shmp->cnt = read(STDIN_FILENO, shmp->buf, BUF_SIZE);
```

```
    if (shmp->cnt == -1)
```

```
        errExit("read");
```

Give for Reader turn

```
    if (releaseSem(semid, READ_SEM) == -1)  
        errExit ("releaseSem");
```

Note: The *for* loop has no terminating value.

Example of shared memory – Writer (5 OF 6)

```
/* Have we reached EOF? We test this after giving the reader  
   a turn so that it can see the 0 value in shmp->cnt. */  
/* The loop terminates when no further data is available from standard  
   input.  
   On the last pass through the loop, the writer indicates to the  
   reader that there is no more data by passing a block of data of  
   length 0 (shmp->cnt is 0). */
```

The diagram illustrates a code snippet with annotations. A blue brace on the left groups two code blocks: an if-statement and a break statement. An arrow points from this brace to a light blue rectangular box containing the text "When we reach 0, break out of the infinite loop".

```
{ if (shmp->cnt == 0)  
    break;  
}  
} /* end of for loop */
```

Example of shared memory – Writer (6 OF 6)

```
/* Wait until reader has let us have one more turn. We then know
   reader has finished, and so we can delete the IPC objects. */
/* Upon exiting the loop, the writer once more reserves its semaphore, so that it
   knows that the reader has completed the final access to the shared memory. */
if (reserveSem(semid, WRITE_SEM) == -1)
    errExit("reserveSem");

/* The writer then removes the shared memory segment and semaphore set. */
if (semctl(semid, 0, IPC_RMID, dummy) == -1)
    errExit("semctl");
if (shmdt(shmp) == -1)
    errExit("shmdt");
if (shmctl(shmid, IPC_RMID, 0) == -1)
    errExit("shmctl");

fprintf(stderr, "Send %d bytes (%d xfrs)\n", bytes, xfers);
exit(EXIT_SUCCESS);
}
```

Example of shared memory – Reader (p 1005 LPI Book)

```
main(int argc, char *argv[])
{
    int semid, shmid, xfrs, bytes;
    struct shmseg *shmp;

    /* Get IDs for semaphore set and shared memory created by writer */

    semid = semget(SEM_KEY, 0, 0);
    if (semid == -1)
        errExit("semget");

    shmid = shmget(SHM_KEY, 0, 0);
    if (shmid == -1)
        errExit("shmget");

    /* Attach shared memory read-only, as we will only read */

    shmp = shmat(shmid, NULL, SHM_RDONLY);
    if (shmp == (void *) -1)
        errExit("shmat");

    /* Transfer blocks of data from shared memory to stdout */

    for (xfrs = 0, bytes = 0; ; xfrs++) {
        if (reserveSem(semid, READ_SEM) == -1)           /* Wait for our turn */
            errExit("reserveSem");

        if (shmp->cnt == 0)                            /* Writer encountered EOF */
            break;
        bytes += shmp->cnt;

        if (write(STDOUT_FILENO, shmp->buf, shmp->cnt) != shmp->cnt)
            fatal("partial/failed write");

        if (releaseSem(semid, WRITE_SEM) == -1)          /* Give writer a turn */
            errExit("releaseSem");
    }

    if (shmdt(shmp) == -1)
        errExit("shmdt");

    /* Give writer one more turn, so it can clean up */

    if (releaseSem(semid, WRITE_SEM) == -1)
        errExit("releaseSem");

    fprintf(stderr, "Received %d bytes (%d xfrs)\n", bytes, xfrs);
```

Get semaphore id

Attach shared
memory

Wait for
Reader turn

Give Writer
a turn

Example of shared memory – Reader

Transfer blocks of data from a system V shared memory segment to *stdout*

(LPI, P. 1005) (1 OF 4)

```
/* LPI page 1005, Listing 48-3 */
```

```
#include "svshm_xfr.h"
```

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

```
    int semid, shmid, bytes, xfrs;
```

```
    struct shmseg *shmp;
```

Declare a shared memory segment

```
/* Get IDs for semaphore set and shared memory created by writer */
```

```
/* Obtain the IDs of the semaphore set and shared memory segment  
that were created by the writer program. */
```

```
    semid = semget(SEM-KEY, 0, 0);
```

```
    if (semid == -1)
```

```
        errExit("semget");
```

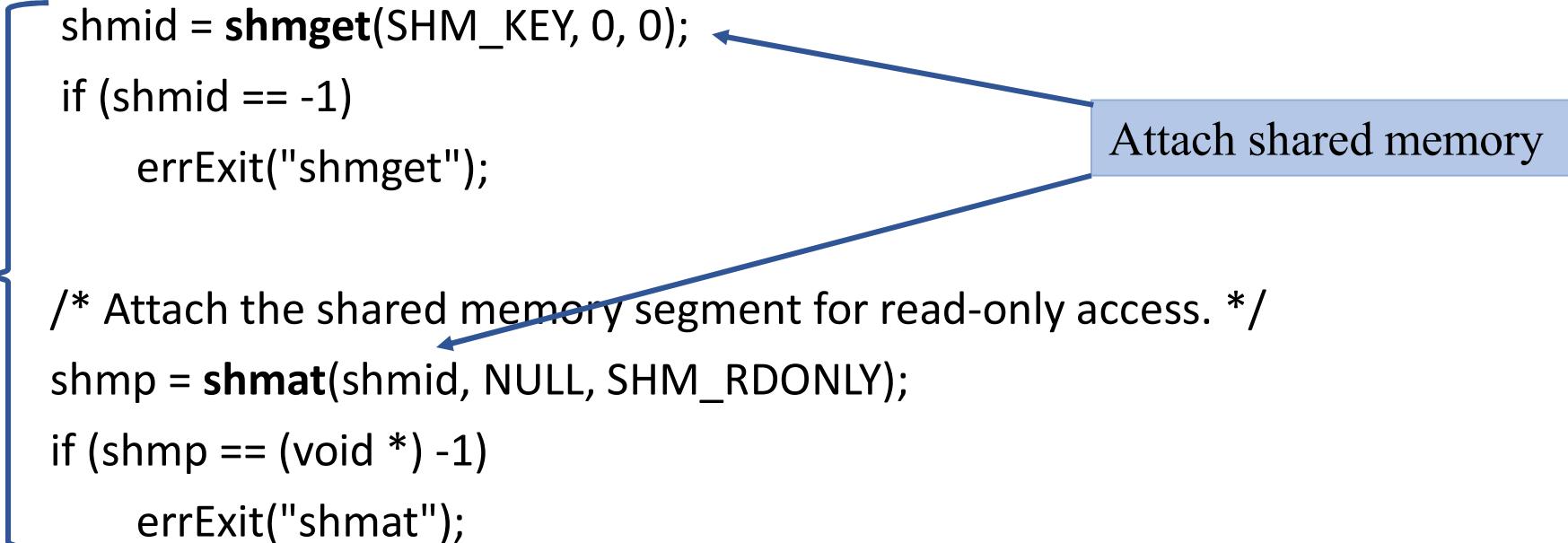
Get semaphore id

Example of shared memory – Reader (2 OF 4)

```
shmid = shmget(SHM_KEY, 0, 0);
if (shmid == -1)
    errExit("shmget");

/* Attach the shared memory segment for read-only access. */
shmp = shmat(shmid, NULL, SHM_RDONLY);
if (shmp == (void *) -1)
    errExit("shmat");
```

Attach shared memory



Example of shared memory – Reader (3 OF 4)

```
/* Transfer blocks of data from shared memory to stdout */
for (xfers = 0, bytes = 0; xfers++) {
    if(reserveSem(semid (READ_SEM) == -1) /* Wait for Reader turn
        errExit("reserveSem");
    if (shmp->cnt == 0) /* Writer encountered EOF */
        break;
    bytes += shmp->cnt;

    if (write(STDOUT_FILENO, shmp->buf, shmp->cnt, != shmp->cnt)
        fatal("partial/failed write");
    if(releaseSem(semid, WRITE_SEM) == -1) /* Give Writer a turn
        errExit("releaseSem");
} /* end of for loop */
```

Example of shared memory – Reader (4 OF 4)

```
/* After exiting the loop, detach the shared memory segment */
if (shmdt(shmp) == -1)
    errExit("shmdt");

/* Release the writer semaphore, so that the writer program can remove the
   IPC objects */
/* Give writer one more turn, so it can clean up */
if (reserveSem(semid, WRITE_SEM) == -1)
    errExit("reserveSem");

fprintf(stderr, "Received %d bytes (%d xfrs)\n", bytes, xfers);
exit(EXIT_SUCCESS);

}
```

Shared memory demo

Notes:

Go to sp2

Bring up the Writer then do ^z

Bring up the Reader with &

Type **jobs**

%l to bring Writer to **fg**

Type words, then Enter.

Get Echo

Ctrl-d (EOF) both go away.

If looking for this code in the on-line distribution, it will be in:

tlpi-dist/svshm

Writer =
svshm_xfr_write

Reader =
svshm_xfr_reader

Message Queues

Message Queue

- A Message Queue is a linked list of message structures stored inside the kernel's memory space and accessible by multiple processes
- Synchronization is provided automatically by the kernel
- New messages are added at the end of the queue
- Each message structure has a long *message type*
- Messages may be obtained from the queue either in a FIFO manner (default) or by requesting a specific *type* of message (based on *message type*)

msgget()

This function creates a message queue

```
#include <sys/msg.h>
```

```
int msgget(key_t key, int msgflg);
```

returns a queue identifier
if successful, otherwise, -1

a key value

permission bits and
IPC_CREAT

msgsnd()

The `msgsnd()` system call is used to send messages to a message queue.

```
#include <sys/msg.h>
```

```
int msgsnd(int msgid, const void *msg_ptr, size_t msgsize, int msgflg);
```

Returns 0 if successful
otherwise -1

```
struct my_message  
{  
    long int message_type;  
    // the data to transfer  
}
```

The message queue
identifier from msgget()

the message size.
Does not include
the type.

IPC_NOWAIT returns without sending
the message if the queue is full when
set. Otherwise the process waits for
space to become available in the queue.

msgrecv()

The msgrecv() system call is used receive messages from a message queue.

```
#include <sys/msg.h>
```

```
int msgrecv(int msgid, void *msg_ptr, size_t msg_sz,  
           long int msgtype, int msgflg);
```

0 if successful
otherwise -1

if 0, the next message is retrieved. If non-zero, the next message with this message type is retrieved

The message queue id

Message size

The message is copied here. Includes the long int.

IPC_NOWAIT if set, the call will return immediately. Otherwise, it waits until a message of the specified type is available to be read.

msgctl()

Performs the control operation specified by cmd on the message queue with identifier msgid

#include <sys/msg.h>

int msgctl(int msgid, int command, struct msgid_ds *buf);

0 if successful
otherwise -1

message queue id

IPC_STAT loads data in msgid_ds structure
IPC_SET gets the data from the msgid_ds structure
IPC_RMID deletes the message queue

13-UNIX

Shared Memory

Shared Memory, Message Queues

The End



14-UNIX

POSIX Threads

Chapter 29

Threads

(1 of 2)

- Like processes, threads are mechanism that permit an application to perform MULTIPLE tasks concurrently (LPI – 29-1).
- Think of a thread as a “procedure” that runs independently from its main program.
- Multi-threaded programs are where several procedures are able to be scheduled to run simultaneously and/or independently by the OS.
- A Thread exists within a process and uses the process resources.

Threads

(2 of 2)

- Threads only duplicate the essential resources they need to be independently schedulable.
- A thread will terminate if the parent process terminates.
- A thread is “lightweight” because most of the overhead has already been accomplished through the creation of the process.

Threads

- Each thread has its own stack, PC, registers
 - Share address space, files,..

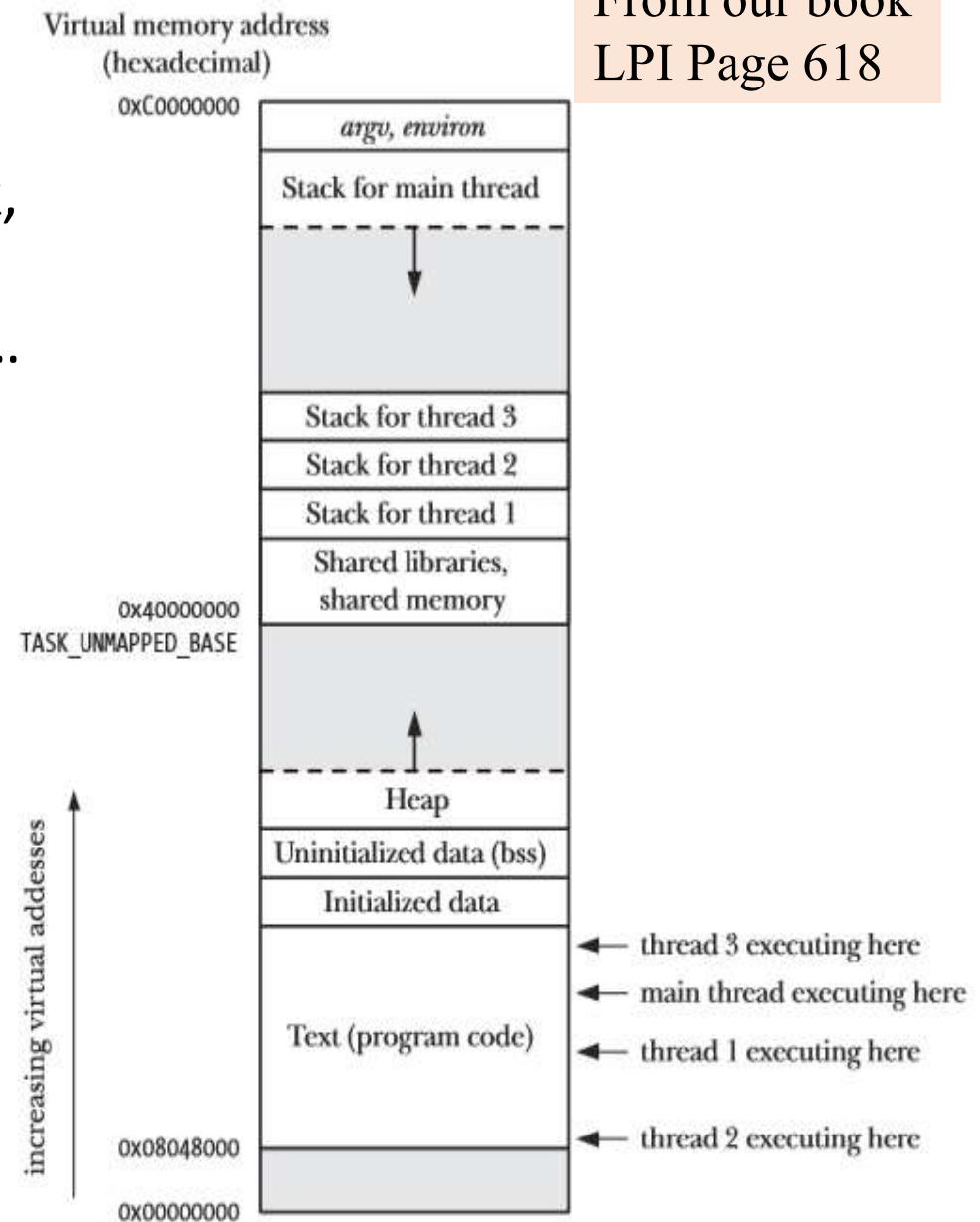
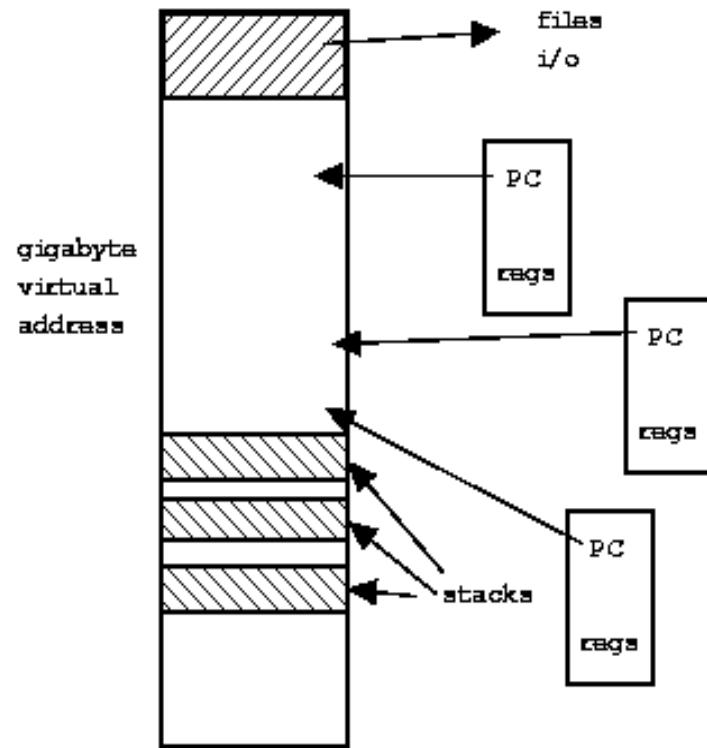


Figure 29-1: Four threads executing in a process (Linux/x86-32)

From our book
LPI Page 618

POSIX Threads (PThreads)

- For UNIX systems, implementations of threads that adhere to the IEEE POSIX 1003.1c standard are Pthreads.
- Pthreads are C language programming types defined in the pthread.h header/include file.
- How to compile:
➤`gcc hello.c -lpthread -o hello`

Why Use Pthreads

- The primary motivation behind Pthreads is improving program performance (**MAY NOT BE A BENEFIT if your system has a single processor**).
- Can be created with much less OS overhead.
- Needs fewer system resources to run.

Threads (clone) vs Forks (TLPI book)

Table 28-3: Time required to create 100,000 processes using *fork()*, *vfork()*, and *clone()*

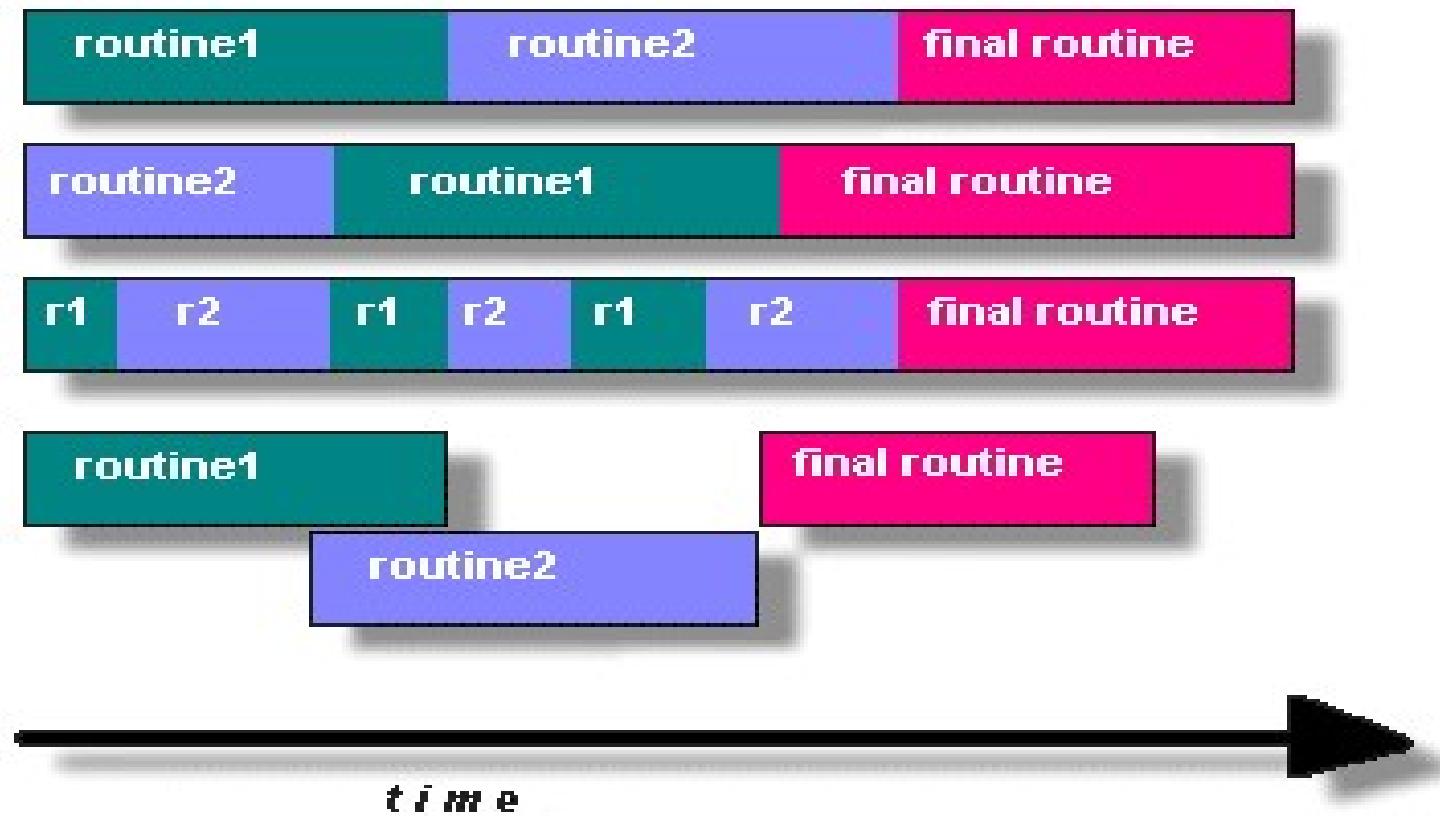
Method of process creation	Total Virtual Memory					
	1.70 MB		2.70 MB		11.70 MB	
	Time (secs)	Rate	Time (secs)	Rate	Time (secs)	Rate
<i>fork()</i>	22.27 (7.99)	4544	26.38 (8.98)	4135	126.93 (52.55)	1276
<i>vfork()</i>	3.52 (2.49)	28955	3.55 (2.50)	28621	3.53 (2.51)	28810
<i>clone()</i>	2.97 (2.14)	34333	2.98 (2.13)	34217	2.93 (2.10)	34688
<i>fork() + exec()</i>	135.72 (12.39)	764	146.15 (16.69)	719	260.34 (61.86)	435
<i>vfork() + exec()</i>	107.36 (6.27)	969	107.81 (6.35)	964	107.97 (6.38)	960

Thread creation is faster than process creation—typically, ten times faster or better. (On Linux, threads are implemented using the *clone()* system call, and Table 28-3, on page 610, shows the differences in speed between *fork()* and *clone()*.) Thread creation is faster because many of the attributes that must be duplicated in a child created by *fork()* are instead shared between threads. In particular, copy-on-write duplication of pages of memory is not required, nor is duplication of page tables.

Designing Pthreads Programs (1 of 2)

- Pthreads are best used with programs that can be organized into discrete, independent tasks which can execute concurrently.
- Example: routine 1 and routine 2 can be interchanged, interleaved and/or overlapped in real time.

Candidates for Pthreads



Designing Pthreads

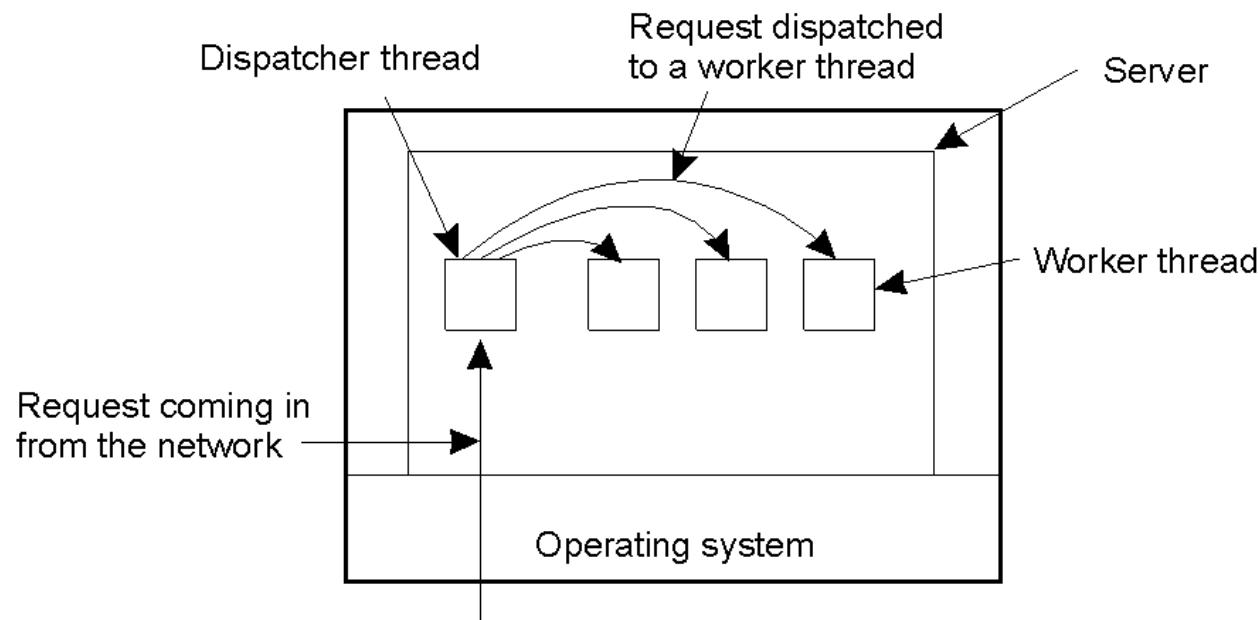
(2 of 2)

- Common models for threaded programs:
 - Manager/Worker: manager assigns work to other threads, the workers. Manager handles input and hands out the work to the other tasks.
 - Pipeline: task is broken into a series of sub-operations, each handled in series but concurrently, by a different thread.

Multi-threaded Server Example

Apache web server: pool of pre-spawned worker threads

- Dispatcher thread waits for requests
- For each request, choose an idle worker thread
- Worker thread uses blocking system calls to service web request



Pthread Management – Creating Threads

- The main() method comprises a single, default thread.
- **pthread_create()** creates a new thread and makes it executable.
- The maximum number of threads that may be created by a process is implementation dependent.
- Once created, threads are peers, and may create other threads.

Create Thread Call

```
#include <pthread.h>  
  
int pthread_create( pthread_t *thread,  
                    pthread_attr_t *attr,  
                    void *(*thread_function)(void *),  
                    void *arg );
```

Returns 0 on success, or a positive error number on error.

1st arg – pointer to the identifier of the create thread

2nd arg – thread attributes. If null, then the thread is created with default attributes

3rd arg – pointer to the C function that the thread will execute once the thread is created

4th arg – the argument of the executed function returns 0 for success

Pthread Management – Terminating Threads

Several ways to terminate a thread:

- The thread is complete and returns
- The `pthread_exit()` method is called
- The `pthread_cancel()` method is invoked
- The `exit()` method is called

The `pthread_exit()` routine is called after a thread has completed its work and it no longer is required to exist.

Thread Termination

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

Retval specifies the return value for the thread.

The value pointed to by *retval* should not be located on the thread's stack, since the contents of that stack become undefined on thread termination.

If the main thread calls *pthread_exit()* or performing a **return** then the other threads continue to execute.

Thread IDs

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

Returns the thread ID of the calling thread.

A thread can obtain its own ID using *pthread_self()*

Thread Joining

```
#include <pthread.h>

int pthread_join (pthread_t thread, void **retval)
```

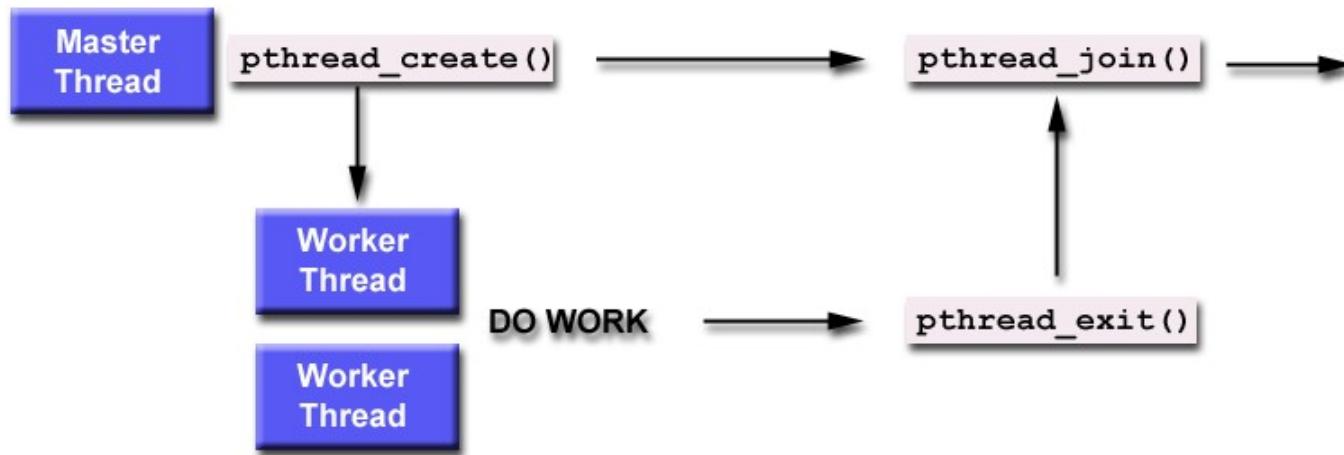
Returns 0 on success, or a positive number on error.

The *pthread_join()* function waits for the thread identified by *thread* to terminate.

Thread Join

(1 of 2)

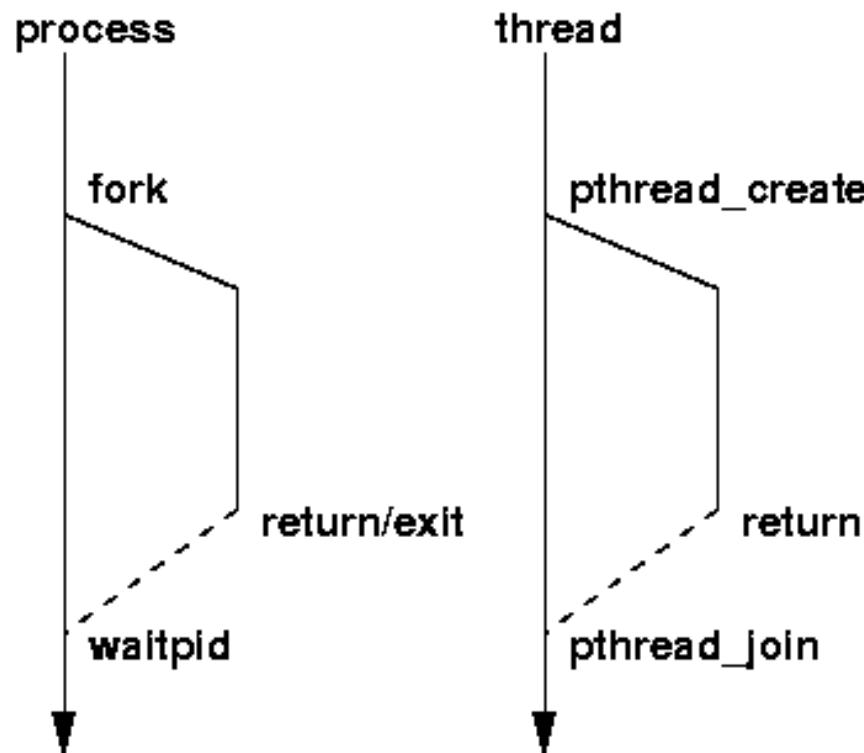
- "Joining" is one way to accomplish synchronization between threads.



- The `pthread_join()` subroutine blocks the calling thread until the specified thread-id thread terminates.

Parallel Processing

Thread Join (2 of 2) – Compared with fork/waitpid



Advantages of Threads

The overhead for creating a thread is significantly less than that for creating a process

Multitasking, i.e., one process serves multiple clients

Switching between threads requires the OS to do much less work than switching between processes

Sharing data between threads is easier than processes

Drawbacks of Threads

Not as widely available as longer established features

Writing multithreaded programs require more careful thought

More difficult to debug than single threaded programs

For single processor machines (1 CPU), creating several threads in a program may not necessarily produce an increase in performance

Sample program

(thread.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
int main(int argc, char **argv){
    pthread_t t1;
    int thread_id = 1;
    if ( (pthread_create(&t1, NULL, (void *)worker, (void *)&thread_id)) != 0) {
        printf("Error creating thread\n");
        exit(EXIT_FAILURE);
    }
    pthread_join(t1, NULL);
    return (EXIT_SUCCESS);
}
void *worker(void *a) {
    int *cnt = (int *)a;
    printf("This is thread %d\n", *cnt);
    pthread_exit(0);
}
```

Special directions for compiling

- The use of **pthread.h** requires an additional flag at compile time, as it is not automatically included as are other include files.
- Example:
gcc -o thread thread.c -lpthread



14-UNIX

POSIX Threads

The End

Getting Started with C

Basic information

The C Language

- C developed in the late 1960's
- ANSI C – American National Standard Institute.
 - Established in 1989.
 - Allowed for portable code that can be transferred from one computer platform to another and still work.

Hello World program

```
/*-----*/  
/* Ruthann Biel      */  
/* Lab 1           */  
#include <stdio.h>  
#include <stdlib.h>  
int main(void)  
{  
    printf("\nLab 1 \n\n");  
    printf("Hi, Ruthann Biel \n\n");  
    return(EXIT_SUCCESS);  
}  
/*-----*/
```

The run will look like this:

Lab 1

Hi, Ruthann Biel

```
/*-----*/  
/* Ruthann Biel */  
/* Lab 1 */
```

Examples of comments which can extend over several lines.

- Can be at end of line of code also
e.g. printf("\n"); /* print newline */
- Alternative form:
printf("/n"); //print newline

Preprocessor Directives – give the compiler the information it needs to run the program.

#include <stdio.h>

Stands for “STandard Input Output”

Needed because we used:

printf

#include <stdlib.h>

Stands for “STandard LIBrary”

Needed because we used:

EXIT_SUCCESS

int main(void)
must be in program

The first module of every C program is called “main”.

Some sources use “void main(void)” with no return.
It does not work with EXIT_SUCCESS, so
we will NOT use this style.

```
{  
...     braces surround BODY of the function  
}
```

=====

Later we will find additional uses for braces.

```
printf("\nLab 1\n\n");
printf("Hi, Ruthann Biel\n\n");
```

Examples of the printf function.

Each declaration and statement MUST end with semicolon.

The format string or control string must be enclosed by double quotes.

```
return EXIT_SUCCESS;
```

Shows a successful end of program

It is optional in ANSI C, but it is **not** optional in this class.

```
/*-----*/  
/* Your name here */  
/* Simple computation program */  
#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>  
  
int main (void)  
{  double x1=1, y1=5, x2=4, y2=7;  
    double side_1, side_2, distance;  
  
    printf("\nJane Smith\n\n");  
    side_1 = x2 - x1;  
    side_2 = y2 - y1;  
    distance = sqrt(side_1*side_1 + side_2*side_2);  
    printf("The distance between the two points "  
          "is %5.2f \n\n", distance);  
    return EXIT_SUCCESS;  
}  
/*-----*/
```

The RUN will look like this:

Jane Smith

The distance between the two points is 3.61



The nitty-gritty details!

Variable & Identifier Name Rules:

- Must begin with an alphabetic character (a-z, A-Z) or underscore (_).
- Digits are OK but not as first character.
- Can be any length, BUT first 31 characters must be unique.
- C is case sensitive.
sum, Sum, SuM, and SUM are all different variables.
- A C Reserved Word or Keyword cannot be used as an identifier.

ANSI C Reserved Words:

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

Examples of:

Valid Names

density

sum3

x_y

x2_2

Volume

Invalid Names

2sum

x&y

x-y

x2.2

1Volume

TYPES

Each variable must have a “type” which tells us the size, precision, and accuracy the variable will be allowed.
The word “type” is an important buzz word in computers.

NOTE: Min & Max values VARY from system to system.

NUMERIC

Integers

short

int

long

unsigned

Floating point

float

double

long double

CHARACTER

char

string

Limits on athena

SIGNED INTEGERS:

short minimum: -32768

short maximum: 32767

int minimum: -2147483648

int maximum: 2147483647

long minimum: -2147483648

long maximum: 2147483647

UNSIGNED INTEGERS:

The lower limit for all unsigned integer is zero.

unsigned short maximum: 65535

unsigned int maximum: 4294967295

unsigned long maximum: 4294967295

More limits on athena

FLOAT PRECISION:

float precision digits: 6

float maximum exponent: 38

float maximum: 3.402823e+038

double precision digits: 15

double maximum exponent: 308

double maximum: 1.797693e+308

long double precision: 18

long double maximum exponent: 4932

long double maximum: 1.189731e+4932

DECLARING VARIABLES:

All variables **MUST** be declared.

Examples:

```
int day;
```

```
int nickels, dimes;
```

```
float x;
```

```
float y1, y2;
```

```
double degrees;
```

```
double a, b, c;
```

ARITHMETIC OPERATORS & USE

Addition + $a = b + c;$

Subtraction - $c = a - b;$

Multiplication * $d = x * y;$

Division / $x = d / y;$

Modulus % $z = f \% g;$
(or Mod)

$29 \% 5 \rightarrow 4$ which is the remainder from
the division of 29 & 5.

Shortened Operator and Arithmetic Precedence

<u>Precedence</u>	<u>Operator</u>	<u>Associativity</u>
1	()	inner-most first
2	unary + -	right to left
3	binary * / %	left to right
4	binary + -	left to right
5	assignment operator =	right to left

Unary involves only one number with the operator.

Ex. -8 $-x$

Binary involves two numbers with the operator in between.

Ex. $9 + 8$ or $9 * -8$ or x / y

GETTING VALUES INTO VARIABLES:

```
int day = 21;           } declarations  
double y1 = 5.0, y2 = 10.0; }
```

OR

```
day = 6;      }  
y1 = 7.2;     } assignments  
y2 = 7.0;     }
```

GENERAL FORM of Assignment:

```
variable_name = value;
```

Assignment Statements

Use the equal sign (=) to move a value from the right side to the left side. (Same as in Java)

```
int i = 5;
```

Conceptually it acts like:

```
int i ← 5;
```

Order of Precedence for Numeric Conversion:

Highest precedence: long double
double
float
long integer
integer
Lowest precedence: short integer

Constants

```
int a, b;
```

```
a = b + 6; /* the 6 is a constant integer */
```

```
double c, d;
```

```
c = d * 2.3; /* 2.3 is, by default, a double */
```

Mixing Numeric Types

int a = 7, b = 3, c;

c = a / b;

now c will have 2

int / int

since $7 / 3$ is 2 r 1

INTEGER DIVISION TRUNCATES!!!

int a = 7, b = 3;

float c;

c = a / b;

but

int / int

so $7 / 3$ yields 2 (as an int)
then

converts it to a float, so ...
final value of c is 2.0

```
float a = 7, b = 3, c;  
c = a / b;
```

float / float
so $7.0 / 3.0 \rightarrow 2.33333$
final value of c is 2.33333

```
float c;  
c = 7 / 3.0;
```

int / double
double takes precedence
acts like $7.0 / 3.0 \rightarrow 2.33333$
final value of c is 2.33333

```
float a = 7.0, b = 3.0;
```

```
int c;
```

```
c = a / b;           so 7.0/ 3.0 → 2.333  
          float/float      final value of c is 2
```

To force the action we want, use **CASTING**.

```
int a = 7, b = 3;  
float c;  
c = (float) a / (float) b;
```

Note that the () go around the **type**, not the variable.

General Form for CASTING:
(type) expression

Use of Precedence

If we do $2 + 4 * 6 - 3$, there are THREE ways it could be done.

$$2 + 4 * 6 - 3$$

$$2 + 24 - 3$$

$$26 - 3$$

$$23$$

$$2 + 4 * 6 - 3$$

$$6 * 3$$

$$18$$

$$2 + 4 * 6 - 3$$

$$6 * 6 - 3$$

$$36 - 3$$

$$33$$

Using Precedence and no parentheses, C would give you the first answer of 23.

To get the second answer of 18, do:

$$(2 + 4) * (6 - 3)$$

To get the third answer of 33, do:

$$(((2 + 4) * 6) - 3) \text{ or } (2 + 4) * 6 - 3$$

Beginning Precedence

Precedence Level	Symbol	Comments	Example
1	()	Done inner-most first. Then left to right	$A + (C * D) + E$
2	+ -	Positive & Negative Both unary. Done right to left	$-A$ or $+A$
3	* / %	Done left to right	$A + C * D + E$
4	+ -	Add & Subtract Both binary Done left to right	$A + C - D + E$

Level 1 is highest. Level 4 is lowest.

Printing in Scientific Notation

X = 157.8926;

Specifier	Value Printer
%. ³ E	1.579E+002
%. ³ e	1.579e+002
%. ² e	1.58e+002
%g	157.893

General Form:

[sign]d.ddde[sign]ddd

Trigonometric Functions

System defined in math.h

```
#include <math.h>
```

$\sin(x)$ x in radians

$\cos(x)$ x in radians

$\tan(x)$ x in radians

$\text{asin}(x)$ arcsine, $-1 \leq x \leq +1$

$\text{acos}(x)$ arccosine, $-1 \leq x \leq +1$

$\text{atan}(x)$ arctangent

$\text{atan2}(y, x)$ arctangent of y / x

Trigonometric Functions

to convert to radians to degrees, or degrees to radians:

```
#define PI      3.14159
```

```
...
```

```
angle_degree = angle_radian * (180/PI);  
angle_radian = angle_degree * (PI/180);
```

Math Functions

Intrinsic arithmetic operators (+ - * / %) are part of the core of C.

All the extra math functions are stored in:

#include <math.h>

All math functions are type double

fabs(x) absolute value

sqrt(x) square root, $x \geq 0$

pow(x,y) exponentiations, x^y
 error: if $x = 0 \ \& \ y \leq 0$
 if $x \leq 0 \ \& \ y$ not an integer

ceil(x) rounds up to next integer

floor(x) rounds down to previous integer

More Math Functions

All the extra math functions are stored in:

#include <math.h>

All math functions are type double ←

$\exp(x)$ ex (2.718282)

$\log(x)$ $\ln x, x > 0$

$\log10(x)$ $\log_{10}x, x > 0$

$\text{abs}(x)$ absolute value of integer x
in **<stdlib.h>**

GETTING THE VARIABLE ON THE SCREEN:

General Form:

```
printf(format_string, argument_list);
```

The format_string has 3 parts:

- characters

- conversion specifiers

- escape sequences

GETTING THE VARIABLE ON THE SCREEN:

Example:

What I want to appear on the screen:

Daniel's age is 23.

To get it:

```
int age = 23;
```

```
printf("Daniel\\'s age is %d. \\n", age);
```

or

same result

```
printf("Daniel\\'s age is %i. \\n", age);
```

where

%d = conversion specifier

age = list of variables (in this case, list
has only one variable in it)

```
printf("Height is %6.2f \nLength is %6.2f \n", height, length);
```

on screen:

Height is 123.45

Length is 6.27

%6.2 f 6 refers to **width** total
 2 refers to precision

123.45 = 6 characters printed on the screen

If we changed it to %8.2 f

8 refers to width total
2 refers to precision

bb123.45 = 8 characters printed on the screen

```
int group = 3;  
float money = 78.25;  
printf("Group %1i raised $%6.2f.\n", group, money);
```

Output:

Group 3 raised \$ 78.25.

Print Conversion Specifiers

int, short %d, %i

short %hd, %hi

long %ld, %li

unsigned int %u

unsigned short %hu

unsigned long %lu

More Print Conversion Specifiers

float, double %f

floating pt.

%e

%E

scientific

%g

%G %e or %f

whichever is shorter

long double %lf

%le %lE

%lg %lG

character %c

string %s

Examples of Conversion Specifiers for printf:

(b stands for Blank)

```
int i = 1, j = 29;
```

```
float x = 333.12345678901234567890;
```

```
double y = 333.12345678901234567890;
```

<u>format exp</u>	<u>how printed</u>	<u>why</u>
%d -j	“-29”	field length 3 by default
%010d i	“0000000001”	padded with zeros
%-12d j	“29 bbbbbbbbbb ”	left adjusted
%12o j	“ bbbbbbbbbb 35”	octal/right adjusted
%-12x j	“1d bbbbbbbbbb ”	hex/left adjusted

Examples of Conversion Specifiers for printf :

(b stands for Blank)

```
int i = 1, j = 29;
```

```
float x = 333.12345678901234567890;
```

```
double y = 333.12345678901234567890;
```

format exp

%f

x

how printed

“333.123444”

why

precision 6 by default

.1f

x

“333.1”

precision 1

20.3f

x

“**bbbbbbbbbbbbb**333.123”

right adjusted

.9f

y

“333.123456789”

precision 9

-20.3e

y

“3.331e+02**bbbbbbbbbb**”

left adjusted

Summary: Conversion Specifiers for printf:

octal	%o
hexadecimal	%x
left adjusted	%-
right adjusted	%+
zero filled	%0

Examples:

Left Adjusted

123
4
67.8
5678

Right Adjusted

123
4
67.8
5678

Zero Filled

int x = 65;

Use a %05d & get: → 00065

Escape Sequences for printf:

\n Line feed or New Line

\a Alert. Beep. Bell.

\b Backspace.

\r Carriage return. Moves to start of line.

\ Concatenate lines.

\\" Print double quotes.

\f Formfeed (ejects printer page)

\t Horizontal tab.

\v Vertical tab.

\\\ Print backslash.

\' Print single quote.

\? Print question mark.

\% Print percent character.

scanf function

(reads values from keyboard)

```
int count;  
scanf("%i", &count);
```

%i - control string

& - address operator REQUIRED for scanf

count – identifier of variable

scanf function

With two variables:

```
float height, length;
scanf("%f%f ", &height, &length);
```

scanf function

printf and scanf often appear in pairs:

```
int age;  
printf("\nEnter your age: ");  
scanf("%i", &age);
```

NOTE;

scanf does **not** like “\n” in the control string.
“\n” is an instruction aimed at the output,
not at input from the keyboard.

scanf is very picky!

- 1) You MUST use the “&” symbol.
- 2) You MUST be sure your conversion specifiers AGREE with your variables.

double NEEDS %lf (that's a lower case L)

int NEEDS %i or %d

float NEEDS %f

CONSTANTS

Values that will not change during program.

Constants can be set up in a program using preprocessor directives.

Examples:

```
#define PI 3.14159  
#define MONTHS_IN_YEAR 12
```

General Form:

```
#define SYMBOLIC_NAME replacement
```

***WRONG WAY:

```
#define PI = 3.1415;
```

```
#define PI = 3.1415; // Everything after the name  
// gets substituted
```

What will happen?

```
x = 2;  
y = x * PI;
```

It will fill in as:

```
y = 2 * = 3.1415; ;
```



the whole phrase gets substituted!
and will NOT work as written.

ANOTHER WAY to do PI:

Use #include <math.h>

You may use these constants in my class,
but know that they are NOT ANSI standard!

```
/* Traditional/XOPEN math constants (double precision) */  
#ifndef __STRICT_ANSI__  
#define M_E                  2.7182818284590452354  
#define M_LOG2E              1.4426950408889634074  
#define M_LOG10E            0.43429448190325182765  
#define M_LN2                0.69314718055994530942  
#define M_LN10              2.30258509299404568402  
#define M_PI                 3.14159265358979323846  
#define M_PI_2              1.57079632679489661923  
#define M_PI_4              0.78539816339744830962  
#define M_1_PI              0.31830988618379067154  
#define M_2_PI              0.63661977236758134308  
#define M_2_SQRTPI          1.12837916709551257390  
#define M_SQRT2             1.41421356237309504880  
#define M_SQRT1_2          0.70710678118654752440  
#endif
```

More Operators: `+=` `-=` `*=` `/=` `%=`

examples:

`x = x + 5;` `x += 5;`

`y = y - 7;` `y -= 7;`

`z = z * 9;` `z *= 9;`

`a = a / 13;` `a /= 13;`

`b = b % 15;` `b %= 15;`

Operator and Arithmetic Precedence

Precedence	Operator	Associativity
1	()	inner-most first
2	+ - Unary	right to left
3	+ - Unary Postfix	left to right
4	+ - Unary Prefix	right to left
5	* / % Binary	left to right
6	+ - Binary	left to right
7	= Assignment Operator	right to left

Operator and Arithmetic Precedence

Unary involves only one number with the operator.

Ex. -8 -x y++

Binary involves two numbers with the operator in between.

Ex. 9 + 8 or 9 * -8

More on PRINTF Statements:

```
int a = 5, b= 9;  
printf ("%i%i\n\n", a, b);
```

OUTPUT:

59

COMMENTS:

Oops, the numbers are bumped right against each other.

More on PRINTF Statements:

```
int a = 5, b= 9;  
printf ("%ibb%i\n\n", a, b);
```

OUTPUT:

5**b**9

Try it again:

b - represents a blank

```
int a = 5, b= 9;  
printf ("%2i%2i\n\n", a, b);
```

OUTPUT:

5 9

b5b9 /* this line showing where the blanks are */

COMMENTS:

Now the numbers have space between them.

Try it again:

b - represents a blank

```
int a = 5, b= 9;  
printf ("%3i%3i\n\n", a, b);
```

OUTPUT:

```
      5   9  
bb5bb9 /* this line showing where the  
          blanks are */
```

COMMENTS:

Now the numbers have more space between them.

Another Problem:

```
int a = 5, b = 9;  
int c = 223, d = 123;
```

```
printf ("%2i%2i\n\n", a, b);  
printf ("%2i%2i\n\n", c, d);
```

OUTPUT:

5 9

223123

COMMENTS:

Not enough room for the three-digit numbers.
Also the numbers do not line up under each other.

Try it again:

b - represents a blank

```
int a = 5, b = 9;  
int c = 223, d = 123;
```

```
printf ("%4i%4i\n\n", a, b);  
printf ("%4i%4i\n\n", c, d);
```

OUTPUT:

bbb5bbb9

b223b123

COMMENTS:

Now the numbers print with space between them.
Also the numbers now line up under each other.

The C Language

- Currently, the most commonly-used language for embedded systems
- “High-level assembly”
- Very portable: compilers exist for virtually every processor
- Easy-to-understand compilation
- Produces efficient code
- Fairly concise



Source: **Embedded Systems Programming Languages**
http://www.eetimes.com/author.asp?section_id=36&doc_id=1323907
9/12/2014

What is API?

Application Program Interface (**API**) is a set of routines, protocols, and tools for building software applications.

An **API** specifies how software components should interact and **APIs** are used when programming graphical user interface (GUI) components.

Why C?

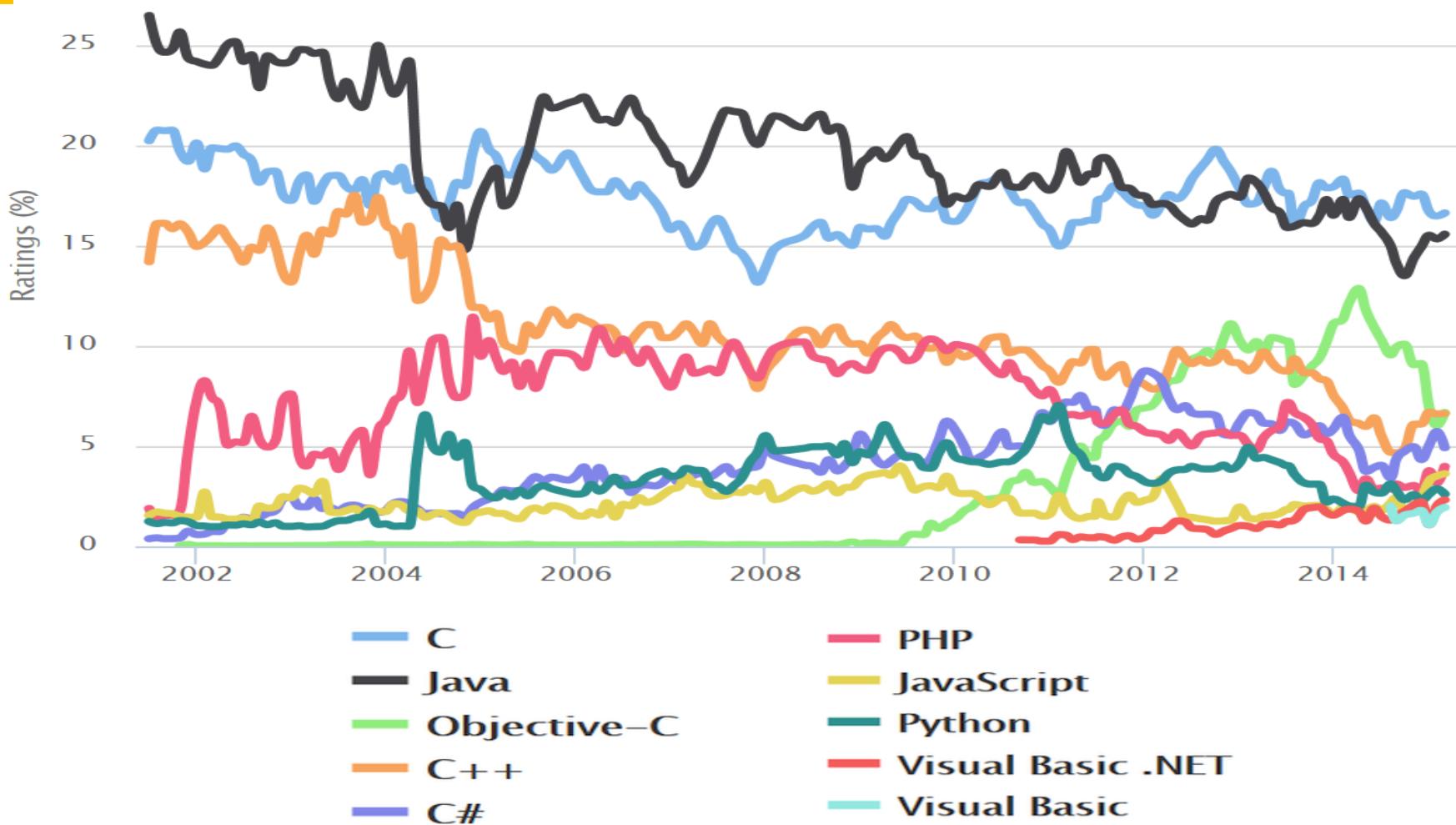
- Many situations where it is *only* language or system available
 - Small, embedded systems, instrumentation, etc.
- Many “low-level” situations that don’t have support for “high-level” languages
 - Operating systems, real-time systems, drivers
- On [Unix-like](#) systems, that API is usually part of an implementation of the [C library](#) (libc), such as [glibc](#), that provides [wrapper functions](#) for the system calls.

API

API stability is guaranteed, source code is portable!



Languages Popularity



Source: https://en.wikipedia.org/wiki/TIOBE_index

Getting Started

Basic information

THE END

Control Structures in C

And a bit on Characters

CHARACTERS

ASCII American Standard Code for Information Interchange

EBCDIC Extended Binary Coded Decimal
Interchange Code (from IBM)

Characters need to be stored in the ones and zeros of the binary system at heart of computer.

A portion of the ASCII chart:

	<u>Integer</u>	<u>Binary</u>
\n (new line)	10	0001010
+	43	0101011
3	51	0110011
B	66	1000010
b	98	1100010

```
int k = 97;  
char c = 'a';
```

```
/* print both as characters */
```

```
printf("value of k: %c; value of c: %c \n", k, c);
```

RESULT:

```
value of k: a; value of c: a
```

```
/* print both as integers */
```

```
printf("value of k: %i; value of c: %i \n", k, c);
```

RESULT:

```
value of k: 97; value of c: 97
```

Declaration of Characters & Integers Examples:

```
char name, a1 = 'b';
```

```
int n1, n2;
```

```
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
    char c1, c2, c3;
    c1 = 'x';
    c2 = '#';
    c3 = '\n';
    printf("%c %c \n", c1, c2);

    putchar(c1);
    putchar(32); /* a space */
    putchar(c2);
    putchar(32); /* a space */
    putchar(c3);
    return EXIT_SUCCESS;
}
/*-----*/
x # (NL)
x # (NL)                                where (NL) means a NewLine
```

In a printf
use **%c** to print single character

These three prints do the SAME thing.

```
putchar(32);
```

```
putchar(' ');
```

```
#define SPACE ' '
putchar (SPACE);
```

They all print a space.

```
char c1, c2;
printf("Enter two characters (without spaces), then"
        "press return: \n");
scanf("%c%c", &c1, &c2);
putchar(c1);
putchar(c2);
printf("\n %d %d \n", c1, c2);
printf("\n %c %c \n", c1, c2);
/*-----*/
```

RESULTS:

Enter two characters (without spaces), then press return:

xy (NL)	from the keyboard
xy	from the putchar
(NL)	from the 1st printf
120 121 (NL)	from the 1st printf
(NL)	from the 1st & 2nd printf
x y (nl)	from the 2nd printf

```
char c1, c2;  
printf("Enter 2 chars. (no spaces), then press "  
      "return: \n ");  
c1 = getchar();  
c2 = getchar();  
putchar(c1);  
putchar(c2);  
putchar('\n');  
/*-----*/
```

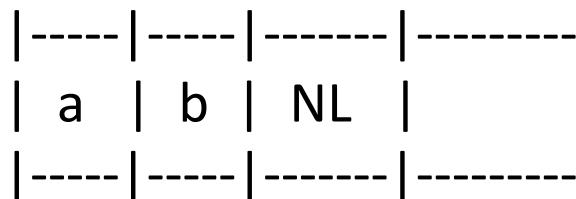
RESULTS:

Enter 2 chars. (no spaces), then press return:

a b (NL) from the keyboard

a b (NL) from the putchars

The Input Buffer:



```
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
    int year;
    char letter_1, letter_2, letter_3;

    printf("Enter a 3-letter nickname and press return: ");
    scanf("%c%c%c", &letter_1, &letter_2, &letter_3);
    printf("\nEnter the current year and press return: ");
    scanf("%d", &year);
    printf("Welcome, %c%c%c. %d is a great year "
           "to study C! \n\n", letter_1, letter_2, letter_3,
           year);
    return EXIT_SUCCESS;
}
/*-----*/
```

RESULTS:

Welcome, Liz. 2017 is a great year to study C!

The ASCII Table

ASCII Character Codes

Dec	Oct	Hex	Chr
000	000	00	NULL
001	001	01	SOH
002	002	02	STX, Start TX
003	003	03	ETX, End TX
004	004	04	EOT
005	005	05	ENQ, Inquire
006	006	06	ACK, Acknowledge
007	007	07	BEL, Bell
008	010	08	BS, Back Space
009	011	09	HT, Horizontal Tab
010	012	0A	LF, New Line(Line Feed)
011	013	0B	VT, Vertical Tab
012	014	0C	FF, Form Feed
013	015	0D	CR, Carriage Return
014	016	0E	SO, Stand Out
015	017	0F	SI, Stand In

<u>Dec</u>	<u>Oct</u>	<u>Hex</u>	<u>Chr</u>
016	020	10	DLE
017	021	11	DC1
018	022	12	DC2
019	023	13	DC3
020	024	14	DC4
021	025	15	NAK, Negative ACK
022	026	16	SYN
023	027	17	ETB
024	030	18	CAN
025	031	19	EM
026	032	1A	SUB
027	033	1B	ESC, Escape
028	034	1C	FS, Cursor Right
029	035	1D	GS, Cursor Left
030	036	1E	RS, Cursor Up
031	037	1F	US, Cursor Down
032	040	20	SP, Space

<u>Dec</u>	<u>Oct</u>	<u>Hex</u>	<u>Chr</u>
033	041	21	!
034	042	22	"
035	043	23	#
036	044	24	\$
037	045	25	%
038	046	26	&
039	047	27	'
040	050	28	(
041	051	29)
042	052	2A	*
043	053	2B	+
044	054	2C	,
045	055	2D	-
046	056	2E	,
047	057	2F	/

<u>Dec</u>	<u>Oct</u>	<u>Hex</u>	<u>Chr</u>
048	060	30	0
049	061	31	1
050	062	32	2
051	063	33	3
052	064	34	4
053	065	35	5
054	066	36	6
055	067	37	7
056	070	38	8
057	071	39	9
058	072	3A	:
059	073	3B	;
060	074	3C	<
061	075	3D	=
062	076	3E	>
063	077	3F	?
064	100	40	@

<u>Dec</u>	<u>Oct</u>	<u>Hex</u>	<u>Chr</u>
065	101	41	A
066	102	42	B
067	103	43	C
068	104	44	D
069	105	45	E
070	106	46	F
071	107	47	G
072	110	48	H
073	111	49	I
074	112	4A	J
075	113	4B	K
076	114	4C	L
077	115	4D	M
078	116	4E	N
079	117	4F	O
080	120	50	P

<u>Dec</u>	<u>Oct</u>	<u>Hex</u>	<u>Chr</u>
081	121	51	Q
082	122	52	R
083	123	53	S
084	124	54	T
085	125	55	U
086	126	56	V
087	127	57	W
088	130	58	X
089	131	59	Y
090	132	5A	Z
091	133	5B	[
092	134	5C	\
093	135	5D]
094	136	5E	^
095	137	5F	_
096	140	60	`

<u>Dec</u>	<u>Oct</u>	<u>Hex</u>	<u>Chr</u>
097	141	61	a
098	142	62	b
099	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q

<u>Dec</u>	<u>Oct</u>	<u>Hex</u>	<u>Chr</u>
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	
125	175	7D	}
126	176	7E	~
127	177	7F	DEL, Delete

Decision Making

Relational Operators

< less than

<= less than or equal to

> greater than

>= greater than or equal to

== compare equal

!= not equal

In C:

a TRUE condition is assigned a value of NON-ZERO.

A FALSE condition is assigned a value of ZERO.

Examples of conditions:

$(a > b)$

$(x == y + z)$

(x) where if $x = 0$, false
if $x = 2$, true/non-zero

Logical Operators:

&& and keyboard location, on the “7” key
|| or keyboard location, on key with “\”
 usually in upper right corner
! not keyboard location, on the “1” key

Results of logical operators (Truth Table):

A	B	A && B	A B	!A	!B
F	F	F	F	T	T
F	T	F	T	T	F
T	F	F	T	F	T
T	T	T	T	F	F

Practice with Conditions:

```
float a = 2.2, b = -1.2;  
int i = 5, done = 1;
```

(! (a == 2 * b))

Will it be true or false?

Practice with Conditions:

```
float a = 2.2, b = -1.2;
```

```
int i = 5, done = 1;
```

(! (a == 2 * b))

Will it be true or false?

!(2.2 == 2 * -1.2)

!(2.2 == -2.4)

!(false)

true

Practice with Conditions:

```
float a = 2.2, b = -1.2;  
int i = 5, done = 1;
```

((a < 10.0) && (b > 5.0))

Will it be true or false?

Practice with Conditions:

```
float a = 2.2, b = -1.2;  
int i = 5, done = 1;
```

((a < 10.0) && (b > 5.0)) Will it be true or false?

((2.2 < 10.0) && (-1.2 > 5.0))
true && false
 false

Practice with Conditions:

```
float a = 2.2, b = -1.2;  
int i = 5, done = 1;
```

((abs(i) > 2) || done) Will it be true or false?

Practice with Conditions:

```
float a = 2.2, b = -1.2;  
int i = 5, done = 1;
```

((abs(i) > 2) || done) Will it be true or false?

```
((abs(5) > 2) || done)  
 5 > 2  ||  1  
  true  ||  true  
      true
```

Precedence:

- 1) () innermost first
- 2) ++ --
++ -- post-increment. left to right
pre-increment. right to left
- 3) ! + -
+= -= *= /= %= left to right. (+ positive, - negative)
right to left.
- m
- 4) * / % left to right
- 5) + - left to right (add and subtract)
- 6) < <= > >= left to right
== != left to right
- 7) **&&** left to right
- 8) **||** left to right
- 9) = right to left (assignment)

The simple *if* structure

The simple IF statement:

```
if(condition)
{
    statement 1;
}
```

if condition is True, do statement 1.

if condition is False, skip statement 1.

Examples:

```
if (a < b)
{
    sum = sum + a;      /* simple if */
}
```

```
if (time > 1.5) {          /* another simple if */
    scanf("%d", &distance);
}
```

Use a Compound Statement or Block

a set of statements enclosed in braces { }

```
if(condition)
{
```

```
    statement 1;
    statement 2;
    statement 3;
```

```
}
```

```
if(condition) {
    statement 1;
    statement 2;
    statement 3;
}
```

Two different styles of indentation.

My version of VIM defaults to the style on the left.

The K&R book uses the style on the right.

Use indentation with a consistent style.

In both styles, the contents inside the braces are indented!!!

Example:

```
if(x != 0.0)          /* compound if */  
{  
    sum = sum +x;  
    count = count + 1;  
    printf("/nEnter another number: ");  
    scanf("%f", &x);  
}
```

Nested IF:

```
if (gpa >= 3.0)
{
    printf("Honor Roll \n");
    if (gpa > 3.5)
    {
        printf("President's List \n");
    }
}
```

Nested IF:

```
if (gpa >= 3.0)
{
    printf("Honor Roll \n");
    if (gpa > 3.5)
    {
        printf("President's List \n");
    }
    if(gpa < 2.0)
    {
        printf("Flirting with trouble \n");
    }
}
```

“Flirting with trouble” can never print from this code.

if-else statement:

```
if (course_code != 2)
{
    printf("No course listed \n");          /* true section */
    printf("No room listed \n");
}
else
{
    printf("Computer Science \n");          /* false section */
    printf("Ruthann Biel \n");
}
```

Two examples – both result in same output.

Example 1:

```
if (marital_status == 's')
{
    if (gender == 'M')
    {
        if (age >= 18)
        {
            if (age <= 26)
            {
                printf("All criteria are met.\n");
            }
        }
    }
}
```

Two examples – both result in same output.

Example 2:

```
if (marital_status == 'S'  
    && gender == 'M'  
    && age >= 18  
    && age <= 26)  
{  
    printf("All criteria are met.\n");  
}
```

```
if (road_status == 'S') /* for slick road */  
{  
    if (temp > 32)  
    {  
        printf("Wet Roads Ahead \n");  
        printf("Stopping Time Doubled \n");  
    }  
    else  
    {  
        printf("Icy Roads Ahead \n");  
        printf("Stopping Time Quadrupled \n");  
    }  
    else  
    {  
        printf("Drive Carefully! \n");  
    }  
}
```

The *else if* structure

if-else-if structure:

```
if (weight <= 50.0)
{
    category = 1;
}
else if (weight <= 125.0)
{
    category = 2;
}
else if (weight <= 200.0)
{
    category = 3;
}
else
{
    category = 4;
}
```

if-else-if structure generic form:

```
if (condition 1)
{
    statements 1;
}
else if (condition 2)
{
    statements 2;
}
    /* repeat else-if as many times as needed within reason */
else    /* the else is optional but often used for catching errors */
{
    last set of statements;
}
```

The Conditional Operator

Alternative for the if-else

Conditional Operator = **?:**

A ternary operator can be used instead of ***if-else***

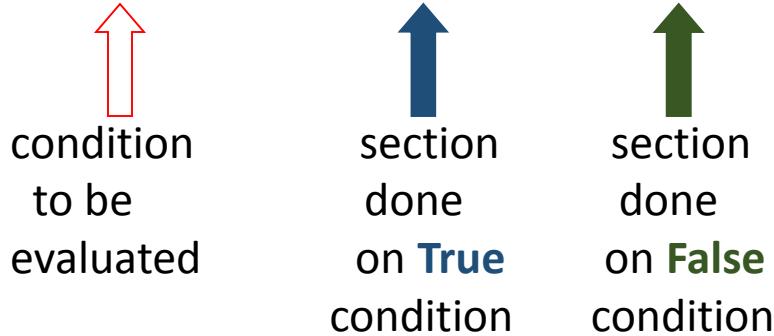
(Ternary means it has 3 parts with 2 operators.)

Conditional Operator - ?:

```
if(count <= 100)          /* if-else way /  
{  
    count +=5;  
}  
else  
{  
    sum = count + hilo;  
}
```

/* Conditional Operator way */

count <= 100 ? count +=5 : sum = count + hilo;



```
#include <stdio.h>          /* voltage.c */
#include <stdlib.h>
int main (void)
{
    float led_voltage;      /* Voltage across LED in volts. */
    float resistor_voltage; /* Voltage across resistor in volts. */
    float source_voltage;   /* Voltage of the source in volts. */
    float circuit_current;  /* Current in the LED in amperes */
    float resistor_value;   /* Value of resistor in ohms. */

    printf("\n\nEnter the source voltage in volts => ");
    scanf("%f", &source_voltage);
    printf("\n\nEnter value of resistor in ohms => ");
    scanf("%f", &resistor_value);

    led_voltage = (source_voltage < 2.3) ? source_voltage: 2.3;
    resistor_voltage = source_voltage - led_voltage;
    circuit_current = resistor_voltage / resistor_value;
    printf ("Total circuit current is %f amperes. \n", circuit_current);
    system("pause");
    return EXIT_SUCCESS;
}
```

Example of the two ways: ?: and if-else

(1)

```
led_voltage = (source_voltage < 2.3) ? source_voltage: 2.3;
```

(2)

```
if (source_voltage < 2.3)
    led_voltage = source_voltage;
else
    led_voltage = 2.3;
```

Examples of RUNs of the previous program:

Enter the source voltage in volts => 2

Enter value of resistor in ohms => 5

Total circuit current is 0.000000 amperes.

Enter the source voltage in volts => 5

Enter value of resistor in ohms => 5

Total circuit current is 0.540000 amperes.

The *switch* structure

The ***switch*** and the ***if-else-if*** produce similar results but they operate in a different way internally.

The ***if-else-if*** tests each condition until a True is encountered. It does that section, then jumps out of the structure.

The ***switch*** creates an internal table, determines where it should jump to, and then does it. It does not repeatedly test for true.

```
int code;
switch (code)
{
    case 10:
        printf ("Too hot – turn equipment off \n");
        break;
    case 11:
        printf("Caution – recheck in 5 minutes. \n");
        break;
    case 13:
        printf("Turn on the circulating fan. \n");
        break;
    default:
        printf("Normal mode of operation. \n");
        break;
}
```

```
int code;
switch (code)
{
    case 10:
        printf ("Too hot – turn equipment off \n");
        break;
    case 11:
    case 12:
        printf("Caution – recheck in 5 minutes. \n");
        break;
    case 13: case 14:
        printf("Turn on the circulating fan. \n");
        break;
    default:
        printf("Normal mode of operation. \n");
        break;
}
/* Note two cases for one print statement, two styles */
```

/* General form of the Switch */

```
switch (controlling expression or variable)
```

```
{
```

```
    case label_1:
```

```
        statements;
```

```
        break;
```

```
    case label_2:
```

```
        statements;
```

```
        break;
```

```
...
```

```
    default:
```

```
        statements;
```

```
        break;
```

```
}
```

```
/* default – optional, recommended*/
```

```
/* break – forces flow-of-control out of the switch statement */
```

Control Structures in C

And a bit on Characters

The End

C-3 Loops

```
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
    int x, sum = 0, count = 0;
    float average;
    printf("Enter a number (zero to end): ");
    scanf("%d", &x);

    while (x != 0)          /* blue shows the loop */
    {
        sum += x;
        count += 1;
        printf("\nEnter a number (zero to end): ");
        scanf("%d", &x);
    }
    average = (float) sum / (float) count;
    printf("\nThe average of these %d numbers is %.2f.\n\n",
           count, average);
    return EXIT_SUCCESS;
}
```

General Form of the while loop:

```
while (condition)
{
    statements;
}
```

If the loop has only one statement, the braces can be omitted.

Most loops have more than one statement.

On to the second type of loop

The do-while loop

```
int main (void)
{
    int x, sum = 0, count = 0;
    float average;

    do
    {
        printf("Enter a number (zero to end: ");
        scanf("%d", &x);
        sum += x;
        count += 1;
    } while (x != 0);

    average = (float) sum / ((float) count -1);
    printf ("The average of %d numbers is %f.",
           count-1, average);
    return EXIT_SUCCESS;
}
```

General Form of the do-while loop:

```
do  
{  
    statements;  
} while (condition);
```



One of the few structures we use that ends with a semicolon.

In the do-while loop, the test happens at the end. So it is guaranteed to do the “statements” section at least once.

On to the third type of loop...

The FOR loop

Degrees to Radians			
0	0.000000	180	3.141593
10	0.174533	190	3.316126
20	0.349066	200	3.490659
30	0.523599	210	3.665192
40	0.698132	220	3.839725
50	0.872665	230	4.014258
60	1.047198	240	4.188791
70	1.221731	250	4.363324
80	1.396264	260	4.537857
90	1.570796	270	4.712389
100	1.745329	280	4.886922
110	1.919862	290	5.061455
120	2.094395	300	5.235988
130	2.268928	310	5.410521
140	2.443461	320	5.585054
150	2.617994	330	5.759587
160	2.792527	340	5.934120
170	2.967060	350	6.108653
		360	6.283186

NOTE: This has been cut and pasted to fit on one slide.

```
/* Print a degree-to-radians table using a FOR loop structure */
#include <stdio.h>
#include <stdlib.h>
#define PI 3.1415
int main (void)
{
    int degrees;
    double radians;
    printf("\nDegrees to Radians \n");

    for (degrees = 0; degrees <= 360; degrees += 10)
    {
        radians = degrees * PI / 180;
        printf("%6i %9.6f \n", degrees, radians);
    }
    return EXIT_SUCCESS;
}
```

General Form of the for loop:

```
for (exp 1; exp 2; exp 3)
{
    statements;
}
```

where:

- exp 1** is used to initialize the loop-control variable
- exp 2** specifies the condition that should be TRUE to continue the loop repetition
- exp 3** specifies the modification to the loop-control variable

Picky details on the *for* loop

The minimum for a FOR loop is:

for (; ;) It must have the two semicolons
 and the ()

if missing:

exp1 – no initialization performed

exp2 – then test is ALWAYS true

exp3 – no for-loop automatic incrementing or decrementing

while (condition) { statements; }	do { statements; } while (condition);
--	--

for (exp 1; exp 2; exp 3) { statements; }
--

break – used to exit any loop or structure
immediately

continue – used to skip remaining statements in
current pass of the loop or structure

For simple short for loops, one can just write down the valid loop counters, and count them up.

```
for (x = 0; x <=18; x+=2)
```

The valid loop counters would be:

0 2 4 6 8 10 12 14 16 18

1 2 3 4 5 6 7 8 9 10 MyCountingLine.

The list consists of 10 numbers, so the loop would execute 10 times.

Computing the number of times a for loop will execute:

(PS: *floor* is a function that rounds down.)

$\text{floor} \left(\frac{\text{final} - \text{initial}}{\text{increment}} \right) + 1$

Example:

for (k = 5; k <= 83; k +=4)

$$\text{floor} \left(\frac{83 - 5}{4} \right) + 1 = \text{floor} \left(\frac{78}{4} \right) + 1 = 19 + 1 = 20$$

Loops & printf & columns.
Using integers.

Lining up numbers under column headers:

CODE:

```
int a = 125, b = 789;
```

```
printf("First Number    Second Number \n");
printf("-----    ----- \n");
printf ("%4i%4i\n\n", a, b);
```

OUTPUT:

First Number	Second Number
-----	-----
125	789

PROBLEM:

The numbers do not line up correctly under the column headers.

There are several solutions. Here is a first try:

CODE:

```
int a = 125, b = 789;
```

```
printf("First Number    Second Number \n");
printf("-----      ----- \n");
printf ("%8i  %8i\n\n", a, b);
```

OUTPUT:

First Number	Second Number
-----	-----
125	789

PROBLEM:

The 125 is almost centered (if that is desired).
The 789 is still in the wrong place.

There are several solutions. Here is another try:

CODE:

```
int a = 125, b = 789;
```

```
printf("First Number      Second Number \n");
printf("-----      ----- \n");
printf ("%8i      %8i\n\n", a, b);
```

OUTPUT:

First Number	Second Number
-----	-----
125	789

PROBLEM:

The 125 is almost centered (if that is desired).
Added 5 more spaces between.
Now the 789 is about right place.

Loops & printf & columns.
Using variables with decimal
points.

Lining up numbers under column headers:

CODE:

```
double a = 125.6, b = 7.89, c = 45.678, d=567.1234;
```

```
printf("1st Column  2nd Column \n");
printf("----- ----- \n");
printf("%f  %f \n", a, b);
printf("%f  %f \n", c, d);
```

OUTPUT 2:

1st Column	2nd Column
-----	-----
125.6000	7.8900
45.6780	567.1234

PROBLEM:

Getting closer to a correct solution, but the spacing is still off.

CODE:

```
double a = 125.6, b = 7.89, c = 45.678, d=567.1234;
```

The most digits before the decimal point = 3

The decimal point will take one space. = 1

The most digits after the decimal point = 4

Solution = **%8.4f**

```
printf("%8.4f %8.4f \n", a, b);  
printf("%8.4f %8.4f \n", c, d);
```

Output 3:

1st Column	2nd Column
------------	------------

-----	-----
-------	-------

125.6000	7.8900
----------	--------

45.6780	567.1234
---------	----------

The decimal points line up but more space is required to get the numbers to line up with the headers.

CODE:

```
double a = 125.6, b = 7.89, c = 45.678, d=567.1234;
```

Both columns ought to shift to the right by 2 spaces.

Solution = %10.4f

```
printf("%10.4f %10.4f \n", a, b);
printf("%10.4f %10.4f \n", c, d);
```

Output 4:

1st Column	2nd Column
-----	-----
125.6000	7.8900
45.6780	567.1234

The decimal points line up.

The numbers line up with the headers.

SUCCESS!

CODE:

```
double a = 125.6, b = 7.89, c = 45.678, d=567.1234;
```

I allowed 3 spaces between the two header lines.

I could have left those three spaces **out**,
and added the 3 to the second set of conversion specifiers.

```
printf("%10.4f%13.4f \n", a, b);
printf("%10.4f%13.4f \n", c, d);
```

Output 5:

1st Column	2nd Column
-----	-----
125.6000	7.8900
45.6780	567.1234



C-3 Loops

THE END

C-4 Using Files

Creating a file with values

Give the file the name of RESIST.DAT

Type these numbers inside the file:

1000
1100
2000
500
1000
2000

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

int main (void)
{
    double r1, r2, r3, r_combo;
    FILE * input_file;           // Declare a pointer variable
    FILE * output_file;         // for each file.

    // more of this program on the next slide
```

```
input_file = fopen ("resist.dat", "r");
if(input_file == NULL)
{
    printf("Error on opening the input file \n");
    exit (EXIT_FAILURE); // ( ) required since exit is a function
}
```

```
output_file = fopen ("resist.out", "w");
if(output_file == NULL)
{
    printf("Error on opening the output file \n");
    exit (EXIT_FAILURE);
}
```

// **more** of this code on the **next** slide

```
/* Now that the files are open, we can use them */
fprintf(output_file, "\nRuthann Biel. Resistance Program. \n\n");

while ((fscanf(input_file, "%lf%lf%lf ", &r1, &r2, &r3)) == 3)
{
    r_combo = 1.0 / (1.0/r1 + 1.0/r2 + 1.0/r3);
    fprintf(output_file, "Three resistors are: %f %f %f \n", r1, r2, r3);
    fprintf(output_file, "Combined Parallel Resistance: %f \n\n",
            r_combo);
}
fclose(input_file);
fclose(output_file);

return EXIT_SUCCESS;
}
/*-----*/
```

This is the contents of resist.out

Ruthann Biel. Resistance Program.

Three resistors are: 1000.000000 1100.000000 2000.000000
Combined Parallel Resistance = 415.094330

Three resistors are: 500.000000 1000.000000 2000.000000
Combined Parallel Resistance = 285.714294

Programs & Files

A program can **read** from a file containing data as input.

A program can **write** data to a file as output (rather than to the screen).

For each file used, one must have a file pointer.

```
FILE * my_data;
```

The Details:

FILE - the word must be capitalized.

asterisk - indicates that it is a pointer variable.

my_data - name of the file pointer.

(created by the programmer, you or me)

Next step.

Associate the **file pointer with a file name** by using an *fopen*

```
my_data = fopen ("body_info.dat", "r");
```

The Details:

my_data - file pointer name

fopen - opens the file & creates a connection
between the file and your program

body_info.dat – the file name as in Windows

“r” - means the file is for “read only”

Error Checking on `fopen`

After each `fopen`, one must error check.

```
if( file_pointer_name == NULL)  
{
```

- print an error message so you will know what is wrong
- do an `exit(EXIT_FAILURE);` to leave the program.

Makes no sense to go on without data

```
}
```

To read the data from the file into the program, use the *fscanf* statement (file scan function)

```
fscanf(my_data, "%lf%lf ", &height, &weight);
```

Similar to a *scanf*, except:

- (1) use *fscanf* instead of *scanf*
- (2) add the file pointer name immediately inside the parentheses

To write the data to a file from the program, use the *fprintf* statement (file print function)

```
FILE * out_file;  
...  
out_file = fopen("results.out", "w");  
...  
fprintf(out_file, "%f%f ", height, weight);
```

Similar to a printf, except:

- (1) use fprintf instead of printf
- (2) add the file pointer name immediately inside the parentheses

When one is done with the files, close them.

```
fclose(my_data);
```

```
fclose(out_file);
```

One can also use **fclose** to re-open a file
for repeated use.

A Variation or Alternative Way:

Style 1

```
FILE *my_data;  
...  
my_data = fopen ("body_info.dat", "r");  
    // Use the file name directly in the fopen
```

Style 2

```
#define IN_FILE_NAME "body_info.dat"  
...  
FILE *my_data;  
...  
my_data = fopen (IN_FILE_NAME, "r");  
    // Use a variable that holds the file name  
    // Notice that the quotes are on the define line  
    // and there are NO quotes in the fopen for the file name
```

```
/* Using the alternative method */
```

```
#include <stdio.h>
#include <stdlib.h>
#define IN_FILE_NAME "body_info.dat"
#define OUT_FILE_NAME "results.out"
int main (void)
{
    FILE * my_data;
    FILE * out_file;
    ...
    my_data = fopen (IN_FILE_NAME, "r");
    if (my_data == NULL)
    {
        printf ("Error opening the input file \n");
        exit (EXIT_FAILURE);
    }                                //more on next slide
```

```
out_file = fopen (OUT_FILE_NAME, "w");
if (out_file == NULL)
{
    printf ("Error opening the output file \n");
    exit (EXIT_FAILURE);
}

...
fscanf(mydata, ...);

...
fprintf(out_file, ...);

...
fclose (my_data);
fclose (out_file);

.....
}
```

Reading a data file:

Contents of the file (in blue):

Date Rain

1	0.0
2	0.1
3	0.9
4	1.5
5	2.0
6	1.1

```
while((fscanf(infile, "%d%f ", &date, &rain)) == 2)
{
    ....
}
```

fscanf returns the number of values read; so here we continually read **two** values, until no more data.



Controlling Files
and their End.

First example – FOR loop.

(1) We have a known number of records or lines in a file, so we can use a for loop.

```
int main (void)
{
    int i, n, max = 0;
    FILE * infile;
    infile = fopen("d.dat", "r");
    if (infile == NULL)
    {
        printf ("Error on input file open\n");
        exit (EXIT_FAILURE);
    }
    for (i = 1; i <= 5; i++)
    {
        fscanf (infile, "%d", &n);
        if (n > max)
            max = n;
    }
    printf ("\nMax is %d \n\n", max);
    fclose (infile);
    return EXIT_SUCCESS;
}
```

FILE CONTENTS:

14
65
24
72
40

Classroom Program
Files_For.c

Second Example – DO WHILE loop

(2) We have a known trailer signal or sentinel signal or a “dummy value” in the file, so we can use a do while loop, continuing to loop until we find the marker at the end.

```
int main (void)
{
    int i, n, max = 0;
    FILE * infile;
    infile = fopen("d2.dat", "r");
    fscanf (infile, "%d", &n);
    max = n;
    do
    {
        if (n > max)
            max = n;
        fscanf (infile, "%d", &n);
    } while (n > -1);

    printf ("Max is %d \n", max);
    fclose (infile);
    return EXIT_SUCCESS;
}
```

FILE CONTENTS:

```
14
65
24
72
40
-1
```

Classroom Program
Files_DoWhile.c

Third example – FOR loop with length in file.

- (3) We have a file where the first value in the file tells us how many values follow in the file, so we can use a for loop.

```
int main (void)
{
    int i, n, max = 0, end;
    FILE * infile;
    infile = fopen("d3.dat", "r");
    fscanf (infile, "%d", &end);

    for (i = 1; i <= end; i++)
    {
        fscanf (infile, "%d", &n);
        if (n > max)
            max = n;
    }
    printf ("Max is %d \n", max);
    fclose (infile);
    return EXIT_SUCCESS;
}
```

FILE CONTENTS:

5
14
65
24
72
40

Classroom Program
Files_For_length.c

Fourth Example – Looking for the end.

(4) We look for a good read from the file. In this case, we are reading one value at a time.

```
while ( (fscanf(my_data, "%f ", &x) ) == 1)
```

If the fscanf returns a “1”,
then we know we got a “good” read.

If the fscanf returns a “0”,
then we know we are out of numbers.

```
int main (void)
{
    int n, max, n_pts = 0;
    FILE * infile;
    infile = fopen("d.dat", "r");
    while( (fscanf (infile, "%d", &n) ) ==1)
    {
        n_pts++;
        if (n_pts == 1)
            max = n;
        if (n > max)
            max = n;
    }
    printf ("Max is %d \n", max);
    fclose (infile);
    system("pause");
    return EXIT_SUCCESS;
}
```

FILE CONTENTS:

14
65
24
72
40

Classroom Program
Files_While_EOF.c

C-4 Files

Input/Output And Using Files

The End

C-5 Functions

Library Functions:

- They come with the system.
- We have already used some of them.
- Examples:
 - **x = sqrt (y);**
 - **w = pow(x,3);**
 - **Sine = sin(start);**

Vocabulary:

Function Prototype A definition or outline of a function to be used.

Each function requires a Function Prototype

The Prototype can tell us:

- if a function will return a value
- what values will be sent in to the function
- what type those values have (int, double, etc)
- the name of the function.

Function Prototype

For functions that **we write**, the prototype is located:

- Above the line “int main(void)”
- Later we will learn to put them in a separate file.

For functions that come with the **system**:

- The prototype is located in the system include file.
- Ex: The prototype for *printf* is located in *stdio.h* and we are not required to re-enter it as long as our program starts with *#include <stdio.h>*

```
/* An example */
/*-----*/
#include <stdio.h>
#include <stdlib.h>
int reverse (int num);      /* function prototype */
/*-----*/
int main (void)
{
    int num_in;

    printf ("\nEnter a 2-digit number: ");
    scanf ("%i", &num_in);

    printf ("\n\nThe number %i reversed is %d \n\n",
           num_in, reverse(num_in));
    return EXIT_SUCCESS;
}
/*---End of main-----*/
```

```
/*-----*/  
// This function will reverse a two digit number  
  
int reverse (int num)  
{  
    int digit1, digit2;  
  
    digit1 = num / 10;  
    digit2 = num % 10;  
    return (digit2 * 10) + digit1;  
}  
/*---End of reverse-----*/
```

The Run:

The number 27 reversed is 72

```
#include <stdio.h>           //Another example
#include <stdlib.h>
void function1 (void);      /* function prototypes */
void function2 (int n, double x);

/*-----*/
int main (void)
{ int m; double y;
  m = 15;
  y = 308.24;
  printf ("The value of m in main is m = %d \n\n", m);

  function1 ( );           /* has no argument */
  function2 (m, y);

  printf ("The value of m in main is still m = %d \n\n", m);
  return EXIT_SUCCESS;
}
/*-----*/
```

Output after the 1st printf in main:

The value of m in main is m = 15

```
/*-----*/  
void function1 (void)  
{  
    printf("function 1 is a void function that does"  
          " not receive values from main. \n\n");  
    return;  
}  
/*-----*/
```

Output after we finish with **function 1**:

The value of m in main is m = 15

function 1 is a void function that does not receive values from main.

```
/*-----*/
```

```
void function2 (int n, double x)
```

```
{
```

```
    int k, m;
```

```
    double z;
```

```
    k = 2 * n + 2;
```

```
    m = 5 * n + 37;
```

```
    z = 4.0 * x - 58.4;
```

```
    printf("function2 is a void function that does receive \n"
```

```
          "values from main. The values received from main are: "
```

```
          "\t n = %d \n\t x = %lf \n\n", n, x);
```

```
    printf("function2 creates three new variables, k, m, and z \n"
```

```
          "These variables have the values : \n"
```

```
          "\t t = %d \n\t m = %d \n\t z = %lf \n\n", k, m, z);
```

```
    return;
```

```
}
```

```
/*-----*/
```

Output – FINAL – after function 2:

The value of m in main is m = 15

function 1 is a void function that does not receive values from main.

function2 is a void function that does receive values from main. The values received from main are:

n = 15

x = 308.240000

function2 creates three new variables, k, m, and z

These variables have the values:

k = 32

m = 112

z = 1174.560000

The value of m in main is still m = 15

General Form of Functions:

Functions that don't return a value:

```
void function_name (parameter declarations) {  
    declarations;  
    statements;  
    return;  
}
```

Function Example: **void function2 (int n, double x)**

The prototype for this sort of function:

```
void function_name (parameter declarations);
```

Prototype Example: **void function2 (int n, double x);**

General Form of Functions:

Functions that return one value & only one value.

```
return_type function_name (parameter declarations)
{
    declarations;
    statements;
    return expression;
}
```

Function Example: int reverse (int num)

The prototype for this sort of function:

```
return_type function_name (parameter declarations);
```

Prototype Example: int reverse (int num);

We will use a return in every function that we write;

It is good engineering practice.

Call-by-value

On passing a value to a function,
the actual value stored in memory is passed to the
sub-function,
not its memory location.

In other words,
main keeps the original copy.

The sub-function gets a xerox copy.
If the sub-function alters its xerox copy,
it does not change the original copy/value in main.

```
/*-----Functions2.c-----*/
#include <stdio.h>
#include <stdlib.h>
int m = 12;      /* file scope variable (global) */

int function1 (int a, int b, int c, int d); /*function prototype */
/*-----*/
int main (void)
{
    int n = 30;
    int e, f, g, h, i;
    e = 1;
    f = 2;
    g = 3;
    h = 4;
    printf("\n\nIn main (before call to function 1):\n"
          " m = %d\n n = %d\n e = %d\n", m, n, e);
```

Let's stop and see the output of that printf:

In main (before the call to function1):

m = 12

n = 30

e = 1

and now on with the code, repeating that printf to put us in context...

```
printf("\n\nIn main (before call to function 1):\n"
      " m = %d\n n = %d\n e = %d\n", m, n, e);
```

```
i = function1(e, f, g, h);
```

```
printf("After returning to main: \n");
printf(" n = %d \n m = %d \n e = %d \n i = %d\n",
      n, m, e, i);
```

```
return EXIT_SUCCESS;
```

```
}
```

```
/*-----end of main -----*/
```

```
/*-----*/  
int function1 (int a, int b, int c, int d)  
{  
    int n = 400;  
    printf("In function1: \n n = %d \n m = %d initially \n"  
          " a = %d initially \n\n", n, m, a);  
    m = 999;  
  
    if (a >= 1) {  
        a += b + m + n;  
        printf(" m = %d after being modified \n"  
              " a = %d after being modified \n\n", m, a);  
        return a;  
    }  
    else {  
        c += d + m + n;  
        return c;  
    }  
}  
/*-----end of function 1-----*/
```

Now to see the WHOLE output:

In main (before the call to function1):

m = 12

n = 30

e = 1

In function1:

n = 400

m = 12 initially

a = 1 initially

m = 999 after being modified

a = 1402 after being modified

After returning to main:

n = 30

m = 999

e = 1

i = 1402

Scope of Variables

Definition of SCOPE

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed.

There are three types of Scope.

Scope of Variables:

(1) **local** variables –

- located inside a function or a block
- an identifier declared within a block unit is only active within the block.
- blocks are surrounded by { }
- an identifier declared within a function is only active within the function.

Scope of Variables:

(2) **global** variables –

- located outside of all functions
- usually on top of the program.
- they hold their values throughout the lifetime of your program
- they can be accessed inside any of the functions defined for the program.

Scope of Variables:

(3) **prototype scope**

- an identifier used in a function prototype
- used only within the one line of the prototype
- is not declared outside that line
- are treated as local variables with-in a function
- they take precedence over global variables.

Where do you Declare Variables?

Outside any *function definition*

.e.g., Prior to the start of the **main()** function

Global/external variables

Within a *function*, after the opening {

Local to the function

Within a *block of code*, after the {

Local to the area surrounded by the {} braces

Random Numbers

RANDOM NUMBERS

In stdlib.h there is a function to generate random numbers:

```
int rand (void);
```

rand generates a random integer
between 0 and RAND_MAX

RAND_MAX is a system-defined integer in stdlib.h
in our system, it is:

```
/* The largest number rand will return (same  
as INT_MAX). */
```

```
#define RAND_MAX      2,147,483,647
```

Random numbers are generated using a seed value.

By default, the seed = 1

Examples:

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

41 18467 6334

```
for (k=1; k <= 10; k++)
```

```
{
```

```
    printf("%i", rand( ) );
```

```
}
```

41 18467 6334 26500 19169

15724 11478 29358 26962 24464

These two examples show that `rand()` is *pseudo random*.

Random numbers are generated using a seed value.

By default, the seed = 1

The same random number sequence will be generated given a certain seed.

This is called **pseudo-randomness**.

We can change the value of seed from the default by using another function *srand* .

Its prototype in stdlib.h is:

void srand (unsigned int);

A different seed will generate a different sequence of random numbers.

Examples:

```
unsigned seed;  
printf("Enter a seed: ');  
scanf("%u", &seed);      /* %u for unsigned int */  
srand(seed);  
printf("%i", rand( ) );
```

seed of 1 → 41

seed of 123 → 440

seed of 5 → 54

So using the different seeds produces
a list of numbers that look random.

```
/*-----*/
/* This program generates and prints ten random */
/* integers between user-specified limits.      */

#include <stdio.h>
#include <stdlib.h>

int rand_int(int a, int b); // function prototype
/*-----*/
int main(void)
{
    /* Declare variables and function prototypes. */
    unsigned int seed;
    int a;
    int b;
    int k;
    char go_on[3] = "y";
```

```
while (go_on[0] == 'y' || go_on[0] == 'Y')
{
    /* Get seed value and interval limits. */
    printf("\nEnter a positive integer seed value: ");
    scanf("%u",&seed);
    srand(seed);
    printf("\nEnter integer limits a and b (a<b): ");
    scanf("%i %i",&a,&b);

    /* Generate and print ten random numbers */
    printf("\nRandom Numbers: \n\n");
    for (k=1; k<=10; k++)
        printf("%i ",rand_int(a,b));
    printf("\n\n");
    printf("Enter \"y\" or \"Y\" for YES if you wish to continue: ");
    scanf("%s", go_on);
}
```

```
    return EXIT_SUCCESS;  
}  
/*-----End of main-----*/
```

```
/* This function generates one random integer      */  
/* between specified limits a and b (a<b).          */
```

```
int rand_int(int a, int b)  
{  
    return (rand( ) % (b - a + 1) + a);  
}
```

```
/*-----End of rand_int-----*/
```

Functions that “return” more
than one value

First a look at a variable.

variable

name → counter



address → 664136

New use of operators:

We will use the address operator (**&**) to pass the address of the variable to a sub-function.

Inside the sub-function, we will use the

indirection operator (*****) (the asterisk)

to refer to the contents of the variable rather than its address.

```
#include <stdio.h>           //Functions3.c
#include <stdlib.h>
#include <math.h>

void find_area(double s, double *b, double *h, double *a);
    /* function prototype */
/*-----*/
int main(void)
{
    double side = 5.1875; /* triangle parts*/
    double base = 3;
    double height = 5;
    double area = 0;

    printf ("\nIn main before function, the values are: \n"
            " Side = %f \n"
            " Base = %f \n"
            " Height = %f \n"
            " Area = %f \n", side, base, height, area);
```

Results of the Run:

In main before function, the values are:

Side = 5.187500

Base = 3.000000

Height = 5.000000

Area = 0.000000

```
/* after the printf statement */

find_area(side, &base, &height, &area);

printf ("\n\n main after function, the values are: \n"
        " Side = %f \n"
        " Base = %f \n"
        " Height = %f \n"
        " Area = %f \n\n", side, base, height, area);

return EXIT_SUCCESS;
}

/*----end of main-----*/
```

```
/*-----*/
/* This function will calculate the area of a triangle */

void find_area(double s, double *b, double *h, double *a)
{
    *a = (*b) * (*h)) / 2.0;
    s = 400; /* Just to see what will happen */

    printf ("\n In find_area after computation, the values are: \n"
           "     Side  = %f \n"
           "     Base  = %f \n"
           "     Height = %f \n"
           "     Area   = %f \n", s, *b, *h, *a);

    return;
}
/*-----end of find_area-----*/
```

Results of the Run:

In main before function, the values are:

Side = 5.187500

Base = 3.000000

Height = 5.000000

Area = 0.000000

In find_area after computation, the values are:

Side = 400.000000

Base = 3.000000

Height = 5.000000

Area = 7.500000

In main after function, the values are:

Side = 5.187500

Base = 3.000000

Height = 5.000000

Area = 7.500000

Now to really understand why the “`&`” and “`*`” pair work (the address operator and the indirection operator).

We **re-run** the above program but this time we will **print** out the actual addresses in memory of the variables we used.

%p prints an address in a notation consistent with the addressing scheme of our computers. (for our computers, this is HEX)

%u prints an unsigned integer (in base ten)

```
/*-----*/  
#include <stdio.h>           //functions4.c  
#include <stdlib.h>  
#include <math.h>  
  
void find_area(double s, double *b, double *h, double *a);  
    /* function prototype */  
/*-----*/  
int main(void)  
{  
    double side = 5.1875; /* triangle parts*/  
    double base = 3;  
    double height = 5;  
    double area = 0;
```

```
printf ("\n In main before function, the values are: \n"
        " Side = %f \n"
        " Base = %f \n"
        " Height = %f \n"
        " Area = %f \n", side, base, height, area);
```

/ NEW Printf follows */*

```
printf ("\n In main, the addresses are: \n"
        " Side = %p %u\n"
        " Base = %p %u\n"
        " Height = %p %u\n"
        " Area = %p %u",
        &side,&side,
        &base,&base,
        &height,&height,
        &area,&area);
```

In main before function, the values are:

Side = 5.187500

Base = 3.000000

Height = 5.000000

Area = 0.000000

In main, the addresses are:

Side = 0012FF78 1245048

Base = 0012FF70 1245040

Height = 0012FF68 1245032

Area = 0012FF60 1245024

```
/* rest of function main */

find_area(side, &base, &height, &area);

printf ("\nIn main after function, the values are: \n"
        " Side = %f \n"
        " Base = %f \n"
        " Height = %f \n"
        " Area = %f \n", side, base, height, area);

return EXIT_SUCCESS;
}

/*----end of main-----*/
```

```
/* This function will calculate the area of a triangle */
void find_area(double s, double *b, double *h, double *a)
{
    *a = (*b * *h) / 2.0;
    s = 400; /* Just to see what will happen */
    printf ("\n In find_area after computation, the values are: \n"
            "     Side  = %f \n"
            "     Base  = %f \n"
            "     Height = %f \n"
            "     Area   = %f \n", s, *b, *h, *a);
    printf ("\n In find_area, the addresses are: \n"
            "     Side  = %p %u\n"
            "     Base  = %p %u\n"
            "     Height = %p %u\n"
            "     Area   = %p %u\n", &s,&s, b,b, h,h, a,a);
    return;
}          /*----end of find_area---*/
```

In `find_area` after computation, the values are:

Side = 400.000000

Base = 3.000000

Height = 5.000000

Area = 7.500000

In `find_area`, the addresses are:

Side = 0012FF00 1244928

Base = 0012FF70 1245040

Height = 0012FF68 1245032

Area = 0012FF60 1245024

In main after function, the values are:

Side = 5.187500

Base = 3.000000

Height = 5.000000

Area = 7.500000

More on Scope of Variables

Storage Classes Specifiers

automatic – the default. We may use the word “auto”.

Ex. auto int x;

external – for global variables initially declared in other files

Ex. extern int count;

static - specifies that memory for this local variable should be kept or saved even after control has returned to the calling function

Ex. static int function_count;

register – specifies use of computer registers, rather than memory IF POSSIBLE.

Ex. register int speed;

typedef – to be covered in class later this semester.

Example of Storage Class:

```
#include ...
int count=0;
...
int main (void) {
    int x, y, z;
    ...
}
/*-----*/
extern int count;
int calc(int a, int b) {
    int x;
    ...
}
/*-----*/
extern int count;
int check(int sum) {
    ...
}
/*-----*/
```

count is a global variable referenced by **calc** and **check**.

x, y. & z are local variables referenced only by **main**.

a, b, x are local variables only referenced by **calc**.

sum is a local variable that can only be referenced by **check**.

There are TWO local variables **x** but they are different variables with different scopes.

Static Variables

Scope Rules for “Static” variables

- Static Variables: use static prefix on functions and variable declarations to limit scope
 - static prefix on external variables will limit scope to the rest of the source file (not accessible in other files)
 - static prefix on functions will make them invisible to other files
 - static prefix on internal variables will create permanent private storage; retained even upon function exit

New operator: sizeof()

The **sizeof** operator may be used to determine the size of a data object or type.

```
/* Demonstrate static variables (4_UNIX)      */\n\n#include <stdio.h>\n#include <stdlib.h>\n\nchar name[100];          /* variable accessible from all files */\nstatic int i;              /* variable accessible only from this\n                           file */\nstatic int max_so_far(int); /* function prototype accessible\n                           only from this file */\n\nint main (void) {\n    int val[] = {14, 25, 10, 100, 20};\n    int i;\n    for (i=0; i<sizeof(val)/sizeof(int); i++)\n        printf("max = %d \n", max_so_far(val[i]));\n    return (EXIT_SUCCESS);\n}\n/*-----*/
```

```
/*-----*/  
/* code for the function */  
  
int max_so_far(int curr) {  
    static int biggest=0; /*Variable whose value is retained  
                         between each function call */  
    if(curr > biggest)  
        biggest=curr;  
    return biggest;  
}  
/*-----*/
```

The run with the static variable:

```
[bielr@athena ClassExamples]> a.out  
max = 14  
max = 25  
max = 25  
max = 100  
max = 100  
[bielr@athena ClassExamples]>
```

C-5 Functions

The End

C-6 ARRAYS

One Dimensional Arrays

An Array named **seconds**:

Contents of the Cell	Name of Each Cell
10	seconds [0]
13	seconds [1]
9	seconds [2]
45	seconds [3]
14	seconds [4]

The array **seconds** contains five values.

Declaring an array:

```
int seconds[5];
```

```
float time[100];
```

Initializing the array at start:

```
int seconds[5] = {10, 13, 9, 45, 14};
```

```
float time[100] = {0.0};
```

```
int a[ ] = {2, 4, 6};
```

Example – Fill an array with values:

```
int j, c = 2, a[100];  
  
for (j = 0; j < 100; j++)  
{  
    a[ j ] = c;  
    c = c + 2;  
}
```

Example using a variable for the length of the array

```
#define A_SIZE 100  
...  
int j, c = 2, a[A_SIZE];  
...  
for (j = 0; j < A_SIZE; j++)  
{  
    a[ j ] = c;  
    c = c + 2;  
}
```

Reading Arrays

```
#define MSIZE 100
...
int c = 0, miles[MSIZE], sum = 0;
FILE *datfile;

datfile = fopen("travel.dat", "r");
if(datfile == NULL)
{
    printf("Error opening file");
    exit EXIT_FAILURE;
}

while( (fscanf(datfile, "%d", &miles[c])) == 1 )
{
    sum += miles[c];
    c++;
}
```

Printing Arrays – example 1:

```
for (k = 0; k < MSIZE; k++)  
{  
    printf("%i \n", miles[k]);  
}
```

Printing Arrays – example 2:

```
for (k = 0; k < c; k++)  
{  
    printf("%d \n", miles[k]);  
}
```

Printing Arrays – example 3:

```
for (k = 0; k < MSIZE; k++)
{
    if (k % 4 == 0)
        printf("\n %f", miles[k]);
    else
        printf("%f  ", miles[k]);
}
printf("\n");
```

// Prints 4 numbers per line.

Precedence of [] :

[] appears high on chart with ()

Both are at the same level

```
/*-----*/  
/* try initialization of array with 2 values */  
  
#include <stdio.h>  
#include <stdlib.h>  
  
int main(void)  
{  
    int i;                  /* loop counter */  
    int a[10] = {1, 2};  
  
    for (i = 0; i < 10; i++)  
        printf("\nPosition %i has %i", i, a[i]);  
    printf("\n\n");  
  
    return EXIT_SUCCESS;  
}
```

The run looks like this:

Position 0 has 1
Position 1 has 2
Position 2 has 0
Position 3 has 0
Position 4 has 0
Position 5 has 0
Position 6 has 0
Position 7 has 0
Position 8 has 0
Position 9 has 0

***If part of an array is initialized,
the remaining array is initialized to zero.***

Functions & Arrays

Function Arguments and Arrays

Arrays are not the same as simple variables.

Simple variables in function call are call-by-value

Value of variables passed to function
(xerox copy concept)

Arrays in function call are call-by-address

The starting address of the array is passed to function
Not just the values!

The function can **change** the values in the array.
(original copy concept)

Just like what we did with the (& *) pair

Function Prototype:

```
double max (double x[ ], int n);
```

where x is an array and n is its size

In main, declare an array and use it in this function call:

```
double y[N]; /* declare array of size N */  
int npts;      /* actual length of array */
```

```
printf("The maximum value is: %f \n", max(y, npts));
```

Array Examples

```
int x[10] = {-5, 4, 3};
```

If initializing sequence is shorter than the array, then the rest of the values are initialized to zero.

Values of x: -5, 4, 3, 0, 0, 0, 0, 0, 0, 0

Show the contents of this array:

```
double z[4];
```

...

```
z[1] = -5.5;
```

```
z[2] = z[3] = fabs(z[1]);
```

Show the contents of this array:

```
double z[4];
```

...

```
z[1] = -5.5;
```

```
z[2] = z[3] = fabs(z[1]);
```

Contents would be:

? -5.5 5.5 5.5

Show the contents of this array:

```
int k;  
double time[9];  
...  
for (k = 0; k < 9; k++)  
{  
    time[k] = (k - 4) * 0.1;  
}
```

Show the contents of this array:

```
int k;  
double time[9];  
...  
for (k = 0; k < 9; k++)  
{  
    time[k] = (k - 4) * 0.1;  
}
```

Contents would be:

-0.4 -0.3 -0.2 -0.1 0.0 0.1 0.2 0.3 0.4

Show what is printed for the following:

```
int k, s[ ] = {3, 8, 15, 21, 30, 41};  
  
for (k = 0; k < 6; k += 2)  
{  
    printf("%i %i \n", s[k], s[k + 1]);  
}
```

Show what is printed for the following:

```
int k, s[ ] = {3, 8, 15, 21, 30, 41};  
          0  1   2   3   4   5 ←position in array  
for (k = 0; k < 6; k += 2)  
{  
    printf("%i %i \n", s[k], s[k + 1]);  
}
```

It would print:

3 8
15 21
30 41

Show what is printed for the following:

```
int k, s[ ] = {3, 8, 15, 21, 30, 41};
```

```
for (k = 0; k < 6; k++)
{
    if (s[k] % 2 == 0)
    {
        printf("%i", s[k]);
    }
}
printf("\n");
```

Show what is printed for the following:

```
int k, s[ ] = {3, 8, 15, 21, 30, 41};
```

```
for (k = 0; k < 6; k++)
{
    if (s[k] % 2 == 0)
    {
        printf("%4i", s[k]);
    }
}
printf("\n");
```

8 30

Two Dimensional Arrays

Two-dimensional Arrays

Arrays with both rows and columns:

Row 0	-1	3	2	6
Row 1	5	3	1	-1
Row 2	10	4	-2	9
	Column 0	Column 1	Column 2	Column 3

Initialization

Row first, then column

```
int c[3][4]; /* the array on the last slide */
```

```
int c[3][4] = {{-1, 3, 2, 6}, {5, 3, 1, -1}, {10, 4, -2, 9}};
```

```
/* initialization of the same array */
```

We use nested loops with two-dim. arrays.
What will be in this array when it is finished?

```
int r, c, x[3][4];  
  
for (r = 0; r < 3; r++)  
{  
    for (c = 0; c < 4; c++)  
    {  
        x[r][c] = r;  
    }  
}
```

We use nested loops with two-dim. arrays.
What will be in this array when it is finished?

```
int r, c, x[3][4];  
  
for (r = 0; r < 3; r++)  
{  
    for (c = 0; c < 4; c++)  
        x[r][c] = r;  
}
```

0 0 0
1 1 1 1
2 2 2 2

Code to fill an Identity Matrix:

```
int r, c, m[4][4];

for (r = 0; r < 4; r++)
{
    for (c = 0; c < 4; c++)
    {
        if (r == c)
            m[r][c] = 1;
        else
            m[r][c] = 0;
    }
}
```

Code to fill an Identity Matrix:

```
int r, c, m[4][4];  
  
for (r = 0; r < 4; r++)  
{  
    for (c = 0; c < 4; c++)  
    {  
        if (r == c)  
            m[r][c] = 1;  
        else  
            m[r][c] = 0;  
    }  
}
```

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Code to add 2 matrices

```
/*-----*/  
#define N 4 /*parts of main */  
void matrix_add(int d[N][N], int e[N][N], int f[N][N]);  
  
int main(void)  
{  int r, c;  
    int d[N][N], e[N][N], f[N][N];  
  
    ...  
    matrix_add(d, e, f);  
    /* loop to print matrix c */  
    for (r = 0; r < N; r++)  
    {  
        for (c = 0; c < N; c++)  
            printf("%i ", f[r][c]);  
    }  
}  
/*----end of main-----*/
```

```
/*-----*/
/* function to add 2 matrices          */
void matrix_add(int d[N][N], int e[N][N], int f[N][N])
{  int r, c;
   for (r = 0; r < N; r++)
   {
      for (c = 0; c < N; c++)
      {
         f[r][c] = d[r][c] + e[r][c];
      }
   }
   return;      /* void return */
}
/*----end of matrix_add-----*/
```

Show the contents
of these arrays

Show the contents of the array:

```
int d[3][1] = {{1}, {4}, {6}};
```

Show the contents of the array:

```
int d[3][1] = {{1}, {4}, {6}};
```

1

4

6

Show the contents of the array:

```
int g[6][2] = {{5,2}, {-2,3}};
```

Show the contents of the array:

```
int g[6][2] = {{5,2}, {-2,3}};
```

5 2

-2 3

0 0

0 0

0 0

0 0

Show the contents of the array:

```
float h[4][4] = {{0.0}};
```

Show the contents of the array:

```
float h[4][4] = {{0.0}};
```

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

Show the contents of the array:

```
int k, p[3][3] = {{0,0,0}};  
...  
for (k = 0; k < 3; k++)  
{  
    p[k][k] = 1;  
}
```

Show the contents of the array:

```
int k, p[3][3] = {{0,0,0}};
```

...

```
for (k = 0; k < 3; k++)  
{  
    p[k][k] = 1;  
}
```

1 0 0
0 1 0
0 0 1

Show the contents of the array:

```
int r, c, g[5][5];  
...  
for (r = 0; r < 5; r++)  
{  
    for (c = 0; c < 5; c++)  
    {  
        g[r][c] = r + c;  
    }  
}
```

Show the contents of the array:

```
int r, c, g[5][5];  
...  
for (r = 0; r < 5; r++)  
{  
    for (c = 0; c < 5; c++)  
    {  
        g[r][c] = r + c;  
    }  
}
```

0	1	2	3	4
1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8

Show the contents of the array:

```
int r, c, g[5][5];  
...  
for (r = 0; r < 5; r++)  
{  
    for (c = 0; c < 5; c++)  
    {  
        g[r][c] = pow(-1, c);  
    }  
}
```

Show the contents of

the array:

```
int r, c, g[5][5];
```

...

```
for (r = 0; r < 5; r++)
{
    for (c = 0; c < 5; c++)
    {
        g[r][c] = pow(-1, c);
    }
}
```

1	-1	1	-1	1
1	-1	1	-1	1
1	-1	1	-1	1
1	-1	1	-1	1
1	-1	1	-1	1

C-6 ARRAYS

THE END

C-7Pointers

Why have pointers?

- Pointers allow different sections of code to share information easily. You can get the same effect by copying information back and forth, but pointers solve the problem better.
- Pointers enable complex "linked" data structures like linked lists and binary trees.
- The use of strings in C require a knowledge of pointers.

Addresses

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int a = 1, b = 2;
    printf("a = %i; address of a = %u \n", a, &a);
    printf("b = %i; address of b = %u \n", b, &b);
    return EXIT_SUCCESS;
}
```

output:

```
a = 1; address of a = 65524
b = 2; address of b = 65522
```

& is called an address operator

%u is conversion specifier for an unsigned integer

Pointer Declaration

Pointer – a variable that contains the memory address of another variable.

A pointer must be defined to point to a specific **type** of variable.

An *int* pointer may not point to a *double* variable as an example.

Examples

Examples:

```
int a, b, *ptr;
```

```
float c, *fptr;
```

Reminders:

- * (asterisk) is called the *dereferencing* operator or *indirection* operator
- In a type declaration statement, the asterisk shows that the variable is being declared a pointer variable.

Example:

```
int a, b, *ptr;
```

a ? b ? ptr ? →

```
int a, b, *ptr;  
ptr = &a;
```

ptr → a ? b ?

Now ptr points to variable a

```
int a = 5, b = 9, *ptr = &a;
```

ptr → a 5 b 9

```
b = *ptr;
```

ptr → a 5 b 5

Take the value from the variable-pointer points to,
variable **a** which contains **5** and place it in the variable **b**

```
b = *ptr;
```

```
b = a;
```

 both do same thing

```
int a = 5, b = 9, *ptr = &a;
```

ptr → a 5

b 9

```
*ptr = b;
```

ptr → a 9

b 9

```
*ptr = b;
```

```
a = b;
```

accomplish the same thing

`ptr` - points to an address.

Ex: `int a, *ptr;`
`ptr = &a;`

`*ptr` - dereferences the pointer;
refers to the ***value*** in the address that `ptr` is
pointing to

Ex: `a = 5;`
`ptr = &a;`

The value in `*ptr` is 5

```
int main(void) {  
    int a = 1, b = 2, *A_ptr = &a;  
    printf("a = %i; address of a = %u \n", a, &a);  
    printf("b = %i; address of b = %u \n", b, &b);  
    printf("A_ptr = %u; address of A_ptr = %u \n", A_ptr, &A_ptr);  
    printf("A_ptr points to the value %i \n", *A_ptr);  
    return EXIT_SUCCESS;  
}
```

output:

a = 1; address of a = 65524
b = 2; address of b = 65522
A_ptr = 65524; address of A_ptr = 65520
A_ptr points to the value 1

Give a memory snapshot after this set of statements is executed

```
int a = 1, b = 2, *pointer;  
pointer = &b;
```

After the first line, the picture is:

```
int a = 1, b = 2, *pointer;
```

a 1

b 2

pointer → ?

After the second line of code, the picture is:

```
int a = 1, b = 2, *pointer;  
pointer = &b;
```

a 1 b 2 ←pointer

pointer contains the address of b

***pointer** contains the value of 2

Give a memory snapshot after this set of statements is executed

```
int a = 1, b = 2, *my_ptr = &b;
```

```
a = *my_ptr;
```

a 1 b 2 ← my_ptr /* after line 1 of code */

a 2 b 2 ← my_ptr /* after line 2 of code */

Give a memory snapshot after this set of statements is executed

```
int a = 1, b = 2, c = 5, *ptr = &c;  
b = *ptr;  
*ptr = a;
```

a 1 b 2 c 5 ← ptr /* after line 1 of code */

a 1 b 5 c 5 ← ptr /* after line 2 of code */

a 1 b 5 c 1 ← ptr /* after line 3 of code */

Give a memory snapshot after this set of statements is executed

```
int a = 1, b = 2, c = 5, *ptr;
```

```
ptr = &c;
```

```
c = b;
```

```
a = *ptr;
```

a 1 b 2 c 5 ptr → ?

/* after 1st line */

a 1 b 2 c 5 ← ptr

/* after 2nd line */

a 1 b 2 c 2 ← ptr

/* after 3rd line */

a 2 b 2 c 2 ← ptr

/* after 4th line */

A pointer can point to only one location,
but several pointers can point to the same location.

```
int x = -5, y = 8, *ptr_1, *ptr_2;  
ptr_1 = &x;  
ptr_2 = ptr_1;
```

x -5 y 8 ptr_1 → ? ptr_2 → ?

x -5 ← ptr_1 y 8 ptr_2 → ?

ptr_2 → x -5 ← ptr_1 y 8

FILE Pointers

File Pointer – a special pointer that holds the starting address of file.

```
FILE * sensor1;  
sensor1 = fopen("sensor1.dat", "r");
```

sensor1 is a pointer variable

```
fscanf(sensor1, "%f %f", &t, &motion);
```

Read data from the file pointed to by **sensor1**

Pointer Address Arithmetic

Pointer Address Arithmetic

- Arithmetic operations can be performed on pointers
 - Increment/decrement pointer (++ or --)
 - Add an integer to a pointer(+ or += , - or -=)
 - Pointers may be subtracted from each other
 - Operations meaningless unless performed on an array

Address Arithmetic #1:

A pointer can be assigned to another pointer of the same type.

```
int x, *p1, *p2;
```

```
p1 = &x;
```

```
p2 = p1;
```

Address Arithmetic #2:

An integer value can be added to or subtracted from a pointer.

`ptr++;` increments the pointer to point to the next value in memory;
only works correctly with arrays

Address Arithmetic #3:

A pointer can be assigned or compared to the integer zero, or equivalently, to symbolic constant NULL which is in <stdio.h>.

```
if (ptr == NULL)
{
    printf("Error \n");
}
```

Address Arithmetic #4:

Pointers to elements of the same array can be subtracted or compared.

```
ptr -= 3;
```

```
...
```

```
if (ptr < ptr + 1)
```

Common Errors

```
int y, *ptr1, *ptr2;
```

The following are all **invalid** statements:

`&y = ptr1;` attempts to change the address of y

`ptr2 = y;` attempts to change ptr2 to a non-address value

`*ptr1 = ptr2;` attempts to move an address to an integer variable

`ptr1 = *ptr2;` attempts to change ptr1 to a non-address value

It is not allowed to mix pointers of different types.

This shows an int with an int pointer,
and a float with a float pointer, using correct procedure.

```
int a, *ptr_a;
```

```
float b, *ptr_b;
```

Memory assignments for elements of **arrays** are guaranteed to be sequential.

We can use a pointer to reference each element of an array.

Assign a pointer to the first element of the array and then reference the elements of the array by incrementing or decrementing the pointer.

Examples:

```
int x[10], *ptr_x;
```

```
ptr_x = &x[0];
```

```
ptr_x++;      increment ptr_x to point to the next  
              value in memory
```

More examples:

```
int x[10], *ptr_x = &x[0];
```

`ptr_x += 1;` increment `ptr_x` to point to the next value in memory

`ptr_x = &x[1];` `ptr_x` is assigned the address of `x[1]`

`ptr_x += k;` `ptr_x` is assigned the address `k` values past the one it was pointing to

Give memory snapshots after this set of statements is executed.

```
double x = 15.6, y = 10.2, *ptr1 = &y, *ptr2 = &x;
```

x 15.6 ← ptr2

y 10.2 ← ptr1

```
*ptr1 = *ptr2 + x;
```

so $15.6 + 15.6 = 31.2$ hence

x 15.6 ← ptr2

y 31.2 ← ptr1

Give memory snapshots after this set of statements is executed.

```
int w = 10, x = 2, *ptr2 = &x;
```



```
*ptr2 -= w;
```

so $2 - 10 = -8$



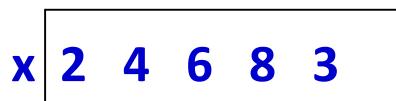
Give memory snapshots after this set of statements is executed.

```
int x[5] = {2, 4, 6, 8, 3};
```

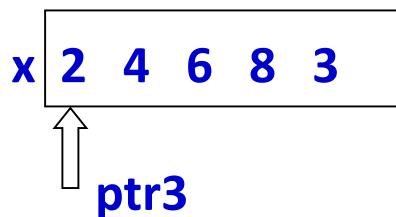
```
int *ptr1 = NULL, *ptr2 = NULL, *ptr3 = NULL;
```

```
ptr3 = &x[0];
```

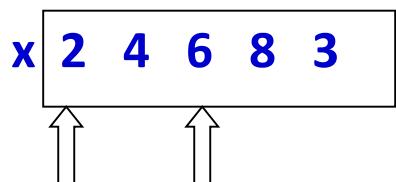
```
ptr1 = ptr2 = ptr3 + 2;
```



ptr1 → NULL ptr2 → NULL ptr3 → NULL



ptr1 → NULL ptr2 → NULL



ptr3 ptr2, ptr1

Give memory snapshots after this set of statements is executed.

```
int w[4], *first = NULL, *last = NULL;  
first = &w[0];  
last = first + 3;
```

w ? ? ? ?

first → NULL last → NULL

w ? ? ? ?

last → NULL

↑
first

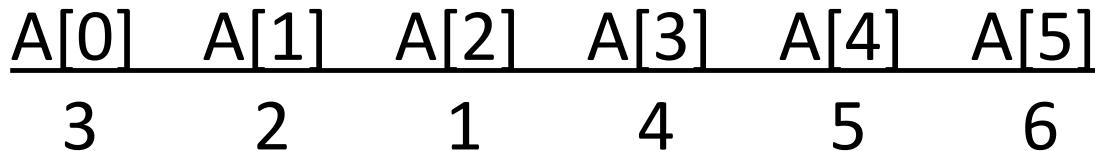
w ? ? ? ?

↑ ↑
first last

Pointers and Arrays

Pointers and Arrays

```
int A[6] = {3, 2, 1, 4, 5, 6}, *ptr;
```



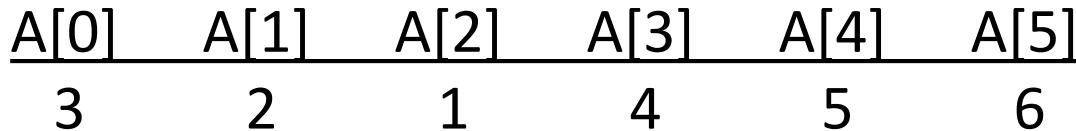
```
ptr = &A[0];
```

ptr + 2 refers to A[2]

ptr + 4 refers to A[4]

Pointers and Arrays

```
int A[6] = {3, 2, 1, 4, 5, 6}, *ptr=&A[0];
```



To sum the array:

```
sum = 0;  
for (k = 0; k < 6; k++)  
{  
    sum += A[k];  
}
```

Or -----

```
sum = 0;  
for (k = 0; k < 6; k++)  
{  
    sum += *(ptr + k);  
}
```

int g[] = {2, 4, 5, 8, 10, 32, 78};
0 1 2 3 4 5 6 → positions in array

int *ptr1 = &g[0];
int *ptr2 = &g[3];

What is the value of:

*g

```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};  
          0 1 2 3 4 5 6 → positions in array
```

```
int *ptr1 = &g[0];  
int *ptr2 = &g[3];
```

What is the value of:

***g**

2 = answer

The name of an array acts like a pointer to the beginning of the array when the array name is missing the brackets [].

```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};  
          0 1 2 3  4  5  6 → positions in  
                                array
```

```
int *ptr1 = &g[0];  
int *ptr2 = &g[3];
```

What is the value of: $*g + 1$

```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};  
          0 1 2 3  4  5  6 → positions in array
```

```
int *ptr1 = &g[0];  
int *ptr2 = &g[3];
```

What is the value of: $*g + 1$

3 = answer

Go to g, position zero.

Dereference getting the 2

Add 1 to the 2 and get 3



```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};
```

0 1 2 3 4 5 6 → positions in array

```
int *ptr1 = &g[0];
```

```
int *ptr2 = &g[3];
```

What is the value of: $*(g + 1)$



```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};
```

0 1 2 3 4 5 6 → positions in array

```
int *ptr1 = &g[0];
```

```
int *ptr2 = &g[3];
```

What is the value of: *(g + 1)

4 = answer.

Go to g, position zero.

Move over one address

Dereference and get the four.

```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};  
          0 1 2 3  4  5  6 → positions in  
                                array
```

```
int *ptr1 = &g[0];  
int *ptr2 = &g[3];
```

What is the value of: **$*(g + 5)$**

```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};  
          0 1 2 3  4  5  6 → positions in  
                                array
```

```
int *ptr1 = &g[0];  
int *ptr2 = &g[3];
```

What is the value of: ***(g + 5)**

32 = answer

Go to g position zero
Move over 5 address
Dereference and get the 32

↓ ↓

```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};  
          0 1 2 3  4  5  6 -> positions in array
```

```
int *ptr1 = &g[0];  
int *ptr2 = &g[3];
```

What is the value of: ***ptr1**

↓ ↓

```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};  
          0 1 2 3  4  5  6 -> positions in array
```

```
int *ptr1 = &g[0];  
int *ptr2 = &g[3];
```

What is the value of: ***ptr1**

2 = answer

Find what ptr1 points to
Dereference and get the 2

```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};  
          0 1 2 3  4  5  6 -> positions in array
```

```
int *ptr1 = &g[0];  
int *ptr2 = &g[3];
```

What is the value of: ***ptr2**

```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};  
          0 1 2 3  4  5  6 -> positions in array
```

```
int *ptr1 = &g[0];  
int *ptr2 = &g[3];
```

What is the value of: ***ptr2**

8 = answer

Find what ptr2 points to
Dereference and get the 8

↓ ↓

```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};  
          0 1 2 3  4  5  6 -> positions in array
```

```
int *ptr1 = &g[0];  
int *ptr2 = &g[3];
```

What is the value of: ***(ptr1 + 1)**

↓ ↓

```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};  
          0 1 2 3  4  5  6 -> positions in array
```

```
int *ptr1 = &g[0];  
int *ptr2 = &g[3];
```

What is the value of: ***(ptr1 + 1)**

4 = answer

Find what ptr1 points to (position zero)
Move over one address (position one)
Deference and get the 4

↓ ↓

```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};  
          0 1 2 3  4  5  6 -> positions in array
```

```
int *ptr1 = &g[0];  
int *ptr2 = &g[3];
```

What is the value of: ***(ptr2 + 2)**

↓ ↓

```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};  
          0 1 2 3  4  5  6 -> positions in array
```

```
int *ptr1 = &g[0];  
int *ptr2 = &g[3];
```

What is the value of: ***(ptr2 + 2)**

32 = answer

Find what ptr2 points to (position 3)
Move over 2 addresses (position 5)
Dereference and get the 32

↓ ↓

```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};  
          0 1 2 3  4  5  6 -> positions in array
```

```
int *ptr1 = &g[0];  
int *ptr2 = &g[3];
```

What is the value of: ***ptr2 + 10**

↓ ↓

```
int g[ ] = {2, 4, 5, 8, 10, 32, 78};  
          0 1 2 3  4  5  6 -> positions in array
```

```
int *ptr1 = &g[0];  
int *ptr2 = &g[3];
```

What is the value of: ***ptr2 + 10**

18 = answer

Find what ptr2 points to (position 3)
Dereference and get 8
Add 8 + 10 and get 18

Pointers and Functions

Pointers and Functions

Functions send arguments by *call-by-value*

The following exceptions use *call-by-address*:

Arrays – Address of array is passed to the function

Pointers – Address of variable, array, or string of characters is passed-to/returned-from a function

or

the pointer is used to step through an array

Example: a function to switch two values

```
void switch_it(int *a, int *b)
{
    int hold;
    hold = *a;
    *a = *b;
    *b = hold;
    return;
}
```

A valid call to this function would be:

```
int x, y;
switch_it(&x, &y);
```

Function Prototype is: **void switch_it(int *a, int *b);**

Below is a call to the switch_it function.
Is it a valid call?

```
float x = 1.5, y = 3.0, *ptr_x = &x, *ptr_y = &y;  
  
switch_it(ptr_x, ptr_y);
```

Will NOT work since x & y are *float*, but the function requires the incoming arguments to be *int*

Function Prototype is: **void switch_it(int *a, int *b);**

Below is a call to the switch_it function.
Is it a valid call?

```
int f = 2, g = 7, *ptr_f = &f, *ptr_g = &g;  
  
switch_it(ptr_f, ptr_g);
```

OK. All *int*. Passes in the addresses of f & g

Function Prototype is: **void switch_it(int *a, int *b);**

Below is a call to the `switch_it` function.
Is it a valid call?

```
int f = 2, g = 7, *ptr_f = &f, *ptr_g = &g;
```

```
switch_it(*ptr_f, *ptr_g);
```

No good! not passing the *address* of the f & g
but rather the *values* of 2 & 7

Function Prototype is: **void switch_it(int *a, int *b);**

Below is a call to the `switch_it` function.
Is it a valid call?

```
int f = 2, g = 7, *ptr_f = &f, *ptr_g = &g;
```

```
switch_it(&ptr_f, &ptr_g);
```

No good! Passing the addresses of the *pointers*
not the addresses of the *integers*.

Function Prototype is: **void switch_it(int *a, int *b);**

Below is a call to the switch_it function.
Is it a valid call?

```
int f = 2, g = 7, *ptr_f = &f, *ptr_g = &g;
```

```
switch_it(&f, &g);
```

OK. the addresses of f and g are being passed.

Function Prototype is: **void switch_it(int *a, int *b);**

Below is a call to the switch_it function.
Is it a valid call?

```
int f = 2, g = 7, *ptr_f = &f, *ptr_g = &g;
```

```
switch_it(f, g);
```

No good. This passing the *values* of f & g,
not the *addresses* of f & g.

Using the **const** Qualifier with Pointers

Using the **const** Qualifier with Pointers

- **const** – a keyword
- **const** qualifier
 - Variable cannot be changed
 - Use **const** if function does not need to change a variable
 - Attempting to change a **const** variable produces an error

Using the `const` Qualifier with Pointers. Examples.

```
int *const myPtr = &x;
```

Type `int *const` – constant pointer to an `int`

ERROR:

```
int *const myPtr = &x;
```

```
myPtr = &b;
```

because we are trying to change the address.

The `*const` freezes the pointer.

Using the **const** Qualifier with Pointers. Examples.

```
const int *myPtr = &x;
```

Regular pointer to a **const int**

ERROR:

```
const int *myPtr = &x;
```

```
*myPtr = 9;
```

because we are not allowed to change the value of x because the position of the * causes the value of x to freeze.

Using the **const** Qualifier with Pointers. Examples.

const int *const Ptr = &x;

const pointer to a const int

Nothing can be changed.

Function Pointers

What are function Pointers?

- C does not require that pointers only point to data, it is possible to have pointers to functions
- Functions occupy memory locations therefore every function has an address just like each variable
- Function pointers are different from regular pointers. They point to a function as opposed to a value. Hence they behave differently.

Why do we need function Pointers?

- Useful when alternative functions may be used to perform similar tasks on data (eg: sorting)
- One common use is in passing a function as a parameter in a function call.
- Can pass the data and the function to be used to some control function
- Greater flexibility and better code reuse

Define a Function Pointer

A function pointer is nothing else than a variable, it must be defined as usual.

```
int (*funcPointer) (int, char, int);
```

funcPointer is a pointer to a function.

The extra parentheses around (*funcPointer) is needed because there are precedence relationships in declaration just as there are in expressions

Assign an address to a Function Pointer

```
//assign an address to the function pointer
int (*funcPointer) (int, char, int);

int firstExample ( int a, char b, int c) {
    printf(" Welcome to the first example");
    return a+b+c;
}
funcPointer= firstExample;      //assignment of address of
                                //the function to a pointer
funcPointer=&firstExample;     //alternative using
                                //address operator
```

Calling a function using a Function Pointer

There are two alternatives

- 1) Use the name of the function pointer
- 2) Can explicitly dereference it

```
int (*funcPointer) (int, char, int);
```

```
// calling a function using function pointer  
int answer= funcPointer (7, 'A' , 2 );  
int answer=(* funcPointer) (7, 'A' , 2 );
```

Example Trigonometric Functions

```
// prints tables showing the values of cos,sin
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
void tabulate(double (*f)(double), double first, double last, double incr);
int main(void) {
    double final, increment, initial;
    printf ("Enter initial value: ");
    scanf ("%lf", &initial);
    printf ("Enter final value: ");
    scanf (%lf", &final);
    printf ("Enter increment : ");
    scanf (%lf", &increment);
    Printf("\n  x  cos(x) \n"
          " ----- ----- \n");
    tabulate(cos, initial,final,increment);
    Printf("\n  x  sin (x) \n"
          " ----- ----- \n");
    tabulate(sin, initial,final,increment);
    return (EXIT_SUCCESS);
}
```

The **main** function in little print.
Bigger print used in following slides.

Example Trigonometric Functions (1 of 4)

```
// prints tables showing the values of cos, sin

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

void tabulate(double (*f)(double), double first,
              double last, double incr);

int main(void)
```

Example Trigonometric Functions (2 of 4)

```
int main(void)
{
    double final, increment, initial;
    // Enter the data at the keyboard
    printf ("Enter initial value: ");
    scanf ("%lf", &initial);
    printf ("Enter final value: ");
    scanf (%lf", &final);
    printf ("Enter increment : ");
    scanf (%lf", &increment);
```

Example Trigonometric Functions (3 of 4)

```
// Print the headers and call tabulate
printf("\n  x  cos(x) \n"
      " ----- ----- \n");
tabulate(cos, initial,final,increment);

printf("\n  x  sin (x) \n"
      " ----- ----- \n");
tabulate(sin, initial,final,increment);
return (EXIT_SUCCESS);
}
```

Trigonometric Functions (4 of 4)

```
// when passed a pointer f, the function prints a table  
// showing the value of f
```

```
void tabulate(double (*f)(double), double first,  
                double last, double incr)  
{
```

```
    double x;  
    int i, num_intervals;  
    num_intervals = ceil( (last -first) /incr );  
    for (i=0; i<=num_intervals; i++) {  
        x= first +i * incr;  
        printf("%10.5f %10.5f\n", x , (*f) (x));  
    }  
}
```

Output of the Example

Enter initial value: 0

Enter final value: .5

Enter increment: .1

X	cos(x)
0.00000	1.00000
0.10000	0.99500
0.20000	0.98007
0.30000	0.95534
0.40000	0.92106
0.50000	0.87758

X	sin(x)
0.00000	0.00000
0.10000	0.09983
0.20000	0.19867
0.30000	0.29552
0.40000	0.38942
0.50000	0.47943

Another Common Use of FuncPtr

- Sorting function (**qsort**) where you pass in a pointer to a comparison function that will return the results of the comparison.
 - Ex: Which argument was larger.

C-7 Pointers

The End

C-8 Structures

Further information available
on Canvas under Reference

File = **structs.pdf**

Record: data *structure* that stores different types of data under a single variable name.

In C, we use the keyword ***struct*** to define records. They can contain *components* (also called *members* or *fields*) that can have different types.

Record

Buzz word. A *record* is one line in a data base.

Example of a Parts Warehouse:

For any given Part, there might be information stored about it in a **record** :

part number

its cost

amount in stock

location in warehouse

name of supplier

Reminder: what do we mean by “type”?

Examples of types we have used:

int

unsigned

float

double

char

So now we are moving on to creating our **own** types!

(and then, on to multi-part types.)

typedef - a mechanism which allows the programmer to explicitly associate a **type** with an **identifier**.

Example 1:

```
typedef int Length_t;
```

```
Length_t len, maxlen;
```

Example 2:

```
typedef int Inches_t, Feet_t;
```

```
Inches_t box_length, box_width;
```

```
Feet_t lot_width, lot_length;
```

Defining Structure Types

A new **type definition** can be defined for the structure, which can then be used to declare variables:

```
typedef struct
{
    int month;
    int day;
    int year;
} date_t;
```

```
date_t birth, current; /* This line declares two type
                           date_t variables */
```

The declaration should list each member on its own line, properly indented.

```
typedef struct
{
    int month;
    int day;
    int year;
} date_t;
```

Common Industry Practice:

The “_t” suffix is not required by C
but certainly makes it easier to keep a program readable.

It is very common in industry & that's what we will use in this class.
(Some companies use all caps.)

Initialization of structs:

Structures can be initialized when declared by putting the values, in the correct order, inside brackets { }.

```
date_t birth = {3, 13, 1989};
```

```
date_t current = {9, 23, 2017};
```

A structure can contain data items of different types:

/* First set up the structure */

```
typedef struct
{
    char    id[20];
    double  price;
    int     current_inv;
}auto_part_t;
```

/* This declares variable *part1* of type auto_part_t */

```
auto_part_t  part1;
```

/*These 3 lines initialize the parts of *part1* */

```
strcpy (part1.id , "A45X"); // string copy function
part1.price = 10.60;
part1.current_inv = 23;
```

Structures within Structures

```
typedef struct {  
    int month;  
    int day;  
    int year;  
} date_t;          /* This sets up the structure date_t */  
  
typedef struct {  
    char name[20];  
    date_t birth;  
} person_t;        /* This sets up the structure person_t */  
  
person_t person;  /* Initialize a variable person of type person_t */
```

To reference the data items in the structure:

- person.name
- person.birth.month
- person.birth.day
- person.birth.year

/ Or similarly create a person type */*

```
typedef struct {  
    char name[20];  
    date_t birth;  
} person_t;
```

/ Declare 2 variables of type person_t */*

```
person_t pers1, pers2;
```

/ To reference the data items pers1 & pers2 of type person_t */*

pers1.name

pers1.birth.month

pers1.birth.day

pers1.birth.year

pers2.name

pers2.birth.month

pers2.birth.day

pers2.birth.year

Arrays of Structures

Declaring an array of structures is the same as declaring an array of any other type of variable.

```
typedef struct
{
    int idnum;          /* employee id */
    char name[20];      /* employee name */
    float rate;         /* employee pay rate */
} pay_rec_t;           /* structure for one employee */
```

Define an array of 10 employees of type pay_rec_t:

```
pay_rec_t employee[10];
int        counter;
double     average;
```

```
/* A program using struct & typedef */
```

```
#include <stdio.h>
#include <stdlib.h>
#define MAXNAME 30
#define NUMRECS 5
```

```
typedef struct          /* a global type definition */
{
    long id;           /* employee id */
    char name[MAXNAME]; /* employee name */
    double rate;        /* employee pay rate */
} pay_rec_t;
...
```

```
int main(void)
{
    int j;
    pay_rec_t employee[NUMRECS] = {
        { 32479, "Abrams, B.", 6.72 },
        { 33623, "Bohm, P.", 7.54 },
        { 34145, "Donaldson, S.", 5.56 },
        { 35987, "Ernst, T", 5.43 },
        { 36203, "Gooding, K.", 8.73 }};

    for (j = 0; j < NUMRECS; j++)
    {
        printf("%li %s %4.2f \n", employee[j].id,
               employee[j].name, employee[j].rate);
    }
    return EXIT_SUCCESS;
}
```

The output would be:

32479 Abrams, B. 6.72

33623 Bohm, P. 7.54

34145 Donaldson, S. 5.56

35987 Ernst, T. 5.43

36203 Gooding, K. 8.73

Structures and Functions

Structures and Functions

```
typedef struct
{
    int hour, minute, second;
} time_t;
```

function prototype:

```
time_t new_time (time_t time_of_day, int elapsed_secs);
```

(example next slide)

First, an example:

Suppose the **current** time is 21:58:32 and **elapsed** time is 97 seconds.

What is the sum of the two times?

Call the function from **main**, which has the following declarations:

```
time_t time_now = {21, 58, 32};  
int secs = 97;
```

Similarly could assign values as follows:

```
time_t time_now;  
int secs;  
  
time_now.hour = 21;  
time_now.minute = 58;  
time_now.second = 32;  
secs = 97;  
  
time_now = new_time(time_now, secs);
```

new_time would return a value of 22:00:09

`/*Here is the function code that would work as in the previous example. */`

```
time_t new_time (time_t time_of_day, int elapsed_secs)
{
    int new_hr, new_min, new_sec;

    new_sec          = time_of_day.sec + elapsed_secs;
    time_of_day.sec = new_sec % 60;

    new_min          = time_of_day.min + new_sec / 60;
    time_of_day.min = new_min % 60;

    new_hr           = time_of_day.hr + new_min / 60;
    time_of_day.hr = new_hr % 24;

    return(time_of_day);
}
```

Structures as Function Arguments

Individual structure members may be passed to a function in the same manner as any scalar (or non-array) variable.

For example, given the structure definition:

```
typedef struct
{
    int id_num;
    double pay_rate;
    double hours;
} emp_t;
```

emp_t emp; /* declaration of a variable *emp* */

the statement

```
display(emp.id_num);
```

passes a copy of the structure member emp.id_num to a function called **display()**;

Similarly,

```
calc_pay(emp.pay_rate, emp.hours);
```

passes copies of *emp.pay_rate*
and *emp.hours*
to a function to calculate the amount of pay
owed the employee.

A copy of the complete structure can also be passed to a function:

```
calc_net(emp);
```

passes a copy of the entire *emp* structure to the function calc_net().

```
/* another example */
```

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

```
typedef struct          /* global type definition */
{
    int id_num;
    double pay_rate;
    double hours;
} employee_t;
```

```
double calc_net(employee_t temp); /* function prototype*/
```

```
...
```

```
...
int main(void)
{
    employee_t emp = {6782, 8.93, 40.5};
    double net_pay;
    net_pay = calc_net(emp);
    printf("The net pay for employee %i is $%6.2f \n\n",
           emp.id_num, net_pay);
    return EXIT_SUCCESS;
}
*-----*/
double calc_net(employee_t temp)
{
    return (temp.pay_rate * temp.hours);
}
*-----*/
```

The output is:

The net pay for employee 6782 is \$361.66

POINTERS AND STRUCTS

When we use *pointers* and *structs* together we use:

the *address operator* (`&`)

the *indirection operator* (`*`)

We also add a **new operator**, the structure pointer operator (`->`)
(a minus sign followed by a greater-than sign).

In general practice we use this new pointer operator
when we are using a pointer to values not in an array,
in place of the dot notation (`.`) that we have been using to get
into a *struct*.

The structure pointer operator is illustrated in the code that follows.

Passing a Structure to a Function as a Pointer

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

```
typedef struct          /* global type definition */
{
    int id_num;
    double pay_rate;
    double hours;
} employee_t;
```

```
double calc_net(employee_t *e); /* function prototype*/
```

...

```
int main(void)
{
    employee_t emp = {6782, 8.93, 40.5};
    double net_pay;

    net_pay = calc_net(&emp);
    printf("The net pay for employee %i is $%6.2f \n\n",
           emp.id_num, net_pay);
    return EXIT_SUCCESS;
}
/*-----*/
double calc_net(employee_t *e)
{
    return (e->pay_rate * e->hours);
}
/*-----*/
```

The output is:

The net pay for employee 6782 is \$361.66

Struct Return Types for Functions

```
#include <stdio.h>
#include <stdlib.h>
typedef struct      /* global type definition */
{
    int id_num;
    double pay_rate;
    double hours;
} employee_t;

// More on next slide
```

```
int main(void)
{
    employee_t emp;
    employee_t get_vals(void); /* function */
    emp = get_vals( );
    printf("The employee id number is %i \n",
           emp.id_num);
    printf("The employee pay rate is $%6.2f \n",
           emp.pay_rate);
    printf("The employee hours are %4.1f \n",
           emp.hours);
    return EXIT_SUCCESS;
}
```

```
/*-----*/
/* This function returns an employee structure */
employee_t get_vals (void)
{
    employee_t one;
    one.id_num  = 6789;
    one.pay_rate = 16.25;
    one.hours    = 40.0;
    return (one);
}
/*-----*/
```

Output:

The employee id number is 6789

The employee pay rate is \$16.25

The employee hours are 40.0

You can read more helpful material in **structs Information**
Which is on Canvas under Reference Materials.

It is a chapter that is eleven pages, and has very good
examples of *structs* and *typedef*.

Rule of Thumb:

If the struct comes into a function as an **array**,
use the *dot notation* (.).

If the struct comes into a function as a **pointer** with an *,
use the *points-into notation* (->).

Extra material on Enumeration Types

enum - keyword - used to declare enumeration types.
provides a means of naming a finite set,
and declaring identifiers as elements of the set.

Declare a type named *day*

```
enum day {sun, mon, tue, wed, thu, fri, sat};
```

Declare variables *d1* and *d2* of type enum day

```
enum day d1, d2;
```

```
d1 = fri; /* allowed */
```

```
if (d1 == d2)... /* allowed */
```

```
/* Compute the next day using function find_next_day */

enum day {sun, mon, tue, wed, thu, fri, sat};

typedef enum day day_t;

day_t find_next_day (day_t d)
{
    day_t next_day;
    switch (d) {
        case sun:
            next_day = mon;
            break;
        case mon:
            next_day = tue;
            break;
        ..... /* and so on */
        return next_day;
    }
}
```

```
/* A Second Version of the same function */  
/* Compute the next day using function find_next_day */
```

```
enum day {sun, mon, tue, wed, thu, fri, sat};
```

```
typedef enum day day_t;
```

```
day_t find_next_day (day d)  
{  
    return (day) (((int) d + 1) % 7);  
}
```

The values in **day** are constants of type *int*.
By default, the first item is assigned a zero,
and each succeeding one has the next integer value.

/*The default value of zero can be changed. */

```
enum day {sun=1, mon, tue, wed, thu, fri, sat };
```

/* Now sun starts as 1, and the group go from 1 to 7. */

Another example:

```
enum trees  
    { oak, maple, cherry, spruce, pine};
```

```
enum trees tree1;
```

Or combine the previous two lines into one line:

```
enum trees  
    { oak, maple, cherry, spruce, pine  
} tree1;
```

Same example, using `typedef`:

```
typedef enum
    { oak, maple, cherry, spruce, pine }
trees_t;
```

```
trees_t tree1;
```

C-8 Structures

The End

C-9 Bitwise Operators

Bit Manipulation in C



The C language has Bitwise Operators.

They allow us to manipulate bits.

May only be applied to **integers**:

- char
- short
- int
- long
- unsigned

Bitwise Operators:

Logical operators

- `~` bitwise *compliment* (unary)
- `&` bitwise *and*
- `^` bitwise *exclusive or*
- `|` bitwise *inclusive or*

Shift operators

- `<<` left shift
- `>>` right shift

Bitwise Complement ~

It inverts the bit string representation;
the 0s become 1s, and the 1s become 0s.

```
int a = 70707;
```

binary representation of **a** is:

```
00000000 00000001 00010100 00110011
```

the expression **~a** results in:

```
11111111 11111110 11101011 11001100
```

so the **int** value of the expression **~a** is -70708

Truth Table for Logical Bit Operators:

Values of:

a	b	a & b	a ^ b	a b
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Bitwise And &

The AND will return a 1 if both bits have a value of 1, else it returns a 0;

```
int a = 33333, b = -77777;
```

a → 00000000 00000000 10000010 00110101 → 33333

b → 11111111 11111110 11010000 00101111 → -77777

a & b → 00000000 00000000 10000000 00100101 → 32805

Masking, using the AND (1 of 2)

Suppose **a** is an unsigned integer variable whose value is 0x6db7.
Extract the **rightmost** 6 bits of the value using an AND (&).
Assign them to unsigned integer variable **b**.
Assign 0s to the 10 **leftmost** bits of **b**.

b = a & 0x3f;

	<i>Base 10</i>
a = 0110 1101 1011 0111	28087
mask = <u>0000 0000 0011 1111</u>	63
b = 0000 0000 0011 0111	55
= 0x37	

Masking, using the AND (2 of 2)

Suppose **a** is an unsigned integer variable whose value is 0x6db7.

Extract the **left**most 6 bits of the value using an AND (&).

Assign them to unsigned integer variable **b**.

Assign 0s to the 10 **right**most bits of **b**.

b = a & 0xfc00;

a = 0110 1101 1011 0111	
mask = <u>1111 1100 0000 0000</u>	
b = 0110 1100 0000 0000	
= 0x6c00	

Base 10

28087

-1024

27648

Bitwise EOR ^

The EOR will return a 1 if both bits have opposing values (1 and 0, or 0 and 1), else it returns a 0;

```
int a = 33333, b = -77777;
```

a → 00000000 00000000 10000010 00110101 → 33333

b → 11111111 11111110 11010000 00101111 → -77777

a ^ b → 11111111 11111110 01010010 00011010 → -110054

Masking, using the EOR (1 of 2)

Suppose **a** is an unsigned integer variable whose value is 0x6db7.
Let us reverse the **rightmost** 8 bits using an OR (^),
and preserve the **leftmost** 8 bits.

This new bit pattern will be assigned to the unsigned integer **b**.

b = **a** ^ 0xff;

	<i>Base 10</i>
a = 0110 1101 1011 0111	28087
mask = <u>0000 0000 1111 1111</u>	255
b = 0110 1101 0100 1000	27976
= 0x6d48	

Masking, using the EOR (2 of 2)

Suppose **a** is an unsigned integer variable whose value is 0x6db7. The expression **a ^ 0x4** will invert the value of the bit number 2 (the third bit from the right) within **a**.

If this operation is carried out repeatedly, the value of **a** will alternate between 0x6db7 and 0x6db3

Thus, using this operation repeatedly will toggle the third bit from the right on and off.

	<u>Base 10</u>
0x6db7 = 0110 1101 1011 0111	28087
mask = <u>0000 0000 0000 0100</u>	4
0x6db3 = 0110 1101 1011 0011	28083
mask = <u>0000 0000 0000 0100</u>	4
0x6db7 = 0110 1101 1011 0111	28087

Bitwise OR |

The EOR will return a 0 if both bits have a value of 0 , else it returns a 1;

```
int a = 33333, b = -77777;
```

a → 00000000 00000000 10000010 00110101 → 33333

b → 11111111 11111110 11010000 00101111 → -77777

a | b → 11111111 11111110 11010010 00111111 → -77249

Masking, using the OR | (1 of 2)

Suppose **a** is an unsigned integer variable whose value is 0x6db7. Transform the corresponding bit pattern into another bit pattern in which:

the **rightmost** 8 bits are all 1s, and

the **leftmost** 8 bits retain their original value.

Assign this new bit pattern to the unsigned integer **b**.

$b = a \mid 0xff;$

	<i>Base 10</i>
$a = 0110\ 1101\ 1011\ 0111$	28087
$\text{mask} = \underline{0000\ 0000\ 1111\ 1111}$	255
$b = 0110\ 1101\ 1111\ 1111$	28159
$= 0x6dff$	

Masking, using the OR | (2 of 2)

Suppose **a** is an unsigned integer variable whose value is 0x6db7. Transform the corresponding bit pattern into another bit pattern in which:

the **left**most 8 bits are all 1s, and

the **right**most 8 bits retain their original value.

Assign this new bit pattern to the unsigned integer **b**.

b = a | 0xff00; } Both accomplish the same thing.

b = a | ~0xff; } (2nd is independent of word size)

	<i>Base 10</i>
a = 0110 1101 1011 0111	28087
mask = <u>1111 1111 0000 0000</u>	-256
b = 1111 1111 1011 0111	-73
= 0ffb7	

Two Examples using both *Complement* and *OR*

```
int a = 33333, b = -77777;
```

a → 00000000 00000000 10000010 00110101 → 33333

b → 11111111 11111110 11010000 00101111 → -77777

$\sim(a \mid b)$ → 00000000 00000001 00100010 11010000 → 74448

$(\sim a \mid \sim b)$ → 11111111 11111111 11111111 11111010 → -6

Left Shift Operator:

Both operands must be integers of some sort.

expr1 << expr2

causes the bit representation of *expr1*
to be shifted to the left
by the number of places specified by *expr2*.

Left Shift Operator Examples: (1 of 2)

```
char c = 'Z';
```

<u>Expression</u>	<u>Representation</u>	<u>Action</u>
c	00000000 00000000 00000000 01011010	un-shifted
c << 1	00000000 00000000 00000000 10110100	left-shifted 1
c << 4	00000000 00000000 00000101 10100000	left-shifted 4
c << 31	00000000 00000000 00000000 00000000	left-shifted 31

Another Left Shift Operator Example: (1 of 2)

Suppose **a** is an unsigned integer variable whose value is 0x6db7.
The expression **b = a << 6;** will shift all bits
of variable **a** six places to the left and
assign the resulting bit pattern to the unsigned integer variable **b.**

lost bits

a = 0110 1101 1011 0111

shift left

a << 6 = 0110 1101 1100 0000 = 0x6dc0

filled with 0s

The leftmost 6 bits are lost.

The six rightmost bits are zero-filled.

Right Shift Operator:

Both operands must be integers of some sort.

expr1 >> expr2

Not symmetric to the left shift operator.

For *unsigned* integral expressions,
0s are shifted at the high end.

For *signed* types, some machines shift in 0s,
while others shift in sign bits.

The sign bit is the high-order bit;
It is 0 for nonnegative integers
It is 1 for negative integers

Right Shift Operator Examples: (1 of 2)

```
int      a = 1 >> 31; /* shift 1 to the high bit */  
unsigned b = 1 >> 31;
```

<u>Expression</u>	<u>Representation</u>	<u>Action</u>
a	00000000 00000000 00000000 01011010	unshifted
a >> 3	00000000 00000000 00000000 00001011	right-shifted 3
b	00000000 00000000 00000101 10100000	unshifted
b >> 3	00000000 00000000 00000000 10110100	right-shifted 3

Another Right Shift Operator Example: (2 of 2)

Suppose **a** is an unsigned integer variable whose value is 0x6db7. The expression **b = a >> 6;** will shift all bits of **a** six places to the right and assign the resulting bit pattern to the unsigned integer variable **b**.

lost bits
a = 0110 1101 1011 0111
shift right

a << 6 = 0000 0001 1011 0110 = 0x1b6
 filled with 0s

The rightmost 6 bits are lost.
The six leftmost bits are zero-filled.

```
#include <stdio.h>          // right_shift.c
#include <stdlib.h>
int main (void)
{
    unsigned a = 0xf05a;
    int b = a;

    printf("\nOriginal numbers: \n\n");
    printf("Unsigned %u. Integer %d.\n", a, b);
    printf("Both in Hex %x. %x.\n", a, b);

    printf("\nAfter the right shift: \n");
    printf("a in Hex %x.\n", a >> 6);
    printf("b in Hex %x.\n", b >> 6);
    return EXIT_SUCCESS;
}
/* the output on next page */
```

/* rightshift.c output*/

[bielr@athena ClassExamples]> **rightshift**

Original numbers:

Unsigned 61530. Integer 61530.

Both in Hex f05a. f05a.

After the right shift:

a in Hex 3c1.

b in Hex 3c1.

[bielr@athena ClassExamples]>

Just like we can do `a += 5;` or `a = a + 5;`
the same works for the bit operators.

```
unsigned a = 0x6db7;
```

Expression	Equivalent Expression	Final Value
<code>a &= 0x7f</code>	<code>a = a & 0x7f</code>	0x37
<code>a ~= 0x7f</code>	<code>a = a ~ 0x7f</code>	0x6dc8
<code>a = 0x7f</code>	<code>a = a 0x7f</code>	0x6dff
<code>a << 5</code>	<code>a = a << 5</code>	0xb6e0
<code>a >>= 5</code>	<code>a = a >> 5</code>	0x36d

sizeof()

- The **sizeof** unary operator is used to obtain the **size of** a variable or datatype
- Used in Lab 9
- Reminder: there are 8 bits in a byte.

```
/*-----(1 of 3)----*/
/* Your Name */
/* Lab 8      */
#include <stdio.h>
#include <stdlib.h>

/* Function Prototypes */
void bitprint (unsigned num);
int circular_shift(unsigned num, int n);
/*-----*/
int main (void)
{
    int left_count;
    unsigned num;          /* the starting number */
    unsigned shifted_num;
```

(2 of 3)

```
do {  
    /* read a unsigned integer */  
    printf("\n\nEnter an unsigned integer value (0 to stop): ");  
    scanf("%d", &num);  
  
    if (num != 0) {  
        printf("\n\nEnter an unsigned integer value for the left shift: ");  
        scanf("%d", &left_count);  
        printf("\n\nOriginal is %i \n\n", num);  
        bitprint(num);  
        shifted_num = circular_shift(num, left_count);  
        bitprint(shifted_num);  
        printf("Shifted it is %i \n", shifted_num);  
    } //end of if  
} while (num != 0); //end of do-while  
printf("\n\n");  
return EXIT_SUCCESS;  
}  
/*-----*/
```

```
void bitprint (unsigned num)
{
    unsigned mask;
    int bit, count, nbits;
    /* determine the word size in bits and set the initial mask */
    nbits = 8 * sizeof(int);           /* finds number of bytes in an unsigned
                                         and changes it to bits */
    mask = 0x1 << (nbits - 1);       /* place 1 in left most position
                                         starting place for the mask */
    for(count = 1; count <= nbits; count++)
    {
        bit = (num & mask) ? 1: 0;   /* set display bit on or off */
        printf("%x", bit);          /* print display bit */
        if(count %4 == 0)
            printf(" ");             /* blank space after every 4th digit */
        mask >>= 1;                 /* shift mask 1 position to the right */
    }
    printf("\n\n");
    return;
}
/*-----*/
```

C-9 Bitwise Operators

Bit Manipulation in C

THE END

C-10 Characters and Strings



REMINDER:

Characters and Integers are closely related.

Declaring Character Variables

1. Single Characters

Stored in binary

Declared as type *char*

EX:

```
char name, a1 = 'a';  
int n1, n2;
```

Declaring Character Variables

2. Character Strings:

Several characters stored in an array of char

Stored in binary

By definition, ends with NULL

Examples on next slide...

Declaring Character Strings. Examples:

```
char my_name[12 ] = “Ruthann”;
```

- C will recognize **my_name** as a string and add the NULLs at the end.
- Will consist of R,u,t,h,a,n,n,\0 ,\0 ,\0 ,\0
 - \0 is the same as **NULL**

```
char filename[ ] = “lab15.dat”;
```

will default to a length of 9 + 1 for NULL = 10

```
char name[30];
```



Now to deal with getting
characters in and out....I/O.

Some choice.

Choice 1:

We can use **printf** and **scanf** for character I/O utilizing the **%c** specifier.

```
char letter;  
scanf ("%c", &letter);  
printf("The letter is: %c \n", letter);
```

Choice 2, more common:

We can use special character functions called **getchar** and **putchar**:

getchar – reads a character from the keyboard and
returns an integer value

putchar – prints a character to the screen

The function prototypes for both of these functions
Included in *stdio.h*

Function prototypes:

```
int getchar(void);  
int putchar(int);
```

Example:

```
int c;  
c = getchar( );  
putchar(c);
```

End of File (EOF)

Defined in stdio.h:

#define EOF (-1)

/ This value is system dependent and will
vary from system to system */*

How to implement EOF at the keyboard:

In Unix/Linux environments, press:

control-d (^d)

It is configurable/changeable.

Now a program to

- read chars from keyboard
- echo back to screen.

It also counts number of chars entered

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int c, count = 0;
    printf("Enter character: (^d to quit) \n");
    c = getchar();
    count++;
    while (c != EOF)
    {
        putchar(c);
        c = getchar();
        count++;
    }
    printf("%i characters printed. \n", count - 1);
    return EXIT_SUCCESS;
}
```

Run the program:

Enter character: (^D to quit)

abcd

abcd 5 chars including newline

z

z 2 chars including newline

1w2e3r \$

1w2e3r \$ 9 chars including newline and space

^d 1 char

16 characters printed.

The 16 does not include the EOF marker/character.

Character I/O from Keyboard (review)

1. To read characters from the keyboard and write to screen:

```
c = getchar( );  
putchar(c);
```

Character I/O from Files

2. To read characters from a file and write to a file:

```
FILE *infile, *outfile;
```

```
infile = fopen("in.dat", "r");
outfile = fopen("out.dat", "w");
```

```
c = fgetc(infile);
```

```
fputc(outfile, c);
```

Character I/O from Files

3. To read chars from a file in a while loop:

```
while ((c = fgetc(infile)) != EOF)
{
    fputc(outfile, c);
}
```

```
/* This program counts words line by line */ Page 1 of 4  
/* count_words.c */
```

```
#include <stdio.h>  
#include <stdlib.h>  
#define FILENAME "Text1.dat"  
#define NEWLINE '\n'  
  
int main(void)  
{  
    int line[100], k = 0, count = 0;  
    FILE *text1;  
    int word_cnt(int x[ ], int npts);  
    text1 = fopen(FILENAME, "r");  
    /* omit error check to save room on slide */  
    ... /* more on next slide */
```

...

Page 2 of 4

```
while ((line[k] = fgetc(text1)) != EOF)
{
    if (line[k] == NEWLINE)
    {
        if (k != 0)
        {
            count += word_cnt(line, k);
        }
        k = 0;
    }
    else
        k++;
}
... /* and more on next slide */
```

```
...  
    if (k != 0)  
    {  
        count += word_cnt(line, k);  
    }  
    printf("\n%i words read. \n\n", count);  
    return EXIT_SUCCESS;  
}
```

The function **word_cnt** is on next slide.

```
/* Function to count number of words in an integer array */
int word_cnt(int x[ ], int npts) {
    int count = 0, k = 0;
    char space = ' ';
    while (k < npts)
    {
        while ((k < npts) && (x[k] == space)) {
            k++;
        }
        if (k < npts) {
            count++;
        }
        while ((k < npts) && (x[k] != space)) {
            k++;
        }
    }
    return count;
}
```

Character Functions in ctype.h

To use these functions, add to your code:

```
#include <ctype.h>
```

Sample of one function

`tolower(c)` returns a lower case letter

```
#include <ctype.h>
```

```
.....
```

```
int c;
```

```
while ((c = getchar() ) != EOF)
```

```
    putchar(tolower(c) );
```

```
.....
```

Various character functions all found in **ctype.h**:

`tolower(c)` returns a lower case letter

`toupper(c)` returns an upper case letter

`isdigit(c)` if (digit) return nonzero
 else return zero

`islower(c)` if (lower case) return nonzero
 else return zero

`isupper(c)` if (upper case) return nonzero
 else return zero

Various character functions all found in **ctype.h**:

`isalpha(c)` if (alphabetic) return nonzero
 else return zero

`isalnum(c)` if (alphanumeric) return nonzero
 else return zero

`iscntrl(c)` if (control char) return nonzero
 else return zero

`isgraph(c)` if (printable) return nonzero
 else return zero

`isprint(c)` if (printable or space) return nonzero
 else return zero

Various character functions all found in **ctype.h**:

`ispunct(c)` if (printable, but not space, letter, or
 digit) return nonzero

else return zero

`isspace(c)` if (white space: space, formfeed, nl,
 cr, horizontal or vertical tab)
 return nonzero

else return zero

`isxdigit(c)` if (hexadecimal digit: 0 1 2 3 4 5 6 7
 8 9 A B C D E F a b c d e f)
 return nonzero

else return zero

Character Arrays

Character Arrays

Character array – each letter stored as individual character element of the array

Character **string** – a character array where the last array element is NULL ('\0')

Character string constants are enclosed in **double quotes**. All three of these do the same thing:

```
char name[5] = "Jane";  
char name[ ] = "Jane";  
char name[ ] = {'J', 'a', 'n', 'e', '\0'};
```

We may get a **compilation error** if we try to put too many characters into an array of a defined length:

```
char name[6] = "Janice";
```

Some compilers give an error like this:

```
error C2117: 'Janice' : array bounds overflow
```

We need a space for the NULL, '\0', at the end.

Our compiler will NOT give an error, but the program may not run correctly.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char a[6]={"234"};
    /* first see what is in the array
       after initialization */

    for (int j=0; j < 6; j++)
    {
        if (a[j] == NULL)
            printf("%i Null\n", j);
        else
            printf("%i \"%c\"\n", j, a[j]);
    }
    printf("\n\n");

    return EXIT_SUCCESS;
}
```

Output:

0	"2"
1	"3"
2	"4"
3	Null
4	Null
5	Null

```
/* Now change one value and look  
   again to see what is there */  
  
a[4] = '9';  
for ( j=0; j < 6; j++)  
{  
    if (a[j] == NULL)  
        printf("%i Null\n", j);  
    else  
        printf("%i \"%c\"\n", j, a[j]);  
}  
return EXIT_SUCCESS;  
}/*-----*/
```

Output:

0	"2"
1	"3"
2	"4"
3	Null
4	"9"
5	Null

Output as it appeared on the screen:

```
/*-----*/
```

```
0 "2"  
1 "3"  
2 "4"  
3 Null  
4 Null  
5 Null
```

```
0 "2"  
1 "3"  
2 "4"  
3 Null  
4 "9"  
5 Null
```



C library string functions found
in `string.h`:

To use these functions, add to your code:

```
#include <string.h>
```

```
size_t strlen(s); // returns length of string s  
// that is filled with values.
```

```
/* ----- */
```

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
int main(void)  
{  
    char a[6] = {"234"};  
    int len;  
    len = strlen(a);  
    printf("\nLength = %i\n", len);  
    return EXIT_SUCCESS;
```

```
}
```

```
/* ----- */
```

Length = 3

char *strcpy(s, t); copies string t onto and over string s

```
/*-----*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char a[10] = {"234"};
    char b[10] = {"ABC"};
    strcpy(a, b);
    printf("\nNew String A = %s\n", a);
    return EXIT_SUCCESS;
}
/*-----*/
```

New String A = ABC

char *strncpy(s, t, n); copies n characters of string t to s

```
/*-----*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int main(void)
```

```
{
```

```
    char a[10] = {"234"};
```

```
    char b[10] = {"ABC"};
```

```
strncpy(a, b, 2);
```

```
    printf("\nNew String A = %s\n", a);
```

```
    return EXIT_SUCCESS;
```

```
}
```

```
/*-----*/
```

New String A = AB4

char *strcat(s, t); concatenates string t to the end of s
(concatenates = to paste on end)

/*-----*/

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int main(void)
```

```
{
```

```
    char a[10] = {"234"};
```

```
    char b[10] = {"ABC"};
```

```
    strcat(a, b);
```

```
    printf("\nNew String A = %s\n", a);
```

```
    return EXIT_SUCCESS;
```

```
}
```

/*-----*/

New String A = 234ABC

char *strncat(s, t, n); concatenates n chars of t to end of s

```
/*-----*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int main(void)
```

```
{
```

```
    char a[10] = {"234"};
```

```
    char b[10] = {"ABC"};
```

```
    strncat(a, b, 2);
```

```
    printf("\nNew String A = %s\n", a);
```

```
    return EXIT_SUCCESS;
```

```
}
```

```
/*-----*/
```

New String A = 234AB

int strcmp(s, t); compares strings s and t;

returns:

a negative value, if **s < t**,

zero, if **s == t**,

a positive value, if **s > t**

This function will be used in the mini-shell program.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    int answer1, answer2;
    int answer3, answer4;

    char a[10] = {"234"};
    char b[10] = {"ABC"};
    char c[10] = {"ABC"};
    char d[10] = {"abc"};
    char e[10] = {"cba"};

answer1 = strcmp(a, b);
answer2 = strcmp(b, c);
answer3 = strcmp(c, d);
answer4 = strcmp(d, a);
answer5 = strcmp(d, e);
```

```
printf("\n ab = %3i"
      "\n bc = %3i"
      "\n cd = %3i"
      "\n ad = %3i\n",
      "\n de = %3i\n",
      answer1, answer2,
      answer3, answer4,
      answer5);
return EXIT_SUCCESS;
}
```

Output:

```
ab = -1
bc = 0
cd = -1
ad = 1
de = -1
```

int strncmp(s, t, n);

compares at most n chars of s to t;

returns:

a negative value, if $s < t$,

zero, if $s == t$,

a positive value, if $s > t$

```

/*-----*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    int answer1, answer2;
    int answer3, answer4;

    char a[10] = {"234"};
    char b[10] = {"ABC"};
    char c[10] = {"ABC"};
    char d[10] = {"abc"};

answer1 = strncmp(a, b, 1);
answer2 = strncmp(b, c, 2);
answer3 = strncmp(c, d, 2);
answer4 = strncmp(d, a, 2);

```

```

printf("\n ab = %3i"
      "\n bc = %3i"
      "\n cd = %3i"
      "\n ad = %3i\n",
      answer1, answer2,
      answer3, answer4);
return EXIT_SUCCESS;
}
/*-----*/

```

Output:

```

ab = -1
bc =  0
cd = -1
da =  1

```

More String Functions

Using Pointer Notation



```
char *strchr(s, c);
```

returns a pointer to the first occurrence
of character c in the string s;

returns NULL if c does not occur in s

```
/*-----*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *where_ptr;

    char a[10] = {"234"};
    where_ptr = strchr(a, '3');

    printf("\nDereference of where_ptr = %c\n",
           *where_ptr);
    return EXIT_SUCCESS;
}
/*-----*/
```

Dereference of where_ptr = 3



```
char * strrchr(s, c);
```

returns a pointer to the last occurrence
of character c in the string s;

returns NULL if c does not occur in s.

```
/*-----*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *where_ptr;
    char a[10] = {"234234"};

where_ptr = strrchr(a, '3');
    printf("\nDereference of where_ptr = %c\n",
           *where_ptr);
    printf("Address of a[1]      = %u\n"
           "Address of a[4]      = %u\n"
           "Address in where_ptr = %u\n",
           &a[1], &a[4], where_ptr);
    return EXIT_SUCCESS;
}
/*-----*/
```

Output:

Dereference of where_ptr = 3
Address of a[1] = 1245041
Address of a[4] = 1245044
Address in where_ptr = 1245044

C library string functions found in **string.h**:

char *strstr(s, t);

returns a pointer to the start of the
string **t** within the string **s**;

returns NULL if **t** does not occur in **s**

```

/*
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *where_ptr;
    char a[10] = {"234234"};
where_ptr = strstr(a, "34");

printf("\nDereference of where_ptr = %c\n",
       *where_ptr);
printf("Address of a[1]      = %u\n"
       "Address of a[4]      = %u\n"
       "Address in where_ptr = %u\n",
       &a[1], &a[4], where_ptr);
    return EXIT_SUCCESS;
}
/*

```

Output:

Dereference of where_ptr = 3	
Address of a[1] = 1245041	
Address of a[4] = 1245044	
Address in where_ptr = 1245041	



The next two functions

- These next functions both search a null-terminated string **a** for occurrences of characters specified by whether they are include in a second null-terminated string **set**.

```
size_t strspn(s, set);
```

It searches the string **s** for the first occurrence
of a character that is **not** included in the string **set**

The value returned is the length of the longest initial segment
of **s** that consists of characters found in **set**.

If every character of **s** appears in **set**,
then the total length of **s** (not counting the terminating null
character) is returned.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    int set_len1, set_len2, set_len3;
    char a[10] = {"abcde"};
    char b[10] = {"bbcce"};
    char c[10] = {"cbbcca"};

set_len1 = strspn(a, "bc");
set_len2 = strspn(b, "bc");
set_len3 = strspn(c, "bc");

    printf("set_len1 = %i\n"
           "set_len2 = %i\n"
           "set_len3= %i\n",
           set_len1, set_len2, set_len3);
    return EXIT_SUCCESS;
}
```

Output:

```
set_len1 = 0
set_len2 = 4
set_len3 = 5
```



```
size_t strcspn(s, set);
```

Similar to **strspn** except that it searches **s** for the first occurrence of a character that is included in the string **set**, skipping over characters that are not in **set**.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{
    int set_len1, set_len2, set_len3;
    char a[10] = {"abcde"};
    char b[10] = {"bbcce"};
    char c[10] = {"asdcbbcca"};

set_len1 = strcspn(a, "bc");
set_len2 = strcspn(b, "bc");
set_len3 = strcspn(c, "bc");

printf("set_len1 = %i\n"
       "set_len2 = %i\n"
       "set_len3 = %i",
       set_len1, set_len2, set_len3);
return EXIT_SUCCESS;
}
```

Output:

```
set_len1 = 1
set_len2 = 0
set_len3 = 3
```



```
char *strupr(s, t);
```

returns a pointer to the first occurrence in string s
of any character of string t;

returns NULL if none of the characters in t occur in s

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *ptr;
    char a[10] = {"abcde"};

ptr = strpbrk(a, "cd");

printf("*pos1 = %c\n"
      "Address of a[2] = %u\n"
      "Address in ptr = %u",
      *ptr, &a[2], ptr);

return EXIT_SUCCESS;
}
```

Output:

```
*pos1 = c
Address of a[2] = 1245042
Address in ptr = 1245042
```



```
char *strtok(s, ct);
```

It extracts tokens from strings.

Searches string **s** for tokens delimited by characters from **ct**.

?returns NULL if none of the characters in t occur in s

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *ptr;
    char s[15] = "abc,de,fghi";
ptr = strtok(s, ",");
    while(ptr != NULL)
    {
        printf("Token = %s\n", ptr);
ptr = strtok(NULL, ",");
    }
    return EXIT_SUCCESS;
}
```

In the loop, the call to strtok starts with a Null. This has strtok start from where it left off in the string, instead of repeatedly starting at the beginning.

Output:

Token = abc
Token = de
Token = fghi

ASCII Character Codes

<u>Dec</u>	<u>Oct</u>	<u>Hex</u>	<u>Chr</u>
000	000	00	NULL
001	001	01	SOH
002	002	02	STX, Start TX
003	003	03	ETX, End TX
004	004	04	EOT
005	005	05	ENQ, Inquire
006	006	06	ACK, Acknowledge
007	007	07	BEL, Bell
008	010	08	BS, Back Space
009	011	09	HT, Horizontal Tab
010	012	0A	LF, New Line(Line Feed)
011	013	0B	VT, Vertical Tab
012	014	0C	FF, Form Feed
013	015	0D	CR, Carriage Return

<u>Dec</u>	<u>Oct</u>	<u>Hex</u>	<u>Chr</u>
014	016	0E	SO, Stand Out
015	017	0F	SI, Stand In
016	020	10	DLE
017	021	11	DC1
018	022	12	DC2
019	023	13	DC3
020	024	14	DC4
021	025	15	NAK, Negative ACK
022	026	16	SYN
023	027	17	ETB
024	030	18	CAN
025	031	19	EM
026	032	1A	SUB
027	033	1B	ESC, Escape
028	034	1C	FS, Cursor Right
029	035	1D	GS, Cursor Left
030	036	1E	RS, Cursor Up
031	037	1F	US, Cursor Down
032	040	20	SP, Space

<u>Dec</u>	<u>Oct</u>	<u>Hex</u>	<u>Chr</u>
033	041	21	!
034	042	22	"
035	043	23	#
036	044	24	\$
037	045	25	%
038	046	26	&
039	047	27	'
040	050	28	(
041	051	29)
042	052	2A	*
043	053	2B	+
044	054	2C	,
045	055	2D	-
046	056	2E	,
047	057	2F	/

<u>Dec</u>	<u>Oct</u>	<u>Hex</u>	<u>Chr</u>
048	060	30	0
049	061	31	1
050	062	32	2
051	063	33	3
052	064	34	4
053	065	35	5
054	066	36	6
055	067	37	7
056	070	38	8
057	071	39	9
058	072	3A	:
059	073	3B	;
060	074	3C	<
061	156	66	n
062	076	3E	>
063	077	3F	?
064	100	40	@

<u>Dec</u>	<u>Oct</u>	<u>Hex</u>	<u>Chr</u>
065	101	41	A
066	102	42	B
067	103	43	C
068	104	44	D
069	105	45	E
070	106	46	F
071	107	47	G
072	110	48	H
073	111	49	I
074	112	4A	J
075	113	4B	K
076	114	4C	L
077	115	4D	M
078	116	4E	N
079	117	4F	O
080	120	50	P

<u>Dec</u>	<u>Oct</u>	<u>Hex</u>	<u>Chr</u>
081	121	51	Q
082	122	52	R
083	123	53	S
084	124	54	T
085	125	55	U
086	126	56	V
087	127	57	W
088	130	58	X
089	131	59	Y
090	132	5A	Z
091	133	5B	[
092	134	5C	\
093	135	5D]
094	136	5E	^
095	137	5F	_
096	140	60	`

<u>Dec</u>	<u>Oct</u>	<u>Hex</u>	<u>Chr</u>
097	141	61	a
098	142	62	b
099	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q

<u>Dec</u>	<u>Oct</u>	<u>Hex</u>	<u>Chr</u>
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	
125	175	7D	}
126	176	7E	~
127	177	7F	DEL, Delete

C-10 Characters and Strings

The End