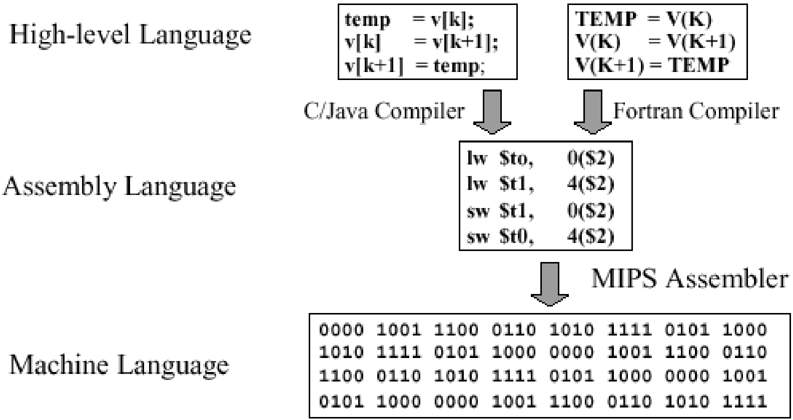
CSC209: Software Tools and Systems Programming

Week 2: C, Unix, and Makefile[1]

Kianoosh Abbasi

[1] Slides are mostly taken from Andi Bergen's in summer 2021.

Assembly and Machine Code



High-level Language

```
temp    = v[k];
v[k]    = v[k+1];
v[k+1]  = temp;
```

```
TEMP  = V(K)
V(K)    = V(K+1)
V(K+1) = TEMP
```

C/Java Compiler

Fortran Compiler

Assembly Language

```
lw  $to,    0($2)
lw  $t1,    4($2)
sw  $t1,    0($2)
sw  $t0,    4($2)
```

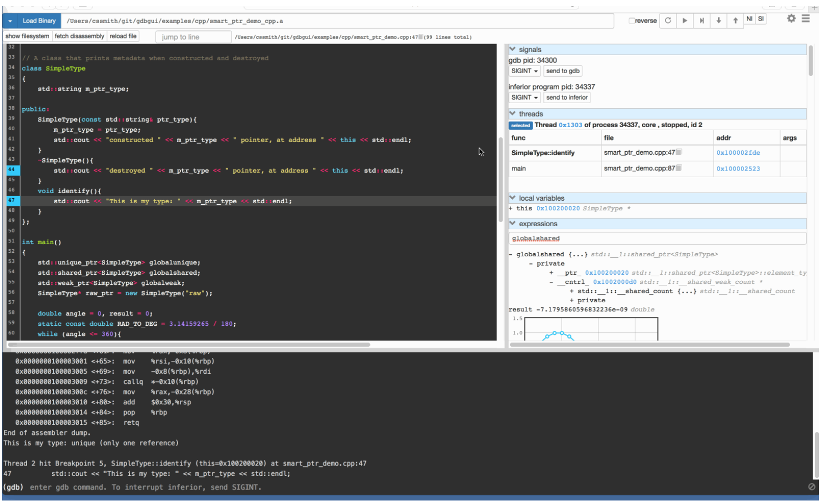
MIPS Assembler

Machine Language

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

PCRS C Visualizer

- More up-to-date C visualizer
- Investing time to learn gdb will pay off handsomely
- gdbgui is installed on lab PCs: very powerful for generating visualizations



/Users/cssmith/git/gdbgui/examples/cpp/smart_ptr_demo_cpp.a

/Users/cssmith/git/gdbgui/examples/cpp/smart_ptr_demo.cpp:47 (99 lines total)

```
// A class that prints metadata when constructed and destroyed
class SimpleType
{
    std::string m_ptr_type;

public:
    SimpleType(const std::string& ptr_type){
        m_ptr_type = ptr_type;
        std::cout << "constructed " << m_ptr_type << " pointer, at address " << this << std::endl;
    }
    ~SimpleType(){
        std::cout << "destroyed " << m_ptr_type << " pointer, at address " << this << std::endl;
    }
    void identify(){
        std::cout << "This is my type: " << m_ptr_type << std::endl;
    }
};

int main()
{
    std::unique_ptr<SimpleType> globalunique;
    std::shared_ptr<SimpleType> globalshared;
    std::weak_ptr<SimpleType> globalweak;
    SimpleType* raw_ptr = new SimpleType("raw");

    double angle = 0, result = 0;
    static const double RAD_TO_DEG = 3.14159265 / 180;
    while (angle <= 360){
```

gdb pid: 34300

inferior program pid: 34337

Thread 0x1303 of process 34337, core , stopped, id 2

func	file	addr	args
SimpleType::identify	smart_ptr_demo.cpp:47	0x100002fde	
main	smart_ptr_demo.cpp:87	0x100002523	

```
+ this 0x100200020 SimpleType *
```

globalshared

```
- globalshared {...} std::__1::shared_ptr<SimpleType>
    - private
        + __ptr_ 0x100200020 std::__1::shared_ptr<SimpleType>::element_ty
        - __cntrl_ 0x1002000d0 std::__1::__shared_weak_count *
            + std::__1::__shared_count {...} std::__1::__shared_count
            + private
result -7.1795860596832236e-09 double
```

```
   0x0000000100003001 <+65>:  mov    %rsi,-0x10(%rbp)
   0x0000000100003005 <+69>:  mov    -0x8(%rbp),%rdi
   0x0000000100003009 <+73>:  callq  *-0x10(%rbp)
   0x000000010000300c <+76>:  mov    %rax,-0x28(%rbp)
   0x0000000100003010 <+80>:  add    $0x30,%rsp
   0x0000000100003014 <+84>:  pop    %rbp
   0x0000000100003015 <+85>:  retq
End of assembler dump.
This is my type: unique (only one reference)

Thread 2 hit Breakpoint 5, SimpleType::identify (this=0x100200020) at smart_ptr_demo.cpp:47
47          std::cout << "This is my type: " << m_ptr_type << std::endl;
(gdb) enter gdb command. To interrupt inferior, send SIGINT.
```

Programming in C: Return Values

```
while (scanf(...) != EOF) { ... }
```

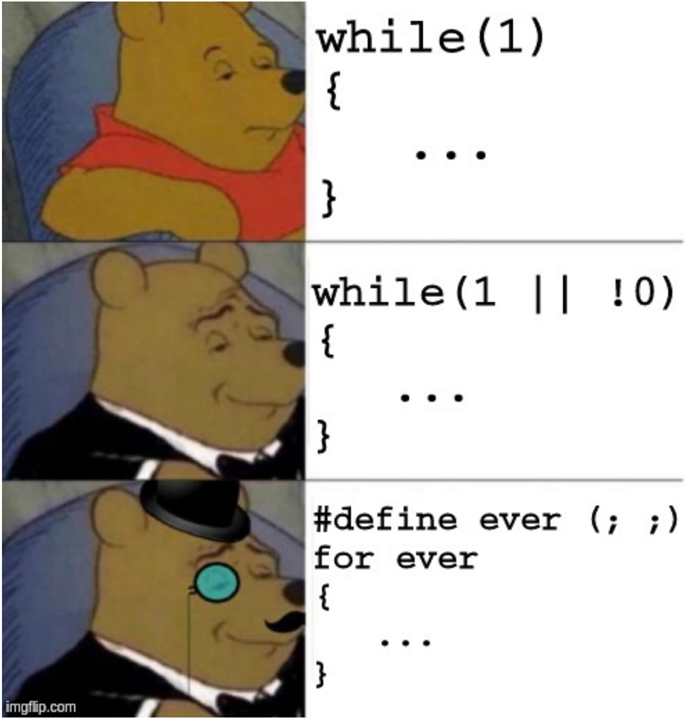
- Almost every library call has a return value
- Always check return values
  - C does not throw exceptions like Java or Python
  - Rightfully be paranoid about whether or not each library call completes successfully

  What does the above code do? Check `man 3 scanf` and scroll to `RETURN VALUE`

Programming in C: Macros

- Return values are often defined as *macros*, e.g., `EOF`
  - These typically “expand” to integer constants
  - Typically defined in `.h` files
  - Already saw an example of this in PCRS:

```
#define DAYS 365
```



```
while(1)
 {
     ...
 }
```

```
while(1 || !0)
 {
     ...
 }
```

```
#define ever (; ;)
for ever
{
    ...
}
```

Compiler Warnings (and Errors) are Your Friends

Common `gcc` compiler flags (all explained in `man gdb`):

- `-g`: Include debugging symbols in compiled program (`gdb` and `valgrind` make use of these)
- `-Wall`: Warn about highly-questionable code
- `-Wextra`: More warnings (sometimes helpful)
- `-Wpedantic`: All possible warnings
- `-Werror`: Treat all warnings as errors
  *Your assignments **must** compile with `-Wall` and `-Werror`*

C: Memory (un)Safety

- C assumes that you know what you're doing
  - A perilous assumption: 70% of security vulnerabilities in Microsoft products are due to **avoidable mistakes that C/C++ allow you to make**
- Example of unsafe code that will compile and run:

```
int arr[10];
arr[-1] = 123;
```

Use gcc flag `-fsanitize=address` to catch memory safety bugs

C: Undefined Behaviour

- *Undefined behaviour* is any operation for which the C standard imposes no requirements
- Example: The contents of uninitialized variables are **undefined**
    - The following code will likely print **garbage values**, but **it will compile and run regardless**:

```c
int a;
printf("%d", a);
```

Use `valgrind` to detect reads on uninitialized variables

Compiling C Programs

- C programs can consist of multiple `.c` files
- Each individual `.c` file can be compiled to an *object file*
- Object files contain “placeholders” for addresses of functions that were *declared* but not *defined*
  - Header (`.h`) files ensure consistency between function declarations across your program’s multiple source files
- The *linker* connects object files together to create an *executable file*

Makefiles (just for reference)

- Makefiles facilitate *building* (i.e., compiling, linking, sometimes testing and packaging) projects consisting of multiple source files
- If only one source file has changed, no need to recompile everything; instead:
  1. Recompile source files that have changed
  2. Relink updated object files to generate new executable file
- **Makefile slides are for reference. You might need to use them in assignments and/or PCRS, but they will NOT be asked at exams.**

Makefile format

A Makefile contains a sequence of *rules*, each in the format:

```
target: prereq_1 prereq_2 ... prereq_n
    action_1
    ...
    action_n
```

Using make

- Makefiles are processed by the `make` program
- Run `make` with no arguments to evaluate first rule
- Run `make TARGET` to execute action(s) defined in rule for `TARGET`
    - Only if `TARGET` prerequisites were modified since last time that `make TARGET` was run
- To force `make TARGET` to recompile code, you can:
    - Update last modified time of prerequisite source files, with `touch`, or
    - Delete prerequisite object files

Makefile Syntax: Defining Variables

You may define variables; e.g., to store compiler flags:

```
CFLAGS= -g -Wall -Werror -fsanitize=address

reverse : reverse.c
    gcc $(CFLAGS) -o reverse reverse.c
```

Makefile Syntax: Automatic (Built-In) Variables

Variable	Meaning
`$@`	Target
`$<`	First prerequisite
`$?`	All out of date prerequisites
`$^`	All prerequisites

```
CFLAGS= -g -Wall -Werror -fsanitize=address

hello: hello.c hello.h
	gcc $(CFLAGS) -o $@ $<
```

Ref.: 10.5.3: Automatic Variables, GNU Make manual

Makefile Example (Assignment 1)

```
FLAGS= -Wall -Werror -fsanitize=address -g
OBJ = simfs.o initfs.o printfs.o simfs_ops.o
DEPENDENCIES = simfs.h simfstypes.h

all : simfs

simfs : ${OBJ}
    gcc ${FLAGS} -o $@ $^

%.o : %.c ${DEPENDENCIES}
    gcc ${FLAGS} -c $<

clean :
    rm -f *.o simfs
```

Makefile Example: Pattern Rules

```
%.o : %.c ${DEPENDENCIES}
	gcc ${FLAGS} -c $<
```

- Most files are compiled in the same way, so we write a pattern rule for the general case
- % expands to the stem of the file name (i.e., without extension)
- `gcc -c` compiles the source file(s), but does not link

Makefile Example: Phony Targets

You may want a command that builds a target:

```
OBJ = simfs.o initfs.o printfs.o simfs_ops.o

simfs: ${OBJ}
    gcc ${FLAGS} -o $@ $^
```

Or a target that doesn't build anything:

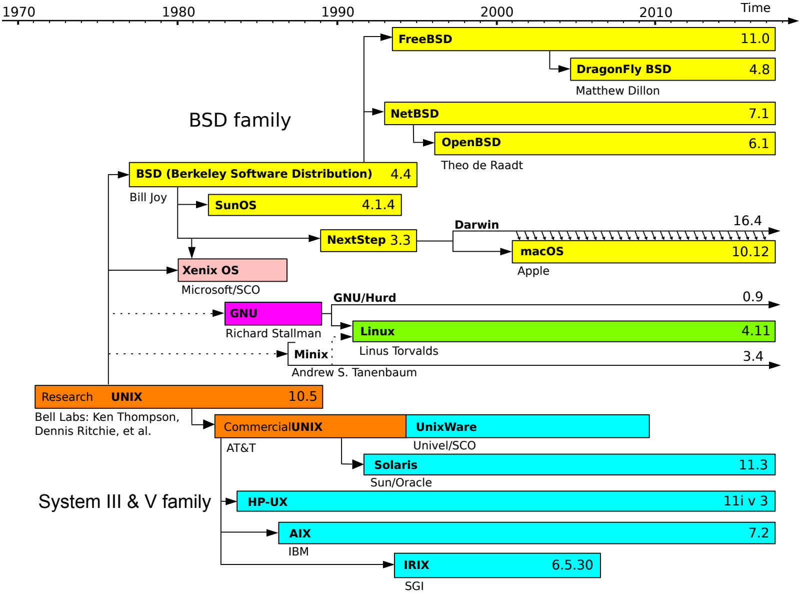
```
clean:
    rm -f *.o simfs
```

Unix and Linux

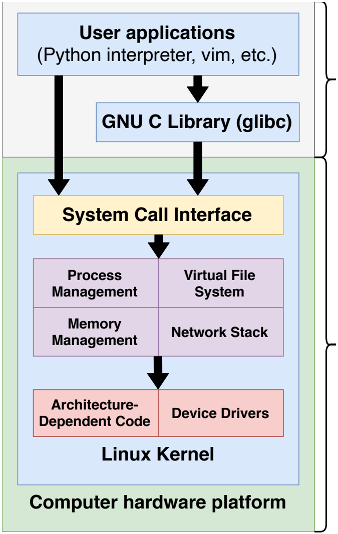
UNIX vs. Linux vs. UNIX-like

- UNIX is a proprietary OS developed by AT&T in 1969
- Free and commercial imitations followed, such as BSD, Linux, Solaris
  - The macOS kernel is a BSD derivative
- We say UNIX to refer to *UNIX-like* OSs, often colloquially called *nix
- Linux is the most widely-used UNIX-like OS: It runs on all kinds of devices, e.g., PCs, smartphones, printers, security cameras, wireless routers. . .

The UNIX Timeline



GNU/Linux: User Space vs. Kernel Space



User applications
(Python interpreter, vim, etc.)

GNU C Library (glibc)

System Call Interface

Process Management

Virtual File System

Memory Management

Network Stack

Architecture-Dependent Code

Device Drivers

Linux Kernel

Computer hardware platform

User space (CSC209)

Kernel space (CSC369)

The UNIX Philosophy

Brief summary of the UNIX philosophy, from A Quarter-Century of UNIX by P. H. Salus, 1994:

- Write programs that do one thing and do it well
- Write programs to work together
  - Expect that the output from your program will be used as input for another (e.g., by piping)
  - Don't require interactive input
- Write programs that handle text streams, because that is a universal interface

Common UNIX Tools/Commands and Abstractions

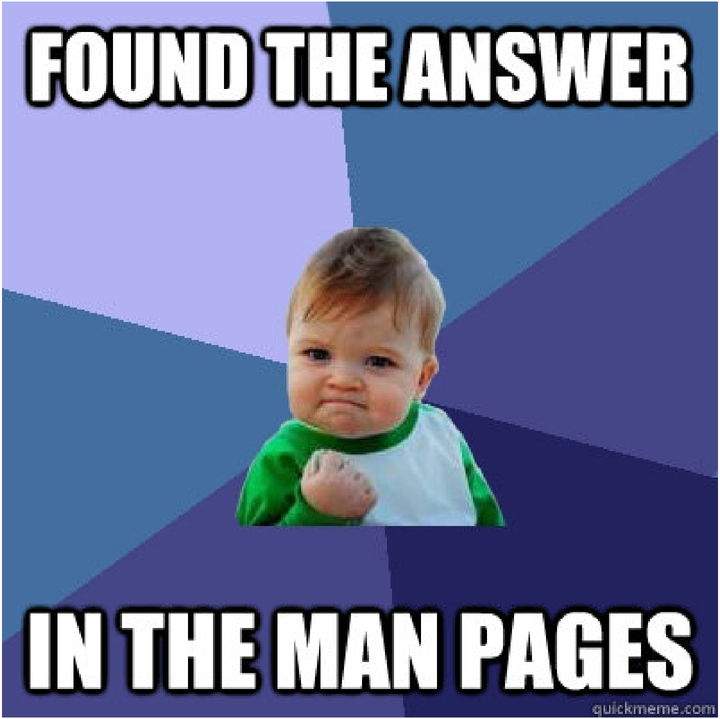
File/directory operations	Text filtering	System Information	Input/Output Abstractions
`cd`, `ls`	`head`	`who`, `last`	stdin
`mkdir`	`tail`	`free`	stdout
`touch`	`sort`	`ps`	stderr
`cp`, `mv`, `rm`	`grep`	`top`	pipes/fifos
`cat`, `diff`	`tr`, `wc`	`type`	sockets

Look these up in the man pages for practice!

How to Learn Linux

Use it.

- Don't worry about memorizing stuff
- Work on your task(s) at hand, look things up as needed
  - Man pages: Comprehensive documentation
  - Arch Wiki: Community-maintained tutorials
- Common tasks will quickly start to become familiar
- A key outcome of your CS degree: Being able to quickly locate the required information to learn new concepts on your own


FOUND THE ANSWER
IN THE MAN PAGES
quickmeme.com

Man pages

- The man pages are sectioned; you will mainly use:
  - 1: General commands
    - e.g., man ls to learn how to use ls
  - 2: System calls
  - 3: Library functions
  - 7: Miscellanea
    - e.g., man gittutorial or man man-pages
- If the command exists in more than one section, specify the section you want:
  - e.g., man 3 printf for the `printf` library function, man 1 printf for the `printf` shell command

Even the `man` command has its own man page: man man

YO DAWG, I HEARD YOU LIKE MAN PAGES
SO I MADE A MAN PAGE FOR
MAN SO YOU CAN MAN WHILE YOU MAN
imgflip.com

You likely won't use any special options, aside from man -k or man -K (to search); man man-pages will be more generally informative.

The Shell Prompt

```
$ gcc -o hello hello.c
```

- The `$` is a *prompt* indicating that the user can enter a command via keyboard input
- Commands can be shell builtins (e.g., `cd`, `ls`, `type`)
  - Check man builtins
- Commands may also launch an executable file, by providing either:
  - The full path to the executable file
  - The name of the executable file; the shell will search for the file in the directories listed in the `PATH` environment variable

Executing Programs in the Shell

```
$ gcc -o hello hello.c $ ./hello
```

- The first line compiles the C program hello.c into an executable file hello
- The second line tells the OS to load the hello program into memory and jump to its *entry point*
  - C compiles to *machine code*
  - Recall CSC207: Java compiles to *bytecode*
- Let's see how the executable file is loaded into memory. . .

Memory Model

- Memory is divided into *segments*
- The executable program code is loaded into the bottom segments:
  - Read/write data
  - Read-only code and data

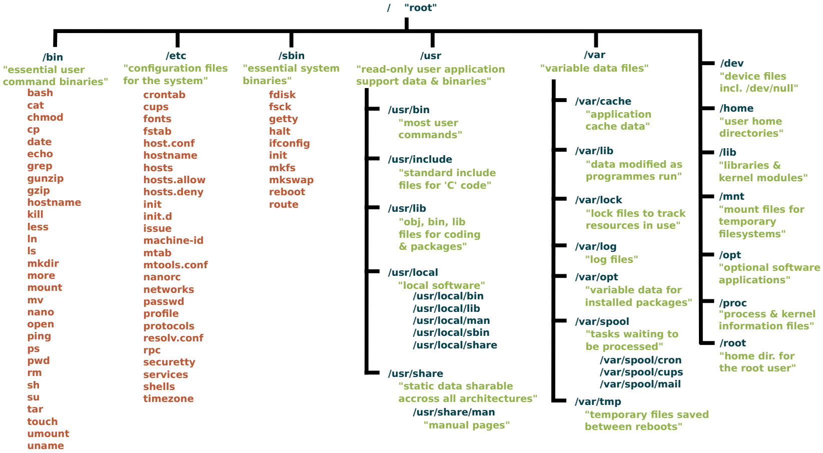
Did You Notice?

```
$ gcc -o hello hello.c $ ./hello
```

- Q: Why is `hello` prefixed by `./`, but `gcc` isn't?
  - A: Current directory is not included in `PATH`
- **Pay attention to detail**: Understand the meaning behind every character
- Even missing (or extra) spaces can cause you hours of grief

Avoid spamming gcc with code until it compiles: Compilers catch syntax errors, but not logical flaws

The UNIX File System Hierarchy



/ "root"

/bin
"essential user command binaries"
bash
cat
chmod
cp
date
echo
grep
gunzip
gzip
hostname
kill
less
ln
ls
mkdir
more
mount
mv
nano
open
ping
ps
pwd
rm
sh
su
tar
touch
umount
uname

/etc
"configuration files for the system"
crontab
cups
fonts
fstab
host.conf
hostname
hosts
hosts.allow
hosts.deny
init
init.d
issue
machine-id
mtab
mtools.conf
nanorc
networks
passwd
profile
protocols
resolv.conf
rpc
securetty
services
shells
timezone

/sbin
"essential system binaries"
fdisk
fsck
getty
halt
ifconfig
init
mkfs
mkswap
reboot
route

/usr
"read-only user application support data & binaries"

/usr/bin
"most user commands"

/usr/include
"standard include files for 'C' code"

/usr/lib
"obj, bin, lib files for coding & packages"

/usr/local
"local software"
/usr/local/bin
/usr/local/lib
/usr/local/man
/usr/local/sbin
/usr/local/share

/usr/share
"static data sharable accross all architectures"
/usr/share/man
"manual pages"

/var
"variable data files"

/var/cache
"application cache data"

/var/lib
"data modified as programmes run"

/var/lock
"lock files to track resources in use"

/var/log
"log files"

/var/opt
"variable data for installed packages"

/var/spool
"tasks waiting to be processed"
/var/spool/cron
/var/spool/cups
/var/spool/mail

/var/tmp
"temporary files saved between reboots"

/dev
"device files incl. /dev/null"

/home
"user home directories"

/lib
"libraries & kernel modules"

/mnt
"mount files for temporary filesystems"

/opt
"optional software applications"

/proc
"process & kernel information files"

/root
"home dir. for the root user"

File System Hierarchy on Lab PCs

- On most UNIX systems, a user `bob`'s home directory is `/home/bob`
  - But on the lab PCs, it is `/student/bob`
- Devices or networked file systems can be *mounted* to directories in your file system tree
  - Your home directory is mounted from the MCS server
  - Run df to see list of mounted devices and network locations

Absolute File Paths

```
/usr/bin/bash
```

- Above: Path to the executable file `bash` (our shell program)
- The leading / represents the *root directory*
- `usr` is a *subdirectory* of /
- `bin` is a subdirectory of `usr`
- `bash` is a file located in `bin`
- The ~ shortcut translates (*expands*) to your home directory, e.g., try cd ~/my_git_repo

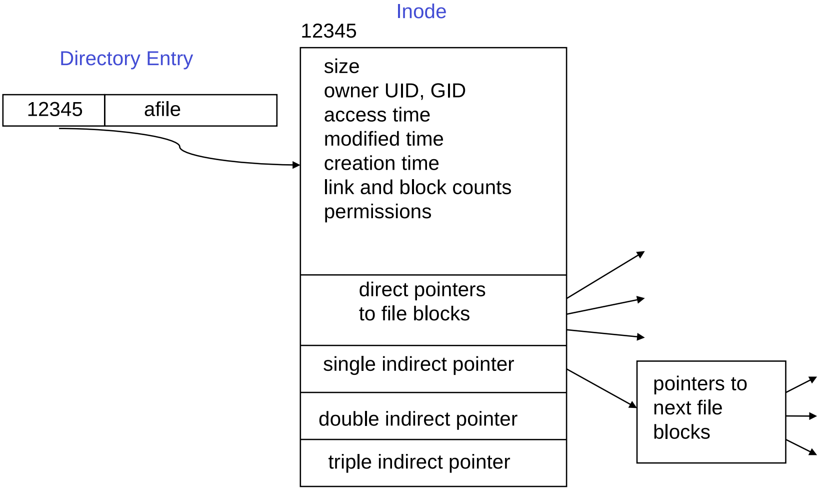
Relative File Paths

- You may also access files *relative* to your *present working directory*
  - `./file1` refers to `file1` in your working directory
  - `../file2` refers to `file2` in the *parent* of your working directory
  - `../../file3` refers to `file3` in... you get the idea
- Run pwd to see your present working directory

What is a Directory?

- A directory is a file that contains *directory entries*
- Directory entries map file names to *inode* numbers
- An inode is a data structure containing information about a file, such as its:
    - Access permissions
    - Size
    - Physical location on disk

Directory Entries and inodes



Inode

Directory Entry

12345 afile

12345

size
owner UID, GID
access time
modified time
creation time
link and block counts
permissions

direct pointers
to file blocks

single indirect pointer

double indirect pointer

triple indirect pointer

pointers to
next file
blocks

Files in UNIX

- "Everything is a File" is a key UNIX feature
  - Files and processes: Principal UNIX abstractions
- UNIX provides a file interface for all Input/Output:
  - Regular files
  - Directories
  - Special files (e.g., `/dev/null`, `/dev/urandom`)
  - Physical Devices (e.g., keyboard, mouse, printer)
    - Try cat /dev/urandom | padsp tee /dev/audio > /dev/null with your volume turned up
  - Pipes for inter-process communication
  - Network sockets

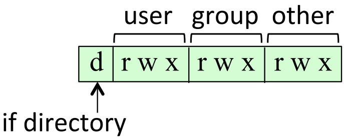
Output Redirection

- Standard I/O streams that every process starts with:
  - `stdin`: By default, reads input from keyboard
  - `stdout`: By default, writes to the console display
  - `stderr`: By default, writes to the console display
- The process treats these streams as files (surprise!)
- Use > to *redirect* `stdout`, and 2> to redirect `stderr`
  - > overwrites the output file, >> appends
  - e.g., try ls >myfiles.txt
- Refer to Section 5.1: Simple redirections, Introduction to Linux

Pipes and Process Substitution

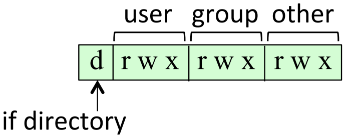
- *Pipes* transfer output from one process to another
  - e.g., ls | grep “pdf”
- *Input redirection* transfers the contents of a file into `stdin` of a process
  - e.g., wc <essay.txt
- *Process substitution* creates a *temporary file* to transfer the output from one or more processes to `stdin` of another process
  - e.g., wc <(ls) or wc <(ls | grep “pdf”)
- Refer to Chapter 23: Process substitution, Advanced Bash-Scripting Guide

UNIX File Permissions



- Each file has a permission string, e.g., `rw-r-xr-x`
- `rwx` flags represent *read*, *write*, & *execute* permissions
- Separate permissions are assigned to three categories of users:
  - The file's owning *user*
  - The file's owning *group*
  - All *other* users

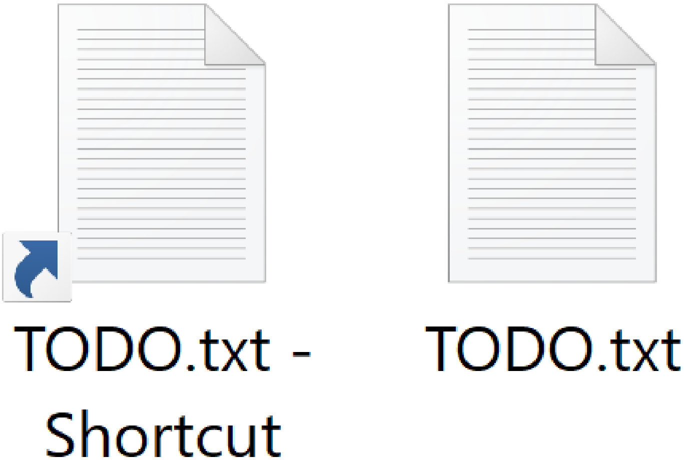
UNIX File Permissions: Directories



- First column: `d` (directory), `l` (link), or `-` (regular file)
- For directories: `r` allows listing its contents (`ls`), `w` allows creating/deleting directory entries, `x` allows entering the directory (`cd`)

Symbolic Links

- *Symbolic links* are files that contain a reference to another file name (i.e., directory entry)
- In Windows terminology, a shortcut:



Hidden Files

```
$ ls
file1  file2  file3  test1  test2

$ ls -a
.  ..  file1  file2  file3  .hidden  test1  test2
```

Files prefixed by a `.` are *hidden* files

Interpreting Directory Listings

```
$ ls -la
total 16
drwxr-xr-x 4 bob staff 4096 Jan  6 20:18 .
drwxr-xr-x 3 bob staff 4096 Jan  6 20:18 ..
-rw-r--r-- 1 bob staff    0 Jan  6 20:16 file1
-rw-r--r-- 1 bob staff    0 Jan  6 20:17 file2
lrwxrwxrwx 1 bob staff    5 Jan  6 20:17 file3 -> file2
-rw-r--r-- 1 bob staff    0 Jan  6 20:18 .hidden
drwxr-xr-x 2 bob staff 4096 Jan  6 20:16 test1
drwxr-xr-x 2 bob staff 8192 Jan  6 20:16 test2
$
```

- From left to right: file permissions, link count, owning user, owning group, file size, last modified date, and file name (symbolic link indicated by ->)
- ls -ali shows inode numbers in the first column

Changing File Permissions

- The file owner (or root user) can change a file's permissions with `chmod`
  - e.g., chmod o+r file.txt grants all other users permission to read `file.txt`
- Octal notation: For each user category, add up the values for `r` (4), `w` (2), and `x` (1)
  - e.g., chmod 754 file.txt grants:
    - `rwx` to the owning user
    - `rx` to the owning group
    - `r` to all other users
- Exercise: man chmod for more `chmod` usage examples

Globbing

- *Globbing* patterns are strings that expand to match multiple file names
    - Similar, but simpler, than regex: see man 7 glob
- `?` matches any single character
- `*` matches any string, including the empty string
- `[list of characters]` matches a single character inside the list, e.g., `[abc]`
- Usage examples:
    - rm *.log: Remove all files ending in `.log`
    - ls *.pdf: List files ending in `.pdf`

Extra Slides

Common Size of C Primitives

Type	`sizeof` (bytes)	bits
`char`	1	8
`int`	4	32
`long int`	8	64
`long long int`	8	64

GNU C compiler (`gcc`) default values (`std=gnu11`) on a 64-bit system. See GNU C Reference Manual.

Note: Compiler and machine dependent.

Hexadecimal, Decimal, Octal, and Binary

- A hexadecimal digit corresponds to 4 binary digits
  - `0x` prefix indicates hex, e.g., `0xFF`
  - `b` prefix indicates binary, e.g., `0b11`
- You may also encounter octal notation
  - `0` prefix, e.g., `012`
  - `\` prefix followed by up to 3 digits, e.g., `\111`
- Try declaring `int x` and assigning values in hex, decimal, octal, and binary
- Tutorial on binary, decimal, and hexadecimal notation

UNIX File Systems

Directory: /home/astleyrick	
File name	**inode number**
lyrics.txt	1234
dog.jpg	3000
same_dog.jpg	3000
shortcut_to_dog.jpg	7000

inode: 1234
Permissions: 640
Size: 100KB
Contents
Hard link count: 1

Never gonna give you up
Never gonna let you down
Never gonna run around and
desert you
Never gonna make you cry
Never gonna say goodbye
Never gonna tell a lie and hurt
you

inode: 7000
Permissions: 640
Size: 100KB
Contents
Hard link count: 1

/home/astleyrick/dog.jpg

inode: 3000
Permissions: 644
Size: 100KB
Contents
Hard link count: 2

Files and inodes

- In UNIX, every file is associated with an *inode*
- An *inode* is a structure that contains key information about the file, including:
  - A unique numeric ID
  - Access permissions
  - Owning user and group

Directory Entries and Links

- A *directory* is a file containing *directory entries*
- A *directory entry* maps a *file name* to an *inode number*
- *Hard links* refer to directory entries that assign one or more file names to the same inode number
- A *symbolic link* is a file that contains a reference to a *file path*, i.e., to a directory entry

Hard Links

- *Hard links* refer to multiple file names that map to the same inode
  - Each inode thus has a *link count*
- Removing a file involves deleting a directory entry, which:
  - *Unlinks* that file name from the inode
  - Decrements the corresponding inode's link count
  - If the link count is 0, the inode and associated file data is deleted
- `.` and `..` are hard links present in every directory
  - What is a directory's minimum link count?

Job Control

- Jobs are programs that were started in the shell
- ctrl+z suspends the *foreground job*
- Append & to a command to start a *background job*
  - e.g., ./hello&
  - Background jobs are killed if the terminal is closed
- jobs lists the status of jobs in the current session
- fg N resumes job number *N* in the foreground
- bg N resumes job number *N* in the background
- kill %N kills job number *N*

Typographical Conventions in Slides

- Commands to be typed: ping utoronto.ca
- Code fragments, commands, function names, variables: `printf`
- File names:
  - When part of commands/code: Same as `code`
  - Other contexts: *emphasized*
- New terms: *emphasized*
- Book titles: underlined