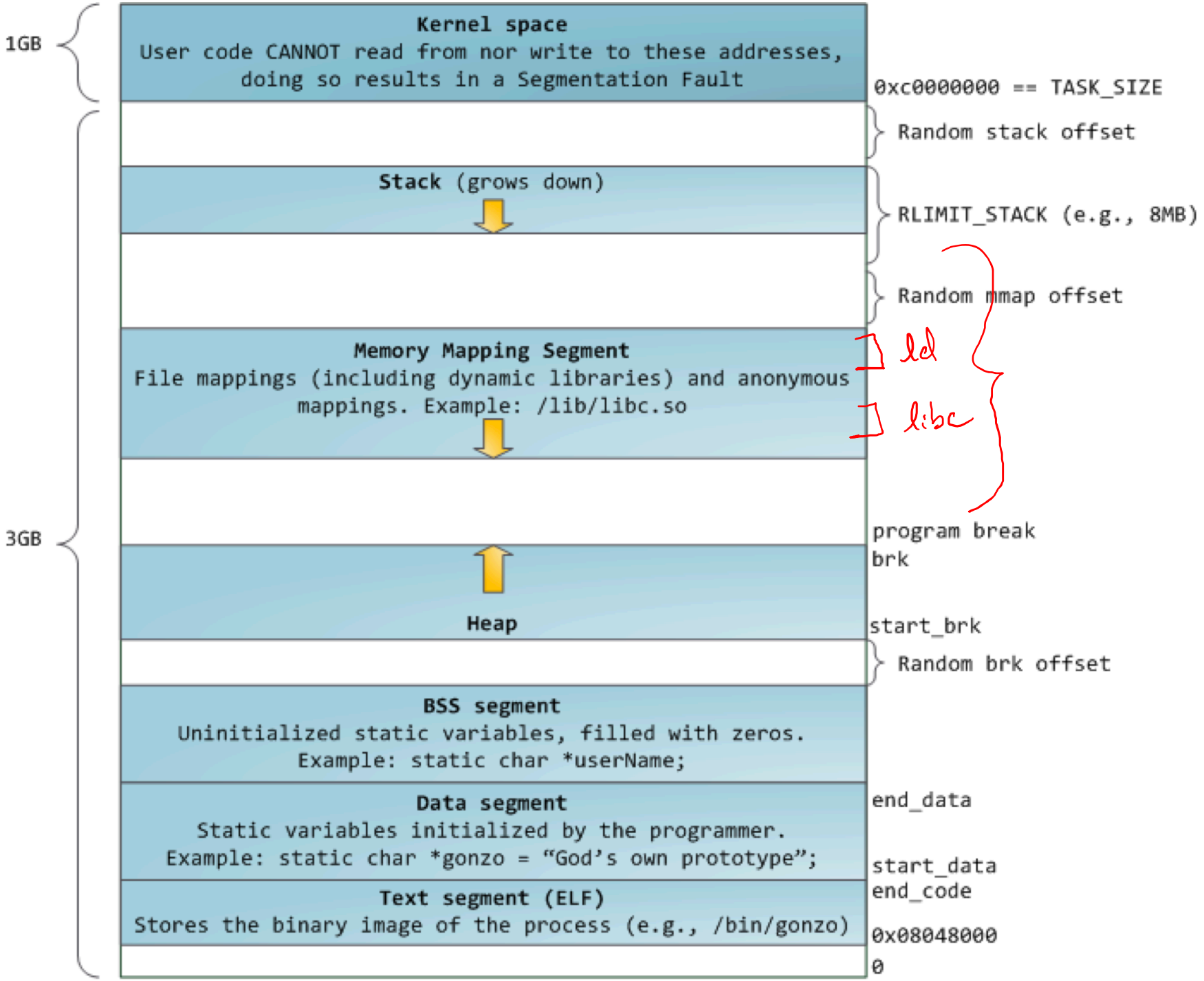
DEMO DAY!

Recall...


1GB
3GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
ld
libc
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
end_data
start_data
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
end_code
0x08048000
0

Demo: probe.c

We'll walk through this demo in class. The code can be found here:

https://github.com/traviswpeters/cs476-code/tree/master/00_intro/probe

You Try!

Think. Pair (Break Out Rooms). Share.

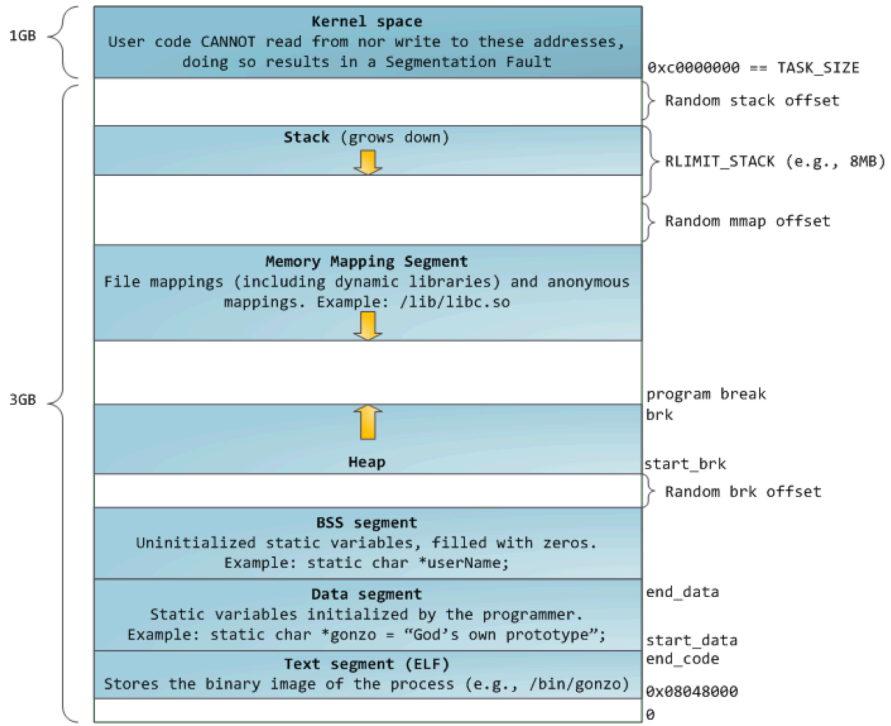
- **Get the code**:

```
[seed@VM][~]$ git clone https://github.com/traviswpeters/cs476-code.git class-code
[seed@VM][~]$ cd class-code/00_intro/probe
[seed@VM][~]$ make
[seed@VM][~]$ ./probe
```

- **Explore**:

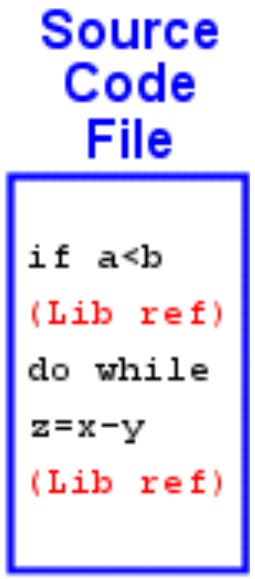
The "probe" program prints addresses for where various things are in memory. According to the output from "probe":

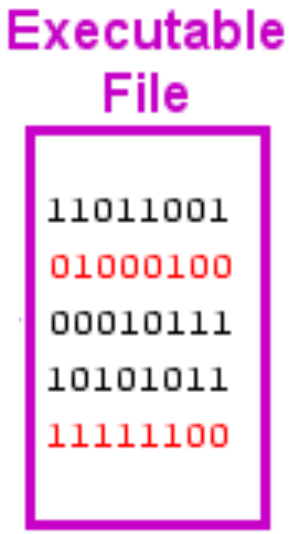
  - Where is "**main**" located in memory?
  - Where is "**printf**" located in memory?
  - Where is "**argv**" located in memory?
  - Where is "**environ**" located in memory?
  - Did you observe anything interesting/unexpected?



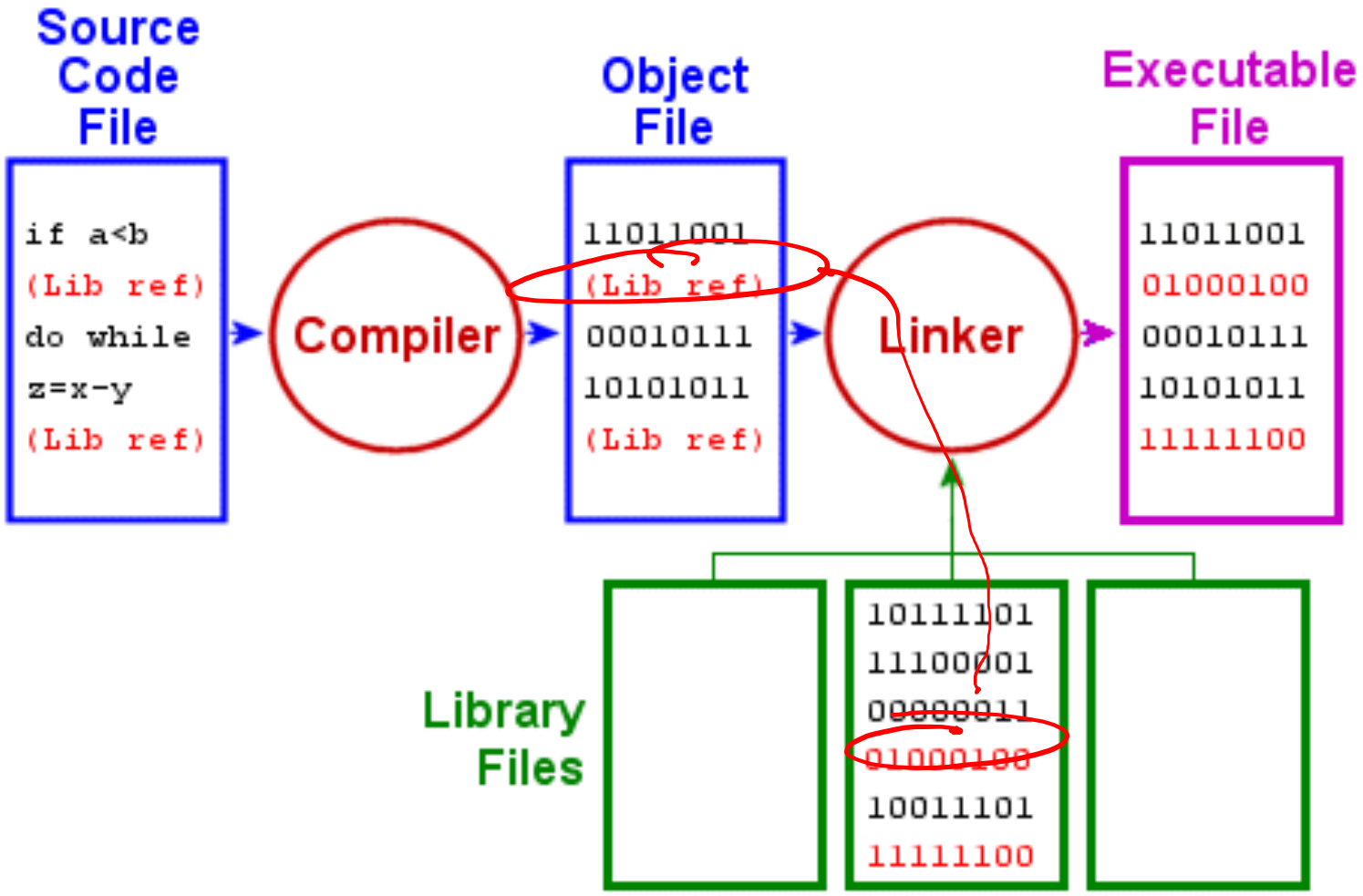
Compiling Code

Compiling C Programs

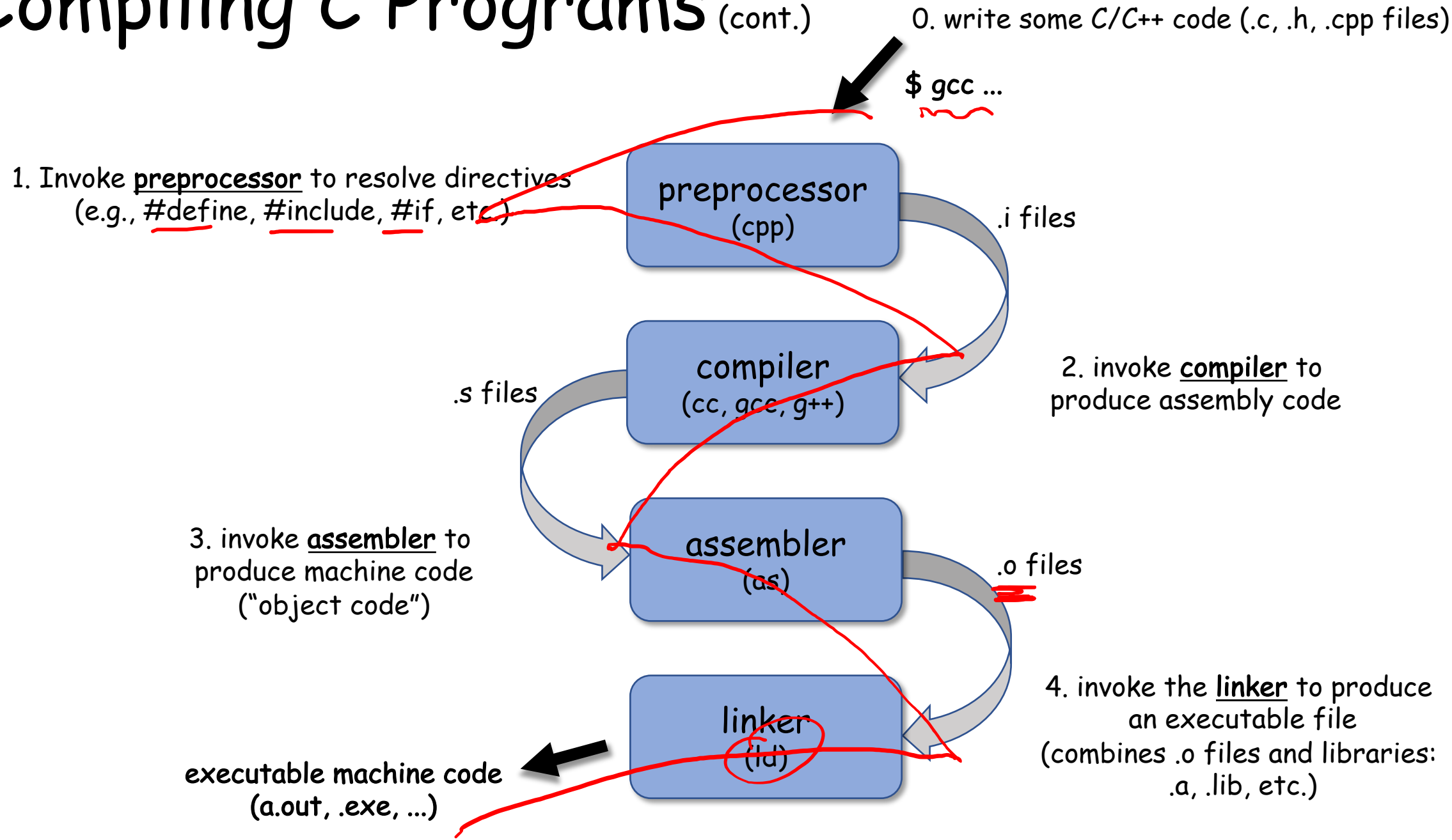




Compiling C Programs (cont.)



Compiling C Programs (cont.)



Demo: compilation_example.c

We'll walk through this demo in class. The code can be found here:

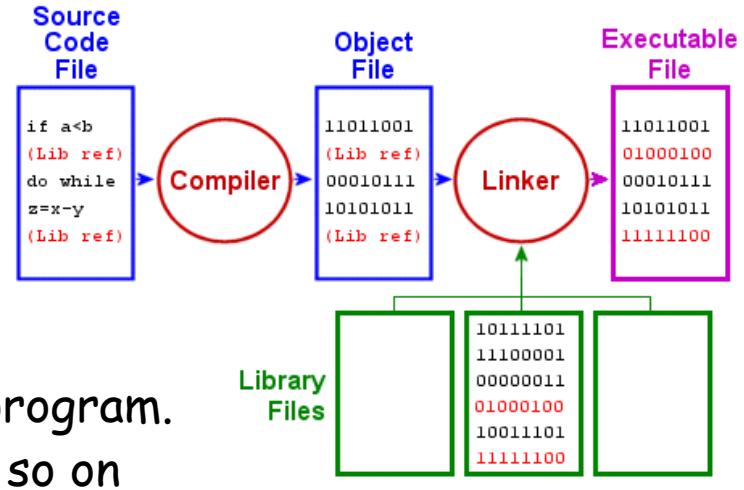
https://github.com/traviswpeters/cs476-code/tree/master/00_intro/pba

Common gcc Flags

- `gcc -o` *OUTFILE* *set the name of the resulting executable (default = a.out)*
- `gcc -E` *Stop after the **preprocessing** stage; do not run the compiler proper.*
- `gcc -S` *stop after the stage of **compilation** proper; do not assemble (produces .s files)*
- `gcc -c` ***compile but do not link** (produces .o files)*
- `gcc -D`*VAR* *acts like **#define** in the source code; it sets the value of a symbol (default is 1)*
- `gcc -I`*../headers* *specify include file in a non-standard directory.*
- `gcc -l`*name* *link a library*
  - *NOTE: library "name" is linked (system search e.g., /usr/lib/lib**name**.a)*
  - *NOTE: this is a lowercase L ("ell") not an uppercase I ("eye")*
  - *NOTE: you can tell gcc to look in a non-standard location first with -L../libs/lib*
  - *NOTE: must be run at the end*

Also check out: `-Wall, -ggdb, -O`*N*`, -m32 -fno-builtin, -masm=intel...`

Libraries



- *A* **library** is…
  - an assortment of **pre-compiled pieces of code** that can be reused in a program.
  - they provide reusable functions, routines, classes, data structures and so on (written by a another programmer), which they can use in their programs.
  - Examples: libc (/lib64/libc.so.6), glibc, libcurl, libcrypt, pthreads, ….
    - so = "shared object"

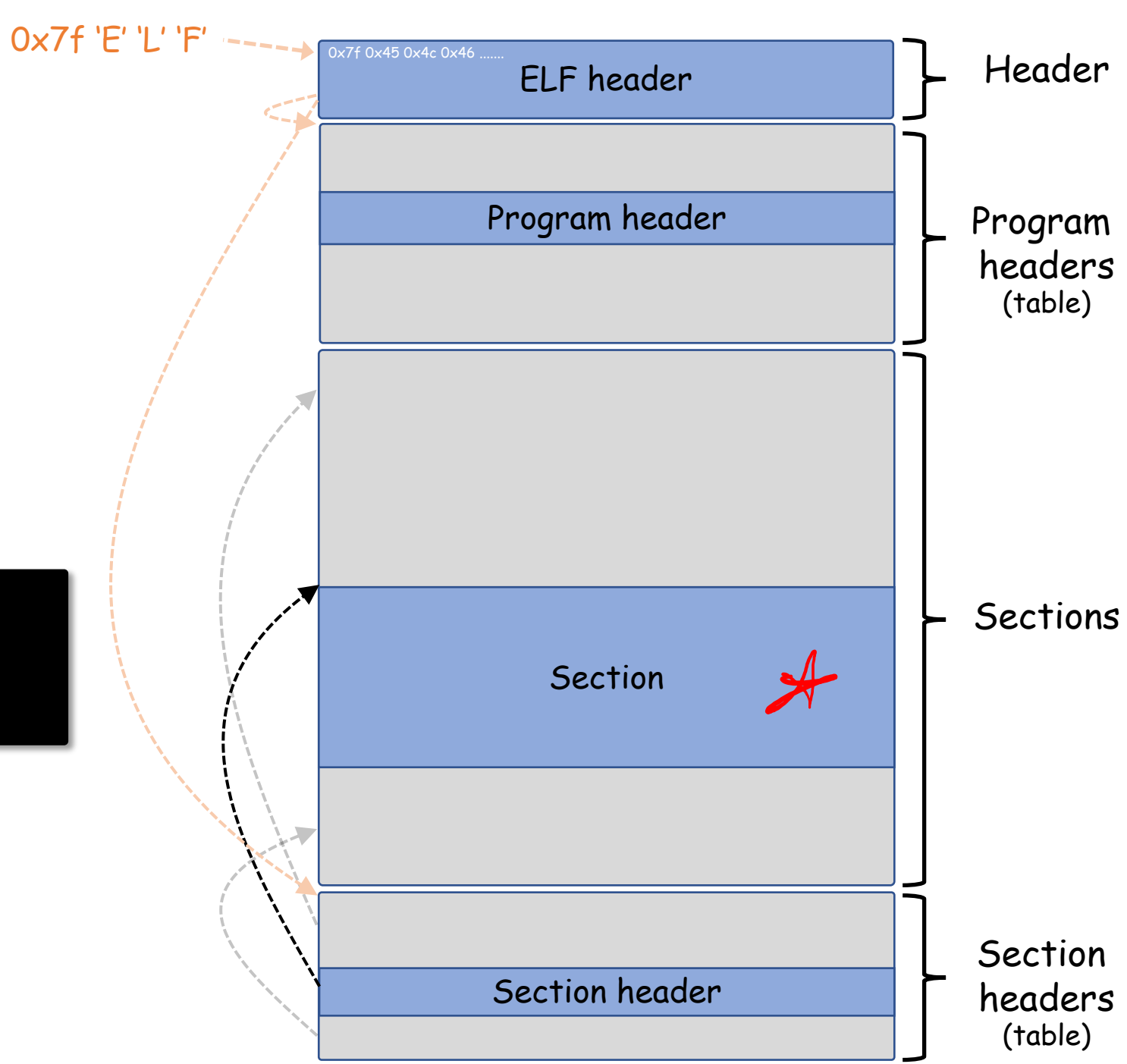
- Linux supports two classes of libraries, namely:
  - **Static Libraries** – are bound to a program statically at compile time.
  - **Dynamic/Shared Libraries** – are loaded when a program is launched and loaded into memory and binding occurs at run time. These can further be categorized into:
    - Dynamically Linked Libraries – here a program is linked with the shared library and the kernel loads the library (in case it's not in memory) upon execution.
    - Dynamically Loaded Libraries – the program takes full control by calling functions with the library.
  - Shared libraries are loaded by ld.so (or ld.so.x) and ld-linux.so (or ld-linux.so.x) programs

Compiled Code

ELF

Executable and Linkable Format

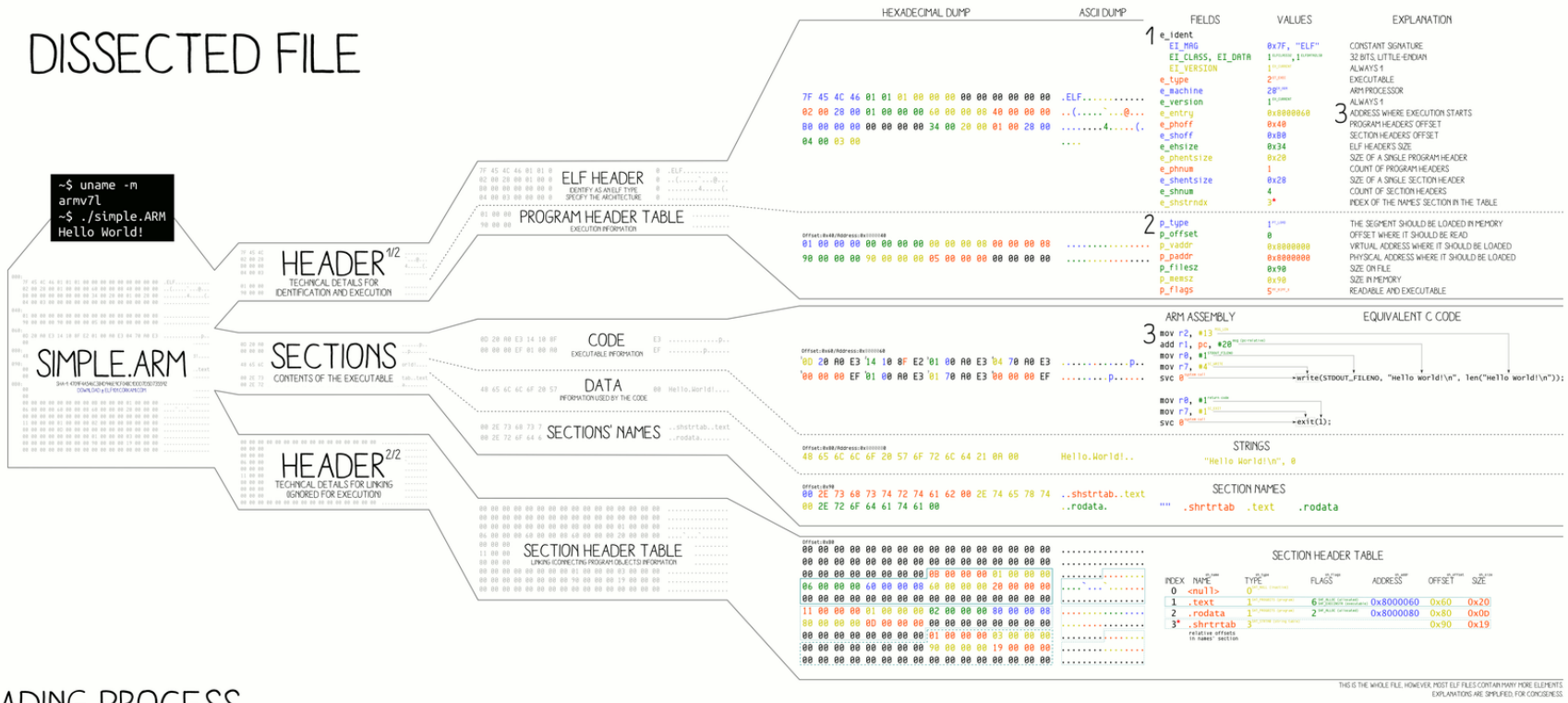
```
[seed@VM][~]$ readelf ...
[seed@VM][~]$ objdump ...
```



ELF[101] a Linux executable walkthrough

ANGE ALBERTINI
CORKAMI.COM

DISSECTED FILE



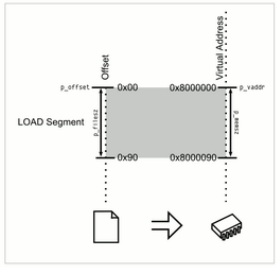
LOADING PROCESS

1 HEADER

THE ELF HEADER IS PARSED
THE PROGRAM HEADER IS PARSED
(SECTIONS ARE NOT USED)

2 MAPPING

THE FILE IS MAPPED IN MEMORY
ACCORDING TO ITS SEGMENT(S)



3 EXECUTION

ENTRY IS CALLED
SYSCALLS[OS] ARE ACCESSED VIA:
- SYSCALL NUMBER IN THE R7 REGISTER[ARM]
- CALLING INSTRUCTION SVC

TRIVIA

THE ELF WAS FIRST SPECIFIED BY U.S.L.[ORG] AND U.I.[ORG]
FOR UNIX SYSTEM V, IN 1989

THE ELF IS USED, AMONG OTHERS, IN:
- LINUX, ANDROID, *BSD, SOLARIS, BEOS
- PSP, PLAYSTATION 2-4, DREAMCAST, GAMECUBE, WII
- VARIOUS OSES MADE BY SAMSUNG, ERICSSON, NOKIA,
- MICROCONTROLLERS FROM ATMEL, TEXAS INSTRUMENTS

VERSION 1.0[A]
2013/12/06

ELF (Executable and Linkable Format)

(if time)

Demo: File Ops, Users, and Groups

We'll walk through this demo in class. The code can be found here:

https://github.com/traviswpeters/cs476-code/tree/master/00_intro (see README.md)