# CSCI 2021: ELF Files and Linking

Chris Kauffman

*Last Updated:*
*Tue Apr 30 16:42:59 CDT 2019*

# Logistics

## Reading Bryant/O'Hallaron

- Ch 7: Linking

## Assignment 5: Questions?

- Memory allocator + Print ELF Symbol Table
- Due last day of classes

## Goals

- Separate Compilation
- ELF Files
- Linking and Loading
- Libraries

| Date | Event |
|---|---|
| Mon 4/29 | Linking |
| Wed 5/1 | Linking / Evals |
| | Lab 13: Linking |
| Fri 5/3 | Linking/Loading |
| Mon 5/6 | Last day of class |
| | A5 Due |

## Final Exams

- Sec 001 (12:20 MWF) Sat 5/11 1:30pm
- Sec 010 (3:35 MWF) Mon 5/13 10:30am

# Exercise: Separate Compilation

```
# COMPILATION 1
> gcc -c func_01.c
> gcc -c main_func.c
> gcc -o main_func main_func.o func_01.o

# COMPILATION 2
> gcc -o main_func main_func.c func_01.c
```

- Describe differences between compilations above
- What is the result in each case?
- How are they different: any *artifacts* created in one but not the other?
- Any advantages/disadvantages to them?

# **Answers**: Separate Compilation

```
# COMPILATION 1
> gcc -c func_01.c
> gcc -c main_func.o
> gcc -o main_func main_func.o func_01.o

# COMPILATION 2
> gcc -o main_func main_func.c func_01.c
```

## Compilation 1: Separate Compilation

- Separately compile `func_01.c` and `main_func.c` to binary
- Results in 2 `.o` **object files**
- Final step is to **link** two objects together to create an executable

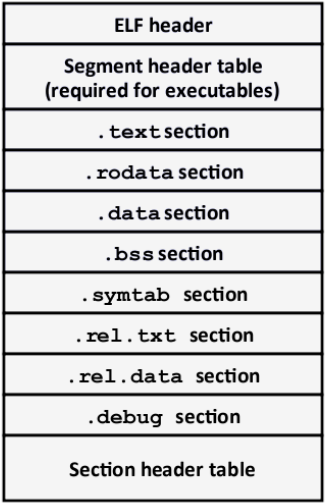
## Compilation 2: "Together" Compilation

- Compile all the C files at once to produce an executable
- Still likely to internally do separate compilation BUT no `.o` files will be produced, only executable

Advantages of Separate Compilation to follow

# Object Files and ELF

- Binary files can't be random so will usually adhere to some standard
- **Executable and Linkable Format (ELF)** is standard for the results of compilation on Unix systems
- Stores program data in a variety of **sections** in binary
- Explicitly designed to allow binary objects to be
  - Executed (programs)
  - Merged with other objects (linked)
- Preceded by a dated format called `a.out`: still default name of gcc output programs



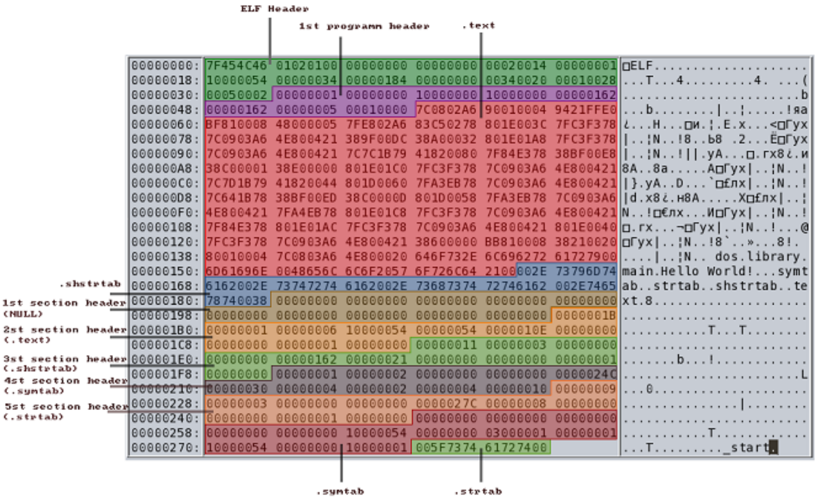
# Brief Tour of ELF Sections

- ELF defines many specific sections that are used in specific circumstances
  - Always ELF Header at the beginning
  - Always Program (Segment) Header Table for executable
  - Always Section Header Table for linkable objects
- Some sections like `.debug` are common but don't appear in ELF specification (have their own DWARF spec)

| Section | Brief Description |
|---|---|
| ELF Header | Global Info (32- or 64-bit, Execuable?, Byte ordering, etc.) |
| Program Header Table | For executable programs, virtual address space info |
| Section Header Table | Descriptions of sections and positions in file |
| `.text` | Opcodes (binary assembly) that can be executed |
| `.rodata` | Read Only data like string constants |
| `.data` | Initialized global variables, space for values |
| `.bss` | Un-initialized global varaiables, no space for values |
| `.symtab` | Table of publicly available symbols for funcs/vars |
| `.strtab` | Null-terminated strings, often names of things in `.symtab` |
| `.shstrab` | Null-terminated strings, often names section headers |
| `.debug` | Debug info from `gcc -g` in DWARF format |
| `.rel.text` | Relocation information for `.text` section |
| `.rel.data` | Relocation information for `.data` section |

# ELF is a Binary Format

- ELF is a binary format so it is NOT easy on the eyes
- Make use of utilities like `readelf` to examine sections
- Can view bytes yourself but it is not usually intelligible



# Linking: Merging Binary Files to One

**Linking** merge multiple `.o` into one `.o` or executable file

- Merge `.text` section with instructions
- Merge `.data` section with global variables
- Merge `.symtab` modifying positions of where things exist, etc.

## Symbol Resolution

- Multiple object files define a symbol, must resolve which definition to use
- Some tricky bugs can arise in resolution

## Relocation

- Adjust offsets of things in symbol table
- Change any instructions which use locations that have changed

Will talk more about algorithms/approach for Resolution and Relocation used by the linker later

# Exercise: Separate Compilation Time

- Mack is building a large application
- Has a `main_func.c` and `func_01.c, func_02.c ...` that define application, up to `func_20.c`
- During build process notices that it takes about 10s for to compile each C file and 20s to link the C files
- After editing files to add features, Mack usually compiles to project like this

```
> gcc -o main_func *.c
```

- **Estimate** his typical build time in seconds
- **Suggest** a way that he might reduce his build time if he has edited only a small number of files

# **Answers**: Separate Compilation Time

Total Build Time `gcc -o main_func *.c`

| Item | Example | Build | Tot |
|---|---|---|---|
| Library C files | `func_01.c` | 20 x 10s | 200s |
| Main C file | `main_func.c` | 1 x 10s | 10s |
| Linking | all `.o` files | 1 x 20s | 20s |
| Total Time | $\sim$ 4min | 22 steps | 230s |

- Explicitly recompiling all C files to object code despite many not changing
- Spends valuable human time waiting to redo the same task as has been done many before

# Answers: Separate Compilation Time

## Exploit Separate Compilation

- Assume already compiled all files, have `func_01.o`, `func_02.o`
- Edit `func_08.c` to add a new feature
- **Don't** recompile C files that haven't changed
- Compile like this

```
> gcc -c func_08.c
> gcc -o main_func *.o
```

| Item | Example | Build | Time |
|---|---|---|---|
| Library `.o` files | `func_01.o` | 19 x 0s | 0s |
| Main `.o` file | `main_func.o` | 1 x 0s | 0s |
| Changed `.c` files | `func_08.c` | 1 x 10s | 10s |
| Linking | all `.o` files | 1 x 20s | 20s |
| Total Time | $\sim$ 30 seconds | 2 steps | 30s |

# Build Systems Exploit Separate Compilation

- Build Systems like `make / Makefile` exploit separate compilation
- Build system establishes a dependency structure
- **Targets** are usually files to create
- **Dependencies** are other files/targets that must be up to date to create a given target
- Only rebuild a target if a dependency **changes**

```
# Typical Makefile gives targets, dependencies,
# commands to create target using dependencies

# TARGET  : DEPENDENCIES
main_func : main_func.o func_01.o func_02.o
        gcc -o main_funcs main_func.o func_01.o func_02.o

main_func.o : main_func.c
        gcc -c main_funcs.c

func_01.o : func_01.c
        gcc -c funcs_01.c
```

# Example Builds from big-compile/

```
> make clean
rm -f *.o main_func

# first compiles, no object files built, build everything
> make main_func
gcc -c main_func.c
gcc -c func_01.c
gcc -c func_02.c
...
gcc -c func_20.c
gcc -o main_func main_func.o func_01.o func_02.o...


# edit func_08.c

# 1 file changed, recompile it and re-link
> make main_func
gcc -c func_08.c    # ONLY NEED TO RECOMPILE THIS
gcc -o main_func main_func.o func_01.o func_02.o...

# no edits, no need to rebuild
> make main_func
make: Nothing to be done for 'main_func'.
```

# Exercise: Initialized vs Uninitialized Data Matters

```
// FILE: big_data.c
long arr[20000] = {1,2,3};
int main(){
  for(int i=0; i<1024; i++){
    arr[i] = i;
  }
  return 0;
}
```

```
// FILE: big_bss.c
long arr[20000] = {};
int main(){
  for(int i=0; i<1024; i++){
    arr[i] = i;
  }
  return 0;
}
```

```
> gcc -c big_data.c  # compile to object
> du -b big_data.o   # print number of bytes
161384  big_data.o

> gcc -c big_bss.c   # compile to object
> du -b big_bss.o    # print number of bytes
1384    big_bss.o
```

- What is the difference between the two files above?
- Why is there such a size difference in the object files

# **Answers**: Initialized vs Uninitialized Data Matters

- ELF `.data` section tracks global variables that is initialized with non-zero values
- Must record every value in global variable so it can be properly set when loaded to run
- `big_data.o` will have a large `.data` section as the line

  `long arr[20000] = {1,2,3};`

  initializes the first few array values, rest will be 0

```
> readelf -S big_data.o
There are 12 section headers, starting at offset 0x27368:
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
...
  [ 3] .data             PROGBITS         0000000000000000  00000080 <--
---->  0000000000027100  0000000000000000  WA       0     0     32
  [ 4] .bss              NOBITS           0000000000000000  00027180 <--
       0000000000000000  0000000000000000  WA       0     0     1
...
```

- `0x27100 = 160000` bytes: entire `arr` array stored in file

## **Answers**: Initialized vs Uninitialized Data Matters

- ELF `.bss` section tracks global variables that are not initialized or initialized to all 0's
- No specific values need be recorded, just instructions on how much space to allocate on starting the program
- `big_bss.o` will have a miniscule `.data` section as the line

  `long arr[20000] = {};`

  initializes to all 0's so `.bss` section

```
  > readelf -S big_bss.o
There are 12 section headers, starting at offset 0x268:
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
...
  [ 3] .data             PROGBITS         0000000000000000  0000007f
       0000000000000000  0000000000000000  WA       0     0     1
  [ 4] .bss              NOBITS           0000000000000000  00000080 <--
---->  0000000000027100  0000000000000000  WA       0     0     32
  [ 5] .comment          PROGBITS         0000000000000000  00000080 <--
       0000000000000012  0000000000000001  MS       0     0     1
...
```

- `arr` array NOT stored in file, significantly smaller `.o` file

# Linker: Multiple .o to Single/Executable

- A `linker` converts multiple `.o` files to
  - To an executable (default)
  - Single `.o` file (`-r` option)
- gcc automatically invokes the linker when creating executables
- Can also manually play with linker: `ld` command
  - SO: Why is the Unix linker called 'ld'?
- Rarely use `ld` by hand: difficult to generate executables properly
- gcc invokes `ld` with *many* additional options / libraries to create executables

```
# Demo merging two .o files with ld

> nm func_01.o   # names in .o file
0000000000000000 T func_01
                 U puts

> nm func_02.o   # names in .o file
0000000000000000 T func_02
                 U puts

# manually link to create combined .o
> ld -r func_01.o func_02.o \
     -o funcs_12.o

> nm funcs_12.o  # names in .o file
0000000000000000 T func_01
0000000000000013 T func_02
                 U puts

# can't create executable with
# undefined symbols and no main()
> ld func_01.o func_02.o \
     -o executable.o
ld: warning: cannot find
 entry symbol _start;
defaulting to 00000000004000e8
func_01.o: In function `func_01':
func_01.c:(.text+0xc): `puts' undefined
func_02.o: In function `func_02':
func_02.c:(.text+0xc): `puts' undefined
```

# Symbol Resolution by the Linker

- One of the jobs of the linker is resolve symbols (names) when merging two relocatable objects (`.o` files)
- A few rules apply
  1. `.o` files can have undefined symbols but executables (mostly) cannot
  2. Symbols are classified as **strong and weak**; can only have one **strong** definition but many weak definitions
  3. Strong definitions are mostly named functions and global variables
  4. Weak definitions are mostly uninitialized global variables and `extern` declarations for global variables, function prototypes
- Only **global stuff** qualify as symbols: functions, global variables
- Local variables inside functions will NOT have symbols associated

# Exercise: Linking Trouble

Consider these two C files

```c
// FILE: x_int.c
int x=0;         // global vars
int y=0;         // strongly defined

void x_to_neg8(); // in different .o

#include <stdio.h>
int main(){
  x_to_neg8();  // set x only
  printf("x: %d\n",x);
  printf("y: %d\n",y);
  return 0;
}
```

```c
// FILE: x_long.c
long x;      // global var

void x_to_neg8(){
  x = -8;    // set global var
}
```

## Predictions

Compile and run: **predict output**

```
> gcc x_int.c x_long.c
/usr/bin/ld: Warning: ...
> ./a.out
x: ??
y: ??
```

# **Answers**: Linking Trouble

- Two files define the sizes of global variable `x` differently

```
// FILE: x_long.c
long x;      // uninitialized, weak symbol
// FILE: x_int.c
int x = 0;   // initialized, strong symbol, prevails
int y = 0;
```

- Linker warns of this during compilation (see below)
- Variable `y` in `x_int.c`, adjacent to 4-byte `x` in memory
- Function `void x_to_neg8()` is in `x_long.c`
- Writes 8 bytes to location `x` clobbering `y`

```
> gcc x_int.c x_long.c
/usr/bin/ld: Warning: alignment 4 of symbol `x'
in /tmp/ccs1zLtj.o is smaller than 8 in /tmp/ccc7ZX9Q.o

> ./a.out
x: -8
y: -1
```

- Message: Global variables are dangerous in linking (and for code design in general) [but you knew that already]

# The Value of Headers and `extern` declarations

- Headers (`.h`) declare global symbols for all C files that will use them
- May declare *external* variables which are defined in another file

```
// FILE: x_to_neg8.h
extern long x;
void x_to_neg8();
-------------------------
// FILE: x_to_neg8.c
#include "x_to_neg8.h"
long x;  // actual global var
void x_to_neg8(){
  x = -8;
}
-------------------------
// FILE: x_main.c
#include "x_to_neg8.h"
// there will be an x var
// and x_to_neg8() func
...
```

- Proper use of headers allow compiler to warn of conflicting definitions

```
// FILE: x_main.c
#include "x_to_neg8.h"
int x = 0; // !!!
...

> gcc -c x_main_bad.c
x_main_bad.c:4:5: error:
conflicting types for 'x'
  int x = 0; // !!!
      ^
x_to_neg8.h:7:13: note:
previous declaration of
'x' was here
  extern long x;
              ^
```

- Without using `.h` header files, compiler can't help as much

# The Immense Journey (apologies to Loren Eisley)

From C source file to running process involves a variety of tools, formats, software and hardware, summarized for Linux below

1. *Compilation:* `gcc` preprocesses `prog.c` file, converts to internal representation, optimizes, produces assembly code (stop at this stage with `-S`)
2. *Assembly:* `gas` invoked by `gcc` to turn a `prog.s` file to a `prog.o` ELF file, may be other `.o` files involved for multiple `.c` files
3. *Linking:* `ld` invoked by `gcc` to link multiple `.o` files to single executable or library, copy in any statically linked library code, indicates if executable has dynamic library dependencies
4. *Stored Program:* Now have an executable program in ELF format stored on disk waiting to be run; call it `prog.out`
5. *Loading:* `ld-linux.so` invoked by shell to load `prog.out` into memory, sets up virtual memory map for `.data, .text, etc.`, initializes `.bss` sections to 0, resolves any dynamic library links required at load time, sets `%rip` to first program instruction
6. *Running:* OS handles remaining behavior of running program (**process**), running, sleeping, exiting, killing on segfaults

# Linker and Loader

## Traditional: Static Linking

- Linker merges `.o` files to create executable
- All global symbols must be resolved: copy text for functions into the executable from libraries
- **Loader** copies executable into memory, set `%rip` to first instruction address, notifies OS to schedule it for execution
- All code/data for running program is in its own memory image

## Modern: Dynamic Linking

- Linker merges `.o` files to create executable
- Global symbols from Dynamic Libraries are left Undefined (`U`)
- Loader copies executable into memory, sets `%rip` but..
- Creates a virtual memory map to definitions for library functions **dynamically linking** to definitions
- Code for running program is spread across its memory image and shared libraries

# gcc: Statically vs Dynamically Linked Executables

- By default gcc produces dynamically linked executables
- With the -static option, will statically link libraries
- Note the differences reported by the file command below

```
> cat hello.c
#include <stdio.h>
int main(int argc, char *argv[]){
  printf("Hello world! I'm a program\n");
  return 0;
}

# compile static dynamically linked vs statically linked
> gcc -o hello_dynamic hello.c
> gcc -o hello_static  hello.c -static

# examine file types
> file hello_static
hello_static: ELF 64-bit LSB executable, x86-64, statically linked

> file hello_dynamic
hello_dynamic: ELF 64-bit LSB shared object, x86-64, dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2
```

# Exercise: Static/Dynamic Program Sizes

- Examine file sizes of two programs below reported by `du`
- Which program is bigger on disk in number of bytes?
- **Why** is there a size difference?

```
# compile static dynamically linked vs statically linked
> gcc -o hello_dynamic hello.c
> gcc -o hello_static  hello.c -static

# examine size of executables in bytes
> du -b hello_*
  9664  hello_dynamic
721424  hello_static
```

# **Answers**: Static/Dynamic Program Sizes

```
# examine size of executables in bytes
> du -b hello_*
  9664  hello_dynamic  #   9,664 bytes
721424  hello_static   # 721,424 bytes
```

- Entire `libc` library (`printf/puts/malloc/etc.`) copied into statically linked version
- Dynamically linked version has undefined references to functions like `puts()` which will be resolved at load/run time

```
# examine symbols/functions
# in static/dynamic executables

> nm hello_static
...
00000000004009dd T main
# T: defined "strong" symbol
...
0000000000408460 W puts
# W: defined "weak" symbol
...

> nm hello_dynamic
...
000000000000064a T main
# T: defined "strong" symbol
...
                 U puts@@GLIBC_2.2.5
# U: undefined
# your funciont is in
# a different castle
```

# Libraries Required at Load/Runtime

- Most executables know ahead of time which dynamic libraries will be needed at run time
- Can examine this with the `ldd` command: print shared object dependencies

```
> gcc -o hello_dynamic hello.c
> gcc -o hello_static  hello.c -static

# examine which libraries will be dynamically linked
# compile static dynamically linked vs statically linked

> ldd hello_static
      not a dynamic executable

> ldd hello_dynamic
      linux-vdso.so.1 (0x00007ffe9b0fb000)
      libc.so.6 => /usr/lib/libc.so.6 (0x00007f6a8c295000)  #printf!
      /lib64/ld-linux-x86-64.so.2 =>
          /usr/lib64/ld-linux-x86-64.so.2 (0x00007f6a8c84e000)
```

# Linking Against Standard Libraries

- At link time, linker must know about library dependencies
- gcc option `-l` will link against a library

```
> gcc do_math.c -lm               # link to math library
> gcc do_pthreads.c -lpthread   # link to threads library
```

- Convention: `lmystuff` links against library `libmystuff.a` or `libmystuff.so`. GCC **always** links `libc`.
- By default Linker selects dynamic libraries if available, static if dynamic not found or `-static` option in effect,
- Compiler/Linker search some known directories for header files and libraries

```
> gcc -v do_math.c -lm        # -v: verbose output
...
#include <...> search starts here:
/usr/lib/gcc/x86_64-pc-linux-gnu/7.2.1/include
/usr/local/include
/usr/lib/gcc/x86_64-pc-linux-gnu/7.2.1/include-fixed
/usr/include
...
LIBRARY_PATH=/lib/:/usr/lib/:...
```

# Creating/Linking Statically Linked Libraries

- Statically Linked Libraries are **archives** with `.a` extension
- Traditional form of program libraries, comprised of a bunch of `.o` files
- Utility `ar` allows creation, modification, inspection of `.a` files
- Most systems include `/lib/libc.a` to allow creation statically linked programs
- System `.a` archives are identical in structure to user-created libraries

```
> gcc -g -Wall -c tree.c
> gcc -g -Wall -c array.c
> gcc -g -Wall -c list.c
> gcc -g -Wall -c util.c

# create archive with ar
> ar rcs libds_search.a \
    tree.o array.o list.o util.o

> file libds_search.a
libds_search.a: current ar archive

# show .o files in archive
> ar t libds_search.a
tree.o array.o list.o util.o

> ar t /lib/libc.a | grep printf.o
vfprintf.o vprintf.o reg-printf.o
fprintf.o  printf.o  snprintf.o
...
```

## Exercise: Linking Against User Libraries

- Lab 14 discussed linking against user-libraries NOT in standard library directories
- Example problems:

```
> ls ds_search_static/
libds_search.a
ds_search.h

# PROBLEM 1
> gcc do_search.c -lds_search
do_search.c:8:10: fatal error:
  ds_search.h: No such file or directory
   #include "ds_search.h"
            ^~~~~~~~~~~~~
compilation terminated.

# PROBLEM 2
> gcc do_search.c -lds_search ...
/usr/bin/ld: cannot find -lds_search
collect2: error: ld returned 1 exit status
```

- How do compiler options resolve these two problems?

# **Answers**: Linking Against User Libraries

```
> ls ds_search_static/
libds_search.a
ds_search.h

# PROBLEM 1
# Use -I to give "includes" directory with header
> gcc do_search.c -lds_search  \
      -I ds_search_static/   # header directory for ds_search.h
/usr/bin/ld: cannot find -lds_search
collect2: error: ld returned 1 exit status

# PROBLEM 2
# Use -L to add a directory to search for libraries
> gcc do_search.c -lds_search \
      -I ds_search_static/   # header directory for ds_search.h
      -L ds_search_static/   # library directory with libds_search.a
> file a.out

a.out: ELF 64-bit LSB shared object, x86-64
```

# Creating Dynamic Libaries

- Dynamically Libraries are **shared objects** with `.so` extension (or `.dll` if you are a Windows user)
- Created by invoking compiler linker with appropriate options
  - Compile option `fPIC` for **position independent code**
  - Link option `-shared` for a shared object
- Dynamic libraries may depend on other dynamic libraries

```
> gcc -g -Wall -fpic -c tree.c
> gcc -g -Wall -fpic -c array.c
> gcc -g -Wall -fpic -c list.c
> gcc -g -Wall -fpic -c util.c

# create shared object with gcc
> gcc -shared -o libds_search.so
  \ tree.o array.o list.o util.o

> file libds_search.so
libds_search.so: ELF 64-bit LSB
shared object, x86-64, ...

# show dependencies
> ldd libds_search.so
    linux-vdso.so.1 (0x00007ffce291e0
    libc.so.6 => /usr/lib/libc.so.6 (
    /usr/lib64/ld-linux-x86-64.so.2 (
```

# Exercise: A Dynamic Hitch

Consider the below hitch in the wonder of dynamic libraries

```
> gcc do_search.c -lds_search \
      -I ds_search_dynamic/   \
      -L ds_search_dynamic/

> a.out
 a.out: error while loading shared libraries:
 libds_search.so: cannot open shared object file:
 No such file or directory

> ldd a.out
  linux-vdso.so.1
  libds_search.so => not found   !!!!
  libc.so.6 => /usr/lib/libc.so.6
  /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2
```

- What went wrong?
- Thoughts on how to resolve?
- Why didn't this happen in the statically linked case?

# **Answers**: A Dynamic Hitch

- Compiler informed that `libds_search.so` was in a non-standard directory
- **Loader** NOT informed of this
- Loader searched `/lib/` and other places, didn't find `libds_search.so` gave up on loading the program
- Must inform loader of non-standard directories for libraries with `LD_LIBRARY_PATH`
- An **environment variable** honored by loader, places to search aside from standard locations
- Environment variables can be set in most shells and are looked for by programs to modify their behaviour
- Default command shell on many Unixes is `bash` with env't var syntax `export VAR=some_value`
- Often set vars in initialization files like `.bashrc` or `.bash_init`

```
export PAGER=less          # a better 'more'
export EDITOR=emacs        # major improvement
export BROWSER=chromium    # open source baby
```

# **Answers**: A Dynamic Hitch

Below is a complete session which fixes the loading problem

```
> a.out
 a.out: error while loading shared libraries:
 libds_search.so: cannot open shared object file:
 No such file or directory

> export LD_LIBRARY_PATH="ds_search_dynamic"

> ldd a.out
 linux-vdso.so.1
 libds_search.so => ds_search_dynamic/libds_search.so  :-)
 libc.so.6 => /usr/lib/libc.so.6
 /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2
> a.out
Searching 2048 elem array, 10 repeats: 1.6470e-01 seconds
```

If distributing a `.so`, either

- Put it install it in a standard location (admin access)
- Notify users of library to adjust `LD_LIBRARY_PATH`

# Exercise: Dynamic Loading Tricks

Consider the following strange sesssion

```
> gcc hello.c
> a.out
Hello World!
My favorite int is 32 and float is 1.234000

> gcc -shared -fPIC -Wl,-soname -Wl,libsamy_printf.so -o libsamy_printf
> export LD_PRELOAD=$PWD/libsamy_printf.so

> a.out
Hello World!
... but most of all, Samy is my hero.
My favorite int is 32 and float is 1.234000
... but most of all, Samy is my hero.
```

Why would compiling another piece of code change the behavior of an **already compiled program**?

# **Answers**: Dynamic Loading Tricks

- One can **interpose** library calls: ask dynamic loader to link a function to a different definition
- Only possible with dynamic linking but a powerful technique
- In this case, re-define `printf()`, similar tricks by `valgrind` for `malloc() / free()`

```
> gcc hello.c
> a.out
> ldd a.out
  linux-vdso.so.1
  libc.so.6 => /usr/lib/libc.so.6
  /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2

> export LD_PRELOAD=$PWD/libsamy_printf.so
> ldd a.out
  linux-vdso.so.1 (0x00007fff591d6000)
  /home/kauffman/2021-S2018/.../libsamy_printf.so  !!!!
  libc.so.6 => /usr/lib/libc.so.6
  libdl.so.2 => /usr/lib/libdl.so.2
  /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2
```

# Valgrind and Your own Malloc

- Valgrind replaces normal `malloc() / free()` with its own version which is slower but allows error checking
- Uses dynamic loading tricks for this so you don't need to recompile your program
- If you complete `el_malloc.c`, you could extend it to a full allocator (would need `realloc()`, use of `sbrk()` for heap management, define `malloc() / free()`)
- Use **library interposition** with `LD_PRELOAD` dynamically link in your own programs
- Brief Instructions in the GNU libc manual on how to do this

# Recall: Globals in Assembly

- Long ago, advised to write following code for global variabls
  `movl  TIME_OF_DAY_SEC(%rip), %edi`
- Load is based on an offset from the Instruction Pointer `rip`
- This kind of code is generated by `gcc` in most cases for globals
- Similarly, will often see in decompiled code the following

  ```
  > objdump -d lcd_update.o
  2f2:  e8 00 00 00 00    callq  2f7 <lcd_update+0x33>
  ...
  31c:  e8 00 00 00 00    callq  321 <lcd_update+0x5d>
  ```

  which looks a little strange
- Why are both call instructions `e8 00 00 ...`?
- Both these deserve some explanation

# Relocation and PC-Relative Address

- Historically, linker would just assign a memory address to everything, global variables and functions
- Problem: forces loading program a fixed virtual memory address, decreases options available to loader/dynamic linker
- `gcc` now generates **relocatable** code by default
- All instructions must be independent of exact memory location where program is loaded
- Enables more flexible virtual memory mapping
- Loader guarantees: **distance between sections is constant**
  - `.text` might be loaded at `0x9000` or at `0x9100` by OS
  - `.text` and `.data` always `0x1000` bytes apart
  - `.text` loaded contiguously at some start address
- Addressing relative to PC gets around requires extra linker work

# Relocation Entries

- ELF files contain **relocation entries**, spots with unknown address that must be "fixed" at link time
- Function calls and global variables induce these
- Compiler inserts notes about byte locations that fixes at link time
  - Where the fix is needed
  - What symbol is needed
  - Extra arithmetic stuff
- Interested in two types of relocation entries
  - `R_X86_64_PC32`: insert address of something relative to `rip`; used for global vars, functions in same C file
  - `R_X86_64_PLT32`: insert address of a **procedure linkage table entry**; used for functions not in same C file
- Linker **inserts addresses** at positions indicated by relocation entries

# Example of Relocation Entries

```
ORIGINAL SOURCE CODE
// file: glob.c
int glob[128];
void glob_scale(int scale){ ... }

void glob_scale_add(int scale,
                    int y[])
{
  glob_scale(scale);          // 66
  for(int i=0; i<128; i++){
    glob[i] += y[i];          // 83
    printf("%d\n",glob[i]);   // e0
  }
}
```

```
RELOCATION ENTRIES
> readelf -r glob.o
Off Type            Sym + Addend
66  R_X86_64_PC32   glob_scale - 4
83  R_X86_64_PC32   glob - 4
e0  R_X86_64_PLT32  printf - 4
...
Above byte positions must have
addresses inserted by the linker
at link time. Currently those
position have 00's as placeholders
until the linker fills them in.
```

```
RELEVANT DISASSEMBLED CODE
> objdump -dx glob.o
0000000000000051 <glob_scale_add>:
  65:   e8 00 00 00 00          callq  6a             # call function
           ^^ 66: R_X86_64_PC32       glob_scale-0x4  # in same file

  80:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax # use global var
                 ^^ 83: R_X86_64_PC32  glob-0x4       # in same file

  df:   e8 00 00 00 00          callq  e4             # call function
           ^^ e0: R_X86_64_PLT32  printf-0x4          # in another file
```

# End Result: Relocatable Code

- Most ELF programs have **no load time constant addresses**
- All functions and variables (locals/globals) are referenced relative to the `rip` (program counter)
- ELF image can be loaded at an starting Virtual Memory Address and run successfully
- Will notice memory address of functions/variables change from run to run but the **difference between locations is constant**

```
> gcc -o glob_main glob_main.c glob.c
> ./glob_main
ADDRESSES
0x563a8c713060: glob variable
0x563a8c5126da: main func
0x563a8c512828: glob_scale func
0x563a8c512879: glob_scale_add func

ADDRESS DIFFERENCES
  200986: glob - main
  200838: glob - glob_scale
     14e: glob_scale - main
      51: glob_scale_add - glob_scale
```

```
> ./glob_main
ADDRESSES
0x564d3da16060: glob variable
0x564d3d8156da: main func
0x564d3d815828: glob_scale func
0x564d3d815879: glob_scale_add func

ADDRESS DIFFERENCES
  200986: glob - main
  200838: glob - glob_scale
     14e: glob_scale - main
      51: glob_scale_add - glob_scale
```

# Wait, what about that PLT thing?

- Minor performance hit for dynamically linked libraries, use of program linkage table (PLT) and global offset table (GOT)
- First call to `printf()` is expensive when it is dynamically linked
- Dynamic linker delays determining address of `printf()` until it is called
- Pseudo-code representing gcc / Linux approach to the right: clever use of 1 level of indirection and GOT table of function pointers

```
void main(){
  ...
  printf(...); // compiled to call_printf()
  ...
}

void *GOT[];   // has addresses of funcs

void call_printf(...){
  int (*func_ptr) = GOT[3]; // get func ptr
  func_ptr(...);            // call func
}

void link_printf(...){   // 1st call only
  void *printf_addr =    // use linker to
     dlsym("printf");    // find printf
  GOT[3] = printf_addr;  // save ptr later
  printf_addr(...);      // call printf
}

void *GOT[] = {   // global table
  ...
  &link_printf,   // for first printf call
  ..
}
```