SI411 Link-Load Lab Name \_\_Toren Hawk\_\_\_

Fall 2025 (AY26)

**Learning Objectives**:

- Understand the difference between static and dynamic linking.

- Use binary tools like objdump, nm, and ldd to examine executable files.

- Be able to use such tools to identify static vs. dynamic linking in pre-built executables.

Instructions:

* Use an “official” CS lab machine for this lab! (remote access okay)
* Enter each answer in the box or space provided
  + For this lab, you may download this file and type your answers or may NEATLY hand-write your answers. If it’s not easily legible, it’s not correct!
* Use your own words. **Do \*not\* copy and paste text from elsewhere.**
* **Turn in hard-copy at start of class of the due date.**
* Staple required for full credit.

Key requirements for all labs:

* Your goal is to understand what and why is happening. Therefore, your answers should be concise but thoughtful.
* For instance, if we ask “Why doesn’t this work?” do not just repeat back the error message. Instead, explain specifically what is going on.

**Lab Total**: 41 points (with 1 point extra credit)

**Part 0: Preliminaries**

Provided files

ctest1.c, ctest2.c, prog.c, mystery1

For Info: compiler flags

-Wall show all errors and warnings

-fPIC force position-independent code

-no-pie ...but don’t use “position independent execution” (breaks ltrace)

-shared produce a shared object

Setup

1. Create a folder with the above files in it.

2. Review the .c source files to see what they do.

3. Do “man gcc”, then use the / key to search for “linker”. Type / again and hit Enter to search for the next occurrence [this is just for practice searching a man page – no need to actually read things yet]

**Part 1: Static Linking**

Learn/review a really useful command

(1 pt) What does the -t argument do for ls? (man ls) \_Sort by time, newest first\_\_

Compile

The ctest source files will be used in libraries. Compile both to generate object files.

gcc -Wall -c ctest1.c ctest2.c

Create a static library Create an “object code archive”. These end in “.a”

ar -cvq libctest.a ctest1.o ctest2.o

See your object files

ls -lt

(1 pt) What file just got created? \_\_libctest.a\_\_\_\_\_\_\_\_

(1 pt) How many bytes is it? \_\_6134\_\_\_

(1 pt) Let’s call that file XYZ. Replace XYZ with the correct filename in this command:

ar tf XYZ

This shows you the library contents – take a minute to examine it.

NOTE: now that you have built a library, you must use it! In particular, you should NOT make direct use of ctest1.c and ctest2. c for the rest of part 1—use your library instead.

Compile with static linking

Try this:

gcc -no-pie -o progA prog.c

(1 pt) Why doesn’t that work?

\_because no-pie forces no dynamically linked position independent executable to be produced so the program does not have an address already allocated\_

(1 pt) Add something to the “gcc” command line to make it work (don’t change anything else). Show your addition below:

gcc -no-pie -o progA prog.c

(1 pt) What file just got created? \_\_\_\_progA\_\_\_\_\_\_\_\_\_\_

(1 pt) How many bytes is it? \_\_\_4368\_\_\_\_\_

Run the program

./progA

(1 pt) What is the output? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

View symbols with nm

man nm # skim the main idea, then come back as needed

nm progA

Note: ‘local’ vs. ‘global/external’ is NOT the same concept as static vs. dynamic. A global symbol is one that is visible to other modules.

The nm command lists the symbols (**n**a**m**ed objects) in a program. For some symbols (like main, ctest1, and ctest2) the output shows the corresponding 'RVA' within the static linked code. RVA stands for *relative virtual address*, which is a fancy way of describing the offset, in bytes, from the beginning of the executable program.

(1 pt) What is the RVA for main?

(1 pt) What “type” of symbol is main?

(1 pt) What does that mean?

(1 pt) What is the RVA for ctest1?

(1 pt) What is the RVA for ctest2?

(1 pt) Find the entry for printf. Why is there no RVA?

(1 pt) Where specifically then can the code for print be found?

(1 pt) Does progA use static linking, dynamic linking, or both? Briefly explain.

**Part 2: Dynamic Linking**

Compile

Compile the test programs into object files that can be used in dynamically link libraries.

gcc -Wall -fPIC -c ctest1.c ctest2.c

Create Library

gcc -shared -Wl,-soname,libctest.so.1 -o libctest.so.1.0 \*.o

Note: the -Wl,…..so.1 is one string since -Wl, is used to pass options to the linker; see **man gcc**.

(1 pt) What file just got created? \_\_\_\_\_\_\_\_\_

(1 pt) How many bytes is it? \_\_\_\_\_\_

Note: at this point, you might copy your “shared-object files” to a common library location (and link to create common aliases) for future use. If you have sudo/root permissions, you can do this, then link against the library in the future. It might look something like:  
sudo cp libctest.so.1.0 /usr/lib …but we don't need to do this step in this lab.

Create Library Links

This will give the linker and loader pointers to see the proper version of your library

ln -sf libctest.so.1.0 libctest.so.1

ln -sf libctest.so.1.0 libctest.so

ls -lt # examine what just got created

Compile and dynamically link

gcc -Wall -L. -no-pie prog.c -lctest -o progB

(1 pt) What does the -L option do (investigate)? \_\_\_\_\_\_\_\_\_\_

Note 1: In the previous command, there is no space between -L and the period.

Note 2: Observe out the naming convention for shared-object libraries: '-lctest' refers to a library whose name is libctest.so(.version.number).

Note 3: Shared-object libraries work the same way in Windows; they're known under the familiar extension '.dll'.

Run

./progB

(1 pt) Why doesn’t this work? \_\_\_\_\_\_\_\_\_\_

Let’s fix that:

export LD\_LIBRARY\_PATH=.

(1 pt) Run again and verify it works. Now Google LD\_LIBRARY\_PATH and explain, in your own words, why setting LD\_LIBRARY\_PATH fixes the problem:

Run again

./progB

(1 pt) Is the program output the same as it was for progA in the 'static' section?\_\_\_\_

Look at all the dynamic libraries the OS is aware of:

man ldconfig (read the description)

ldconfig -v

View symbols with nm

nm progB

(1 pt) Does main still have an RVA? If so, is it about the same as with progA?

|  |
| --- |
|  |

(1 pt) Do ctest1 and ctest2 still have an RVA? If so, is it about the same as with progA? If not, why?

|  |
| --- |
|  |

See the difference in the assembly code

objdump -M intel -d progA | grep '<ctest1>'

objdump -M intel -d progB | grep '<ctest1>'

(1 pt) What's different in the output of these two commands?

|  |
| --- |
|  |

For ONE of these cases a constant that you see as part of the assembly instruction should be the same as a constant that you already seen in an earlier step of this lab.

(1 pt) a. Is this true for progA or for progB?

(1 pt) b. Why does it make sense that this constant is known and can be “hard coded” into the executable?

|  |
| --- |
|  |

(1 pt) Static vs. dynamic linking was supposed to make a difference in the size of the executable. In this particular case, is progA or progB bigger? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(1 pt) Why is the difference so small in this particular situation? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

See the difference in ldd

man ldd

ldd progA

ldd progB

(1 pt) What is the difference in output, and why?

|  |
| --- |
|  |

See the difference in library calls

(1 pt) What does the ltrace tool show (in general)?

|  |
| --- |
|  |

Try running it against both executables.

ltrace ./progA

ltrace ./progB

(1 pt) What is the difference in output, and why?

|  |
| --- |
|  |

See the difference in dynamic relocation records

Relocation records are list of references inside an executable that have to be “fixed up” when the executable is loaded. In other words, memory addresses of things like data objects or function calls, that need to be resolved. They may be stored in a section called .reloc in the executable. Take a look to them using the following commands:

objdump -R progA

objdump -R progB

(1 pt) How is the output for progB different, and why?

|  |
| --- |
|  |

What does linking really do anyway?

Above, we created static and dynamic libraries. Suppose however that we just want to build an executable and don’t wish to create any additional libraries. We could do this:

gcc -no-pie -o progC ctest1.c ctest2.c prog.c

(1 pt) Examine progC. Is it more similar to progA or progB?

(1 pt) What technique/tool did you use to decide this?

|  |
| --- |
|  |

(1 pt) WHY is progC so similar to the earlier executable we built (progA or progB)? (what causes it to be so similar?)

|  |
| --- |
|  |

(1 pt) (small extra credit) It’s possible to make a small change to the command line shown above so that progC is actually identical to either progA or progB (verify with diff). If you discover this, write the new command line here:

Detective work – download mystery1

For mystery1:

1. (1 pt) Do this: chmod u+x mystery1  
   Now, will this executable run? If not, what specifically is missing?
2. (1 pt) Does this executable use static linking, dynamic linking, or both?

1. (1 pt) If static linking is used, what function(s) are statically linked? (ignore names starting with underscore)
2. (1 pt) If dynamic linking is used, what function(s) are dynamically linked? (ignore names starting with underscore)