IC411 Link-Load Lab Name \_\_\_\_Brandon Sipes\_\_\_\_\_\_

Fall AY2019

**Summary**: In this lab, you'll examine the difference between static and dynamic linking. You'll create both a static and a dynamic library, and link the same test code against them, observing differences in the disassembled code.

**Learning Objectives**:

- Understand the difference between static and dynamic linking.

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

**Estimated Completion Time**: 1.0-1.5 hours **Lab Weight**: 15 points

**Submission**: This lab worksheet with complete, typewritten answers. Please use a distinct font color for your answers. Thanks.

**Tasks**:

Example Files

ctest1.c

ctest2.c

prog.c

For Info: Compiler Flags

-Wall show all errors and warnings

-fPIC force position-independent code

-shared produce a shared object

Setup

Create a desktop folder called 'linking', with the above files in it. Review source files to see what they do, then:

Compile

Compile the test programs .

gcc -Wall -c ctest1.c ctest2.c

**STATIC LINKING**

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 -l

Verify library contents

ar tf libctest.a

Compile with static linking

gcc -o prog prog.c libctest.a

Run the program What is the output:

./prog

Valx=5

Valx=100

View symbols with nm

nm prog | grep ctest

nm prog | grep main

The nm command lists the symbols (named objects) in a program. The output shows the 'RVA' for the listed functions in the statically 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. The 'T' in the output indicates a symbol is in the code (text) section.

What is the address of main?

00000000000006aa

What is the address of ctest1?

0000000000000720

What is the address of ctest2?

0000000000000735

**DYNAMIC LINKING**

**Compile**

Compile the test programs into dynamically linked shared object file. Ends in “.so”

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

Create Library

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

See Library

ls -l | grep so

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

Compile and dynamically link

gcc -Wall -L. prog.c -lctest -o prog2

Note 1: In the previous command, there is no space between -L and the link location, which may be your library location. In this case, it's just the current folder, indicated by '.' .

Note 2: Check 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'.

Update the environment variable for dynamic link libraries (will affect only this shell session)

export LD\_LIBRARY\_PATH=.

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

man ldconfig (read the description)

ldconfig -v

Run

./prog2 Is the program output the same as it was for prog in the 'static' section? yes

View symbols with nm

nm prog2 | grep ctest

Note: in the nm output, 'U' indicates a symbol is undefined

Do you see any RVAs in this listing? Why or why not?

No. Because everything is dynamically linked so there does not have to be an offset of where it is relative to the start of the program.

See the difference in the assembly code

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

objdump -M intel -d prog2 | grep '<ctest1>'

What's different in the output of these two commands?

The first one is simply a call to the function ctest1 that is in the file. The second one is a jump to the memory location of where to find the link to ctest1.

Why?

Because in a statically linked program the code is added in much in the same way as a function is. Where as in a dynamically linked program, it has to go to the Global Offset Table to get the address of the program.

See the difference in ldd

If you're not sure what ldd shows you, try 'man ldd'

ldd prog

ldd prog2

What is the difference in output, and why?

Prog2 requires libctest.so.1 because it needs a reference of where to find the ctest code.

See the difference in library calls

man ltrace

Read the man entry. What does ltrace show?

ltrace shows the library calls.

Try running it against both executables.

ltrace ./prog

ltrace ./prog2

What is the difference in output, and why?

prog2 additionally makes library calls ctest1 and ctest2. Instead of prog’s function calls to ctest1 and ctest2.

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 prog

objdump -R prog2

How is the output for prog2 different, and why?

prog2 also has the records for ctest1 and ctest2 because it needs to have a link to where they are so that the program can jump to them.

Step through each program with a debugger to follow the Instruction Pointer

Execute the following commands.

gdb -q prog

(gdb) b main // set a breakpoint at main()

(gdb) b ctest1 // set a breakpoint at ctest1()

(gdb) b ctest2 // set a breakpoint at ctest2()

(gdb) r // run prog from start to 1st breakpoint, main()

(gdb) info reg // look at registers.

rip value: \_0x5555555546ae\_

(gdb) c // continue to 2nd breakpoint, ctest1()

(gdb) info reg // look at registers.

rip value: \_0x555555554724\_

(gdb) c // continue to 3rd breakpoint, ctest2()

(gdb) info reg // look at registers.

rip value: \_0x555555554739\_

(gdb) c // continue to end, exit normally

(gdb) q // quit gdb

Now do the same with prog2:

Execute the following commands.

gdb -q prog2

(gdb) b main // set a breakpoint at main()

(gdb) b ctest1 // set a breakpoint at ctest1()

(gdb) b ctest2 // set a breakpoint at ctest2()

(gdb) r // run prog from start to 1st breakpoint, main()

(gdb) info reg // look at registers.

rip value: \_0x55555555480e\_

(gdb) c // continue to 2nd breakpoint, ctest1()

(gdb) info reg // look at registers.

rip value: \_0x7ffff7bd35ae\_

(gdb) c // continue to 3rd breakpoint, ctest2()

(gdb) info reg // look at registers.

rip value: \_0x7ffff7bd35c3\_

(gdb) c // continue to end, exit normally

(gdb) q // quit gdb

Note the location of the IP (instruction pointer, here rip) when you debug through prog2! How does it differ from prog1, and why?

The instruction pointer was relativly the same for main in both of the programs but in prog2, for ctest1 and ctest2 the instruction pointer was no longer in the same general location.