

CSIS0230A Principles of Operating Systems (Class A)

Tutorial 8

Static and shared libraries

In this tutorial we will build an executable from multiple C source files. We will find that it can be done with or without libraries, and will examine the trade-offs between using static and shared libraries. To see whether the library is actually “shared” during program execution, we also need to read the page tables of the processes. So the kernel we prepare during the last tutorial is put into `/home/os/bzImage`, and the **first thing that you should do is to boot this kernel**. Then follow the steps below.

Part 1: Executable programs and its memory map

1. In the directory `num_str` there are two C files `num_str_tab.c` and `num_str.c`, together with their header files. The first C file contains a table with the names of some numbers, and the second contains a two functions `num_name_str()` and `num_ord_str()` that use the table to convert numbers to their names or ordinal names. Both uses the function `real_num_to_str()` to achieve the task. The first step is to compile these files, **independently**. Remember that these files are not complete C programs: they don't have a `main()` function. So you must tell the compiler to stop after producing the `.o` files.
2. In the home directory there are two C files `test_name.c` and `test_ord.c`, to test each of the functions in `num_str.c`. Compile them and link them to the `.o` files within the `num_str` directory (you will have to use `-I` to ask `gcc` to look for header files in `num_str/`). Make sure the program can run and produce reasonable output. Record their file sizes.
3. Read the object file `num_str_tab.o` in the `num_str` directory, using `readelf -a`. Note that there are a lot of relocation entries with no symbol name. Inspect `num_str_tab.c` to guess what they are. Explain why they need to be relocated. (Hint: if needed, try to compile the C file to assembly and inspect it.)
4. Should these relocations be resolved once the executable file is produced? Verify your answer by viewing the executable using `readelf`.

Part 2: Running the program

1. Examine the executable file using `readelf`, paying special attention to the program header. Compare it with the `/proc/NNN/maps` of an executing copy of the program. (NNN is the PID of the process running the program, as usual. Find it using `ps x`.) Are they the same?
2. Now we run two copies of the same program (say, `num_name`). Can their physical memory be shared? Find the answer by using the `pagetable` program when running two copies of the program. Explain your observation. Where the relocation information is stored? (Check using `readelf` to find the addresses of the relocations entries.)
3. Suppose we run the two different programs, which shares two common `.o` files. Can the physical memory be shared for the common object files used? Explain.
4. Now link the program again, this time statically (using `-static`). Compare the size of the executable program with the size of the original executable. Repeat the last step.

Part 3: Building a static library

1. Use `ar` to build a static library containing the two object files within the `num_str` directory. Call the resulting library `num_str.a`. Move the library to the home directory, and compile the test programs again. Does the program executable size change?
2. Note that the `test_name` program does not use the `num_ord_str()` function, and the `test_ord` program does not use the `num_name_str()` function. So there are useless code in both executable programs. This is because if an object file is included, all its functions are included. By splitting the `num_str.c` file into three parts, rebuild the library and compile the programs against the new library, try to reduce the executable file size.

Part 4: Building a shared library

1. Use `gcc` with the **-shared** flag to build a shared library containing the object files within the `num_str` directory. Call the resulting library `num_str.so`. Move the library to the home directory, and compile the two test programs with it. Does the program executable size change?
2. Run the program. (You will have to type `export LD_LIBRARY_PATH=.` before running the program, since the shared library is in non-standard path.) Read the memory map of the program. Note the change in the virtual memory allocation. Is it appropriate to use shared-library for this library?
3. Now we would like to check whether the memory within the shared library is really shared by two processes, running either the same or different test programs. Do it by using the `pagetable` program.
4. The result of the last step is due to the fact that the code is not position independent (PIC), requiring relocations to be done within the code segment. Try to compile the files in the `num_str` directory again, this time with the flag `-fPIC`. Rebuild the shared library and repeat the last step.
5. Do you expect the relocations due to the `num_str_tab` to remain in the shared library? Check your answer by inspecting the shared library using `readelf`.
6. Compare the assembly code generated by compiling the `num_str.c` file with and without `-fPIC`. What are the differences?