BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI (RAJASTHAN)

IS F462 – Network Programming

- What are shared libraries?
  - Shared libraries are the techniques for placing library functions into a single unit that can be shared by multiple processes at run time.
  - One way of building a program is simply to compile each of the source files to produce corresponding object files, and then link all of these object files together to produce the executable program.

  - `$ gcc -g -c program.c module1.c module2.c module3.c`
  - `$ gcc -g -o program program.o module1.o mudule2.o module3.o`

- Static Libraries
  - Static libraries, also known as `archives`, provide the following benefits :
    - We can place a commonly used object files into a single library file that can then be used to build multiple executables, without needing to recompile the original source files when building each applications.
    - Link commands become simpler. Instead of listing a long series of objects files on the link command line, we specify just the name of the static library. Linker then knows how to search the static library and extract the objects required by the executable.

  - Creating and maintaining a static shared library :
    - In simple words a static library is simply a file holding copies of all the object files added to it. The archive also records various attributes of each of its component objects files, including file permissions, numeric user and group ids and last modification time.
    - By convention static libraries have the name <lib_name>.a
      - `$ ar options archive object_file.`

(Find the files named `checkPrime.c`, `main.c` and `checkPrime.h` in the programs folder) Use the following commands for creating a static library.

- `$ gcc -g -c checkPrime.c`
- `$ ar r lib.a checkPrime.o`
- `$ rm checkPrime.o`
- `$ gcc -g -c main.c`

- `$ gcc -g -o main main.o lib.a -lm`

Q . Try creating your own module named `findGCD` and integrate it with the `main.c` used above. Using the concepts of static library.

- Command for listing the table of contents of the archive file.
  - `$ ar tv <lib_file>.a`
- Delete a named module from the archive
  - `$ ar d <lib_file>.a  mod.o`

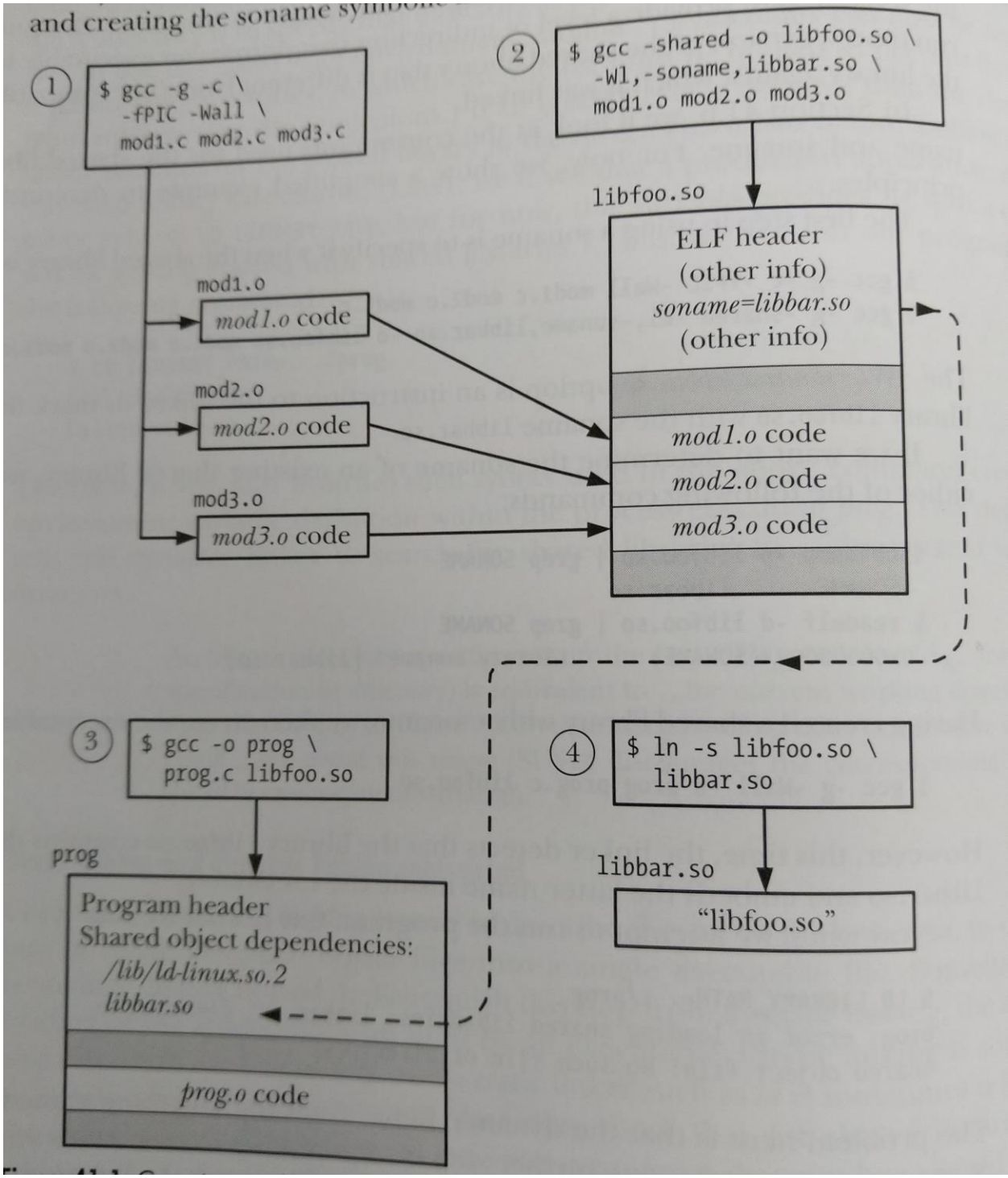
Shared Libraries

- When a program is built by linking against a static library, the resulting executable file includes copies of all the object files that were linked into the program.
- When multiple executables use the same object modules, each executable has its own copy of the object modules.
- This leads to redundancy of code and has the following disadvantage
  - Disk space is wasted
  - Virtual memory is wasted at the time of running
  - If some change is required to any one of the object modules in a static library, then all the executables using that modules must be relinked in order to incorporate the changes.

- Creating a shared library
  - Since the executable file no longer contains the copies of the object files that it requires, it must have some mechanism for identifying the shared library that it needs at run time.
  - This is done by embedding the name of the shared library inside the executable during the link phase.
  - The list of programs shared library dependencies is referred to as its dynamic dependency list.

  - `$ gcc -g -c -fPIC -Wall module1.c module2.c module3.c`
  - `$ gcc -g -shared -o <library>.so module1.o module2.o module3.o`

  - Embedding the name of the library inside the executable happens automatically when we link our program with the shared library.
  - `$ gcc -g -Wall -o prog prog.c <library>.so`
  - $ ./prog
  - ./prog : error in loading shared libraries



- Dynamic linking
  - Resolving the embedded library name at the run time. This task is done by dynamic linker or run-time linker.
  - The dynamic linker is itself a shared library, named `/lib/ld-linux.so.2`
  - The dynamic linker examines the list of shared libraries required by a program and uses a set of predefined rules in order to find the library files

in the file system. Some of these rules specify the set of standard directories in which the shared libraries normally reside.
    - For example many shared libraries reside in the `/lib` and `/usr/lib`
    - The error message above occurs because our library resides in the current working directory, which is not part of the standard list of the dynamic linker.

    - The `LD_LIBRARY_PATH` environment variable
        - One way of informing the dynamic linker that a shared library resides resides in a nonstandard directory is to specify that directory as part of a colon-separated list of directories in the LD_LIBRARY_PATH environment variable.
        - `$LD_LIBRARY_PATH=. ./prog`

- Installing the shared library
    - Copy the shared library to one of these directories ie `/usr/lib or /lib or /usr/local/lib.`
    - After this we can run the program normally ie no need of setting the LD_LIBRARY_PATH variable.

- Try running the following commands
    - `$ gcc -g -c -fPIC checkPrime.c`
    - `$ gcc -g -shared -o libCheckPrime.so checkPrime.o`
    - `$ gcc -g -o main main.c libCheckPrime.so -lm`
    - `LD_LIBRARY_PATH=. ./main`
    - `sudo mv libCheckPrime.so /usr/lib`
    - `./main`

Q . Try creating your own module named `findGCD` of two numbers and integrate it with the `main. Using the concepts of the shared library.`

Dynamically Loaded Libraries

- When as executable starts, the dynamic linker loads all of the shared libraries in the program's dynamic dependency list. But sometimes it is useful to load libraries at a later time.
- The dlopen API enables a program to open a shared library at the run time. Search for a function by name in that library, and then call a function.

- Opening the shared library
    - `#include <dlfcn.h>`
    - `Void *dlopen(const char* libfilename, int flags);`

        - Returns library handler on success, or NULL on error.
    - If the shared library specified by `libfilename` contains dependencies on the other shared libraries `dlopen()` also automatically loads those libraries.
    - The flag argument is a bitmask that must include exactly one of the constants `RTLD_LAZY` or `RTLD_NOW.`
    - `RTLD_LAZY`
        - Undefined function symbols in the library should be resolved only as the code is executed. If the piece of code requiring a particular symbol is not executed, that symbol is never resolved.
        - Lazy resolution is performed only for function references, references to variables are always resolved immediately.
    - `RTLD_NOW`
        - All the undefined symbols in the library should be immediately resolved before `dlopen()` completes, regardless of whether they will ever be required. As a consequence, opening the library is slower, but any potential undefined function symbol errors are detected immediately instead of at some time later.

    - `$ gcc -o dynload dynload.c -ldl`
    - `$ gcc -g -c -fPIC function.c`
    - `$ gcc -h -shared -o function.so function.o`
    - `$  ./dynload ./function.so myFunction`

- First argument is the library name and the second one is the program name.
- `dysym() return the address of the symbol.`