Link 7.A Static Linking

Young W. Lim

2019-01-22 Tue

Outline

Based on

"Self-service Linux: Mastering the Art of Problem Determination",
Mark Wilding
"Computer Architecture: A Programmer's Perspective",
Bryant & O'Hallaron

Compling 32-bit program on 64-bit gcc

- gcc -v
- gcc -m32 t.c
- sudo apt-get install gcc-multilib
- sudo apt-get install g++-multilib
- gcc-multilib
- g++-multilib
- gcc -m32
- objdump -m i386

TOC: Static Linking Examples

1. addvec.c and mutvec.c
2. libvector.a
3. main.c
4. p

addvec.c and multvec.c !

```
/*::::: addvec.c ::::::::::::::::::::::::::*/
void addvec(int *x, int *y, int *z, int n)
{
  int i;

  for (i=0; i<n; i++)
    z[i] = x[i] + y[i];

}

/*::::: multvec.c ::::::::::::::::::::::::::*/
void multvec(int *x, int *y, int *z, int n)
{
  int i;

  for (i=0; i<n; i++)
    z[i] = x[i] * y[i];

}
```

libvector.a

- gcc -c addvec.c
  - addvec.o
- gcc -c multvec.c
  - multvec.o
- ar rcs libvector.a addvec.o multvec.o
  - libvector.a

main.c

```
/*::::: vector.h ::::::::::::::::::::::::::::*/
void addvec(int *x, int *y, int *z, int n);
void multvec(int *x, int *y, int *z, int n);


/*::::: main.c ::::::::::::::::::::::::::::::*/
#include <stdio.h>
#include "vector.h"

int x[2] = { 1, 2};
int y[2] = { 3, 4};
int z[2];


int main() {

  addvec(x, y, z, 2);
  printf("z= [%d %d]\n", z[0], z[1]);

}
```

p

- gcc -O2 -c main.c
  - main.o
- gcc -static -o p main.o ./libvector.a
  - p
- ./p
  - z= [4 6]

TOC: Linking with Static Libraries

Static libraries

- assumption
  - the linker reads a colleciton of relocatable object files and links them together into an output file
- in practice
  - a static library is just a *packaging* mechanism that *organizes* related object modules into a single file
  - object files are supplied as *inputs* to the linker
  - the linker *copies* those object modules into a library
  - the application program *references* those object modules in the library

Object files - input to a static library

- object files are merely collection of blocks of bytes
  - some blocks contain program code (`.data`)
  - others contain program data (`.text`)
  - others contain data structures
    that guide the linker and the loader

Static linking process

- static linker *concatenates* blocks together
  decides on *run-time locations* for the concatenated blocks
  modifies various locations within the code and data blocks
- static linker has a *minimal understanding* of the target machine
- the compiler and the assembler that generate object files
  has already done most of the work

Static linker

- the static linker `ld` takes as input
  - a collection of relocatable object files
  - command line arguments
- geneartes as output
  - a fully linked executable object file
  - that can be loaded and run
- performs two tasks
  - symbol resolution
  - relocation

Symbol resolution - static linker's task

- object files define and reference symbols
- the purpose of symbol resolution is
  to associate each symbol reference
  with exactly one symbol definition

- function calls and the function definition
- global variable accesses and the global variable definition

Relocation - static linker's task

- compilers and assembler generates
  code and data sections that start at address zero
- the linker relocates these sections
  by associating a memory location
  with each symbol definition
- modifying all of the references to those symbols
  so that they point to this memory location

ANSI C `libc.a`

- `libc.a` library
  - an extensive collection of standard I/O
    - `atoi`, `pritnf`, `scanf`
  - string manipulation
    - `strcpy`
  - integer math functions
    - `random`
- `libm.a` library
  - an extensive collection of floating-point math functions
    - `sin`, `cos`, `sqrt`

advantages of static libraries (1)

- related functions can be compiled
  into separate object modules
  then packaged in a single static library file
- application program can then use
  any of the functions defined in the library
  by specifying the file name on the command line
- `gcc main.c /usr/lib/libc.a /usr/lib/libm.a`

advantages of static libraries (2)

- at link time, the linker will
  only copy the object modules
  that are referenced by the program
  which reduces the size of the executable
  on disk and in memory
- the application programmers only need
  to include the names of a few library files
- C compiler drivers always pass libc.a to the linker
  so the reference to `libc.a` is unnecessary

Unix archive

- an archive on Unix systems
  - static libraries are stored on disk
    in a particular file format : archive
  - a collection of concatenated relocatable object files
  - a header describes the size and locaton
    of each member object file
  - archive filenames are denoted with the `.a` suffix

Linking with static library examples (1)

- `main2.c` ⇒ `main2.o`
  - translators (`cpp`, `ccl`, `as`)
- `libvector.a` → `addvec.o`
- `libc.a` → `printf.o`
  - any other modules called by `printf.o` also
- `main2.o`, `addvec.o`, `printf.o`, other object files ⇒ p2
  - linker (`ld`)

```
void addvec(int *x, int *y)
```

Linking with static library examples (2)

- source files : `main2.c`, `vector.h`
- static libraries : `libvector.a`, `libc.a`
- relocatable object files : `main2.o`, `addvec.o`, `printf.o`, any other modules called by `printf.o`
- fully linked executable object file : `p2`

TOC: Resolving refereces with Static Libraries

Symbol resolution phase

- during the symbol resolution phase, the linker scans the relocatable object files and archives left to right ($\Rightarrow$) in the same order that they appear on the command line
- the linker maintains
  - a set E of relocatable object files that will be merged to form the executable
  - a set U of unresolved symbols symbols referred to but not yet defined
  - a set D of symbols that have been defined in the previous input files
  - initially, all the set E, U, D are empty

for each input file f

- for each input file f on the command line,
  the linker determines if f is an object file
  or an archive
- for input file f, the linker
  - adds f to E
  - updates U and D
    to reflect the symbol definitions and references in f
    and proceeds to the next input file

for each input archive f

- the linker attempts to match
  the unresolved symbols in U
  against the symbols defined
  by the members of the archive
- any member object files which are
  not contained in E are discarded
  and the linker proceeds to the next input file

for each member in the archive *f*

- if some archive member *m* defines a symbol
  that resolves a reference in *U*
  then *m* is added to *E*
- then the linker updates *U* and *D*
  to reflect the symbol definitions and references in *m*
- this process iterates over
  the member of object files in *f*
  until a fixed point is reached
  where *U* and *D* no longer change

for undefined symbol set U is not empty

- if U is nonempty when the linker finishes
  scanning the input files on the command line,
  it prints an error and terminates
- otherwise, it merges and relocates
  the object files in E
  to build the output executable file

Link time error

- the ordering of libraries and object files
  on the command line is significant
- if the library that defines a symbol
  appear on the command line
  before the object file that references the symbol
  then the reference will not be resolved
  and the linking will fail

```
void addvec(int *x, int *y)
```

Linke time error example

- when `libvector.a` is processed, U is empty
  therefore, no member object files from `libvector.a`
  is added to E
- the reference to `addvec` is never resolved
  error message

```
gcc -static ./libvector.a main2.c

in function 'main' :
undefined reference to 'addvec'

gcc -static main2.c ./libvector.a
```

Ordering libraries on the command line

- if the members of different libraries are independent
  (no member references a symbol defined by other member)
  then the libraries can be placed
  at the end of the command line in arbitrary order
- if they not independent, they must be ordered
  so that for each symbol s that is referenced
  externally by a member of an archive,
  at least one definition of s
  follows a reference to s

Ordering libraries on the command line example (1)

- `foo.c` calls functions in `libx.a` and `libz.a` which call functions in `liby.a`
- `libx.a` and `libz.a` must precedes `liby.a` on the command line

```
gcc foo.c libx.a libz.a liby.a
```

Ordering libraries on the command line example (2)

- libraries can be repeated on the command line
  if necessary to satisfy the dependence requirements
- `foo.c` calls a function in `libx.a`
  which calls a function in `liby.a`
  which again calls a function in `libx.a`
- then `libx.a` must be repeated on the command line

```
gcc foo.c libx.a liby.a libx.a
```

- alternatively, `libx.a` and `liby.a` can be combined
  into a single archive