Link 8.A Dynamic Linking

Young W. Lim

2019-02-07 Thr

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: Dynamic linking with shared library example

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];

}
```

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]);

}
```

compiler flags for dynamic linking

- `-fPIC` flag directs the compiler
  to generate position independent code
- `-shared` flag directs the linker
  to create a shared object file

compiling commands

- ```
  gcc -g -m32 -Wall -fPIC -c addvec.c
  gcc -g -m32 -Wall -fPIC -c multvec.c
  gcc -g -m32 -shared -o libvector.so addvec.o multvec.o

  gcc -g -m32 -Wall -c main.c
  gcc -g -m32 -o dynamicp main.o ./libvecotr.so

  LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./
  export LD_LIBRARY_PATH
  ```
  ```

analyzing commands

- ```
  $ readelf --segments nmain_dyn.out
  $ objdump -d -s dynamicp
  $ objdump -d -j .plt.got dynamicp
  $ objdump -d -j .plt.got dynamicp
  $ gdb ... disas, x/a 0x...., c
  $ cat /proc/<pid>/map
  ```
  ```

shared object `libvector.so`

1. `gcc -m32 -c -fPIC addvec.c`
   - `addvec.o`
2. `gcc -m32 -c -fPIC multvec.c`
   - `multvec.o`
3. `gcc -m32 -shared -o libvector.so addvec.o multvec.o`
   - `libvector.so`

executable object p

1. `gcc -m32 -c main.c`
    - `main.o`
2. `gcc -m32 -o p main.o ./libvector.so`
    - `p`
3. `LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./`
4. `export LD_LIBRARY_PATH`
    - `./p`

Steps of dynamic linking

1. main2.c, vector.h $\Rightarrow$ main2.o
   - translators (cpp, ccl, as)
1. main2.o, libc.so, libvector.so $\Rightarrow$ p2
   - linker (ld)
1. p2 $\Rightarrow$ partially linked p2 in memory
   - loader (execve)
1. p2, libc.so, libvector.so $\Rightarrow$ fully linked executable in memory
   - dynamic linker (ld-linux.so)

Inputs and outputs dynamic linking steps

1. Linker `ld` inputs
   - relocatble object file :
     `main2.o`
   - relocation and symbol table information :
     `libc.so`, `libvector.so`
2. Loader `execve` input
   - partially linked executable object file :
     `p2`
3. Dynamic linker `ld-linux.so` inputs
   - loaded
     `p2`
   - code and data :
     `libc.so`, `libvector.so`
4. fully linke executable in memory

swap.c

```
/*::::: swap.c ::::::::::::::::::::::::::::*/
extern int buf[];

int *p0 = &buf[0];
int *p1;

void swap()
{
  int tmp;

  p1 = &buf[1];

  tmp = *p0;
  *p0 = *p1;
  *p1 = tmp;

}
```

main.c

```
/*::::: main.c ::::::::::::::::::::::::::::::*/
void swap();

int buf[2]  = {1, 2};

int main()
{
  swap();

  return 0;
}
```

compiling commands

- ```
  gcc -m32 -Wall -fPIC -c swap.c -o swap_pic.o
  gcc -shared -m32 -o libswap.so swap_pic.o

  gcc -m32 -Wall -c main.c
  gcc -m32 -o swap_dyn.out main.o ./libswap.so

  LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./
  export LD_LIBRARY_PATH
  ```
  ```

analyzing commands

- ```
  $ readelf --segments swap_dyn.out
  $ objdump -d -s swap_dyn.out
  $ objdump -d -j .plt.got swap_dyn.out
  $ objdump -d -j .plt.got swap_dyn.out
  $ gdb ... disas, x/a 0x...., c
  $ cat /proc/<pid>/map
  ```
  ```

TOC: Shared Libraries

Shared libraries

- an object module
  - that can be loaded at run time
  - at an *arbitrary* memory address
  - linked with a *program in memory*
- also referred as shared object
  with .so suffix
- dynamic liking is performed by a dynamic linker
  contained in the ld-linux.so interpreter
- corresponds to DLLs (Dynamic Link Libraries) on MS Window

Shared libraries - only a single copy

- a sigle copy of `.so` file :
  there is exactly <u>one</u> `.so` file
  for a particular library
- in case of static libraries
  the contents are copied

Shared libraries - shared in two ways

- shared in two different ways
- the code and data in a `.so` file are shared by *all the executable files* that reference the library
- a single copy of the `.text` section of a shared library in memory can be shared by *different running processes*

TOC: Dynamic Linking

1. First link statistically and then link dynamically
2. The first partial link (static)
3. The second complete link (dynamic)
4. The second complete link (relocation)
5. execution of dynamically linked file

First link statically and then link dynamically

- basic idea :
  - link some thing *statically*
    when the executable file is *created*
  - then complete the linking process *dynamically*
    when the program is *loaded* into memory

The first partial link (static)

- none of the code or data sections are actually copied from `libvector.so` into the executable file
- the linker copies some information about
  - relocation
  - symbol table
- this will allow references to code and data in `libvector.so` to be resolved at run time

The second complete link (dynamic)

- when the loader loads and runs the executable
  it loads the *partially* linked executable
- if the executable contains `.interp` section (interpreter)
  which contains the path name of the dynamic linker
- the dynamic linker itself is a shared object `ld-linux.so`
- instead of passing control to the application
- the loader loads and runs the dynamic linker

The second complete link (relocation)

- shared libraries are loaded in the area starting at address 0x40000000

1. relocating the text and data of libc.so into some memory segment
1. relocating the text and the data of libvector.so into another memory segment
2. relocating any references in p2 to symbols defned by `libc.so` and `libvector.so`

execution of dynamically linked file

- finally, the dynamic linker passes control to the application
- from this point, the locations of
  the shared libraries are fixed
  and do not change during execution of the program

TOC: Compiler options and paths for dynamic linking

-fPIC compile option for dynamic linking

- Generate position-independent code (PIC) suitable for use in a shared library, if supported for the target machine.
- PIC code accesses all constant addresses through a global offset table (GOT).
- The dynamic loader resolves the GOT entries when the program starts

-fpic vs -fPIC compile options

- the dynamic loader is not part of GCC;
  it is part of the operating system.
- If the GOT size for the linked executable exceeds
  a machine-specific *maximum size*, -fpic does not work;
  in that case, recompile with -fPIC instead.
- -fno-pic suppress producing a position independent object

-fno-pic compile option

- `-fno-pic` suppress producing a position independent object
  - does not use the GOT for global variables
  - $\mathtt{R}_3 86_3 2$ relocation type is used
    instead of $\mathtt{R}_3 86_G OT32X$

PIC pseduo-assembly examples

- PIC : this would work whether the code was at address 100 or 1000
  CURENT+10 : pc-relative addressing

```
100: COMPARE REG1, REG2
101: JUMP_IF_EQUAL CURRENT+10
...
111: NOP
```

- Non-PIC : this will only work if the code is at address 100
  111 : absolute addressing

```
100: COMPARE REG1, REG2
101: JUMP_IF_EQUAL 111
...
111: NOP
```

https://stackoverflow.com/questions/5311515/gcc-fpic-option

PIC code and data section characteristics

- The code section
  - no absolute addresses that need relocation
  - only self relative addresses.
- The data section
  - not shared between multiple processes
    because it often contains writeable data.
  - contain pointers or addresses that need relocation.

`https://stackoverflow.com/questions/5311515/gcc-fpic-option`

PIC public function and data characteristics

- All public functions and public data can be overridden in Linux.
- If a function in the main executable has the same name
  as a function in a shared object,
  then the version in main will take precedence,
  not only when called from main,
  but also when called from the shared object.

- when a global variable in main has the same name
  as a global variable in the shared object,
  then the instance in main will be used,
  even when accessed from the shared object.

`https://stackoverflow.com/questions/5311515/gcc-fpic-option`

-shared flag for dynamic linking

- -shared
  - Create a shared library.
  - This is currently only supported on ELF, XCOFF and SunOS platforms.
- -soname=name
  - When creating an ELF shared object, set the internal DT_SONAME field to the specified name.
  - When an executable is linked with a shared object which has a DT_SONAME field, then when the executable is run the dynamic linker will attempt to load the shared object specified by the DT_SONAME field rather than the using the file name given to the linker.

-static linker option

- -static
  - Do not link against shared libraries.
  - You may use this option multiple times on the command line:
  - it affects library searching for -l options which follow it.
  - This option also implies --unresolved-symbols=report-all.
  - This option can be used with -shared.
    - Doing so means that a shared library is being created but that all of the library's external references must be resolved by pulling in entries from static libraries.
  - can observe absolute addresses for external global variables as with -no-pie

-no-pie linker option

- -no-pie
  - not produce a position independent executable
    by default, a position independent executable is produced
  - can observe absolute addresses for external global variables

locating shared libraries (1)

1. Any directories specified by `-rpath-link` options.
   - only effective at link time
1. Any directories specified by `rpath` options.
   - used at runtime
   - supported by native linkers
   - supported by cross linkers
     that are configured with `--with-systroot`
1. On an ELF system, for native linkers
   if the -rpath and -rpath-link options were not used
   search the contents of the environment variable
   `LD_RUN_PATH`

locating shared libraries (2)

1. On SunOS, if the `-rpath` option was not used, search any directories specified using `-L` options.

1. For a native linker, search the contents of the environment variable `LD_LIBRARY_PATH`

1. For a native ELF linker, the directories in `DT_RUNPATH` or `DT_RPATH` of a shared library are searched for shared libraries needed by it. The `DT_RPATH` entries are ignored if `DT_RUNPATH` entries exist.

locating shared libraries (3)

1. The default directories,
   normally `/lib` and `/usr/lib`.

1. For a native linker on an ELF system,
   if the file `/etc/ld.so.conf` exists,
   the list of directories found in that file.

Some links

https://stackoverflow.com/questions/25084855/
how-does-gcc-shared-option-affect-the-output
https://unix.stackexchange.com/questions/475/
how-do-so-shared-object-numbers-work
https://stackoverflow.com/questions/12237282/
whats-the-difference-between-so-la-and-a-library-files

TOC: Loading and linking shared libraries from Applications

Dynamic Linker vs. Applications

- the dynamic linker loads and links shared libraries *when application is loaded*, just before it executes
- an applications can also request the dynamic linker to load and link arbitrary shared libraries *while the application is running* without having to link in the applications against those libraries at compile time

Dynamic Linker Interface

```
#include <dlfcn.h>

void *dlopen(const char *filename, int flag);
      returns ptr to handle if OK, NULL on error

void *dlsym(void *handle, char *symbol);
      returns ptr to symbol if OK, NULL on error

int dlclose (void *handle);
      returns zero if OK, -1 on error

const char *delerror(void);
      returns error message if previous call
      to dlopen, dlsym, dlclose failed
      NULL if previous call is OK
```

dlopen (1)

```
void *dlopen(const char *filename, int flag)
```

- loads and links the shared library `filename`
- the external symbols in `filename` are resolved
  using libraries previously opened with the `RTLD_GLOBAL` flag
- if the current was compiled with `rdynamic` flag,
  then its global symbols are also available for symbol resolution

dlopen (2)

```
void *dlopen(const char *filename, int flag);
```

- the `flag` argument must include
  - `RTLD_NOW`
    tells the linker to resolve references immediately
  - `RTLD_LAZY`
    tells the linker to defer symbol resolution
    until the code from the library is executed
  - `RTLD_GLOBAL` flag can be or'ed

dlsym

```
void *dlsym(void *handle, char *symbol);
```

- inputs
  - a handle to a previously opened shared library
  - a symbol name
- returns the address of the symbol if it exists or NULL otherwise

int dlclose (void *handle);

- unloads the shared library
  if no other shared libraries are still using it

dlerror

const char *delerror(void);

- returns a string describing the most recent error
  that occurred as a result of calling
  `dlopen`, `dlsym`, `dlclose`
  or `NULL` if no error occurred

an example for an application's dynamic linking (1)

```
#include <stdio.h>
#include <dlfcn.h>

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

int main() {
  void *handle;
  void (*addvec) (int*, int*, int*, int);
  char *error;

  handle = dlopen("./libvector.so", RTLD_LAZY);
  if (!handle) {
    fprintf(stderr, "%s\n", dlerror());
    exit(1);
  }
```

an example for an application's dynamic linking (2)

```
  addvec = dlsym(handle, "addvec");
  if ((error = dlerror()) != NULL) {
    fprintf(stderr, "%s\n", error);
    exit(1);
  }

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

  if (dlclose(handle) < 0) {
    fprintf("stderr, "%s\n", dlerror());
    exit(1);
  }

  return 0;
}
```

- declaration

```
void *handle;
void (*addvec) (int*, int*, int*, int);
char *error;
```

- loading a shared library

```
handle = dlopen("./libvector.so", RTLD_LAZY) ;
```

- locating address of a fuction

```
addvec = dlsym(handle, "addvec") ;
```

- unloading the shared library

```
dlclose(handle) ;
```

compiler options

- `#include <dlfcn.h>`
- `-ldl`

- `gcc -o p3 dlex.c -ldl`