Dynamic Linking and Loading(1A)

Young Won Lim
11/20/18

Copyright (c) 2010-2018 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice.

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];

}
```

gcc -shared -fPIC -o libvector**.so** addvec.c multvec.c

"Computer Architecture: A Programmer's Perspective", Bryant & O'Hallaron

main.c

```
/*::::: 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]);
}
```

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

gcc -O2 -c main2.c

gcc -o p2 main2.o **./libvector.so**

"Computer Architecture: A Programmer's Perspective", Bryant & O'Hallaron

Shared Library

gcc -shared -fPIC -o libvector.so addvec.c multvec.c

gcc -O2 -c main2.c

gcc -o p2 main2.o **./libvector.so**

-shared : Produce a shared object which can then be linked with other objects to form an executable. Not all systems support this option. For predictable results, you must also specify the same set of options used for compilation (-fpic, -fPIC, or model suboptions) when you specify this linker option

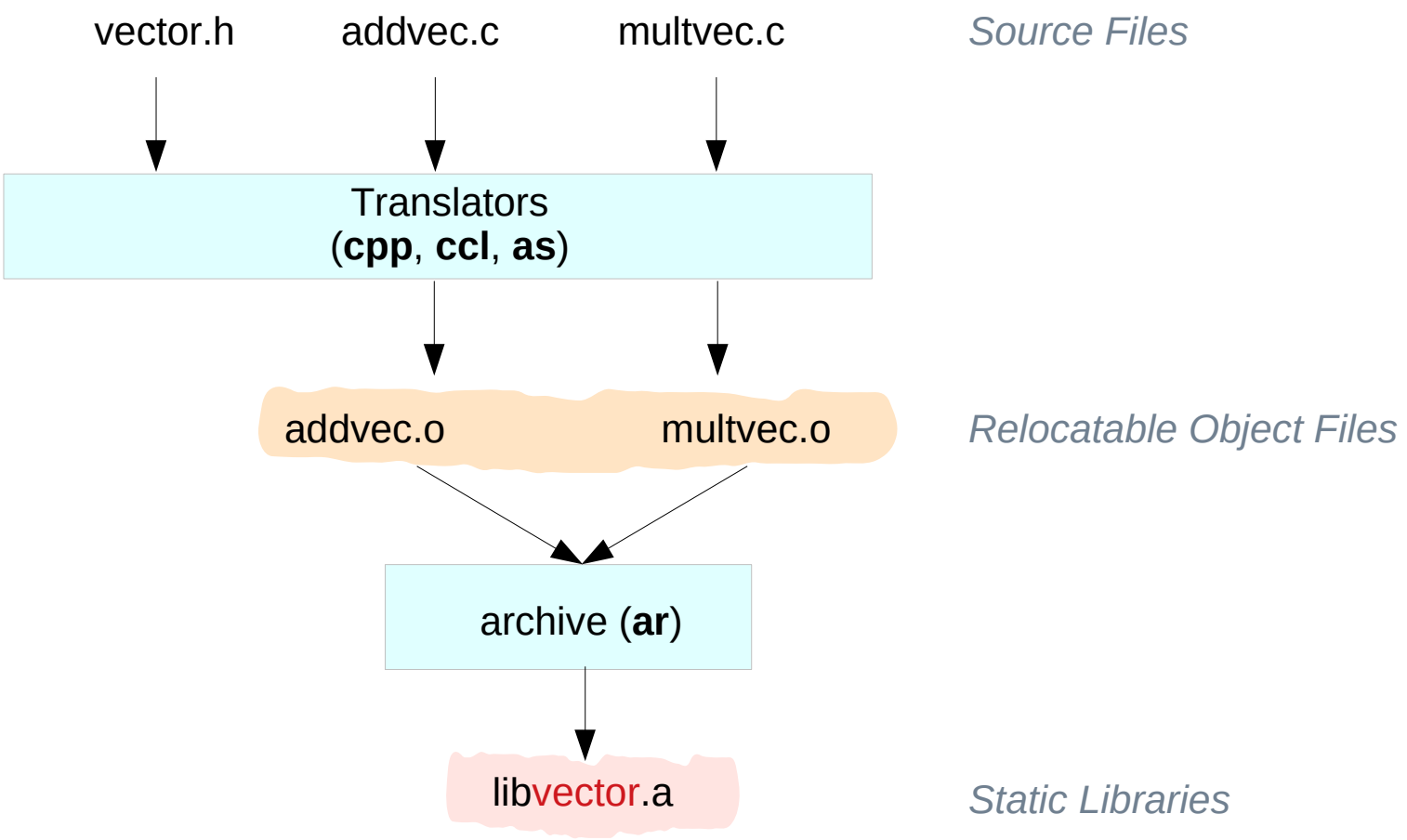
linker option

-fPIC : If supported for the target machine, emit position-independent code, suitable for dynamic linking and avoiding any limit on the size of the global offset table. Position-independent code requires special support, and therefore works only on certain machines.

compiler option

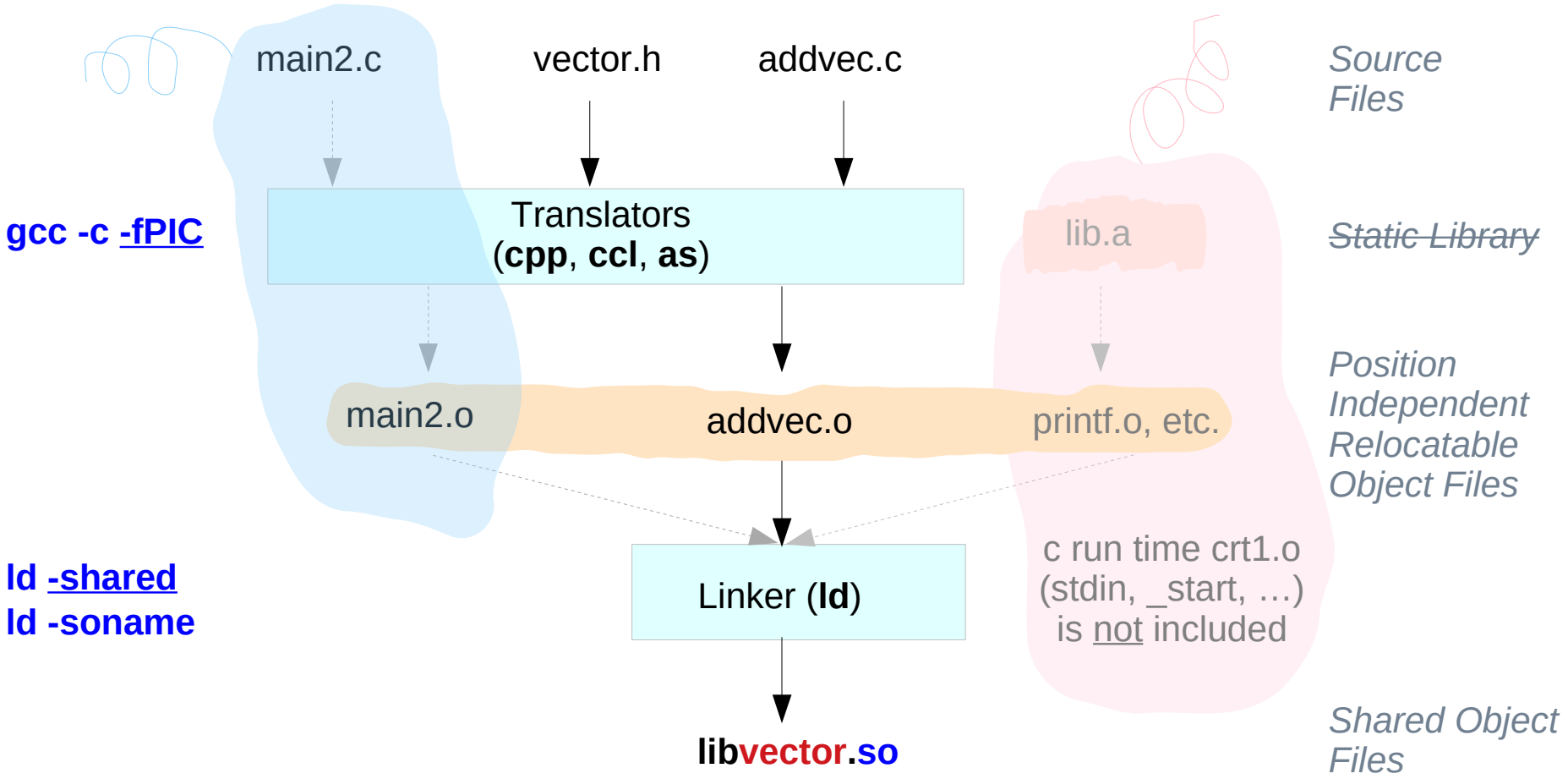
"Computer Architecture: A Programmer's Perspective", Bryant & O'Hallaron

Static Library



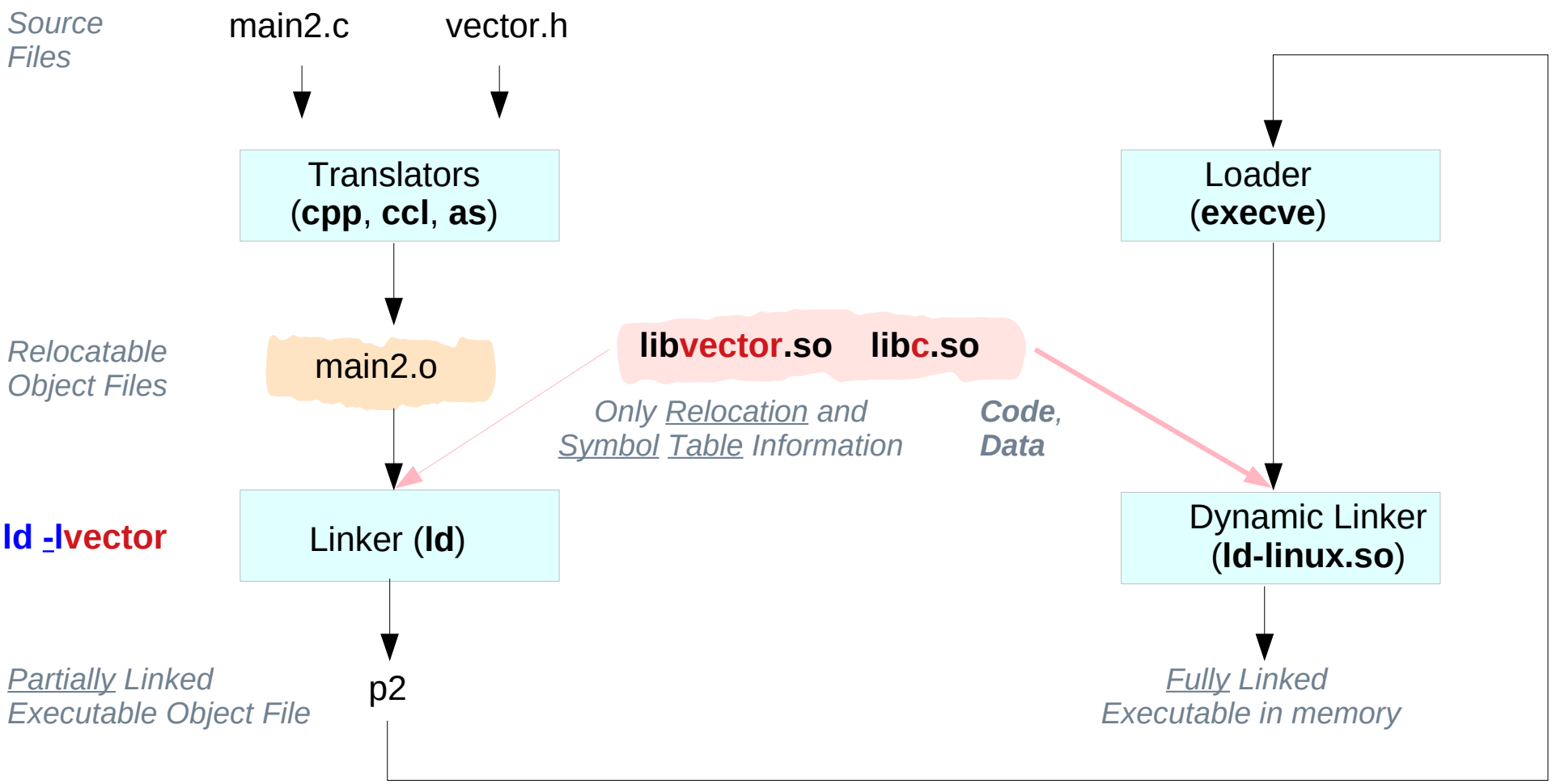
"Computer Architecture: A Programmer's Perspective", Bryant & O'Hallaron

Shared Library v.s. Executable File



"Computer Architecture: A Programmer's Perspective", Bryant & O'Hallaron

Dynamic Linking with Shared Libraries



"Computer Architecture: A Programmer's Perspective", Bryant & O'Hallaron

Another example : source codes (1)

sum.c

```
int sum(int a, int b) { return a+b; }
```

mul.c

```
int mul(int a, int b) { return a*b; }
```

sub.c

```
int sub(int a, int b) { return a-b; }
```

div.c

```
int div(int a, int b) { return a/b; }
```

ttt.h

```
int sum(int, int);
int sub(int, int);
int mul(int, int);
int div(int, int);
```

```
gcc -c  -fPIC sum.c sub.c   mul.c div.c
gcc -shared libttt.so a sum.o sub.o mul.o div.o
```

Shared Object Library File

Another example : source codes (2)

main1.c

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

int main(void) {
        int a=10; int b=20;
        printf("%d \n", sum(a,b));
        printf("%d \n", sub(a,b));
        printf("%d \n", mul(a,b));
        printf("%d \n", div(a,b));
        return 0;
}
```

main2.c

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

int main(void) {
        int i;

        for (i=0; i<10; ++i)
                printf("%d \n", sum(i,20));
        return 0;
}
```

gcc -c main1.c; **gcc** -o **p1** main1.o -L./ -l./ **-lttt**

gcc -c main2.c; **gcc** -o **p2** main2.o -L./ -l./ **-lttt**

LD_LIBRARY_PATH=$**LD_LIBRARY_PATH**:**./;** **export LD_LIBRARY_PATH**

./p1 *Partially Linked Executable Object File*

./p2 *Partially Linked Executable Object File*

Creating a shared library

gcc -c -fPIC sum.c ➡ sum.o
gcc -c -fPIC sub.c ➡ sub.o
gcc -c -fPIC mul.c ➡ mul.o
gcc -c -fPIC div.c ➡ div.o

gcc -shared libttt.so a sum.o sub.o mul.o div.o ➡ **libttt.so**

Shared Object Library File

Linking with a shared library

gcc -c main1.c
gcc -c main2.c

gcc -o p1 main1.o **-L./** -I./ **-lttt** ⬅ **./libttt.so**
gcc -o p2 main2.o **-L./** -I./ **-lttt** ⬅ **./libttt.so**

LD_LIBRARY_PATH=$**LD_LIBRARY_PATH**:**./**
export **LD_LIBRARY_PATH**

./p1 *Partially Linked Executable Object File*
./p2 *Partially Linked Executable Object File*

Names of a shared library

```
gcc -shared -Wl,-soname,libttt.so.1 -o libttt.so.1.0.1 add.o sub.o mul.o div.o
```

link name: libttt.so ---------- gcc -lttt *Partial Link*

ln -s

soname : libttt.so.1 ---------- ld-linux.so *Full Link*

ldconfig -n ./

file name: libttt.so.1.0.1

```
ldconfig -n    ./              (libttt.so.1 → libttt.so.1.0.1)

ln -s libttt.so.1 libttt.so    (libttt.so → libttt.so.1)
```

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

1. Create a shared object library

```
gcc -shared \
      -Wl,-soname,libvector.so.1 \
      -o libvector.so.1.0.1 \
      addvec.c multvec.c

mv  libvector.so.1.0.1   /opt/lib
```

libvector.so.1.0.1

SONAME :
libvector.so.1

The shared library **libvector.so**.1.0.1
will be dynamically linked
with the **soname** of **libvector.so**.1

run time binding

need a symbolic link

http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html

2. Make symbolic links

```
cd   /opt/lib
```

Method 1

```
ln -s libvector.so.1      libvector.so
ldconfig -n /opt/lib
```

Method 2

```
ln -s libvector.so.1      libvector.so
ln -s libvector.so.1.0.1  libvector.so.1
```

Method 3

```
ln -s libvector.so.1.0.1  libvector.so
ln -s libvector.so.1.0.1  libvector.so.1
```

link name: libvector.so
soname : libvector.so.1
file name: libvector.so.1.0.1

http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html

3. Compile main and link with the shared object library

cd ~/
gcc -o p2 main2.c -L/opt/lib -lvector

p2

NEEDED:
libvector.so.1

ldd ./p2

soname actual file name

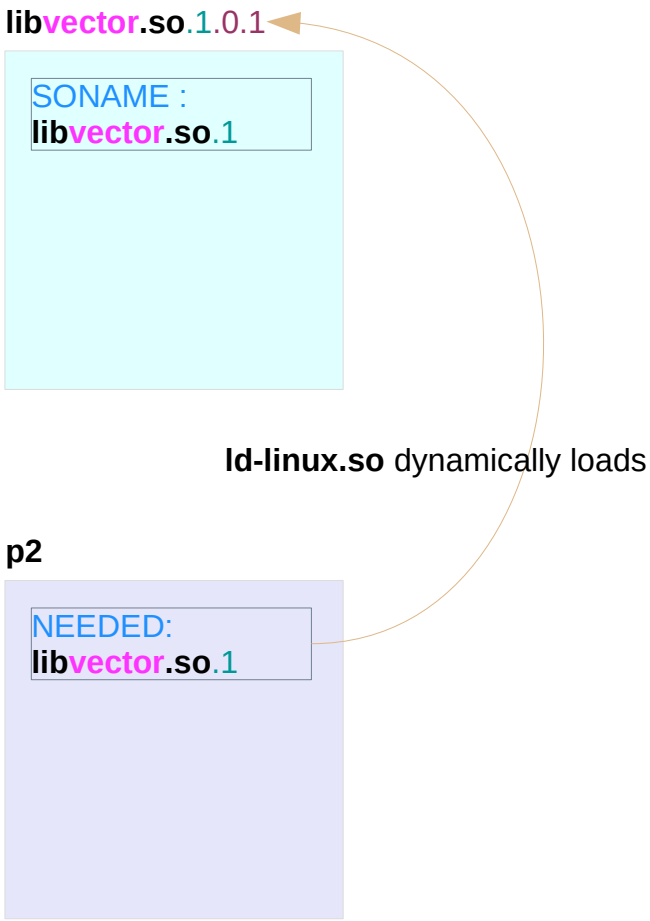
libvector.so.1=> **/opt/lib/libvector.so**.1 (0x...)
libc.so.6 => /lib64/tls/libc.so.6 (0x...)
/lib64/ld-linux-x86-64.so.2 (0x...)

http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html

4. Run the program

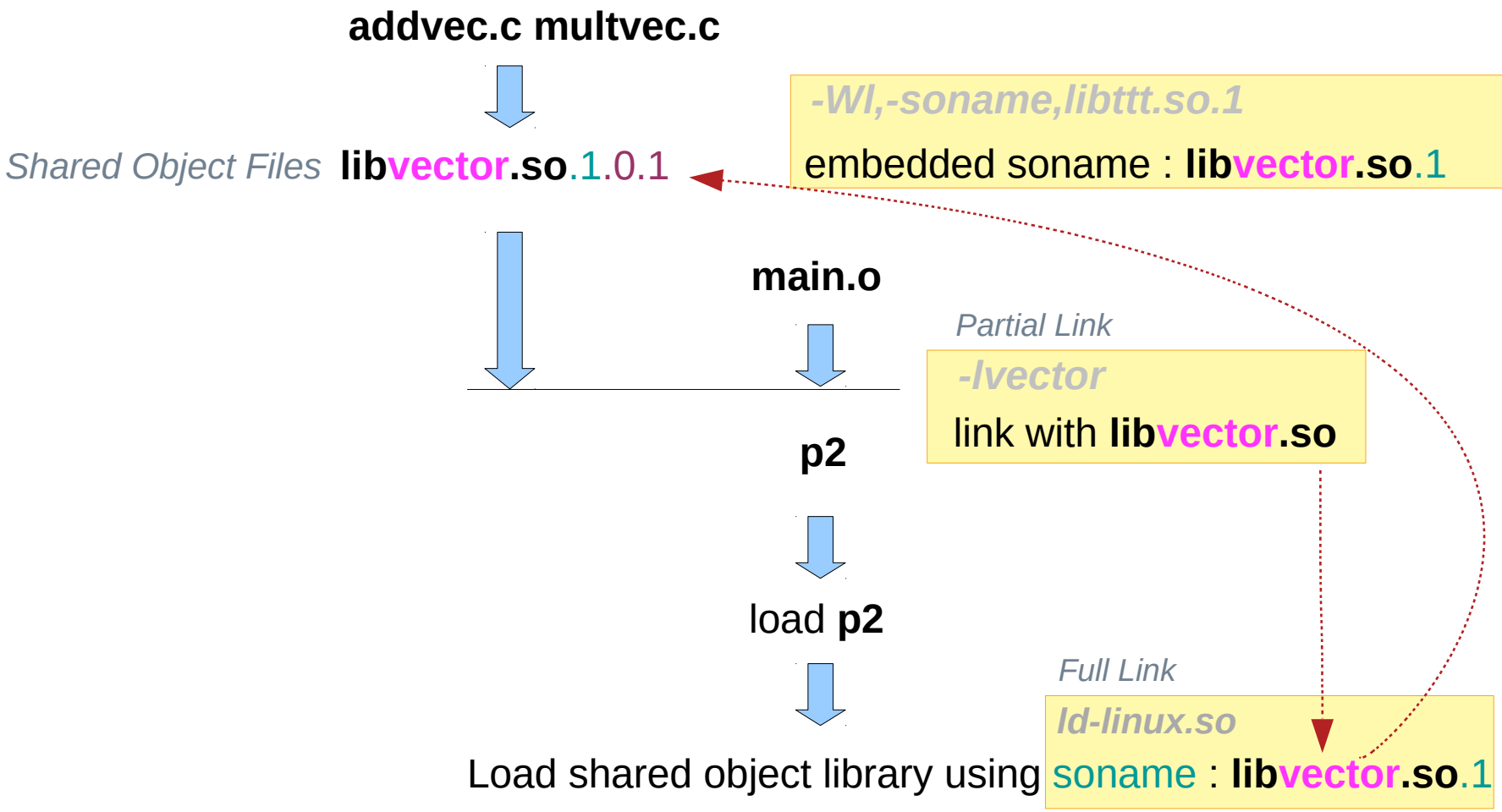
LD_LIBRARY_PATH=$**LD_LIBRARY_PATH**:**./opt/lib**:
export **LD_LIBRARY_PATH**

./p2



http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html

Symbolic Links (1)



http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html

Symbolic Links (2)

ln -s libvector.so.1 libvector.so

symbolic link name : **libvector.so**

directory : **-L/opt/lib**

gcc -o p2 main2.c -L/opt/lib -lvector

link name: **libvector.so**

soname : **libvector.so.1**

file name: **libvector.so.1.0.1**

ln -s libvector.so.1.0.1 libvector.so.1 *or*
ldconfig -n /opt/lib

symbolic link name : **libvector.so.1**

directory : **LD_LIBRARY_PATH**

ld-linux.so (dynamic linker) loads shared library using soname : **libvector.so.1**

http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html

Another Dynamic Linking Example

foo.h

```
#ifndef foo_h__
#define foo_h__
extern void foo(void);
#endif  // foo_h__
```

foo.c

```
void foo(void)
{
    puts("Hello, I'm a shared library");
}
```

main.c

```
#include <stdio.h>
#include "foo.h"

int main(void)
{
    puts("This is a shared library test...");
    foo();
    return 0;
}
```

https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html

Creating an executable file

$ **gcc -c** -Wall -Werror **-fPIC** foo.c

$ **gcc -shared** -o **libfoo.so** foo.o

$ **gcc** -Wall -o test main.c **-lfoo**
/usr/bin/ld: cannot find -lfoo
collect2: ld returned 1 exit status

$ **gcc -L**/home/username/foo -Wall -o **test** main.c **-lfoo** *Partial Link*

$ **./test** *Dynamic Link*
./test: error while loading shared libraries: libfoo.so: cannot open shared object file: No such file or directory

https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html

Executing with **LD_LIBRARY_PATH**

$ **./test** *Dynamic Link*

./test: error while loading shared libraries: libfoo.so: cannot open shared object file: No such file or directory

$ **echo $LD_LIBRARY_PATH**

$ **LD_LIBRARY_PATH**=/home/username/foo:**$LD_LIBRARY_PATH**

$ **./test** *Dynamic Link*

./test: error while loading shared libraries: libfoo.so: cannot open shared object file: No such file or directory

$ **export** **LD_LIBRARY_PATH**=/home/username/foo:$**LD_LIBRARY_PATH**

$ **./test** *Dynamic Link*

This is a shared library test...
Hello, I'm a shared library

https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html

Executing with **rpath**

```
$ unset LD_LIBRARY_PATH
$ gcc -L/home/username/foo -Wl,-rpath=/home/username/foo -Wall -o test main.c -lfoo
$ ./test
This is a shared library test...
Hello, I'm a shared library
```

https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html

Executing with **/usr/lib**

```
$ cp /home/username/foo/libfoo.so /usr/lib
$ chmod 0755 /usr/lib/libfoo.so
$ ldconfig
$ ldconfig -p | grep foo                    print
libfoo.so (libc6) => /usr/lib/libfoo.so
$ unset LD_LIBRARY_PATH
$ gcc -Wall -o test main.c -lfoo
$ ldd test | grep foo
libfoo.so => /usr/lib/libfoo.so (0x00a42000)
$ ./test
This is a shared library test...
Hello, I'm a shared library
```

https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html

Linker option -Wl

-Wl,option ……. **l** in linker

Pass option as an option to the **linker**.
commas split into multiple options
use this syntax to pass an **argument** to the **linker option**.

-Wl,-Map,output.map
passes **-Map output.map** to the linker.

with the GNU linker
-Wl,-Map=output.map

-Wl,-soname,libttt.so.1
passes **-soname libttt.so**.1 to the linker.

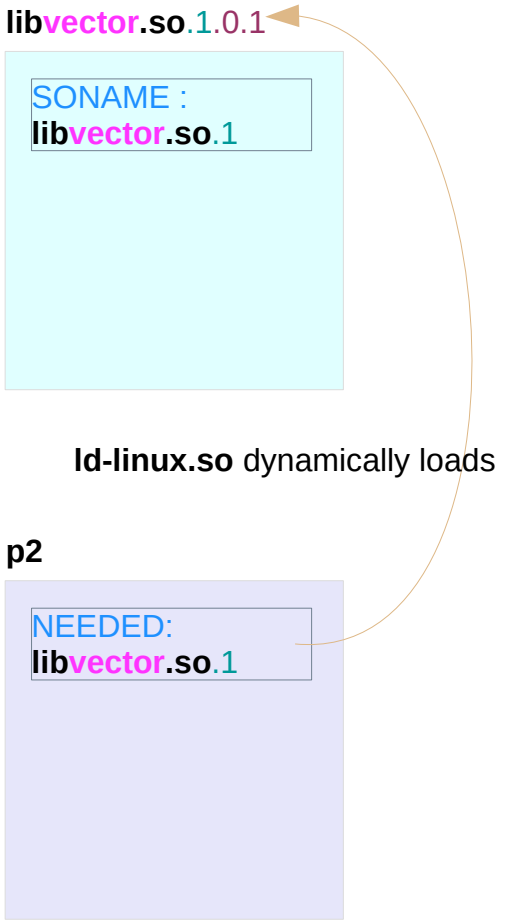
ld -soname

when creating an **ELF shared object** (**.so**),
set the internal SONAME field to the specified **so name**.

```
gcc -shared \
      -Wl,-soname,libvector.so.1 \
      -o libvector.so.1.0.1 \
      addvec.c multvec.c
```

when an **executable** is linked with this **shared object** (**.so**)
which has a SONAME field,
then when the **executable** is run,
the **dynamic linker** (**ld-linux.so**)
will attempt to load the **shared object**
specified by the NEEDED field (**so name**)
rather than the using the file name given to the linker.

```
gcc o p2 main2.c -L/opt/lib -lvector
```



Soname at compile time

soname is used to indicate
what **binary api compatibility** your library support.

SONAME is used at compilation time by **linker**
to determine the *necessary* **library file**
– *the actual target library version*.

gcc ***-lvector*** *(partial link)*

gcc -lNAME will seek for **lib**NAME**.so** link or file
then capture its SONAME that will certainly be more specific

libvector.so links to **libvector.so.0.1.4**
that contains SONAME **libvector.so.0**

libvector.so ……..link name
libvector.so.0……...soname
libvector.so.0.1.4 …….file name

https://stackoverflow.com/questions/12637841/what-is-the-soname-option-for-building-shared-libraries-for/14613602

Soname at run time

A **shared object** file is linked
to get an **executable file**

the SONAME is then set
into **ELF dynamic section** NEEDED

which specifies the necessary library files or links
that must be loaded to execute the ELF file

At run time SONAME field is disregarded, (*full link: ld-linux.so*)
so only the **link** or the **file** existence is enough.

SONAME field is enforced only at link/build time
and not at run time.

At run time NEEDED field is used to load files

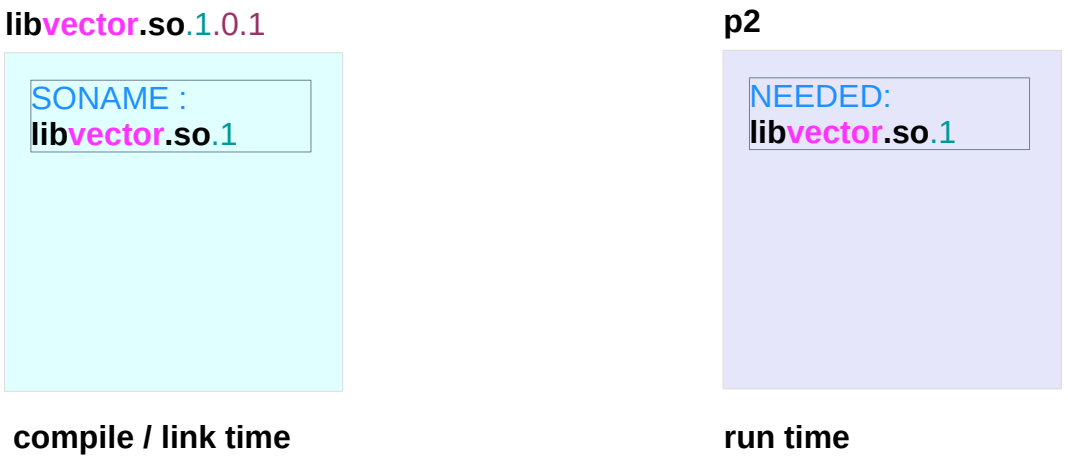
p2

https://stackoverflow.com/questions/12637841/what-is-the-soname-option-for-building-shared-libraries-for/14613602

Soname at run time

objdump -p file | grep SONAME
SONAME of **library** can be seen (shared object)

objdump -p file | grep NEEDED
NEEDED of **binaries** can be seen (executable object)



https://stackoverflow.com/questions/12637841/what-is-the-soname-option-for-building-shared-libraries-for/14613602

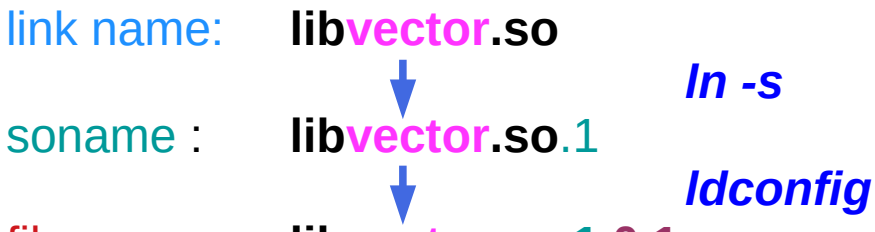
Soname and ldconfig

on Linux the SONAME entry provides a hint
for the **runtime-linker** system on
how to create appropriate **links** in /lib, /lib64 etc.

*ld -**soname**=name*

Running the command **ldconfig**
tries to create a symbolic link named with SONAME
which is also taken into the **run-time linker cache**.

The newest one of the libraries
tagging the same SONAME wins the link-race.



https://stackoverflow.com/questions/12637841/what-is-the-soname-option-for-building-shared-libraries-for/14613602

Soname Format

In linux real life **SONAME** as a specific form :

lib[NAME][API-VERSION]**.so**.[major-version]

libvector.so.0

major-version is only one digit integer value
that increase at each major library change.
API-VERSION is empty by default

the **real filename** include minor versions and subversions

lib[NAME][API-VERSION]**.so**.[major-version].[minor-version]

libvector.so.0.1.4

https://stackoverflow.com/questions/12637841/what-is-the-soname-option-for-building-shared-libraries-for/14613602

Providing soname

not providing a **soname** is a bad practice
    since renaming of file will change its behavior.

If some **software** relies on the specific SONAME and
    you want to renew a library
you have to provide this SONAME
    to get **ldconfig** stick on this new library
    (if **ldconfig** is used to rebuild the cache and the links)

https://stackoverflow.com/questions/12637841/what-is-the-soname-option-for-building-shared-libraries-for/14613602

Soname Examples

Let's assume **libA.so** depends on **libB.so**,
and they all in a directory
(the directory cannot be found by dynamic linker).
If you didn't set **soname** then **dlopen** doesn't work:

auto pB = **dlopen**("./libB.so", RTLD_LAZY | RTLD_GLOBAL);
auto pA = **dlopen**("./libA.so", RTLD_LAZY | RTLD_GLOBAL);

Because **runtime linker** cannot find libB.so,
so pA is set to NULL.

https://stackoverflow.com/questions/12637841/what-is-the-soname-option-for-building-shared-libraries-for/14613602

Program Loading

The **execution** of a program starts inside the kernel,
in the **exec** system call.

There the **file type** is looked up and
the appropriate **handler** is called.

The **binfmt-elf** handler then loads the **ELF header** and
the **program header table** (**PHT**),
followed by lots of sanity checks.

The kernel then loads the parts specified
in the **LOAD** directives in the **PHT** into memory.

https://greek0.net/elf.html

Program Loading

If an **INTERP** entry is present,
the **interpreter** is loaded too.

Statically linked binaries
can do without an interpreter;

Dynamically linked programs
always need **/lib/ld-linux.so** as **interpreter**

because it includes some **startup code**,
loads **shared libraries** needed by the binary,
and performs **relocations**.

Finally **control** can be transfered to the **program**,
to the **interpreter**, if present, otherwise to the **binary** itself.

https://greek0.net/elf.html

Dynamic Linking and the ELF Interpreter

the **dynamic linker**
contained within the **interpreter ld-linux.so**

looks at the **.dynamic** section
whose address is stored in the **PHT**

finds the **NEEDED entries**
at the **.dynamic** section

decides which **libraries** have to be loaded
before the program can be run,

the ***REL*** entries giving
the address of the **relocation tables**,
the **VER*** entries containing
symbol versioning information, etc.

https://greek0.net/elf.html

Dynamic Linking and the ELF Interpreter

the **dynamic linker**
loads the needed **libraries**
performs **relocations**

(either directly at program startup or later,
as soon as the relocated symbol is needed,
depending on the relocation type).

Finally **control** is transferred to the address given
by the symbol **_start** in the **binary**.

Normally some **gcc/glibc startup code** lives there,
which in the end calls **main()**.

https://greek0.net/elf.html

ldconfig

ldconfig - configure dynamic linker **run-time bindings**
Synopsis

ldconfig creates the necessary **links** and **cache**
to the most recent **shared libraries** found in the **directories** specified
on the **command line**,
in the file **/etc/ld.so.conf**, and
in the trusted directories (**/lib** and **/usr/lib**).

The **cache** is used by the **run-time linker**, **ld.so** or **ld-linux.so**.

ldconfig checks the **header** and **filenames** of the **libraries** it encounters when determining which versions should have their links updated.

ldconfig will attempt to deduce the type of ELF libs (i.e., libc5 or libc6/glibc) based on what C libs, if any, the library was linked against.

https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html

ldconfig -n

ldconfig -n *directory_with_shared_libraries*

Process only the directories specified on the command line.
Don't process the trusted directories (/usr/lib, /lib)
nor those specified in /etc/ld.so.conf. Implies -N.

ldconfig -N

Don't rebuild the cache. Unless -X is also specified, links
are still updated.

ldconfig -p, --print-cache

Print the lists of directories and candidate libraries stored in
the current cache.

ld.so, ld-linux.so – dynamic linker/loader (1)

The **dynamic linker** can be run
either indirectly by running some
dynamically linked program or library
no command-line options
in the ELF case,
the dynamic linker which is stored in the **.interp** section

or directly by running:
/lib/ld-linux.so.* [OPTIONS] [PROGRAM [ARGUMENTS]]

find and load the **shared libraries** needed by a program,
prepare the program to run, and then run it.

https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html

ld.so, ld-linux.so – dynamic linker/loader (2)

linux binaries require **dynamic linking** (linking at run time)
unless the **-static** option was given to **ld** during **compilation**.

ld.so handles **a.out** binaries
ld-linux.so* handles **ELF**
    /lib/ld-linux.so.1 for libc5,
    /lib/ld-linux.so.2 for glibc2

https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html

ldd

ldd - print shared library **dependencies**

ldd [OPTION]... FILE...

ldd prints the **shared libraries** required by
each **program** or
shared library
specified on the command line.

```
$ ldd /bin/ls
        linux-vdso.so.1 (0x00007ffcc3563000)
        libselinux.so.1 => /lib64/libselinux.so.1 (0x00007f87e5459000)
        libcap.so.2 => /lib64/libcap.so.2 (0x00007f87e5254000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f87e4e92000)
        libpcre.so.1 => /lib64/libpcre.so.1 (0x00007f87e4c22000)
        libdl.so.2 => /lib64/libdl.so.2 (0x00007f87e4a1e000)
        /lib64/ld-linux-x86-64.so.2 (0x00005574bf12e000)
        libattr.so.1 => /lib64/libattr.so.1 (0x00007f87e4817000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f87e45fa000)
```

https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html

ldd example

When a dynamically linked application is loaded by the operating system,
it must locate and load the **dynamic libraries** it needs for execution.
On linux, that job is handled by **ld-linux.so.2**.
see the libraries used by a given application with the **ldd** command:

```
$ ldd `which ls`
        linux-gate.so.1      =>  (0xb7fff000)
        librt.so.1           => /lib/librt.so.1 (0x00b98000)
        libacl.so.1          => /lib/libacl.so.1 (0x00769000)
        libselinux.so.1      => /lib/libselinux.so.1 (0x00642000)
        libc.so.6            => /lib/libc.so.6 (0x007b2000)
        libpthread.so.0      => /lib/libpthread.so.0 (0x00920000)
         /lib/ld-linux.so.2 (0x00795000)
        libattr.so.1         => /lib/libattr.so.1 (0x00762000)
        libdl.so.2           => /lib/libdl.so.2 (0x0091a000)
        libsepol.so.1        => /lib/libsepol.so.1 (0x0065b000)
```

https://unix.stackexchange.com/questions/122670/using-alternate-libc-with-ld-linux-so-hacks-cleaner-method

ldd example

When **ls** is loaded, the OS passes control to **ld-linux.so.2**
instead of normal **entry point** of the application.
ld-linux.so.2 searches for and loads the unresolved libraries,
and then it passes control to the application **starting point**.

The **ELF** specification provides the functionality for dynamic linking.
GCC includes a special ELF program header called **INTERP**,
which has a p_type of PT_INTERP.
This header specifies the **path** to the **interpreter**.

https://unix.stackexchange.com/questions/122670/using-alternate-libc-with-ld-linux-so-hacks-cleaner-method

elf example

$ **readelf -l a.out**

Elf file type is EXEC (Executable file)
Entry point 0x8048310
There are 9 program headers, starting at offset 52

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x00120	0x00120	R E	0x4
INTERP	0x000154	0x08048154	0x08048154	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x08048000	0x08048000	0x004cc	0x004cc	R E	0x1000
LOAD	0x000f0c	0x08049f0c	0x08049f0c	0x0010c	0x00110	RW	0x1000

https://unix.stackexchange.com/questions/122670/using-alternate-libc-with-ld-linux-so-hacks-cleaner-method

elf example

The ELF specification requires that
if a PT_INTERP section is present,
the OS must create a process image of the of the interpreter's file segments,
instead of the application's.

Control is then past to the interpreter,
which is responisble for loading the dynamic libraries.

The spec offers some amount of flexiblity in how control may be given.

For x86/Linux, the argument passed to the dynamic loader is
a pointer to an mmap'd section.

https://unix.stackexchange.com/questions/122670/using-alternate-libc-with-ld-linux-so-hacks-cleaner-method

Dynamically Loaded Libraries

#include <**dlfcn**.h>

void ***dlopen**(const char *filename, int flag);
opens a library and prepares it for use
returns ptr to handle if OK, NULL on error

void ***dlsym**(void *handle, char *symbol);
looks up the value of a symbol in a given (opened) library
returns ptr to symbol if OK, NULL on error

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

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

https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html

dlopen() flags

In dlopen(), the value of flag must be either RTLD_LAZY, or RTLD_NOW

RTLD_LAZY : resolve undefined symbols
as code from the dynamic library is executed

inscrutable errors if there are unresolved references

RTLD_NOW : resolve all undefined symbols
before dlopen() returns and fail if this cannot be done

good for debugging
makes opening the library take slightly longer
(but it speeds up lookups later)

https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html

DL Library Examples (1)

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

int main(int argc, char **argv) {
    void *handle;
    double (*cosine)(double);       // function pointer
    char *error;

    handle = dlopen ("/lib/libm.so.6", RTLD_LAZY);       // math library
    if (!handle) {
        fputs (dlerror(), stderr);
        exit(1);
    }
```

http://tldp.org/HOWTO/Program-Library-HOWTO/dl-libraries.html

DL Library Examples (2)

```
    cosine = dlsym(handle, "cos");
    if ((error = dlerror()) != NULL)  {
        fputs(error, stderr);
        exit(1);
    }

    printf ("%f\n", (*cosine)(2.0));          // function pointer
    dlclose(handle);
}
```

gcc -o foo foo.c -ldl

http://tldp.org/HOWTO/Program-Library-HOWTO/dl-libraries.html

References

[1] An Introduction to GCC, B. Gough, http://www.network-theory.co.uk/docs/gccintro/
[2] Unix, Linux Programming Indispensable Utilities, CW Paik