Using ELF in `glibc 2.1`

Ulrich Drepper*
Cygnus Solutions
Sunnyvale, CA

March 1st, 1999

*drepper@cygnus.com

1 What is ELF

What is ELF?

- ELF is a binary format
- available on 30+ platforms
- designed with cross-platform development in mind
- does not restrict the internal structure
- can be used for four different file types
  - relocatable object file
  - executables
  - dynamic executables
  - core dumps

Slide 1 Using ELF in glibc 2.1 CYGNUS®

The ELF binary format was defined right from the beginning in publically available documents. The base document is available at:

`ftp://download.intel.com/design/perftool/tis/elf11h.zip`

This site also contains the Dwarf (the debugging format for ELF) and the OMG specification.

The specifications for architectures other than ix86 for available from the vendors. Often they are not publically available so it is not possible to give links here. But looking through the manuals coming with a system should be enough. E.g., Sun provides the documentation in a document called "Linker and Library Guide".

SCO (the official keeper of ELF) has released in April 1998 a draft for a revised ELF standard. It brings a few extensions. The draft can be seen at:

`http://www.sco.com/developer/gabi/contents.html`

2 How to work with ELF?

Linker

- take one or more input files and any number of shared objects and archives
- combine all object files plus startup code
- resolve all references in the input files from the input files, shared objects or archives, from left to right
- references found in shared objects create Procedure Linkage Table (PLT) entry and are resolved by jump into PLT
- resulting executable is relocated according to fixes start address
- start address of program is normally first byte in code section but can be anywhere else

Slide 2 Using ELF in glibc 2.1 CYGNUS®

Elf files are generated by the assembler and the linker. The GNU binutils package contains versions of these programs which can be configured for most platforms to generate ELF.

Program Start: Kernel View

To load a dynamically linked binary the kernel

- loads the dynamic loader
- maps the user program
- puts auxiliary vector on the new process' stack
- starts dynamic loader program

→ program code is loaded at address the linker relocated it for

→ the dynamic loader is relocated, can be loaded anywhere

Slide 3 Using ELF in glibc 2.1 CYGNUS®

To see how the Linux kernel handles loading ELF binaries one has to read the file `fs/binfmt_elf.c`. The function `create_elf_tables` is responsible for creating the user stack which includes creating the auxiliary vector. One might want to run an application with

```
env LD_SHOW_AUXV=1 /path/to/binary
```

and compare the output with the code in this function.

`load_elf_interp` is the function responsible for loading the dynamic loader in case a dynamically linked binary is started. The code should be easy to follow once the remainder of this article is read. Loading a binary in the kernel and at user level it not very different. The function also has to look for the program header and load the individual segments.

The function `do_load_elf_binary` is the main function of the file. It is responsible for loading an ELF binary, be it dynamically linked or not. One can see that dynamically linked binaries are recognized by having a `PT_INTERP` entry in the program table. This functions calls the other functions mentioned here.

3 How the dynamic loader works

The responsibilities of the dynamic loader is simply to finish the work the linker did. The linker left some symbols unresolved but put some extra information in the binary to allow the dynamic loader to do its work. The tasks the dynamic loader has to perform are:

- load all dependencies of the program and the dependencies themselves
- relocate as much as necessary at startup time
- create the internal data structures and prepare for runtime relocation

Program Start: Dynamic Loader View

- **relocate itself**
- **read information from auxiliary vector**
- **build data structures for the binary as loaded by the kernel**
- **find, load, and relocate dependencies; proceed recursively**
- **jump to start address given in auxiliary vector**

Slide 4 **Using ELF in glibc 2.1** **CYGNUS®**

Loading in the Dynamic Loader

(not necessary for the program itself)

- load first block of the object (Elf32_Ehdr/Elf64_Ehdr)
- using this information make sure the program headers is loaded (normally at the beginning of the file as well) (array of Elf32_Phdr/Elf64_Phdr)
- find all load commands (PT_LOAD) and mmap the segments appropriately
- locate dynamic section using PT_DYNAMIC
- initialize hash table
- setup PLT/GOT (Global Offset Table)
- relocate

Slide 5 Using ELF in glibc 2.1 CYGNUS

In the following a few of the ELF data structures will be shown. We will always show the 32bit version only. The 64bit definition is very similar and can normally be generated by replacing `32` by `64` in all the names.

The `Elf32_Ehdr` contains the ELF header and this structure can be found at the very beginning of every ELF file.

```
#define EI_NIDENT       16

typedef struct {
  unsigned char e_ident[EI_NIDENT];
  Elf32_Half    e_type;
  Elf32_Half    e_machine;
  Elf32_Word    e_version;
  Elf32_Addr    e_entry;
  Elf32_Off     e_phoff;
  Elf32_Off     e_shoff;
  Elf32_Word    e_flags;
  Elf32_Half    e_ehsize;
  Elf32_Half    e_phentsize;
  Elf32_Half    e_phnum;
  Elf32_Half    e_shentsize;
  Elf32_Half    e_shnum;
  Elf32_Half    e_shstrndx;
} Elf32_Ehdr;
```

the `e_ident` field contains some magic bytes for identification and then some very basic information about the binary which allows the kernel to recognize very easily whether the binary can actually run on the system.

Symbolic Index Name	Value	Purpose
`EI_MAG0`	0	File identification
`EI_MAG1`	1	File identification
`EI_MAG2`	2	File identification
`EI_MAG3`	3	File identification
`EI_CLASS`	4	File class
`EI_DATA`	5	Data encoding
`EI_VERSION`	6	File version
`EI_PAD`	7	Start of padding bytes

The bytes `EI_MAG0` to `EI_MAG3` always contain the bytes `0x7f`, `'E'`, `'L'`, and `'F'`. The byte at offset `EI_CLASS` contains one of the following values:

Name	Value	Meaning
`ELFCLASSNONE`	0	Invalid class
`ELFCLASS32`	1	32-bit object
`ELFCLASS64`	2	64-bit object

The Data Encoding byte at contains one of the following values:

Name	Value	Meaning
`ELFDATANONE`	0	Invalid data encoding
`ELFDATA2LSB`	1	Two-complement, least significant byte first
`ELFDATA2MSB`	2	Two-complement, most significant byte first

`e_type`: used to identify the type of the binary:

Name	Value	Meaning
`ET_NONE`	0	No file type
`ET_REL`	1	No file type
`ET_EXEC`	2	No file type
`ET_DYN`	3	No file type
`ET_CORE`	4	No file type
`ET_LOPROC`	0xff00	No file type
`ET_HIPROC`	0xfff	No file type

`e_machine`: contains a value identifying the architecture. The `elf.h` file which comes as part of `glibc` provides a long list of symbols starting with `EM_`.

`e_version`: contains zero for now. This could change if an incompatible update of the ELF standard is published.

`e_entry`: the address of the entry point of the binary.

`e_phoff`: offset of the program header table or zero.

`e_shoff`: offset of the section header table or zero.

`e_flags`; processor specific flags.

`e_ehsize`: ELF header size in bytes.

`e_phentsize`: size of one entry in the program header table.

`e_phnum`: number of elements of program header table.

`e_phentsize`: size of one entry in the section header table.

`e_phnum`: number of elements of section header table.

`e_shstrndx`: index in section header table for section name string table.

mmap'ing

- shared objects can be loaded anywhere
- → mmap of the first segment does not specify a load address
- all other segments must be at a fixed, predetermined offset based on the first mmap
- → first mmap region is made large enough to cover all other segments as well; then mprotect is used to make the region unusable with PROT_NONE

3.1 Locating shared objects

The part of the dynamic linker people will most probably have contact with is the part responsible for locating the shared objects to be loaded. The following slide summarizes the rules:

How to locate shared objects?

- **DT_RPATH in object which caused the reference**
  special path elements:
  - **$ORIGIN as in $ORIGIN/../lib: relative to loading object**
  - **$PLATFORM as in $ORIGIN/../lib/$PLATFORM**
- **DT_RPATH of the executable**
- **LD_LIBRARY_PATH environment variable**
- **/etc/ld.so.cache, generated by ldconfig**
- **builtin paths (/lib, /usr/lib)**

Security: when a binary is executed SUID objects in non-safe directories from LD_LIBRARY_PATH are ignored

Slide 7 **Using ELF in glibc 2.1** **CYGNUS**

Many people have different opinions on how the lookup should happen. The way it is implemented prescribed by the ELF standard. Therefore nothing will change. We might add some new features in future, though.

One new feature which is not available on all systems is the interpretation of `$ORIGIN` and `$PLATFORM` in path specifications. This features was developed and implemented on Solaris 2.6. Therefore programs who want to use this feature (which is *highly* advised!) must take precautions for the case it is not available.

The use of the `ldconfig` output in `/etc/ld.so.conf` is also non-standard. It is not available on non-GNU systems.

Some people claim that the concept of using `DT_RPATH` is broken. I don't see it this way. `DT_RPATH` works perfectly if one has not move around the affected binaries. And this should almost never be the case.

In case one explicitly wants to provide a relocatable binaries the new features of the interpreted token in the path helps to avoid all problems.

One more point: people argueing against `DT_RPATH` forget that it makes loading binaries much faster as well. If a `DT_RPATH` is available the dynamic loader does not have to look through directories which do not contain the wanted shared object. At last but not least: `DT_RPATH` adds security. Since it is not possible to overwrite the value it is made sure that the shared objects the implementer intend to be used are really used (of course only if the installation was successful).

3.2 A neat trick

`glibc 2.1` extends the functionality of loading shared objects a bit further:

Using Machine Capabilities

- most architectures introduce new features over time
- portable implementations cannot take advantage of them

→ automatically use appropriate version

- kernel hardware capabilities are used; important set can be masked with LD_HWCAP_MASK

  For x86: fpu,vme,de,pse,tsc,msr,pae,...,mmx,fxsr,amd3d

- with LD_HWCAP_MASK=0x800010 searched directories are:

```
/foo/bar/tsc/mmx
/foo/bar/tsc
/foo/bar/mmx
/foor/bar
```

tsc = 1 << 4

mmx = 1 << 23

Slide 8 Using ELF in glibc 2.1 CYGNUS

The legacy of long-licing processor families (like Intel ix86) causes sometimes problems and hinders sometimes optimal use of the system. With the mechanism in place it is possible to use to provide several different versions of a shared object and the dynamic linker picks the best one.

E.g., a library performing graphic operations on an ix86 machine can take advantage of the MMX instructions. But those are not available generally and so the improved library cannot always be used. With the extension one can generate two different libraries and place them in appropriate directories:

`/foo/bar/mmx/libgraph.so`
`/foo/bar/libgraph.so`

If now a program requiring `libgraph.so` is started with

env LD_HWCAP_MASK=0x800000 LD_LIBRARY_PATH=/foo/bar program

it will pick the library `/foo/bar/mmx/libgraph.so` if the processor of the system has MMX capabilities. Otherwise it will use the default version `/foo/bar/libgraph.so`.

3.3 Accessing all the Information

An ELF file can contain a lot of different information. Things are organized in sections. At runtime the dynamic linker has to access groups of related sections. To allow this efficently each dynamic ELF binary contains a Dynamic section.

Dynamic Section

- **array with elements of type Elf32_Dyn/Elf64_Dyn**
- **pointed to by PT_DYNAMIC in program header**
- **last entry is DT_NULL (no direct value with size)**
- **available information (among others):**
  - ◇ **relocation information (DT_REL, DT_RELA)**
  - ◇ **string table (DT_STRTAB)**
  - ◇ **DT_SYMBOLIC, DT_TEXTREL flags**
  - ◇ **dependencies (DT_NEEDED)**
  - ◇ **object name (DT_SONAME)**
- **easily extendable**

Slide 9 Using ELF in glibc 2.1 CYGNUS

The type of the elements of the section is as follows:

```
typedef struct
{
  Elf32_Sword   d_tag;                  /* Dynamic entry type */
  union
    {
      Elf32_Word d_val;                 /* Integer value */
      Elf32_Addr d_ptr;                 /* Address value */
    } d_un;
} Elf32_Dyn;
```

The number of tags in used is steadily growing since many platform and processor specific inventions are added. The standard describes the following types:

DT_NULL	0	Marks end of dynamic section
DT_NEEDED	1	Name of needed library
DT_PLTRELSZ	2	Size in bytes of PLT relocs
DT_PLTGOT	3	Processor defined value
DT_HASH	4	Address of symbol hash table
DT_STRTAB	5	Address of string table
DT_SYMTAB	6	Address of symbol table
DT_RELA	7	Address of Rela relocs
DT_RELASZ	8	Total size of Rela relocs
DT_RELAENT	9	Size of one Rela reloc
DT_STRSZ	10	Size of string table
DT_SYMENT	11	Size of one symbol table entry
DT_INIT	12	Address of init function
DT_FINI	13	Address of termination function

`DT_SONAME`	14	Name of shared object
`DT_RPATH`	15	Library search path
`DT_SYMBOLIC`	16	Start symbol search here
`DT_REL`	17	Address of Rel relocs
`DT_RELSZ`	18	Total size of Rel relocs
`DT_RELENT`	19	Size of one Rel reloc
`DT_PLTREL`	20	Type of reloc in PLT
`DT_DEBUG`	21	For debugging; unspecified
`DT_TEXTREL`	22	Reloc might modify .text
`DT_JMPREL`	23	Address of PLT relocs
`DT_NUM`	24	Number used
`DT_LOPROC`	0x70000000	Start of processor-specific
`DT_HIPROC`	0x7fffffff	End of processor-specific

3.4 Symbol Lookup Preparation

Since potentially many symbols can be defined and used in ELF programs the symbol lookup mechanism should be made as efficient as possible. For string lookup this means one should use hash tables. Each ELF file providing symbols contains a hash table:

Hashing Table

- **`nchain ==` number of entries in symbol table**
- **preparation code setups up easy access**
- **Lookup process:**
  - **determine hash value H for symbol**
  - **use `bucket[H%nbucket]` as index for symbol table**
  - **if not the right symbol get value `chain[bucket[H%nbucket]]` as new index I**
  - **test `symtab[I]` and proceed with `I := chain[I]` if not successful**
  - **stop when `I == STN_UNDEF`**

```
nbucket
nchain
bucket[0]
  ⋮
bucket[nbucket-1]
chain[0]
  ⋮
chain[nchain-1]
```

Slide 10 **Using ELF in glibc 2.1** CYGNUS®

The generation of the hash table is often implemented quite poorly. This leads to large tables with many holes and also to tables with long chains. Many elements in a chain means that unsuccessful searches much test many elements before realizing the searched symbol is not in the current object.

For GNU ld the author has implemented in later 1998 an optimizer which is enabled when `-O1` is passed as a parameter to the linker. The optimizer tries to generate an good hash table which is not too large and, especially, has no long chains. The optimizations are available in latest snapshots of the GNU binutils and will be in the next release after 2.9.1.

3.5 Why all the complexity?

It is at first non-obvious why the ELF format and it's implementation goes to such a length to implement dynamic loading and symbol resolving. The approach of letting the linker do the work was sufficent.

PLT / GOT basics

- **Purpose: prevent text section from being modified and therefore made non-sharable**

→ **references to objects which address is not known at link time must be *indirect***

- **data references are all relocated at program starttime**
- **relocation of code references can be delayed, but can be forced to happen at program start time**

GOT (and for some platforms PLT) are the places where changes are made to address the symbol definitions found at runtime

Slide 11 **Using ELF in glibc 2.1** CYGNUS®

PIC / non-PIC

(Position Independant Code)

Example code here for x86:

	non-PIC	PIC
Data references	`movl foo,%eax`	`movl foo@GOT(%ebx),%eax`
Code references	`call bar`	`call bar@PLT`

- **although PIC function call does not require `%ebx` register directly it is still needed in the PLT code**
- `movl foo@GOT(%ebx),%eax` **means:**

  load the word from memory position `%ebx+foo@GOT` where `foo@GOT` is offset of `foo` is offset table

Slide 12 **Using ELF in glibc 2.1** CYGNUS®

The requirements on today's systems is different. Today many hundred programs can run on one system and they often share major parts of the program code (multiple copies of one program, use of the same library in multiple programs). With the old approach it was impossible to take advantage of this duplication. The slides in the section explain the basics behind the mechanisms to ensure we can share as much code as possible, even for loaded programs.

3.6 How is it implemented

The implementation of the Procedure Linkage Table various greatly over all the platforms. Some implementations are simpler than others. Several of the design decisions are enforced by the processor. We will describe here the i.x86 implementation.

PLT Details

non-PIC PLT

```
.PLT0:pushl got_plus_4
      jmp * got_plus_8
      nop; nop
      nop; nop
.PLT1:jmp * name1_in_GOT
      pushl $offset
      jmp .PLT@PC
.PLT2:jmp * name2_in_GOT
      pushl $offset
      jmp .PLT@PC
```

PIC PLT

```
.PLT0:pushl 4(%ebx)
      jmp *8(%ebx)
      nop; nop
      nop; nop
.PLT1:jmp *name1@GOT(%ebx)
      pushl $offset
      jmp .PLT0@PC
.PLT1:jmp *name1@GOT(%ebx)
      pushl $offset
      jmp .PLT0@PC
```

The GOT entries for the jump symbols in PLT are initialized by the linker and only have to be adjusted for the load address by the dynamic loader

Slide 13 **Using ELF in glibc 2.1** CYGNUS®

The first entry `.PLT0` is special and is not used directly. Assume now the symbol `bar@PLT` from the last section's example refers to the entry `.PLT1`. At the first call of the function the `jmp` instruction jumps to the following instruction. This is the way the GOT entries are initilized.

Here it pushes an value on the stack, which identifies the symbol to be relocated. It is the offset of the symbol in the relocation table. Then the execution continues with a jump to `.PLT0` where another work is pushed. In the `glibc` implementation this is a pointer to the link map for the current shared object. Finally processor executes the function pointed to by the third GOT entry. This is initialized at object load time by the dynamic loader to point to the runtime resolver function.

Once the resolver has found a symbol definition it overwrites the GOT entry used in the jump at address `.PLT1` and so directs future calls to the PLT entry directly to the resolved function.

3.7 The Lookup Function

The heart of the dynamic linker is the lookup functions. It's address is stored (on ix86) in the third word of the GOT.

Lookup Function

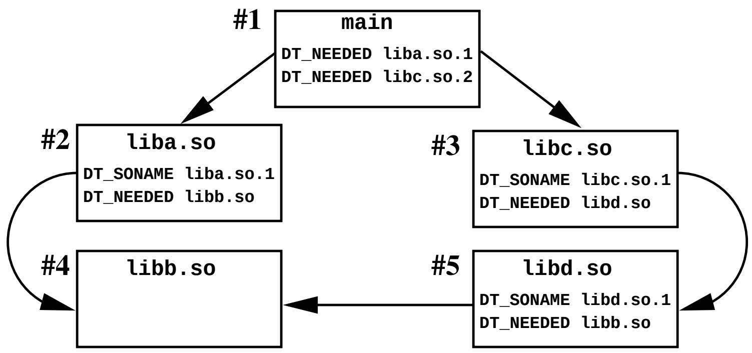
- **Parameters:**
  - **reference to relocation entry (for x86 of type Elf32_Rel with tag R_386_JMP_SLOT)**
  - **implementation defined value, passed by code in .PLT0 (in glibc: points to the link map of the object)**
- **getting the symbol name:**
  - **Get entry in relocation table with DT_JMPREL and offset**
  - **Get start of symbol table using DT_SYMTAB**
  - **Get index in symbol with ELF32_R_SYM from reloc entry**
  - **Get string table with DT_STRTAB**
  - **Get name at offset given by st_name from symbol entry**
  - **Determine address for result using r_offset from reloc entry**
  - **Determine scope for the object with the reference**

Slide 14 **Using ELF in glibc 2.1** **CYGNUS**

The function searches not only in one object for the symbol but instead in a possibly large numbers. The set of objects searched is called the search scope:

Scope

```
gcc -o libb.so -shared b.c
gcc -o libd.so -shared d.c -lb -Wl,-soname,libd.so.3
gcc -o liba.so -shared a.c -lb -Wl,-soname,liba.so.1
gcc -o libc.so -shared c.c -ld -Wl,-soname,libc.so.2
gcc -o main main.c -la -b
```



Slide 15 **Using ELF in glibc 2.1** **CYGNUS**

it is important the the programmer understands the way the scope is constructed. If s/he makes wrong assumptions about the search scope the program might behave differently than expected. To complicated things even more there are possibilities for both the programmer and the user to directly influence the search scope:

Modifying the Scope

- **LD_PRELOAD environment:**

  allows to map extra shared objects after the executable but before and dependency

- **DT_SYMBOLIC flags (passed as -Wl,-Bsymbolic):**

  the current shared object is searched searched first even before the application itself

- **-rdynamic:**

  GNU ld does by default not generate a PLT table with the global symbols for the application; this option changes this

- **RTLD_GLOBAL flag for dlopen:**

  the dynamically loaded objects and the dependencies are put in the global scope, added at the end

Slide 16 **Using ELF in glibc 2.1** CYGNUS®

The search scope is not fixed. It can change over time and this makes things even more complicated. With this knowledge it should now be possible a pseudo-code description of the lookup algorithm.

Basic Lookup Algorithm

```
found = false

forall maps of the scope
  idx = hash value of name % nbucket
  while idx != SN_UNDEF
    sym = element idx of symbol table
    if name of sym matches name of the wanted symbol
      if definition of sym is strong
        return sym
      remember definition sym
      found = true
      leave while loop
    idx = chain[idx]

if found
  return remembered definition

return failure
```

Slide 17 **Using ELF in glibc 2.1** CYGNUS®

4 Take Influence on the Lining Process

Now that we know how the dynamic linker works it is time to understand which decisions the programmer (and in part the user) make influence the runtime behaviour. The goal is to have programs which run as fast as possible.

4.1 Position Independent Code

Never say "PIC Code"!. This is duplication.

-fPIC or not -fPIC (or -fpic)

- **it is possible to compile entirely without -fPIC shared objects do not depend on this**
- **DT_TEXTREL will be set when generating a shared object with code not code compiled as PIC**
- **before relocation text segment must be made writable**
- **relocations will modify MAP_COPY/MAP_PRIVATE memory and make it unsharable**
- **longer startup time due to copy-on-write of text pages and since all relocations must be performed at startup time**
- **behaviour of a non-PIC shared object differs in one case: "late binding" of symbols not possible**

→ **still useful for freezing interfaces**

Slide 18 **Using ELF in glibc 2.1** CYGNUS®

Position independent code is on some platforms the default. MIPS is one example. On many platforms PIC is more expensive at runtime than non-PIC. On ix86, for example, it require one of the only seven available registers. It also requires a quite slow piece of code in every global function which computes the GOT offset (stored in the dedicated register) at every function call.

Despite these disadvantages it is still worth using PIC. The startup time is significantly reduces (especially for large applciations) since the pages containing the code are not changed and can therefore be shared. For large and often used libraries like the C library this is a major advantage.

Using non-PIC in shared objects is still useful in some situations. Keeping code in archives has the problem that one cannot take advantage of the symbol versioning. The symbol versioning mechanism relies on the version refernces in shared objects. If a reference in an object file in a archive is for a function with an ABI change converting the archive into a shared object helps. It converses the references.

4.2 Other Cost Factors

When looking at the implementation of the dynamic linker carefully one can identify the factors which influence the runtime behaviour of an application. Without loosing functionality the following problems can be addressed:

Cost Factors

- **loading of every shared object requires several filesystem operations; among them 2-3 expensive mmap calls**
- **non-PIC shared objects require copy-on-write**
- **all non-jump relocations must happen at load time**
- **many shared objects mean many scopes**
- **large numbers of exported symbols mean large hash table and long chains**
- **weak definitions do not stop lookup immediately**

Slide 19 **Using ELF in glibc 2.1** **CYGNUS®**

The main factor is certainly the number os symbols which require lookup or which are in the tables we have to look through.

Reducing number of symbols

functions to global functions which are not expected to be used from the outside are handle suboptimal:

- ◇ **instead of a direct jump an indirect jump through the PLT is used**
- ◇ **the use of DT_SYMBOLIC helps only avoiding searching the possibly long list of scopes but still means an indirect jump**
- ◇ **DT_SYMBOLIC also changes the behaviour for all other symbols and possible user-intervention (LD_PRELOAD)**

→ **Solution: not exporting all symbols**

- ◇ **smaller PLT and hash table**
- ◇ **linker can convert indirect jump to direct jump**

Slide 20 **Using ELF in glibc 2.1** **CYGNUS®**

5 Symbol Versioning

The mechnisms to prevent symbols from being exported from a shared object was actually a side -product of a more general mechanism. This mechanism is one of the biggest achievements in `glibc 2.1`. Symbol Versioning allows to hide ABI changes from the using programs.

Symbol Versioning

- **allow defining more than one implementation for one symbol**
- **references are to specific versions or to no version at all (in which case the base version is used)**
- **changes in ABIs can be hidden**
- **dynamic linker can exactly determine whether available shared objects are sufficently recent**
- **use of dlvsym instead of dlsym allows to use versioning even from programming interface**

Implementation:

every use and definition of a symbol has a reference to a specific version

Slide 21 **Using ELF in glibc 2.1** **CYGNUS**

A much simpler form was implemented by Sun in their Solaris OS. Their implementation is only capable of determining whether the used library version is good enough for the program. The `glibc` implementation can do this as well but also provides versioning on sy,bol level.

Before we explain how to use symbol versioning in own code a few general words. Each library has its own version. Even if different libraries use the same version name the versions are not the same. One can think a a version as a tuple consiting of library name and version name. In the `glibc` package all libraries use names like `GLIBC_2.1` but they are different. One can see this when looking at the versions required by a binary which needs more than one library:

```
drepper@myware drepper$ ldd -v /usr/bin/makedb
        libdb.so.3 => /lib/libdb.so.3 (0x4001b000)
        libc.so.6 => /lib/libc.so.6 (0x40059000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)

        Version information:
        /usr/bin/makedb:
                libc.so.6 (GLIBC_2.1) => /lib/libc.so.6
                libc.so.6 (GLIBC_2.0) => /lib/libc.so.6
                libdb.so.3 (GLIBC_2.1) => /lib/libdb.so.3
        /lib/libdb.so.3:
                libc.so.6 (GLIBC_2.1) => /lib/libc.so.6
```

```
        libc.so.6 (GLIBC_2.0) => /lib/libc.so.6
/lib/libc.so.6:
        ld-linux.so.2 (GLIBC_2.1) => /lib/ld-linux.so.2
        ld-linux.so.2 (GLIBC_2.0) => /lib/ld-linux.so.2
```

Using the symbol versioning in own code is quite symbol. One only has to define the actual function with a different name and then use the `.symver` assembler pseudo instruction. This will only work with the GNU assembler!

Versioning Specification

- **in source code:**

```
asm (".symver internal_name_old, external_name@version_old");
asm (".symver internal_name_new, external_name@@version_new");
```

The @@ indicates the default version

- **linker map:**

```
version_new {
  global:
    ...
  local:
    *;
} version_old;
```

- **name of linker map passed using -Wl,--version-script=name**
- **linker always picks default version when linking**
- **dynamic loader looks for version matching the version in the reference**

Slide 22 **Using ELF in glibc 2.1** CYGNUS®

At any time a program must only define one default version for each symbol. The the version names after the `@` in must also all be unique for each symbol.

The linker map must match the version definition in the code otherwise strange things can happen. For each version used in a `.symver` definition there must be an appropriate entry in the linker map. The names in the linker map are used as `glob` expression, therefore can contain `*` etc. This is used in the `local:` part. It is advisable to follow this example. It means that all symbols which are not explicitly defined to be global are local. This allows the linker to omit them from the PLTs and optimize calls of these functions if available. For big libraries with many internal functions this is a major advantage.

6 Profiling of Shared Objects

Another new feature of the dynamic loader in `glibc 2.1` is the avilability of profiling of shared objects. So far only statically linked applications can be profiled. With the new dynamic loader it is possible to profile individual shared objects.

Profiling

- **setting environment variable LD_PROFILE to the name (normally SONAME) of a shared objects allows profiling of all functions from this shared object**
- **allows profiling of a running system without major performance loss**
- **all running applications using this shared object contribute to the same output**
- **no code changes necessary**
- **dynamic loader defines special lookup function which never changes the GOT/PLT to avoid rerunning the relocation**
- **sprof program can be used to examine the result output is very similar to the output of gprof**

Slide 23 Using ELF in glibc 2.1 CYGNUS

A sample output of a use of this feature follows:

```
drepper@myware drepper$ LD_PROFILE=libc.so.6 /bin/echo hi
hi
drepper@myware drepper$ sprof -p libc.so.6
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
  0.00      0.00     0.00       21     0.00           getenv
  0.00      0.00     0.00        4     0.00           malloc
  0.00      0.00     0.00        2     0.00           _IO_new_do_write
  0.00      0.00     0.00        2     0.00           __overflow
  0.00      0.00     0.00        2     0.00           atexit
  0.00      0.00     0.00        2     0.00           realloc
  0.00      0.00     0.00        2     0.00           strdup
  0.00      0.00     0.00        1     0.00           _IO_default_xsputn
  0.00      0.00     0.00        1     0.00           _IO_doallocbuf
  0.00      0.00     0.00        1     0.00           _IO_setb
  0.00      0.00     0.00        1     0.00           __fxstat64
  0.00      0.00     0.00        1     0.00           __libc_init_first
  0.00      0.00     0.00        1     0.00           __libc_start_main
```

```
  0.00      0.00     0.00        1     0.00             __secure_getenv
  0.00      0.00     0.00        1     0.00             bindtextdomain
  0.00      0.00     0.00        1     0.00             cfree
  0.00      0.00     0.00        1     0.00             exit
  0.00      0.00     0.00        1     0.00             flockfile
  0.00      0.00     0.00        1     0.00             fputs
  0.00      0.00     0.00        1     0.00             funlockfile
  0.00      0.00     0.00        1     0.00             getopt_long
  0.00      0.00     0.00        1     0.00             getpid
  0.00      0.00     0.00        1     0.00             mempcpy
  0.00      0.00     0.00        1     0.00             setlocale
  0.00      0.00     0.00        1     0.00             textdomain
  0.00      0.00     0.00        1     0.00             write
```

Profiling using this mechanism is not 100% comparable with the static binary profiling. Differences are, among others, that only functions with PLT entries are profiled. Local functions are not. But they still can appear in the output if the alarm timer went off while executing such a not exported function.

But the possibility to have arbitrary many programs using the production version of the library is something extremely useful. Some future extensions planned to improve the library performance will use the output produced by this profiling.

7 Debugging the Dynamic Loader Work

Sometimes the dynamic loader behaves differently than expected. This is normally a big problem since debugging the dynamic loader is everything but trivial. To allow debugging of the complex mechanisms implemented in the dynamic loader the version in `glibc 2.1` allows the user to ask for debugging message while the loader is working.

Debugging the ELF Loading

- **defining the environment variable LD_DEBUG selects output from the dynamic loader while doing its work**
- **any combination of the following flags is legal (separated by commas)**
  - **bindings: show result of resolving process with symbol name and the names of the involved files**
  - **files: show information about the loading process and the the loaded files**
  - **libs: brief information about libraries which habe to be loaded and which filenames are tried during the search**
  - **reloc: show information about relocation process**
  - **symbols: detailed information about resolving process**
  - **versions: show which shared object versions are requested**

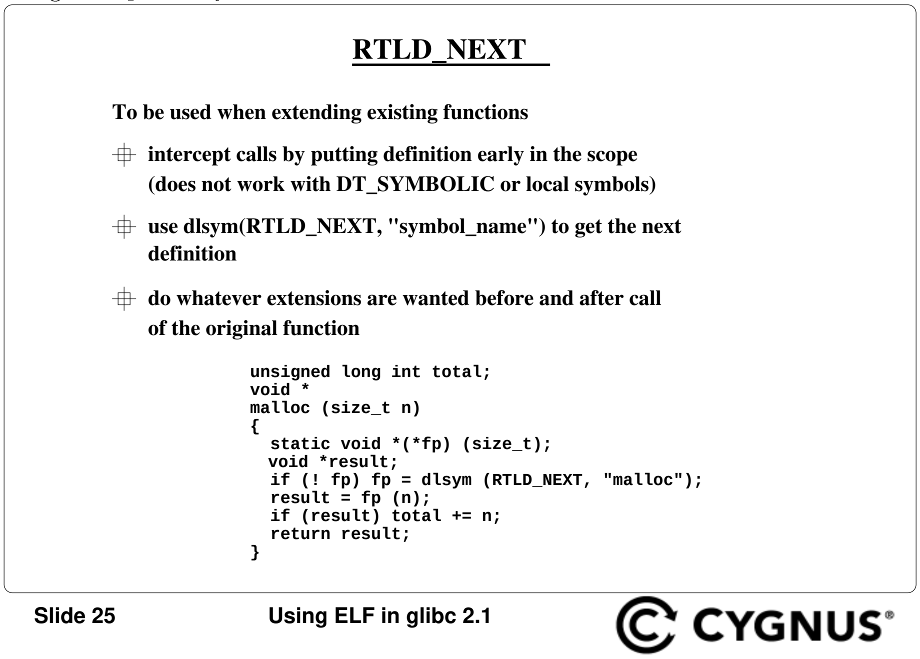
Slide 24 **Using ELF in glibc 2.1** CYGNUS®

This feature is mainly useful for people maintaining the dynamic loader code to "remote debug" a problem. It is in most cases not possible to ask the person reporting the problem to start the debugger. But the feature is also useful for normal users who, e.g., want to optimize there system or are looking for problems in their own programs.

This feature, as many others, was initially implemented in Sun"s dynamic loader of the Solaris OS.

8 The most often used feature: RTLD_NEXT

To cleanly extend the functionality of a symbol it is necessary to be able to call the implementation of the function one wants to extend. Sun added in their Solaris OS the possibility to ask the `dlsym` function about the "next" definitoin using the special symbol `RTLD_NEXT`.



This method is described here once more in detail since dirty tricks which worked for `glibc 2.0` do not work anymore for `glibc 2.1`. When extending, say, the function `open` in `glibc 2.0` one could call the function `__open` to get the original definition since this was an internal alias of `open`. Now `__open` is not available anymore and one has to use the *right* way: `RTLD_NEXT`.