# Dynamic Linking

## Program Interpreter

An executable file may have one PT_INTERP program header element. During exec(BA_OS), the system retrieves a path name from the PT_INTERP segment and creates the initial process image from the interpreter file's segments. That is, instead of using the original executable file's segment images, the system composes a memory image for the interpreter. It then is the interpreter's responsibility to receive control from the system and provide an environment for the application program.

The interpreter receives control in one of two ways. First, it may receive a file descriptor to read the executable file, positioned at the beginning. It can use this file descriptor to read and/or map the executable file's segments into memory. Second, depending on the executable file format, the system may load the executable file into memory instead of giving the interpreter an open file descriptor. With the possible exception of the file descriptor, the interpreter's initial process state matches what the executable file would have received. The interpreter itself may not require a second interpreter. An interpreter may be either a shared object or an executable file.

- A shared object (the normal case) is loaded as position-independent, with addresses that may vary from one process to another; the system creates its segments in the dynamic segment area used by mmap(KE_OS) and related services. Consequently, a shared object interpreter typically will not conflict with the original executable file's original segment addresses.
- An executable file is loaded at fixed addresses; the system creates its segments using the virtual addresses from the program header table. Consequently, an executable file interpreter's virtual addresses may collide with the first executable file; the interpreter is responsible for resolving conflicts.

## Dynamic Linker

When building an executable file that uses dynamic linking, the link editor adds a program header element of type PT_INTERP to an executable file, telling the system to invoke the dynamic linker as the program interpreter.


The locations of the system provided dynamic linkers are processor–specific.

Exec(BA_OS) and the dynamic linker cooperate to create the process image for the program, which entails the following actions:

- Adding the executable file's memory segments to the process image;
- Adding shared object memory segments to the process image;
- Performing relocations for the executable file and its shared objects;
- Closing the file descriptor that was used to read the executable file, if one was given to the dynamic linker;
- Transferring control to the program, making it look as if the program had received control directly from exec(BA_OS).

The link editor also constructs various data that assist the dynamic linker for executable and shared object files. As shown above in ''Program Header,'' these data reside in loadable segments, making them available during execution. (Once again, recall the exact segment contents are processor-specific. See the processor supplement for complete information.)

- A `.dynamic` section with type `SHT_DYNAMIC` holds various data. The structure residing at the beginning of the section holds the addresses of other dynamic linking information.
- The `.hash` section with type `SHT_HASH` holds a symbol hash table.
- The `.got` and `.plt` sections with type `SHT_PROGBITS` hold two separate tables: the global offset table and the procedure linkage table. Sections below explain how the dynamic linker uses and changes the tables to create memory images for object files.

Because every ABI-conforming program imports the basic system services from a shared object library, the dynamic linker participates in every ABI-conforming program execution.

As ''Program Loading'' explains in the processor supplement, shared objects may occupy virtual memory addresses that are different from the addresses recorded in the file's program header table. The dynamic linker relocates the memory image, updating absolute addresses before the application gains control. Although the absolute address values would be correct if the library were loaded at the addresses specified in the program header table, this normally is not the case.

If the process environment [see `exec`(BA_OS)] contains a variable named `LD_BIND_NOW` with a non-null value, the dynamic linker processes all relocation before transferring control to the program. For example, all the following environment entries would specify this behavior.

- `LD_BIND_NOW=1`
- `LD_BIND_NOW=on`
- `LD_BIND_NOW=off`

Otherwise, `LD_BIND_NOW` either does not occur in the environment or has a null value. The dynamic linker is permitted to evaluate procedure linkage table entries lazily, thus avoiding symbol resolution and relocation overhead for functions that are not called. See ''Procedure Linkage Table'' in this part for more information.

## Dynamic Section

If an object file participates in dynamic linking, its program header table will have an element of type `PT_DYNAMIC`. This ''segment'' contains the `.dynamic` section. A special symbol, `_DYNAMIC`, labels the section, which contains an array of the following structures.

**Figure 2-9: Dynamic Structure**

```
typedef struct {
        Elf32_Sword     d_tag;
        union {
                Elf32_Word      d_val;
                Elf32_Addr      d_ptr;
        } d_un;
} Elf32_Dyn;

extern Elf32_Dyn _DYNAMIC[];
```

For each object with this type, `d_tag` controls the interpretation of `d_un`.

`d_val` These `Elf32_Word` objects represent integer values with various interpretations.

`d_ptr` These `Elf32_Addr` objects represent program virtual addresses. As mentioned previously, a file's virtual addresses might not match the memory virtual addresses during execution. When interpreting addresses contained in the dynamic structure, the dynamic linker computes actual addresses, based on the original file value and the memory base address. For consistency, files do *not* contain relocation entries to ''correct'' addresses in the dynamic structure.

The following table summarizes the tag requirements for executable and shared object files. If a tag is marked ''mandatory,'' then the dynamic linking array for an ABI-conforming file must have an entry of that type. Likewise, ''optional'' means an entry for the tag may appear but is not required.

**Figure 2-10: Dynamic Array Tags, `d_tag`**

| Name | Value | `d_un` | Executable | Shared Object |
|---|---|---|---|---|
| DT_NULL | 0 | ignored | mandatory | mandatory |
| DT_NEEDED | 1 | d_val | optional | optional |
| DT_PLTRELSZ | 2 | d_val | optional | optional |
| DT_PLTGOT | 3 | d_ptr | optional | optional |
| DT_HASH | 4 | d_ptr | mandatory | mandatory |
| DT_STRTAB | 5 | d_ptr | mandatory | mandatory |
| DT_SYMTAB | 6 | d_ptr | mandatory | mandatory |
| DT_RELA | 7 | d_ptr | mandatory | optional |
| DT_RELASZ | 8 | d_val | mandatory | optional |
| DT_RELAENT | 9 | d_val | mandatory | optional |
| DT_STRSZ | 10 | d_val | mandatory | mandatory |
| DT_SYMENT | 11 | d_val | mandatory | mandatory |
| DT_INIT | 12 | d_ptr | optional | optional |
| DT_FINI | 13 | d_ptr | optional | optional |
| DT_SONAME | 14 | d_val | ignored | optional |
| DT_RPATH | 15 | d_val | optional | ignored |
| DT_SYMBOLIC | 16 | ignored | ignored | optional |

**Figure 2-10: Dynamic Array Tags,** `d_tag` (continued)

| Name | Value | d_un | Executable | Shared Object |
|---|---|---|---|---|
| DT_REL | 17 | d_ptr | mandatory | optional |
| DT_RELSZ | 18 | d_val | mandatory | optional |
| DT_RELENT | 19 | d_val | mandatory | optional |
| DT_PLTREL | 20 | d_val | optional | optional |
| DT_DEBUG | 21 | d_ptr | optional | ignored |
| DT_TEXTREL | 22 | ignored | optional | optional |
| DT_JMPREL | 23 | d_ptr | optional | optional |
| DT_LOPROC | 0x70000000 | unspecified | unspecified | unspecified |
| DT_HIPROC | 0x7fffffff | unspecified | unspecified | unspecified |

DT_NULL — An entry with a DT_NULL tag marks the end of the _DYNAMIC array.

DT_NEEDED — This element holds the string table offset of a null-terminated string, giving the name of a needed library. The offset is an index into the table recorded in the DT_STRTAB entry. See ''Shared Object Dependencies'' for more information about these names. The dynamic array may contain multiple entries with this type. These entries' relative order is significant, though their relation to entries of other types is not.

DT_PLTRELSZ — This element holds the total size, in bytes, of the relocation entries associated with the procedure linkage table. If an entry of type DT_JMPREL is present, a DT_PLTRELSZ must accompany it.

DT_PLTGOT — This element holds an address associated with the procedure linkage table and/or the global offset table. See this section in the processor supplement for details.

DT_HASH — This element holds the address of the symbol hash table, described in ''Hash Table.'' This hash table refers to the symbol table referenced by the DT_SYMTAB element.

DT_STRTAB — This element holds the address of the string table, described in Part 1. Symbol names, library names, and other strings reside in this table.

DT_SYMTAB — This element holds the address of the symbol table, described in Part 1, with Elf32_Sym entries for the 32-bit class of files.

DT_RELA — This element holds the address of a relocation table, described in Part 1. Entries in the table have explicit addends, such as Elf32_Rela for the 32-bit file class. An object file may have multiple relocation sections. When building the relocation table for an executable or shared object file, the link editor catenates those sections to form a single table. Although the sections remain independent in the object file, the dynamic linker sees a single table. When the dynamic linker creates the process image for an executable file or adds a shared object to the process image, it reads the relocation table and performs the associated actions. If this element is present, the dynamic structure must also have DT_RELASZ and DT_RELAENT elements. When relocation is ''mandatory'' for a file, either DT_RELA or DT_REL may occur (both are permitted but not required).

DT_RELASZ — This element holds the total size, in bytes, of the DT_RELA relocation table.

DT_RELAENT This element holds the size, in bytes, of the DT_RELA relocation entry.

DT_STRSZ This element holds the size, in bytes, of the string table.

DT_SYMENT This element holds the size, in bytes, of a symbol table entry.

DT_INIT This element holds the address of the initialization function, discussed in ''Initialization and Termination Functions'' below.

DT_FINI This element holds the address of the termination function, discussed in ''Initialization and Termination Functions'' below.

DT_SONAME This element holds the string table offset of a null-terminated string, giving the name of the shared object. The offset is an index into the table recorded in the DT_STRTAB entry. See ''Shared Object Dependencies'' below for more information about these names.

DT_RPATH This element holds the string table offset of a null-terminated search library search path string, discussed in ''Shared Object Dependencies.'' The offset is an index into the table recorded in the DT_STRTAB entry.

DT_SYMBOLIC This element's presence in a shared object library alters the dynamic linker's symbol resolution algorithm for references within the library. Instead of starting a symbol search with the executable file, the dynamic linker starts from the shared object itself. If the shared object fails to supply the referenced symbol, the dynamic linker then searches the executable file and other shared objects as usual.

DT_REL This element is similar to DT_RELA, except its table has implicit addends, such as Elf32_Rel for the 32-bit file class. If this element is present, the dynamic structure must also have DT_RELSZ and DT_RELENT elements.

DT_RELSZ This element holds the total size, in bytes, of the DT_REL relocation table.

DT_RELENT This element holds the size, in bytes, of the DT_REL relocation entry.

DT_PLTREL This member specifies the type of relocation entry to which the procedure linkage table refers. The d_val member holds DT_REL or DT_RELA, as appropriate. All relocations in a procedure linkage table must use the same relocation.

DT_DEBUG This member is used for debugging. Its contents are not specified for the ABI; programs that access this entry are not ABI-conforming.

DT_TEXTREL This member's absence signifies that no relocation entry should cause a modification to a non-writable segment, as specified by the segment permissions in the program header table. If this member is present, one or more relocation entries might request modifications to a non-writable segment, and the dynamic linker can prepare accordingly.

DT_JMPREL If present, this entries's d_ptr member holds the address of relocation entries associated solely with the procedure linkage table. Separating these relocation entries lets the dynamic linker ignore them during process initialization, if lazy binding is enabled. If this entry is present, the related entries of types DT_PLTRELSZ and DT_PLTREL must also be present.

DT_LOPROC through DT_HIPROC
Values in this inclusive range are reserved for processor-specific semantics.

Except for the DT_NULL element at the end of the array, and the relative order of DT_NEEDED elements, entries may appear in any order. Tag values not appearing in the table are reserved.

## Shared Object Dependencies

When the link editor processes an archive library, it extracts library members and copies them into the output object file. These statically linked services are available during execution without involving the dynamic linker. Shared objects also provide services, and the dynamic linker must attach the proper shared object files to the process image for execution. Thus executable and shared object files describe their specific dependencies.

When the dynamic linker creates the memory segments for an object file, the dependencies (recorded in DT_NEEDED entries of the dynamic structure) tell what shared objects are needed to supply the program's services. By repeatedly connecting referenced shared objects and their dependencies, the dynamic linker builds a complete process image. When resolving symbolic references, the dynamic linker examines the symbol tables with a breadth-first search. That is, it first looks at the symbol table of the executable program itself, then at the symbol tables of the DT_NEEDED entries (in order), then at the second level DT_NEEDED entries, and so on. Shared object files must be readable by the process; other permissions are not required.

NOTE

Even when a shared object is referenced multiple times in the dependency list, the dynamic linker will connect the object only once to the process.

Names in the dependency list are copies either of the DT_SONAME strings or the path names of the shared objects used to build the object file. For example, if the link editor builds an executable file using one shared object with a DT_SONAME entry of lib1 and another shared object library with the path name /usr/lib/lib2, the executable file will contain lib1 and /usr/lib/lib2 in its dependency list.

If a shared object name has one or more slash (/) characters anywhere in the name, such as /usr/lib/lib2 above or directory/file, the dynamic linker uses that string directly as the path name. If the name has no slashes, such as lib1 above, three facilities specify shared object path searching, with the following precedence.

- First, the dynamic array tag DT_RPATH may give a string that holds a list of directories, separated by colons (:). For example, the string /home/dir/lib:/home/dir2/lib: tells the dynamic linker to search first the directory /home/dir/lib, then /home/dir2/lib, and then the current directory to find dependencies.
- Second, a variable called LD_LIBRARY_PATH in the process environment [see exec(BA_OS)] may hold a list of directories as above, optionally followed by a semicolon (;) and another directory list. The following values would be equivalent to the previous example:
  - LD_LIBRARY_PATH=/home/dir/lib:/home/dir2/lib:
  - LD_LIBRARY_PATH=/home/dir/lib;/home/dir2/lib:
  - LD_LIBRARY_PATH=/home/dir/lib:/home/dir2/lib:;

  All LD_LIBRARY_PATH directories are searched after those from DT_RPATH. Although some programs (such as the link editor) treat the lists before and after the semicolon differently, the dynamic linker does not. Nevertheless, the dynamic linker accepts the semicolon notation, with the

semantics described above.

- Finally, if the other two groups of directories fail to locate the desired library, the dynamic linker searches `/usr/lib`.

**NOTE** For security, the dynamic linker ignores environmental search specifications (such as `LD_LIBRARY_PATH`) for set-user and set-group ID programs. It does, however, search `DT_RPATH` directories and `/usr/lib`.

## Global Offset Table

Position-independent code cannot, in general, contain absolute virtual addresses. Global offset tables hold absolute addresses in private data, thus making the addresses available without compromising the position-independence and sharability of a program's text. A program references its global offset table using position-independent addressing and extracts absolute values, thus redirecting position-independent references to absolute locations.

Initially, the global offset table holds information as required by its relocation entries [see ''Relocation'' in Part 1]. After the system creates memory segments for a loadable object file, the dynamic linker processes the relocation entries, some of which will be type `R_386_GLOB_DAT` referring to the global offset table. The dynamic linker determines the associated symbol values, calculates their absolute addresses, and sets the appropriate memory table entries to the proper values. Although the absolute addresses are unknown when the link editor builds an object file, the dynamic linker knows the addresses of all memory segments and can thus calculate the absolute addresses of the symbols contained therein.

If a program requires direct access to the absolute address of a symbol, that symbol will have a global offset table entry. Because the executable file and shared objects have separate global offset tables, a symbol's address may appear in several tables. The dynamic linker processes all the global offset table relocations before giving control to any code in the process image, thus ensuring the absolute addresses are available during execution.

The table's entry zero is reserved to hold the address of the dynamic structure, referenced with the symbol `_DYNAMIC`. This allows a program, such as the dynamic linker, to find its own dynamic structure without having yet processed its relocation entries. This is especially important for the dynamic linker, because it must initialize itself without relying on other programs to relocate its memory image. On the 32-bit Intel Architecture, entries one and two in the global offset table also are reserved. ''Procedure Linkage Table'' below describes them.

The system may choose different memory segment addresses for the same shared object in different programs; it may even choose different library addresses for different executions of the same program. Nonetheless, memory segments do not change addresses once the process image is established. As long as a process exists, its memory segments reside at fixed virtual addresses.

A global offset table's format and interpretation are processor-specific. For the 32-bit Intel Architecture, the symbol `_GLOBAL_OFFSET_TABLE_` may be used to access the table.

**Figure 2-11: Global Offset Table**

```
extern Elf32_Addr       _GLOBAL_OFFSET_TABLE_[];
```

The symbol _GLOBAL_OFFSET_TABLE_ may reside in the middle of the .got section, allowing both negative and non-negative ''subscripts'' into the array of addresses.

## Procedure Linkage Table

Much as the global offset table redirects position-independent address calculations to absolute locations, the procedure linkage table redirects position-independent function calls to absolute locations. The link editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. Consequently, the link editor arranges to have the program transfer control to entries in the procedure linkage table. On the SYSTEM V architecture, procedure linkage tables reside in shared text, but they use addresses in the private global offset table. The dynamic linker determines the destinations' absolute addresses and modifies the global offset table's memory image accordingly. The dynamic linker thus can redirect the entries without compromising the position-independence and sharability of the program's text. Executable files and shared object files have separate procedure linkage tables.

**Figure 2-12: Absolute Procedure Linkage Table**

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

**Figure 2-13: Position-Independent Procedure Linkage Table**

```
.PLT0:pushl 4(%ebx)
      jmp   *8(%ebx)
      nop; nop
      nop; nop
.PLT1:jmp   *name1@GOT(%ebx)
      pushl $offset
      jmp   .PLT0@PC
.PLT2:jmp   *name2@GOT(%ebx)
      pushl $offset
      jmp   .PLT0@PC
      ...
```

**NOTE** As the figures show, the procedure linkage table instructions use different operand addressing modes for absolute code and for position-independent code. Nonetheless, their interfaces to the dynamic linker are the same.

Following the steps below, the dynamic linker and the program ''cooperate'' to resolve symbolic references through the procedure linkage table and the global offset table.

1. When first creating the memory image of the program, the dynamic linker sets the second and the third entries in the global offset table to special values. Steps below explain more about these values.
2. If the procedure linkage table is position-independent, the address of the global offset table must reside in `%ebx`. Each shared object file in the process image has its own procedure linkage table, and control transfers to a procedure linkage table entry only from within the same object file. Consequently, the calling function is responsible for setting the global offset table base register before calling the procedure linkage table entry.
3. For illustration, assume the program calls `name1`, which transfers control to the label `.PLT1`.
4. The first instruction jumps to the address in the global offset table entry for `name1`. Initially, the global offset table holds the address of the following `pushl` instruction, not the real address of `name1`.
5. Consequently, the program pushes a relocation offset (*offset*) on the stack. The relocation offset is a 32-bit, non-negative byte offset into the relocation table. The designated relocation entry will have type `R_386_JMP_SLOT`, and its offset will specify the global offset table entry used in the previous `jmp` instruction. The relocation entry also contains a symbol table index, thus telling the dynamic linker what symbol is being referenced, `name1` in this case.
6. After pushing the relocation offset, the program then jumps to `.PLT0`, the first entry in the procedure linkage table. The `pushl` instruction places the value of the second global offset table entry (*got_plus_4* or `4(%ebx)`) on the stack, thus giving the dynamic linker one word of identifying information. The program then jumps to the address in the third global offset table entry

(*got_plus_8* or `8(%ebx)`), which transfers control to the dynamic linker.

7. When the dynamic linker receives control, it unwinds the stack, looks at the designated relocation entry, finds the symbol's value, stores the ''real'' address for `name1` in its global offset table entry, and transfers control to the desired destination.
8. Subsequent executions of the procedure linkage table entry will transfer directly to `name1`, without calling the dynamic linker a second time. That is, the `jmp` instruction at `.PLT1` will transfer to `name1`, instead of ''falling through'' to the `pushl` instruction.

The `LD_BIND_NOW` environment variable can change dynamic linking behavior. If its value is non-null, the dynamic linker evaluates procedure linkage table entries before transferring control to the program. That is, the dynamic linker processes relocation entries of type `R_386_JMP_SLOT` during process initialization. Otherwise, the dynamic linker evaluates procedure linkage table entries lazily, delaying symbol resolution and relocation until the first execution of a table entry.

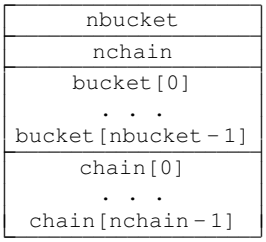
**NOTE**

Lazy binding generally improves overall application performance, because unused symbols do not incur the dynamic linking overhead. Nevertheless, two situations make lazy binding undesirable for some applications. First, the initial reference to a shared object function takes longer than subsequent calls, because the dynamic linker intercepts the call to resolve the symbol. Some applications cannot tolerate this unpredictability. Second, if an error occurs and the dynamic linker cannot resolve the symbol, the dynamic linker will terminate the program. Under lazy binding, this might occur at arbitrary times. Once again, some applications cannot tolerate this unpredictability. By turning off lazy binding, the dynamic linker forces the failure to occur during process initialization, before the application receives control.

## Hash Table

A hash table of `Elf32_Word` objects supports symbol table access. Labels appear below to help explain the hash table organization, but they are not part of the specification.

**Figure 2-14: Symbol Hash Table**




The `bucket` array contains `nbucket` entries, and the `chain` array contains `nchain` entries; indexes start at 0. Both `bucket` and `chain` hold symbol table indexes. Chain table entries parallel the symbol table. The number of symbol table entries should equal `nchain`; so symbol table indexes also select chain table entries. A hashing function (shown below) accepts a symbol name and returns a value that may be used to compute a `bucket` index. Consequently, if the hashing function returns the value *x* for some name, `bucket[`*x*`%nbucket]` gives an index, *y*, into both the symbol table and the chain table. If the symbol table entry is not the one desired, `chain[`*y*`]` gives the next symbol table entry with the same hash value. One can follow the `chain` links until either the selected symbol table entry holds the desired

name or the `chain` entry contains the value `STN_UNDEF`.

**Figure 2-15: Hashing Function**

```
unsigned long
elf_hash(const unsigned char *name)
{
        unsigned long   h = 0, g;

        while (*name)
        {
                h = (h << 4) + *name++;
                if (g = h & 0xf0000000)
                        h ^= g >> 24;
                h &= ~g;
        }
        return h;
}
```

## Initialization and Termination Functions

After the dynamic linker has built the process image and performed the relocations, each shared object gets the opportunity to execute some initialization code. These initialization functions are called in no specified order, but all shared object initializations happen before the executable file gains control.

Similarly, shared objects may have termination functions, which are executed with the `atexit`(BA_OS) mechanism after the base process begins its termination sequence. Once again, the order in which the dynamic linker calls termination functions is unspecified.

Shared objects designate their initialization and termination functions through the `DT_INIT` and `DT_FINI` entries in the dynamic structure, described in ''Dynamic Section'' above. Typically, the code for these functions resides in the `.init` and `.fini` sections, mentioned in ''Sections'' of Part 1.

**NOTE**

Although the `atexit`(BA_OS) termination processing normally will be done, it is not guaranteed to have executed upon process death. In particular, the process will not execute the termination processing if it calls `_exit` [see `exit`(BA_OS)] or if the process dies because it received a signal that it neither caught nor ignored.