Biblioteki wspólne w systemach UNIX, ładowanie dynamiczne bibliotek

Marcin Owsiany `marcin@owsiany.pl`

2003-11-17

Spis treści

1 Ogólne informacje o ładowaniu i wykonaniu kodu

1.1 Ważne pojęcia

1.1.1 Pliki obiektowe

Są to pliki przeznaczonych do bezpośredniego wykonania przez procesor (dla celów tego opracowania kategoria ta nie obejmuje on skryptów i innych plików wymagających jakiegoś rodzaju maszyny wirtualnej).

1.1.2 Linkery i loadery

Granica między tymi pojęciami jest dosyć słabo zarysowana, a co więcej zmieniała się wraz z historią komputerów (patrz rozdział 1.2).

Generalnie jednak można powiedzieć, że linker (program konsolidujący) zajmuje się powiązaniem modułów programu i tłumaczeniem adresów symbolicznych (zwanych po prostu „symbolami"), natomiast loader — załadowaniem i uruchomieniem programu.

Przykładem loadera może być `ld.so`, ładujący biblioteki dynamiczne, wewnętrzny loader systemu X Window, czy `insmod(8)`, zajmujący się ładowaniem modułów jądra.

Aby odróżnić linkowanie wykonywane przy tworzeniu programu od linkowania mającego miejsce w momencie jego uruchamiania, w pierwszym przypadku używa się nazwy programu: link editor, a w drugim — runtime linker.

1.1.3 Kod PIC i non-PIC

W systemach UNIX można rozróżnić dwa główne modele kodu:

Absolute code. W tym modelu instrukcje mogą zawierać adresy bezwzględne. Kod musi być załadowany w konkretne miejsce w wirtualnej przestrzeni adresowej, tak aby adresy bezwzględne odpowiadały adresom wirtualnym.

Position-independent code. Kod nie jest przywiązany do konkretnego adresu wirtualnej przestrzeni adresowej i może się wykonywać poprawnie niezależnie od miejsca, w jakie został załadowany. Model ten wykorzystuje dwa mechanizmy:

- instrukcje zawierają adresy względem wartości w rejestrze EIP
- jeśli potrzebny jest adres bezwzględny, oblicza go przy pomocy specjalnego kodu generowanego przez kompilator

Informacje potrzebne do przeprowadzania obliczeń adresów bezwzględnych znajdują się w globalnej tabeli offsetów (ang. *global offset table*).

1.1.4 Architektura

W teorii słowo to oznacza pewien model maszyny, ale w praktyce używane jest prawie wyłącznie na określenie konkretnej implementacji (np. i386, SPARC, m68k), i w takim też znaczeniu będzie używane w tym opracowaniu.

1.2 Wprowadzenie

1.2.1 Powstanie linkerów i loaderów

Można powiedzieć, że pierwsze pojawiły się loadery, bo o ile na początku najlepsi programiści ręcznie programowali maszyny używając przełączników, to dość szybko okazało się, że w miarę wzrostu objętości kodu i danych, dużo wygodniej przechowywać je na nośnikach zewnętrznych. Stąd loader, czyli kod ładujący program i uruchamiający go.

Udręką pierwszych programistów było ciągłe przeliczanie adresów wykorzystywanych w instrukcjach — jeśli okazało się, że do procedury trzeba było dodać jedną instrukcję, cały program trzeba było przeliczać na nowo. Powodem było to, że nazwy były przyporządkowane adresom zbyt wcześnie w cyklu tworzenia programu.

Dlatego wymyślono narzędzie zwane asemblerem, które umożliwiało posługiwanie się symbolicznymi adresami danych czy miejsc w programie. Przeliczaniem adresów przy łączeniu programu z kilku modułów zajmował się program zwany linkerem. Dwiema podstawowymi funkcjami linkera są więc:

- relokacja (dostosowywanie adresów zawartych w instrukcjach aby „pasowały” przy łączeniu modułów), oraz
- wyszukiwanie bibliotek zawierających symbole do których odwoływał się program.

1.2.2 Rozwój

Do tego momentu każdy program miał do dyspozycji całą przestrzeń adresową komputera. Wraz z pojawieniem się systemów operacyjnych sprawy zaczęły się komplikować — program musiał dzielić przestrzeń adresową z systemem operacyjnym, a często także z innymi programami. W efekcie na etapie tworzenia programu nie było wiadomo pod jaki adres zostanie załadowany. Dlatego część funkcjonalności linkerów, związana z relokacją rozprzestrzeniła się także na loadery, które musiały przeprowadzić większość relokacji w momencie ładowania.

Kolejną „rewolucją” w tym obszarze było pojawienie się sprzętowej relokacji i pamięci wirtualnej. Miało to dwie główne konsekwencje.

Z jednej strony spowodowało to uproszczenie linkerów i loaderów, ponieważ programy znów zaczęły mieć do dyspozycji całą przestrzeń adresową.

Jednak z drugiej strony w systamach takich działało jednocześnie wiele kopii tego samego kodu, pojawiła się potrzeba powtórnego wykorzystania kodu na bardzo niskim

poziomie. Stąd biblioteki wspólne, których używało wiele programów, a na dysku i w pamięci przechowywane były tylko raz.

1.2.3 Stronicowanie a programy i biblioteki

Ze względów wydajnościowych ważne jest, aby jak największa część pliku obiektowego była udostępniona procesom „tylko do odczytu". System operacyjny nie musi wtedy wymieniać stron do obszaru wymiany, ponieważ nigdy nie stają się one „brudne". Wystarczy tylko, że wczyta później ponownie potrzebną stronę z dysku.

Pojawia się tu konflikt: kodu wspólnego (a więc w szczególności bibliotek) nie można relokować różnie dla różnych programów. Można go rozwiązać na dwa sposoby:

- stosowanie mechanizmu COW (copy-on-write), które de facto nie jest wyjściem, gdyż powoduje stworzenie kopii obrazu pliku, tylko nieco później,
- rezerwację pewnej części przestrzeni adresowej dla konkretnej biblioteki,
- stosowanie kodu PIC (patrz rozdział 1.1.3).

Te dwa rozwiązania są stosowane odpowiednio w bibliotekach statycznych i dynamicznych.

1.2.4 Biblioteki statyczne i dynamiczne

Biblioteki statyczne, które zawsze były ładowane w ten sam obszar przestrzeni adresowej, stanowiły poważny problem, ponieważ administrator musiał utrzymywać bazę adresów bibliotek i pilnować, aby w przypadku żadnego programu nie pojawił się konflikt. Dla takiego modelu projektowany był format `A.OUT`.

Powstała więc koncepcja bibliotek dynamicznych, które ostatecznie były linkowane dopiero w momencie uruchomienia programu, lub nawet dopiero w momencie próby skorzystania z danego symbolu.

2 Implementacje

2.1 Format ELF

ELF (ang. *Executable and Linking Format*) jest stosunkowo nowym (lata 90-te) formatem plików obiektowych. Jest on zdefiniowany w [1], w rozdziale 4 i 5. Ma on następujące cechy:

- 32-bitowe słowa, ale jest projektowany tak, aby łatwo dał się rozszerzyć na 64-bitowe słowa,
- niezależność podstawowych struktur danych od endianowości architektury,

- wyrównanie odpowiednich danych (ang. *alignment*) wymagane przez poszczególne architektury,
- łatwa rozszerzalność na nowe architektury,
- nie wykorzystuje pól bitowych (dzięki temu ułatwia przenośność).

Wyróżniamy cztery rodzaje plików obiektowych:

Relocatable (relokowalne). Tworzone przez kompilatory i asemblery. Przed dalszym wykorzystaniem muszą zostać przetworzone przez linker (konsolidator), ponieważ zawarty w nich kod nie zawiera poprawnych adresów.

Executable (wykonywalne). Można je załadować do pamięci i wykonać. Funkcję tą wykonuje program zwany loaderem.

Shared (wspólne). Ten typ obejmuje dynamiczne biblioteki wspólne.

Core. Oznacza, że plik zawiera zrzut pamięci programu. ELF nie definiuje dla tego typu żadnych struktur poza samym nagłówkiem.

Jak widać, ELF obejmuje typy plików, które są przetwarzane zarówno przez kompilatory, asemblery i linkery a także loadery, których obszary zainteresowań różnią się między sobą:

- linkery, asemblery i kompilatory traktują plik jako zbiór logicznych *sekcji* opisanych przez tablicę nagłówków sekcji (ang. *section header table*).
- system operacyjny, a co za tym idzie — także loadery, traktują plik jako zestaw *segmentów*, opisanych przez tablicę nagłówków programu (ang. *program header table*), zwaną też tablicą nagłówków sekcji.

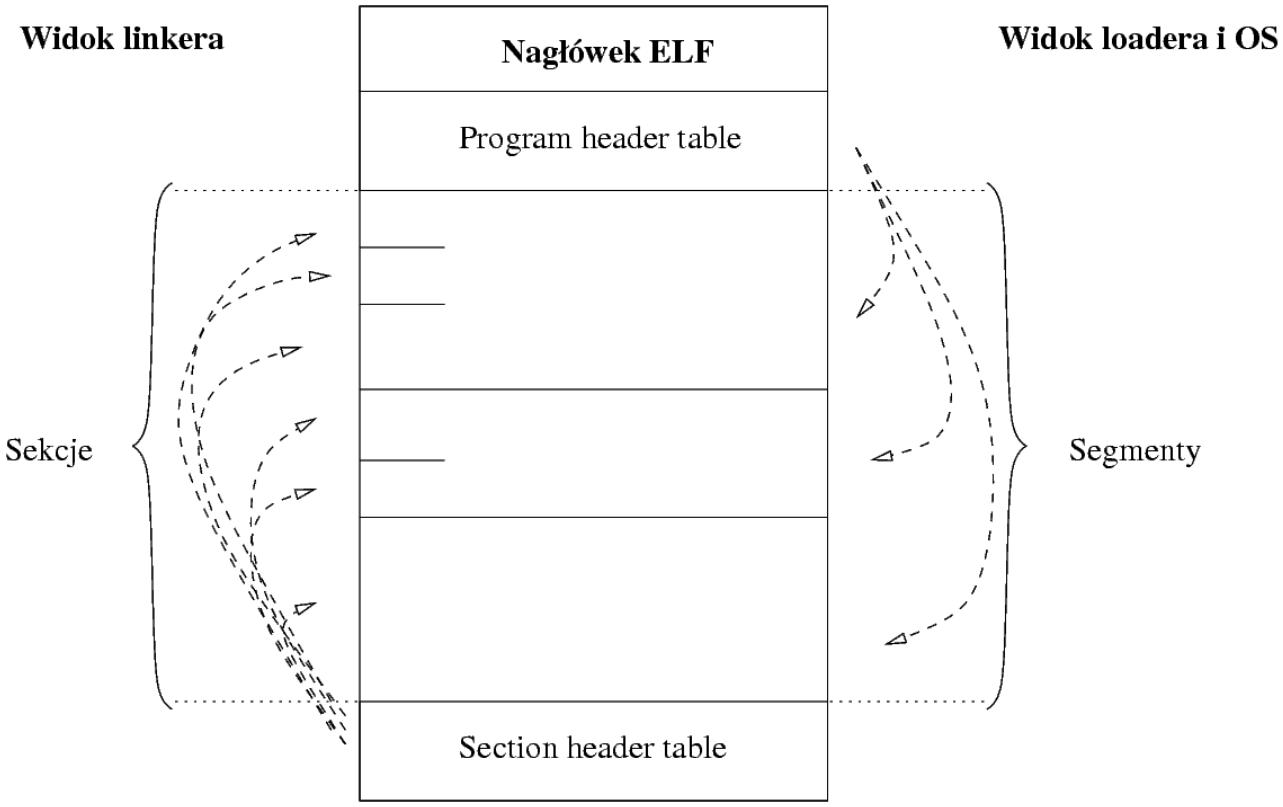
Posiadanie przez poszczególne obiekty poszczególnych części pliku ELF obrazuje tabela 1 (Uwaga: obiekty wykonywalne posiadają tablicę nagłówków sekcji zazwyczaj tylko jeśli mają pozostawioną tablicę symboli — do celów diagnostycznych).

Typ obiektu	**Sekcje**	**Segmenty**
Relocatable	tak	nie
Executable	nie	tak
Shared	tak	tak

Tabela 1: Zależności między typem obiektu a częściami pliku

Z tego powodu format ELF cechuje pewna „dwoistość” — patrz rysunek 1. Umożliwia ona wydajne korzystanie z tego formatu przez oba typy programów.

Nagłówek ELF jest skonstruowany tak, że bezproblemowo można odczytać go na dowolnej architekturze i zawiera następujące informacje:



Rysunek 1: Dwa widoki formatu ELF

Rysunek 2: `readelf -h /usr/lib/libgadu.so.2`:

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x30b0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          80096 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           40 (bytes)
  Number of section headers:         22
  Section header string table index: 21
```

- magic number
  `0x7F E L F ...`, klasa i wersja pliku, itp
- system operacyjny
- typ obiektu
- architekturę
- ...

Przykładowy nagłówek ELF przedstawiony jest na rysunku 2.

2.1.1 Sekcje

Pole `e_shoff` zawiera offset tabeli nagłówków sekcji od początku pliku, `e_shnum` określa ilość nagłówków, a `e_shentsize` ich wielkość.

Sekcje zawierają wszystkie dane w pliku ELF oprócz:

- nagłówka ELF,

- tablicy nagłówków sekcji,
- tablicy nagłówków segmentów.

Istotne informacje na temat poszczególnych sekcji (typ, zawartość) zwierają właśnie ich nagłówki.

Przykładami ważnych sekcji są:

- tablica napisów (ang. *string table*) — są to właściwie skonkatenowane napisy w formacie null-terminated, niż tablica
- tablica symboli importowanych i eksportowanych przez dany obiekt. Używa ona offsetów tablicy napisów do określenia nazw symboli.

  Informacje te czyta na przykład loader dynamiczny w celu określenia potrzebnych obiektów.

2.1.2 Segmenty

Pole `e_phoff` zawiera offset tabeli nagłówków segmentów od początku pliku, `e_phentsize` wielkość nagłówka, a `e_phnum` — ich ilość w tablicy.

Każdy z segmentów składa się z jednej lub więcej sekcji, mających wspólne cechy (np. wykonywalność, dostęp do zapisu, sposób ładowania danych).

Istotny jest zwłaszcza segment typu `PHT_DYNAMIC`. Może on między innymi zawierać:

- zero lub jeden wpis typu `DT_SONAME`. Oznacza on offset w tablicy napisów, pod którym znajduje się specjalna nazwa danej biblioteki (zwana właśnie „soname"). Nazwa ta służy do rozróżnienia między bibliotekami.
- zero lub więcej wpisów typu `DT_NEEDED`. Oznaczają one offset w tablicy napisów, pod którym znajduje się nazwa (soname) *wymaganej* do działania danego obiektu biblioteki dynamicznej.

  Kolejność występowania wpisów typu `DT_NEEDED` jest istotna, gdyż decyduje o zakresie widoczności symboli.

Przykładowy segment zawierający sekcję `.dynamic`, zawiera rys. 3.

2.2 Biblioteki statyczne

Biblioteki statyczne wbrew pozorom bardzo różnią się od bibliotek dynamicznych. Są to po prostu archiwa w formacie `ar` zawierające obiektowe pliki relokowalne oraz indeks przyspieszający dostęp do nich.

Rysunek 3: `readelf -d /usr/lib/libgadu.so.2`:

```
Dynamic segment at offset 0x13500 contains 24 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libssl.so.0.9.7]
 0x00000001 (NEEDED)                     Shared library: [libcrypto.so.0.9.7]
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 0x00000001 (NEEDED)                     Shared library: [libpthread.so.0]
 0x0000000e (SONAME)                     Library soname: [libgadu.so.2]
 0x0000000c (INIT)                       0x2908
 0x0000000d (FINI)                       0xdf50
 0x00000004 (HASH)                       0x94
 0x00000005 (STRTAB)                     0x1670
 0x00000006 (SYMTAB)                     0x770
 [...]
 0x00000000 (NULL)                       0x0
```

3 Dynamic loader

W tym rozdziale zostanie przedstawiony proces ładowania i uruchamiania procesu, ze szczególnym naciskiem na obsługę bibliotek wspólnych.

3.1 Rola jądra

3.1.1 Ładowanie obrazu pliku

Wywołanie systemowe `exec()`, po sprawdzeniu uprawnień procesu do uruchomienia pliku, ładuje nagłówek ELF, który wskazuje na tablicę nagłówków programu.

Jądro ładuje tą tablicę i `mmap()`-uje poszczególne segmenty do pamięci, nadając im odpowiednie uprawnienia (dostęp do zapisu, wykonywalność itp).

3.1.2 Start interpretera programu

W przypadku programów linkowanych dynamicznie jeden z segmentów (`p_type == PT_INTERP`) jest de facto żądaniem załadowania odpowiedniego interpretera programu. Może to być program lub biblioteka. W praktyce w systemach GNU/Linux prawie zawsze jest to `/lib/ld-linux.so.2`, a w Solaris — `/usr/lib/ld.so.1` czyli tzw. *dynamic loader*. Interpreter nie może żądać załadowania kolejnego interpretera.

Jeśli interpreter jest obiektem wspólnym, jądro ładuje jego segmenty przy pomocy `mmap()`. W tym przypadku nie będzie on kolidował z właściwym programem.

Jeśli interpreter jest obiektem wykonywalnym, jest ładowany w stały adres i może

zdażyć się, że będzie kolidował z właściwym programem. W takim przypadku rozwiązanie tej kolizji należy do obowiązków interpretera.

Jądro ładuje interpreter i przekazuje mu na stosie tzw. wektor dodatkowy (ang. *auxiliary vector*) zawierający informacje na temat tego gdzie znaleźć program oraz jak przekazać mu kontrolę po zakończeniu działania.

W końcu jądro przekazuje kontrolę interpreterowi, gdyż sam program nie jest jeszcze kompletny.

3.2 Zadania loadera dynamicznego

Zadania loadera dynamicznego to:

- ewentualne załadowanie właściwego programu,
- określenie wymaganych obiektów i załadowanie ich,
- zrelokowanie programu i bibliotek,
- zamknięcie deskryptora pliku przekazanego przez jądro, o ile w ten sposób właściwy program został przekazany interpreterowi,
- zainicjalizowanie programu i bibliotek w odpowiedniej kolejności,
- oddanie kontroli programowi w taki sposób, jakby został wywołany bezpośrednio przez `exec()`.

Ponieważ loader ten łączy w sobie pewne działania loadera i linkera, nazywany bywa także linkerem dynamicznym.

3.3 Dostępne dane

Linker ma do dyspozycji następujące sekcje:

.dynamic , typu `SHT_DYNAMIC`, zawierająca adresy innych informacji dotyczących ładowania dynamicznego.

.hash , typu `SHT_HASH`, zawierająca tablicę haszującą symboli (patrz rozdział 3.5),

.got , typu `SHT_PROGBITS`, zawierająca globalną tablicę offsetów (patrz rozdział 1.1.3),

.plt , typu `SHT_PROGBITS`, zawierająca tablicę linkowania procedur (ang. *procedure linkage table*).

3.4 Ładowanie bibliotek

Runtime linker analizuje segment `PHT_DYNAMIC` (patrz rozdział 2.1.2) danego programu, aby poznać wszystkie biblioteki, których on żąda.

Tylko względna kolejność występowania wpisów typu `DT_NEEDED` jest istotna, gdyż decyduje o zakresie widoczności symboli. Istotne są także wpisy `DT_RPATH`, zawierające offset ścieżki przeszukiwania bibliotek.

Następnie linker ładuje (de facto mapuje) kolejno obiekty wymienione we wpisach `DT_NEEDED` pierwszego poziomu, a następnie drugiego poziomu (wymienione w obiektach wymaganych). Obiekty załadowane wcześniej zostaną pominięte jeśli zostaną wymienione więcej razy.

Nazwy wymienione we wpisach `DT_NEEDED` są albo równe tym wymienionym w `DT_SONAME` bibliotek, albo (jeśli nazwa zawiera conajmniej jeden znak „/") — wprost ścieżkami do tych obiektów.

3.4.1 Mechanizmy poszukiwania bibliotek

Jeśli nazwa nie jest ścieżką, to plik, który ma zostać załadowany określają (w systemach GNU/Linux) kolejno następujące mechanizmy:

- ścieżka podana przez element `DT_RPATH`,
- ścieżka podana w zmiennej środowiskowej `LD_LIBRARY_PATH`,
- skompilowane informacje z pliku `/etc/ld.so.cache`. Plik ten jest z kolei przygotowywany przez administratora systemu z wykorzystaniem programu `ldconfig`, na podstawie ścieżek wymienionych w pliku `/etc/ld.so.conf`.
- domyślny katalog `/usr/lib`
- domyślny katalog `/lib`

Dodatkowo loader dynamiczny *przed wszystkimi innymi bibliotekami* ładuje biblioteki wymienione w zmiennej `LD_PRELOAD` i pliku `/etc/ld.so.preload`.

Dzięki wymienionym wyżej cechom można stosować takie sztuczki jak „podmiana symboli". Wystarczy napisać własną bibliotekę i umieścić ją w ścieżce wyszukiwań wcześniej od tej, która definiuje dany symbol. W takim przypadku można odwołać się do oryginalnego symbolu przy pomocy `dlsym(RTLD_NEXT, ''nazwa'')`. Taką „sztuczkę" wykorzystuje na przykład biblioteka SOCKS, która umożliwia przezroczyste korzystanie z proxy SOCKS przez dowolną aplikację.

3.4.2 Bezpieczeństwo

Oczywiście takie zachowanie loadera ma poważne konsekwencje w przypadku stosowania programów z bitem SETUID/SETGID, gdyż użytkownik mógłby podmienić dowolną

funkcjonalność uprzywilejowanego programu „podstawiając" na scieżce przeszukiwań bibliotek swoją bibliotekę. Z tego powodu loader stosuje specjalne środki ostrożności (ignoruje niektóre zmienne środowiskowe).

3.4.3 Dodatkowe istotne mechanizmy w linkerze dynamicznym systemu Solaris

W systemie Solaris ścieżki do wymaganych obiektów wspólnych podane jako `DT_NEEDED`, mogą zawierać zmienne, które są podstawiane przez linker w czasie ładowania programu.

Na przykład biblioteka `/lib/libc.so.1` wymaga biblioteki dodatkowej:

```
0x7ffffffd (AUXILIARY)  Auxiliary library: [/usr/platform/$PLATFORM/lib/libc_psr.so.1]
```

W systemie `SunOS student 5.8 Generic_108528-19 sun4u sparc SUNW,Ultra-80` ładowany zostanie plik `/usr/platform/SUNW,Ultra-80/lib/libc_psr.so.1`.

Plik konfiguracyjny runtime linkera znajduje się w systemie Solaris w pliku `/var/ld/ld.config` dla obiektów 32-bitowych, lub `/var/ld/64/ld.config` dla obiektów 64-bitowych. Alternatywne pliki konfiguracyjne można podać przy pomocy zmiennej `LD_CONFIG`.

Zmienna środowiskowa `LD_LOADFLTR` umożliwia „podstawienie" alternatywnej biblioteki w miejsce innej.

Zmienna `LD_NOLAZYLOAD` powoduje ignorowanie flagi „lazy loading" ustawionej przez link editor, i załadowanie danej zależności tak szybko jak to możliwe.

Zmienna `LD_PROFILE` umożliwia profiling wybranej biblioteki.

Dodatkowo system ten zawiera bilbiotekę `/usr/lib/0@0.so.1`, umożliwiającą poprawne działanie bardzo starym programom, pochodzącym z czasów, gdy w systemach UNIX dereferencja zerowego wskaźnika nie powodowała błędu, ale zwracała pusty napis.

3.5 Relokacja

O ile ładowania bibliotek i inicjalizacji nie da się raczej przyspieszyć, to relokacja jest dość kosztownym procesem. Jej złożoność jest conajmniej rzędu $O(rs)$, gdzie r jest liczbą relokacji, a s — liczbą symboli zdefiniowanych we wszystkich powiązanych obiektach.

Co więcej relokacja jest skomplikowana, ponieważ to, jaki adres zostanie przyporządkowany danemu symbolowi zależy od zakresu (ang. *lookup scope*), który z kolei zależy od kolejności ładowania bibliotek.

Jednym sposobów przyspieszenia relokacji jest stosowanie tablicy haszującej, gdzie hasze obliczane są z nazwy symbolu. W obrębie poszczególnych kubełków haszujących symbole są wyszukiwane przy pomocy pełnego porównania nazw.

Wynika z tego, że długie nazwy symboli oraz nazwy symboli o długich identycznych przedrostkach wydłużają czas relokacji. Niestety obie te cechy wymusza obecna implementacja C++. Jest to jeden z powodów, dla których programy pisane w tym języku mogą działać wolniej.

Warto wiedzieć, że linker GNU stara się przyspieszyć relokację stosując dużą ilość małych kubełków przy żądaniu optymalizacji od kompilatora.

3.6 Inicjalizacja

Linker wywołuje wszystkie funkcje inicjalizacyjne dostarczane przez obiekty wspólne i (jeśli ma to miejsce) sam program. Domyślnie są one wywoływane w odwrotej kolejności niż topologicznie posortowane zależności. W przypadku istnienia cykli, są one wywoływane w porządku sortowania po usunięciu cyklu.

Aby poznać kolejność inicjalizacji bibliotek można użyć narzędzia `ldd(1)`.

3.7 Przekazanie kontroli aplikacji

Loader może stosować technikę zwaną „lazy relocation" (in. „late binding"), unikając w ten sposób linkowania symboli funkcji, które nigdy nie zostaną wywołane. Jednak jeśli w środowisku programu znajduje się zmienna `LD_BIND_NOW` i ma ona niepustą wartość, to linker odnajduje wszystkie nazwy tak wcześnie jak to możliwe (jeszcze przed przekazaniem kontroli właściwemu programowi).

Właściwy program może explicite korzystać w czasie działania z usług linkera w celu otwarcia dodatkowych obiektów (`dlopen()`) i dowiązania potrzebnych symboli (`dlsym()`).

3.8 Finalizacja

Linker wywołuje funkcje finalizacyjne bibliotek w kolejności odwrotnej do kolejności wywoływania funkcji inicjalizacyjnych.

4 `dlopen()` and friends

Opisane wyżej mechanizmy są realizowane w przezroczysty sposób przez linker dynamiczny. Programista ma jednak możliwość bardziej elastycznego ładowania kodu. Służą do tego funkcje `dlopen()`, `dlsym()` i `dlclose()`.

Oto przykładowe użycie tych funkcji:

```
void    *handle;
int     *iptr, (*fptr)(int);

/* open the needed object */
handle = dlopen("/foo/bar/libfoo.so", RTLD_LOCAL | RTLD_LAZY);

/* find the address of function and data objects */
*(void **)(&fptr) = dlsym(handle, "my_function");
iptr = (int *)dlsym(handle, "my_object");

/* invoke function, passing value of integer as a parameter */
(*fptr)(*iptr);
```

Istnieją dwie zarezerwowane wartości specjalne dla pierwszego argumentu `dlsym()`:

`RTLD_DEFAULT` — szukanie symbolu wykonywane jest w zakresie globalnym, to znaczy wynikowy obiekt ma takie samo znaczenie jakby został użyty bezpośrednio,

`RTLD_NEXT` — określa obiekt następny po tym, który jest zdefiniowany przez daną nazwę. Takieg zachowanie umożliwia tworzenie hierarchii obiektów, które oddelegowują działania do swych poprzedników. Patrz rozdział 3.4.1.

5 Pisanie, tworzenie i zarządzanie bibliotekami

Ten rozdział zawiera bardziej praktyczne informacje przydatne zwłaszcza dla programistów i administratorów.

5.1 Pisanie kodu biblioteki

W zasadzie pisanie kodu biblioteki nie różni się od pisania kodu zwykłego programu. Współdzielenie biblioteki przez wiele procesów jest dla piszącego przezroczyste, ponieważ wszystkie dane biblioteki są albo tylko do odczytu, albo umieszczone w segmencie „copy on write", a więc dostępne tylko dla danego procesu.

Inną kwestią jest pisanie bibliotek „thread-safe", ale tutaj też w zasadzie obowiązują takie same reguły jak przy pisaniu bezpiecznych ze względu na wątki programów (trzeba zwracać specjalną uwagę na dane statyczne).

5.2 Tworzenie bibliotek dynamicznych

Kod dla bibliotek wspólnych należy kompilować do plików obiektowych z flagą `-fPIC`, co wymusza generowanie kodu „position-independent" (patrz rozdział 1.1.3). Co prawda może się wydawać, że biblioteki kompilowane bez tej flagi działają mimo wszystko, ale faktem jest, że wcale nie musi tak być na wszystkich architekturach. Narażone są na to zwłaszcza maszy HP PA-RISC, w których są problemy z linkowaniem kodu PIC z kodem non-PIC.

```
cc -fPIC -o plik1.o plik1.c
[...]
cc -fPIC -o plikm.o plikm.c
```

Następnie należy zlinkować tak wytworzone pliki relokowalne w bibliotekę:

```
cc -shared -o libfoo.so.N -Wl,-z,defs -Wl,-soname,libfoo.so.N \
  plik1.o ... plikm.o -llib1 ... -libx
```

Flaga `-Wl,-z,defs` zabrania linkerowi generowania biblioteki w przypadku gdy nie jest możliwe odnalezienie jakichś symboli (importowanych z bibliotek `lib1` do `libx`).

Flaga `-Wl,-soname,libfoo.so.N` ustawia wpis `DT_SONAME` w bibliotece na `libfoo.so.N`. `N`, czyli tak zwana wersja ABI, powinno być równe 0 w pierwszej wersji biblioteki. Patrz rozdział 5.5.

5.3 Przydatne narzędzia

`readelf` — wypisuje „niskopoziomowe" informacje na temat zawartości pliku obiektowego

`file` — podaje m.in. informację czy plik zawiera tablice symboli i debugowania

`strip` — „obdziera" plik z niepotrzebnych tablic symboli, dzięki czemu zmniejsza jego wielkość

`ldd` — pokazuje jakie biblioteki wymagane są przez dany obiekt i które zostaną załadowane

`strings` — wypisuje napisy zawarte w pliku obiektowym

`nm` — wypisuje symbole zawarte w pliku obiektowym

`strace`, `ltrace` — śledzą wykonanie programu

5.4 `libtool`

`libtool` jest nakładką na toolchain wykonaną w postaci skryptu shella, która ukrywa przed programistą komplikacje związane z obsługą bibliotek na danej platformie. Ponadto ułatwia tworzenie takich bardziej zaawansowanych obiektów jak pluginy itp.

Więcej informacji na ten temat dostępne jest w [2].

5.5 Zgodność binarna bibliotek

Dany program X, zlinkowany (przy tworzeniu) z biblioteką Y może być linkowany dynamicznie z dowolną wersją tej biblioteki, pod warunkiem, że jest ona binarnie zgodna z wersją, z którą X był linkowany przy jego tworzeniu.

Za interfejsy biblioteki uznajemy:

- funkcje (i ich prototypy) — zarówno nazwy jak i typy argumentów i zwracanych wartości,
- niestatyczne zmienne globalne — nazwy i typy,
- definicje struktur, unii i typów

Biblioteki *tracą* binarną zgodność przy:

- usuwaniu interfejsów,
- zmianie typów interfejsów,
- istotnej niezgodnej zmianie semantyki działania.

Autor biblioteki aby zanaczyć utratę zgodności binarnej ma obowiązek „podbić" wersję ABI (application binary interface). Ponieważ zmiana wersji ABI powoduje zmianę soname, programy muszą być przekompilowane z nową wersją biblioteki.

Takie działanie ma na celu ochronę użytkowników biblioteki przed używaniem niezgodnej wersji biblioteki ze starymi programami.

W celu uniknięcia konieczności częstych zmian ABI zaleca się separację danych eksponowanych użytkownikowi biblioteki od danych prywatnych (prywatne wskaźniki), lub czasem tworzenie struktur „na wyrost", co umożliwi późniejsze dokładanie do nich pól.

Stosuje się także zaawansowane techniki jak wersjowanie symboli (ang. *symbol versioning*), które w systemie GNU/Linux jest rozwinięciem wersjowania z systemu SunOS.

Literatura

[1] The Santa Cruz Operation. *System V application binary interface. Edition 4.1.* `http://www.caldera.com/developers/devspecs/gabi41.pdf`, 1997.

[2] GNU. libtool manual. info manual.

[3] Ulrich Drepper. GNU C library version 2.3, `http://people.redhat.com/drepper/lt2002talk.pdf`. 2002.

[4] The Santa Cruz Operation. *System V application binary interface. Intel386 Architecture Processor Supplement.* http://www.caldera.com/developers/devspecs/abi386-4.pdf, 1997.

[5] Gilad Ben-Yossef. Dynamic linking, `http://www.benyossef.com/presentations/dlink/`. 2002.

[6] John R. Levine. *Linkers and loaders.* Morgan Kaufmann Publishers, 2000.

[7] Caldera. *Developer's topics.*

[8] SunOS 5.8. ld.so.1(8). man page.

[9] Różni autorzy. ld.so(8). man page.

[10] Junichi Uekawa. Debian library packaging guide, http://www.netfort.gr.jp/ dancer/column/libpkg-guide/libpkg-guide.html. 2003.

[11] The GNOME project. Gnome programming guidelines, http://developer.gnome.org/doc/guides/programming-guidelines/binary.html.