9 Kompilacja, biblioteki

Proces kompilacji

Program napisany w języku C czy C++ nie jest bezpośrednio uruchamiany na komputerze. Kod źródłowy musi zostać wpierw przetłumaczony na język maszynowy. Proces ten ogólnie nazywamy *kompilacją*, acz pod takim terminem kryje się też jeden z jej etapów.

Proces kompilacji składa się z następujących etapów:

* **Preprocessing** – wczytywanie wszystkich `include`-owanych plików, podstawianie `define`-ów i wykonywanie makr.
* **Kompilacja** – parsowanie kodu, sprawdzanie typów, optymalizacja. Pojedynczy plik źródłowy zostaje zamieniony na tzw. pliki obiektowy *object file*. Plik obiektowy zawiera kod maszynowy z „dziurami". Dziury te to referencje do zmiennych i funkcji, które nie są w nim zdefiniowane.
* **Linkowanie** – łączenie wielu plików obiektowych w jedną całość. Wszystkie zmienne i funkcje powinny być zdefiniowane, luki w plikach obiektowych zostają wypełnione.

Kompilując kod w języku C lub C++, wszystkie te kroki można wykonać jednym programem. Zwykle jest to `gcc` dla języka C, oraz `g++` dla języka C++.

Kompilatory GCC dominują w środowisku linuxowym, warto jednak wiedzieć, że alternatywa jest możliwa. Przykładem jest `clang` i `clang++` oparte o kompilator LLVM. Pod Windowsem dominuje Visual C/C++. Innym przykładem są kompilatory specjalizowane dla danej architektury, np. `icc` dla procesorów Intel.

Załóżmy, że mamy niewielki projekt napisany w języku C, składający się z plików:

main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "power.h"
int main(int argc, char** argv) {
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    printf("%d\n",power(a,b));
    return 0;
}
```

power.h:

```
int power(int base, int exp);
int sqr(int x);
```

power.c:

```
#include "power.h"
int power(int base, int exp) {
    if (exp==0) return 1;
    if (exp % 2) return
       base*power(base,exp-1);
    else return
       sqr(power(base,exp/2));
}
```

sqr.c:

```
#include "power.h"
int sqr(int x) { return x*x; }
```

Najprostrzy sposób by taki projekt skompilować i zlinkować, to po prostu wywołać:
`gcc main.c power.c sqr.c`

Zostanie wtedy utworzony plik `a.out` który jest kompletnym programem. Jeśli chcemy określić inną nazwę pliku docelowego, przekazujemy parametr `-o <nazwa_pliku>`:

```
gcc main.c power.c sqr.c -oprogram
gcc main.c power.c sqr.c -o program   # (spacja po -o nie ma znaczenia)
```

Możemy jednakże ten sam zestaw skompilować sekwencją:

```
# kompilacja plikow obiektowych
gcc -c main.c -o main.o     # bez -c sie nie powiedzie
gcc -c power.c -o power.o
gcc -c sqr.c -o sqr.o

# zlinkowanie w koncowy program
gcc main.o power.o sqr.o -o program
```

Formy pośrednie

Jeśli potrzebujemy, możemy przetworzyć plik źródłowy języka C do pośredniej postaci. Możliwe parametry to:

`-E`	uruchamia jedynie preprocsor
`-S`	kompiluje kod do asemblera
`-S -emit-llvm`	clang kompiluje do asemblera wirtualnej maszyny LLVM
`-c`	kompiluje kod to pliku obiektowego

W pierwszym etapie, preprocesor C jest wykonany na danym pliku źródłowym. Powstały plik zawiera wszystkie makra i działania preprocesora już zrealizowane. W szczególności, wszystkie includowane pliki są połączone w jedną całość. Aby kolejne etapy kompilacji mogły się zorientować skąd kod pochodzi (np. by wyświetlić błąd), w tak przerobionym pliku pojawiają się informacje o pochodzeniu linii.

```
$ gcc -E main.c > main.e

$ head -9 main.e
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "main.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
```

```
$ wc -l main.e
1920 main.e

$ tail -9 main.e
# 1 "power.h"
int power(int base, int exp);
# 4 "main.c" 2
int main(int argc, char** argv) {
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    printf("%d\n",power(a,b));
    return 0;
}
```

Pojawienie się sekwencji `<linia> <plik> <flagi>` dotyczy bezpośrednio występującej po niej linii i oznacza kolejno:

* Number linii w pliku źródłowym
* Nazwę pliku źródłowego
* Dowolna kombinacja flag:
  * **1** – początek nowego pliku
  * **2** – powrót do pliku po pierwotnego pliku po wykonaniu `#include`
  * **3** – tekst nagłówka systemowego (tak by nie wypisywać ostrzeżeń)
  * **4** – domyślne otoczenie `extern ""C""`

W kolejnych etapach kompilator tworzy **kod asemblera**. Asembler to język niskiego poziomu, czesto powiązany już ściśle z architekturą na której pisany jest dany program. Asembler jest jednak nadal czytelny i pozwala na niskopoziomowe manipulacje kodem.

W następnym kroku kod asemblera konwertowany jest na **język maszynowy**. Ponieważ jednak pojedyńczy plik może korzystać z funkcji i symboli zdefiniowanych gdzie indziej, nie jest to jeszcze program – tylko wspomniany wczesniej **plik obiektowy**.

Pliki obiektowe

Plikiem obiektowym nazywamy dowolny plik zawierający fragmenty kodu wykonywalnego. Plik ten podzielony jest na segmenty, najczęściej są to:

* **Header** – nagłówek opisujący zawartość pliku
* **Code segment**/**Text segment** – zawiera skompilowany, maszynowy kod
* **Data segment** – zainicjowane zmienne statyczne i globalne
* **BBS segment** – zmienne statyczne i globalne inicjowane samymi zerami

Dodatkowe segmenty mogą zawierać informacje o zewnętrznych definicjach wykorzystywanych w kodzie, informacji o sposobach relokacji obiektu (gdy dwa lub więcej plików obiektowych jest łączonych), informacje o debugowaniu.

Pliki obiektowe powstają w wyniku kompilacji pojedyńczych plików źródłowych. Takie pliki można łączyć (*linking*) tworząc większe pliki obiektowe zwykle zwanymi bibliotekami. Plik obiektowy definiujący wszystkie symbole które używa, oraz zawierający punkt startu (*entry point*) jest programem. Ponadto, plik obiektowy może być tworzony i przekształcany w pamięci RAM – co następuje przy dynamicznym linkowaniu.

W systemie linux, wszystkie pliki `.o` (pojedyńcze twory kompilacji), `.a` (biblioteki statyczne), `.so` (biblioteki dynamiczne) i wykonywalne programy mają taką samą strukturę pliku obiektowego.

Komenda `nm` pozwala nam zajrzeć w zawartość danego pliku obiektowego.

```
$ nm main.o
                 U atoi
0000000000000000 T main
                 U power
                 U printf
```

Dowiadujemy się, że `main.o`:

* korzysta z trzech niezdefiniowanych symboli (`U`): `atoi`, `power`, `printf`,
* definiuje symbol `main` pod zadanym adresem.

Jak widać plik `main.o` zawiera tylko elementy które używaliśmy w naszym pliku źródłowym. Gdy sprawdzimy zawartość pliku wykonywalnego, powstałego po zlinkowaniu `main.o`, `power.o` i bibliotek standardowych, lista będzie znacznie dłuższa i ciekawsza:

```
$ nm main
                 U atoi@@GLIBC_2.2.5
0000000000601040 B __bss_start
0000000000601040 b completed.7585
0000000000601030 D __data_start
0000000000601030 W data_start
00000000004004a0 t deregister_tm_clones
0000000000400520 t __do_global_dtors_aux
....
                 U __libc_start_main@@GLIBC_2.2.5
0000000000400566 T main
00000000004005c8 T power
                 U printf@@GLIBC_2.2.5
00000000004004e0 t register_tm_clones
0000000000400628 T sqr
0000000000400470 T _start
0000000000601040 D __TMC_END__
$ nm main | wc -l
38
$ wc main -c
8776 main
```

Wyjaśnienie oznaczeń literowych jest następujące:

* `B` – symbol jest w sekcji BBS. Są to niezainicjalizowane zmienne statyczne i globalne.
* `D` – zainicjalizowana zmienna w sekcji Data.
* `R` – zainicjalizowana zmienna w sekcji read-only. Czyli innymi słowy: stałe.
* `W` – symbol słaby (weak), który może zostać nadpisany przez inny o tej samej nazwie podczas linkowania
* `T` – symbol w sekcji text – reprezentujący fragment kodu źródłowego. Symbolem T oznaczane są wszystkie funkcje.
* `U` – Undefined – symbol który nie ma podanej definicji.

Jak widać, wszystkie funkcje które zdefiniowaliśmy w różnych plikach `.c` są teraz w jednym pliku. Może dziwić natomiast, że funkcje `printf` i `atoi`, które przecież używamy w naszym programie pozostają niezdefiniowane. Dzieje się tak dlatego, że domyślnie standardowa biblioteka C linkowana jest *dynamicznie*. Gdy program zostanie uruchomiony, jego plik przenoszony jest do pamięci operacyjnej komputera. Dopiero tam łączony jest on z biblioteką standardową.

Komenda `readelf` pozwala na dokładniejszą analizę plików obiektowych. Za jej pomocą

możemy wypisać zawartość dynamicznej tablicy symboli:

```
$ readelf --dyn-syms main
Symbol table '.dynsym' contains 5 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND
        printf@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND
        __libc_start_main@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND
        __gmon_start__
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND
        atoi@GLIBC_2.2.5 (2)
```

Kompilując można zażądać by biblioteka standardowa została połączona statycznie:

```
$ gcc main.o power.o -static -o main
$ nm main | wc -l
1751
$ nm main | grep "\( main$\)\|\( printf$\)\|\( atoi$\)"
000000000040dcb0 T atoi
00000000004009ae T main
000000000040f8c0 T printf
$ wc main -c
908784 main
```

Wszystkie funkcje są teraz w sekcji text. Tak stworzony plik można lepiej zoptymalizować, nie zależy on też już od żadnych zewnętrznych bibliotek. Natomiast poważną wadą jest rozmiar pliku – w wyniku statycznego linkowania wzrósł on o około 900KB.

Biblioteki statyczne

Biblioteka statyczna to po prostu kolekcja wielu plików obiektowych złączonych w jedną całość. Tworząc bibliotekę statyczną nie wywołujemy linkera, tylko umieszczamy wszystkie pliki obiektowe obok siebie.

```
$ ar rcs libpower.a power.o sqr.o
```

Argumenty `ar` są następujące:

* Pierwszy argument iformuje co chcemy zrobić z archiwum:
  * **d** – usunąc wybrane pliki z archiwum
  * **m** – przesunąć pliki (zmienić kolejność) plików w archiwum
  * **r** – wstawić pliki do archiwum, usuwając ewentualne duplikaty
  * **t** – wypisać pliki w archiwum
  * **c** – utworzyć archiwum jeśli nie istnieje
  * **s** – zaktualizować indeks symboli w archiwum
* Drugi argument to nazwa archiwum który tworzymy. Standardowo biblioteki statyczne zaczynają się od `lib` i kończa się `.a` (or „archiwum")
* Pozostałe argumenty to pliki obiektowe które chcemy dodać

Teraz możemy sprawdzić zawartość całego archiwum:

```
$ nm libpower.a

power.o:
0000000000000000 T power
                 U sqr

sqr.o:
0000000000000000 T sqr
```

jak widać oba pliki obiektowe są w środku. Nie nastąpiło natomaist linkowanie. Plik `power.o` dalej zawiera niezdefiniowane `sqr`, mimo że plik obok właśnie to definiuje.

Tak powstały plik archiwum można wykorzystać w kolejnym kroku kompilacji w dwojaki sposób.

* Można podać bibliotekę tak jak zwykły argument to gcc:
  `gcc main.o libpower.a -o main`
* Podłączyć bibliotekę parametrem `-l`. Taka biblioteka jest wyszukiwana w standardowych katalogach bibliotek. Jeśli chcemy dołączyć w ten sposób bibliotekę znajdującą się w innym katalogu, trzeba wpierw podać ten katalog parametrem `-L`, np:
  `gcc main.o -L. -lpower -o main`
  Zauważmy, ze przy parametrze `-l` nie piszemy już początkoweg `lib` ani końcowego `.a`. Wartości te są dopisywane automatycznie.

Tak naprawdę rozwiązanie drugie jest bardziej uniwersalne: jak zobaczymy później, w ten sam sposób linkujemy bibliotekę dynamiczną. Jeśli dana biblioteka występuje w formie zarówno statycznej co i dynamicznej to ta druga wersja ma priorytet, chyba że dodamy argument `-static` do `gcc`.

Gdy uruchamiamy linkera z plikami obiektowymi wymienionymi w argumentach, są one dołączane bezwarunkowo. Nie jest tak jednak dla plików zawartych w archiwum! Plik obiektowy z archiwum będzie zawarty jedynie wtedy, gdy w danej chwili – czytając od lewej – pojawił się jakiś niezdefiniowany symbol który dany plik archiwum definiuje. Jeśli więc napiszemy:

```
gcc libpower.a main.o -o main
```

To biblioteka `libpower.a` będzie sprawdzona jako pierwsza, ale wtedy nie ma jeszcze żadnego niezdefiniowanego symbolu. Z tego powodu zostanie ona w całości pominięta. Później dopiero pojawia się `main.o` ze swoim niezdefiniowanym `power`, prowadząc do błedu:

```
main.o: In function ‘main’:
main.c:(.text+0x46): undefined reference to ‘power’
collect2: error: ld returned 1 exit status
```

Dlatego takie archiwa trzeba łączyć we właściwej kolejności. Przy cyklach zależności trzeba takie archiwa wymienić wielokrotnie.

Biblioteki dynamiczne

Biblioteki dynamiczne to specjalne pliki obiektowe które są łączone z programem w pamięci operacyjnej komputera, gdy program już działa. Takie podejście ma kilka zalet:

* Kod programu nie zawiera kodu biblioteki. Gdy wiele programów korzysta z tej samej biblioteki, oszczędność miejsca na dysku może być spora.
* Program nie musi być aktualizowany za każdym razem gdy biblioteka zostanie zmieniona. Korekty takie jak np. naprawa bugów, sprawniejsza implementacja, lub uwzględnienie nowych możliwości jądra systemu zostaną automatycznie uwzględnione przy kolejnym uruchomieniu programu. Pełna rekompilację trzeba wykonać tylko gdy zmieni się interfejs biblioteki.

Natomiast jedyną wadą jest brak możliwości optymalizacji. Nawet najlepszy kompilator nie jest wstanie wymyśleć co dana funkcja biblioteczna robi, jeśli jej kod zostanie zlinkowany dopiero podczas uruchomienia programu. Dlatego linkowanie statyczne (z włączonym tzw. „link-time optimisation") stosowane jest tam gdzie czas działania programu jest krytyczny. W większości programów stosowanych na co dzień tak nie jest.

Zanim stworzymy taką bibliotekę, musimy nieco zmienić sposób w jaki kompilowany jest kod źródłowy. Biblioteka dynamiczna wymaga bowiem aby wszystkie instrukcie operujące na adresach, takie jak instrukcje skoku czy odwołania do wartości w kodzie powinny działać niezależnie od tego w którym miejscu pamięci dany kod się znajduje. Dlatego odwołania absolutne (np. skocz pod adres 0x148725fd) trzeba zamienić na odwołania relatywne (np. skocz o 0x00005ad6 bajtów wprzód). Kompilację z odwołaniami relatywnymi wybieramy parametrem `-fPIC` (*Position Independent Code*)

```
$ gcc power.c -fPIC -c o power.o
$ gcc sqr.c -fPIC -c o sqr.o
```

Gdy mamy już odpowiednie pliki obiektowe, możemy je połączyć w bibliotekę dynamiczną:

```
$ gcc -shared power.o sqr.o -o libpow.so
```

Jeśli na tym etapie dostajecie komunikat w formie:

```
/usr/bin/ld: power.o: relocation R_X86_64_PC32 against symbol ‘power’
   can not be used when making a shared object; recompile with -fPIC
```

oznacza to, że któryś z plików obiektowych nie został skompilowany z parametrem `-fPIC`. Komunikat nawet podpowiada co trzeba zrobić.

Gdy już mamy bibliotekę dynamiczną, można ją wykorzystać budując program:

```
$ gcc main.c -c -o main.o
$ gcc main.o -L. -lpow -o main
```

Taki układ `-L` i `-l` już spotkaliśmy przy linkowaniu statycznym. `-L` to katalog dodatkowy w którym poszukiwane będą biblioteki, a `-l` to rdzeń nazwy biblioteki którą chcemy wykorzystać.

Jednakże, w tym przypadku definicje zawarte w bibliotece dynamicznej nie zostaną umieszczone w kodzie programu głównego. Jedynie informacja o tym, że odpowiedni symbol znajduje się w pliku `libpow.so` zostanie zapisany. Proces linkowania zostanie przeprowadzony automatycznie w momencie uruchomienia programu.

Możemy upewnić się:
```
$ readelf --dyn-syms libpow.so | grep power
    12: 00000000000006c0    96 FUNC    GLOBAL DEFAULT   12 power
$ readelf --dyn-syms main | grep power
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND power
```

Mamy więc potwierdzenie, że symbol `power` pojawia się w obu plikach: `main` i `libpow.so`, ale jest zdefiniowany jedynie w tym drugim.

Gdy spróbujemy uruchomić program, dostaniemy następujący komunikat:
```
$ ./main 2 4
./main: error while loading shared libraries: libpow.so: cannot open
   shared object file: No such file or directory
```

Komunikat ten został wygenerowany przez nasz program podczas automatycznego linkowania bibliotek dynamicznych. Błąd ten jest spowodowany tym, że program poszukuje biblioteki `libpow.so` jedynie w standardowych katalogach, takich jak `/usr/local/lib` i `/usr/lib` – a tam nie ma naszej biblioteki.

Możemy dodać własny katalog w którym biblioteki mają być wyszukiwane. Najlepiej zrobić to podczas linkowania, dodając parametr `-rpath <katalog>` do linkera.
```
$ gcc main.o -L. -lpow -Wl,-rpath=. -o main
```

Parametr `-Wl` powoduje przekazanie następującym po nim parametrze (w naszym przypadku `-rpath`) bezpośrednio do programu linkującego (`ld`). Tak skompilowany program zadziała już poprawnie!

Jeśli nie chcemy "na sztywno" zapisywać katalogu w programie, można to też kontrolować zmienną środowiskową `LD_LIBRARY_PATH`, która wymienia wszystkie katalogi dodatkowe w których powinna być wyszukiwana dana biblioteka dynamiczna. Możemy zatem wpisać:
```
$ export LD_LIBRARY_PATH=.
$ ./main 2 4
16
```

Teraz biblioteka została skutecznie załadowana i program został uruchomiony.