

* + cpp -o main\_p.c main.c
    - Uruchamia preprocesor języka C. W tym przypadku „importuje” wszystkie komponenty, które są uwzględnione w main.c. Na początku pliku napisaliśmy #include „stdio.h”, więc preprocesor załączył elementy tego pliku w main\_p.c
  + gcc -S main\_p.c
    - Uruchamia kompilator gnu C. Ma za zadanie (-S) skompilować main\_p.c do main\_p.s bez linkowania itd.
  + As -o main.o main\_p.s
    - Uruchamia assembler. Konwertuje kod z języka assemblera main\_p.s do main.o jako formę binarną (zera i jedynki).
  + Gcc -o main main.o
    - Uruchamia linker. Scala wszystkie kody plików \*.o, które należą do programu w jeden program wykonalny.

Więcej:

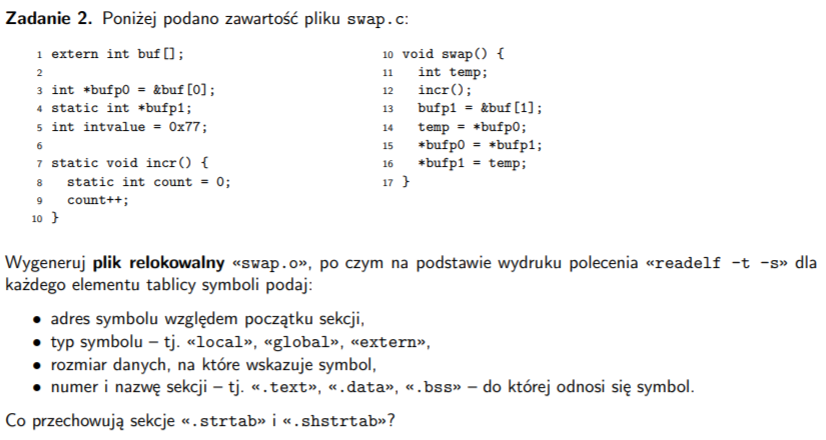
1. Preprocessing is the first step. The preprocessor obeys commands that begin with # (known as directives) by:
   1. removing comments
   2. expanding macros
   3. expanding included files

If you included a header file such as #include <stdio.h>, it will look for the stdio.h file and copy the header file into the source code file.

The preprocessor also generates macro code and replaces symbolic constants defined using #define with their values.

1. Compiling is the second step. It takes the output of the preprocessor and generates assembly language, an intermediate human readable language, specific to the target processor.
2. Assembly is the third step of compilation. The assembler will convert the assembly code into pure binary code or machine code (zeros and ones). This code is also known as object code.
3. Linking is the final step of compilation. The linker merges all the object code from multiple modules into a single one. If we are using a function from libraries, linker will link our code with that library function code.

* Kod pochodzi z biblioteki stdio.h. Preprocesor po prostu wkleił wszelkie potrzebne definicje i składowe, których używa nasz program do pliku main\_p.c. Dzięki temu program może z nich korzystać (np. z funkcji printf).
* Main\_p.s zawiera w sobie kod assemblera (kod pośredni, zrozumiały dla człowieka), który potem będzie tłumaczony na kod zer i jedynek. Typ etykietom możemy przyporządkować poprzez ich zapis. Etykiety typu: .L to etykiety lokalne. Przykładowo FB oraz FE to odpowiedno „function Begin” oraz „function end”. Etykiety są wstawiane wszędzie tam, gdzie potrzebujemy odwołania do pamięci.
* Global oraz set\_global mają według tablicy położenie na adresie 0. Dopiero potem linker będzie wyliczał odpowiednie adresy relatywnie do reszty kodu. W odniesieniu do naszego programu global jest pierwszym symbolem w sekcji .data, więc ma adres 0 (oraz rozmiar 4, bo to int). Gdyby był kolejny symbol typu int, byłby na adresie 4 względem początku sekcji .data. Tak samo set\_global. Jest na adresie 0 względem początku sekcji .text, ma rozmiar 13 bajtów. Dlatego też funkcja main jest położona na 13 bajcie (jest zdefiniowana po set\_global).
* Ponieważ dopiero później będziemy wyliczać jakie są odpowiednie położenia (przy linkowaniu plików \*.o). Adresy w pliku wykonywalnym:
  + .data: 0x4020
  + .text: 0x1050



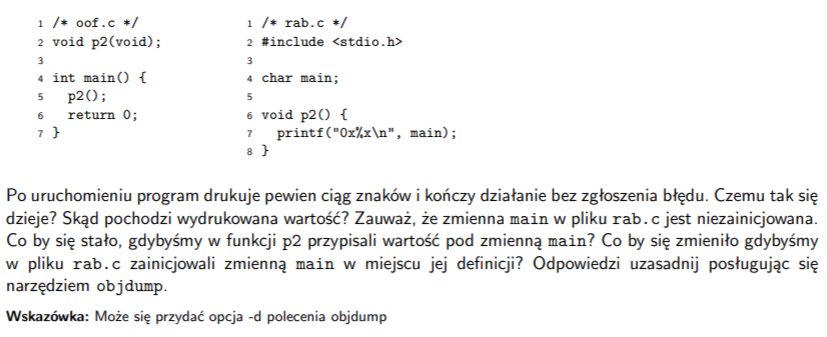
* Wypisuję tylko symbole, które występują w pliku źródłowym:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Nazwa | Adres wzgl. Pocz.sekcji | Typ | Rozmiar danych, na które wskazuje symbol | Numer sekcji | Nazwa sekcji |  |  |  |  |
| buf | 0x0 | NOTYPE | 0 | UND | ----------- |  |  |  |  |
| Bufp0 | 0x0 | OBJECT | 8 | 5 | .data.rel |  |  |  |  |
| Bufp1 | 0x0 | OBJECT | 8 | 4 | .bss |  |  |  |  |
| intvalue | 0x0 | OBJECT | 4 | 3 | .data |  |  |  |  |
| Incr() | 0x0 | FUNC | 22 | 1 | .text |  |  |  |  |
| Count.1961 | 0x8 | OBJECT | 4 | 4 | .bss |  |  |  |  |
| Swap() | 0x16 | FUNC | 77 | 1 | .text |  |  |  |  |

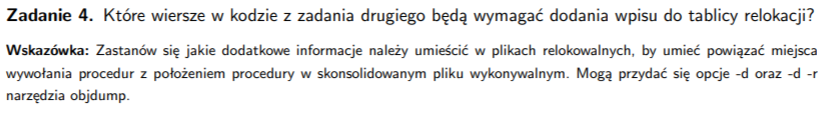
.strtab – Zawiera ciągi znakowe z programu oraz nazwy związane z tablicą symboli.

.shstrtab - Zawiera nazwy sekcji.





Program drukuje pierwszy bajt funkcji main z oof.c. Linker po prostu dowiązuje do symbolu char main z pliku rab.c funkcję main z oof. Dzieje się tak dlatego, że char main jest zmienną niezainicjowaną, więc jest to słaby symbol. Funkcja main to silny symbol, więc linker wybiera ten silny i jego używa przy wiązaniu. Gdybyśmy zainicjowali zmienną main w rab.c, dostalibyśmy błąd, ponieważ nie mogą występować dwa silne symbole o tej samej nazwie. Objdump -d pokazuje, że pierwszym bajtem funkcji main jest 0x48, taka też wartość zostaje wypisana.

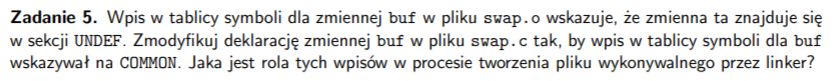


Linker przy próbie utworzenia pliku wykonywalnego wyrzuca dwa błędy:

* Undefined reference to „main”
* Undefined reference to „buf”

Pierwszy błąd możemy rozwiązać dodając po prostu funkcję main od kodu (żeby program wiedział skąd startować). Drugi błąd rozwiązujemy poprzez dodanie deklaracji buf np. w innym pliku.

Objdump -d -r wypisuje w wyniku wyraźnie linijki dotyczące buf.

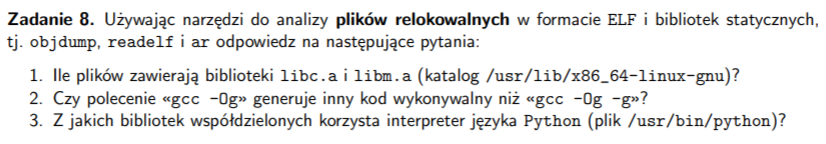


Wystarczy np. usunąć słowo kluczowe „extern”. Wtedy będzie to zmienna globalna, niezainicjowana.

UNDEF-Linker musi poszukać definicji zmiennej w innych plikach, które są częścią programu. W danym pliku \*.o nie ma zdefiniowanej wartości tej zmiennej, oznaczone jest, żeby szukać jej w jakimś innym pliku.

COMMON-Niezainicjowane globalne zmienne (nie statyczne) idą do tej sekcji. Mogą to być np. zmienne globalne, które ma znaleźć linker szukając symboli dla „extern”.

Dzięki odpowiedniemu oznaczaniu linker wie jak dalej postąpić z danym symbolem. Czy ma szukać definicji gdzieś indziej, czy definicja powinna być w danym pliku. Także numery sekcji oznaczają czym jest dany symbol. Zmienne są przechowywane w innych miejscach niż funkcje.



1. ar -t libc.a | wc -l -> 1718

ar -t libm.a | wc -l -> (???) – wyrzuca file format not recognized

1. Opcja -g równa się wpisaniu -g2 a ominiecie -g to to samo co wpisanie -g0.

Higher optimization levels both result in an increase of the code and debugging sections. (-O0 < -O1 < -Og < -O2 < -O3).

(Uzywajac zadania 1)

Gcc -Og main.c -o program1

Gcc -Og -g main.c -o program2

Objdump –syms program1 | grep debug -> 0

Objdump –syms program2 | grep debug -> 5

Czyli po prostu dodane są linijki do łatwiejszego debugowania.

1. Readelf –dyn-syms /usr/bin/python3.7 : 2424 wpisy symboli

Ldd: : 10 bibliotek