



PROGRAMOWANIE NISKOPOZIOMOWE

KONSPEKT LABORATORYJNY

---

# Binutils, biblioteki statyczne i dynamiczne

---

*Autorzy:*

Gabriel Górski

Robert Gałat

29 kwietnia 2018

## Spis treści

<b>1</b>	<b>Informacje do zadań</b>	<b>2</b>
1.1	Biblioteki statyczne . . . . .	2
1.2	Biblioteki współdzielone . . . . .	2
1.3	Zmienne środowiskowe dla LD . . . . .	2
1.4	Binutils . . . . .	3
1.5	Dynamiczne ładowanie . . . . .	3
<b>2</b>	<b>Zadania</b>	<b>5</b>
2.1	Biblioteki statyczne . . . . .	5
2.2	Biblioteki współdzielone . . . . .	5
2.3	Binutils . . . . .	5
2.4	Pluginy i dynamiczne ładowanie . . . . .	5

## 1 Informacje do zadań

W tym miejscu zakładamy, że ogólnie rozumiana teoria z seminarium jest znana uczestnikom laboratorium. Poniżej znajdują się rzeczy przydatne w wykonywaniu zadań z laboratorium.

### 1.1 Biblioteki statyczne

Jeśli chcemy utworzyć bibliotekę statyczną, potrzebujemy pliki obiektowe które mają się w niej znaleźć. Komenda wygląda następująco

```
ar rcs libNAZWA.a plik1.o plik2.o plik3.o
```

Żeby wykorzystać tą bibliotekę należy ją **zlinkować** z resztą plików obiektowych (na tym etapie symbol **main** *musi* się zawierać w którymś z nich).

Wykorzystanie biblioteki: Flaga **-L** umożliwia dodanie ścieżek przeszukiwań bibliotek dla linkera statycznego. Dla nagłówków istnieje analogiczna flaga: **-I** — przydatna na etapie kompilacji.

```
gcc -L . plik.o -o plik -lNAZWA
```

### 1.2 Biblioteki współdzielone

Podobnie ma się sprawa z bibliotekami dynamicznymi, tutaj jednak należy pamiętać o większej ilości niuansów.

Po pierwsze składowe pliki obiektowe biblioteki należy skompilować z flagą **fPIC**.

Tworzenie biblioteki:

```
gcc -shared plik1.o plik2.o plik3.o -o libNAZWA.so
```

Aby wykorzystać bibliotekę w czasie linkowania należy zrobić dokładnie to samo

```
gcc -L . plik.o -o plik -lNAZWA
```

### 1.3 Zmienne środowiskowe dla LD

Uruchomienie programu zbudowanego z wykorzystaniem bibliotek współdzielonych wymaga ich obecności w czasie działania. Jeśli owa biblioteka znajduje się poza

domyślnymi katalogami przeszukiwań, wtedy musimy wykorzystać zmienne środowiskowe które wpłyną na działanie LD. Aby dodać ścieżki przeszukiwań możemy zrobić:

```
LD_LIBRARY_PATH=. ./plik
```

Dodatkowo, mając plik wykonywalny korzystający z bibliotek współdzielonych, możemy zmusić linker dynamiczny by — zanim zrobi wszystko wg reguł ustalonych przy linkowaniu statycznym — **z pierwszeństwem** załadował inne symbole.

Przykład:

*Plik wykonywalny 'plik' korzysta z biblioteki współdzielonej dostarczającej symbol 'abc'; biblioteka SAMPLE dostarcza symbol 'abc' o takiej samej sygnaturze, ale o innej implementacji*

```
LD_PRELOAD=./libSAMPLE.so ./plik
```

W takiej sytuacji, wykorzystany zostanie symbol *abc* z biblioteki SAMPLE bo owa biblioteka została *preload-owana* przed pozostałymi.

## 1.4 Binutils

Pliki binarne, a w szczególności obiektowe możemy analizować różnymi narzędziami. Poniżej wymienione są wybrane tego typu programy

- **nm** — listowanie symboli z pliku obiektowego
- **objdump** — w zależności od wybranych flag wyświetla w czytelny sposób określone informacje o fragmentach pliku obiektowego
- **readelf** — kompleksowe narzędzie do analizy struktury i zawartości pliku obiektowego

## 1.5 Dynamiczne ładowanie

Istnieje potężny i jeszcze bardziej elastyczny mechanizm umożliwiający ładowanie bibliotek współdzielonych. Zamiast dodawać do pliku wykonywalnego informację o bibliotekach na etapie linkowania statycznego lub wykorzystywać zmienne środowiskowe, sam program może posiadać odpowiednie funkcjonalności do manipulacji bibliotekami wedle własnych upodobań.

W systemach opierających się o jądro Linux takie możliwości dostarcza biblioteka **dl** i komplementarny do niego nagłówek **dlfcn.h**.

Do wykonania zadań będziemy korzystać z następujących funkcji:

- `void *dlopen (const char *__file, int __mode)`

Umożliwia otworenie biblioteki współdzielonej.

- `__file` określa nazwę biblioteki współdzielonej którą chcemy wczytać
- `__mode` określa zasadę doczytywania symboli (natychmiastową lub *leniwo* tj. w razie potrzeby). Tutaj jako argument będziemy przekazywać **RTLD\_NOW**.

Wartość zwracana to jest *handle* tj. referencja do biblioteki by móc się do niej odwoływać

- `int dlclose (void *__handle)`

Umożliwia zamknięcie biblioteki współdzielonej. Jako argument przyjmuje *handle* do otwartej biblioteki którą chcemy zamknąć. Zwraca kod błędu.

- `void *dlsym (void *__restrict __handle, const char *__restrict __name)`

Umożliwia pobranie symbolu z biblioteki

- `__handle` określa *handle* do biblioteki z której będziemy pobierać symbol. Dostępne są także *pseudohandle's* jako makra określające specjalne zachowanie przy zawołaniu *dlsym*. Np **RTLD\_NEXT** spowoduje pobranie wybranego symbolu z **następnej** biblioteki w kolejności wczytywania przez linker dynamiczny.
- `__name` to nazwa pożądanego symbolu.

Wartość zwracana jest wskaźnikiem nieokreślonym na symbol. Jeśli ma wartość **NULL**, oznacza to błąd (brak symbolu etc).

Jeśli symbolem jest funkcja, taki wskaźnik należy **zrzutować** na typ wskaźnikowy na funkcję o **odpowiedniej sygnaturze**

## 2 Zadania

### 2.1 Biblioteki statyczne

- Celem zadania jest uzupełnienie pliku **run.sh** tak, aby umożliwiał on kompilację biblioteki statycznej, oraz zlinkowanie projektu do programu wynikowego. [1a]
- Celem zadania jest uzupełnienie pliku **run.sh** tak, aby stworzyć biblioteki statyczne oraz zlinkować je z *głównym* plikiem obiekowym. Czy zauważasz coś ciekawego? Jeśli tak, to czy potrafisz to wyjaśnić? [1b]

### 2.2 Biblioteki współdzielone

- W tym zadaniu należy utworzyć bibliotekę dynamiczną, zlinkować wobec niej plik obiekowy, a następnie otrzymany plik wykonywalny należy uruchomić — pamiętaj o odpowiednich flagach kompilacji i linkowania! [2]
- Celem zadania jest podmienienie implementacji funkcji która była w bibliotece z poprzedniego zadania.

Należy to zrobić bez modyfikacji pliku wykonywalnego z poprzedniego zadania tj. poprzez wykorzystanie funkcjonalności linkera dynamicznego.

Wprowadź własną implementację tej funkcji. [3]

### 2.3 Binutils

- W tym zadaniu należy dokonać kompilacji pliku *relocatableFile.c* a następnie przeanalizować wygenerowany plik binarny programem *nm* oraz *objdump* [4]

### 2.4 Pluginy i dynamiczne ładowanie

- Celem zadania jest uzupełnienie pliku *main.c* w taki sposób aby uruchomić funkcję z biblioteki *libgoo.so*, która powinna zostać załadowana w czasie działania programu. [5]
- Celem zadania jest uzupełnienie brakujących części obsługi pluginów, oraz napisanie własnego pluginu, wzorując się na przygotowanym przykładzie

Do uzupełnienia są następujące funkcje:[6]

- `apply_hook()` {PluginManager/PluginManager.c}
- `initPlugi()` {PluginManager/PluginLoader.c}