



Politechnika Wrocławska

Wydział Elektroniki,
Fotoniki i Mikrosystemów

Laboratorium informatyki

Ćwiczenie nr 13. Implementacja i konsolidacja bibliotek programistycznych

Zagadnienia do opracowania:

- biblioteka standardowa języków *C* i *C++*
- biblioteki statyczne i dynamiczne
- biblioteka a framework
- interfejs programistyczny (API)
- położenie funkcji bibliotecznych w wirtualnej przestrzeni adresowej

Spis treści

1	Cel ćwiczenia	2
2	Wprowadzenie	2
2.1	Biblioteki programistyczne	2
2.2	Biblioteka standardowa	12
2.3	Funkcje biblioteczne a pamięć programu	12
3	Program ćwiczenia	18
4	Dodatek	20
4.1	Biblioteki dynamiczne jako wtyczki	20

1. Cel ćwiczenia

Celem ćwiczenia jest opanowanie umiejętności implementacji oraz konsolidacji własnych bibliotek programistycznych w językach *C/C++*.

2. Wprowadzenie

2.1. Biblioteki programistyczne

Biblioteki programistyczne (*ang. libraries*) stanowią zbiór zdefiniowanych *funkcji*, *stałych* i *typów danych*, wykorzystywanych w kodzie źródłowym aplikacji. Skompilowany kod **biblioteki** nie stanowi samodzielnego pliku wykonywalnego – nie posiada zdefiniowanej funkcji *main()*. **Biblioteki programistyczne** pozwalają na użycie tego samego kodu przez wiele różnych aplikacji. Przykładem może być **Standardowa Biblioteka Sza-blonów** języka *C++*, dostarczająca sprawdzone implementacje *kontenerów* (struktur danych) czy popularnych *algorytmów*. **Biblioteki** można podzielić na:

- **statyczne** (*ang. static libraries*) – dołączane do aplikacji na etapie **konsolidacji** kodu, w wyniku czego kod **bibliotek statycznych** jest zawierany w wynikowym kodzie wykonywalnym programu. Skutkuje to zwiększeniem rozmiaru aplikacji. W systemie operacyjnym Windows **biblioteki statyczne** mają najczęściej rozszerzenie **.lib** albo **.obj**, natomiast w systemach z rodziny Unix rozszerzenie **.a** (z *ang. archive*);
- **dynamiczne** (*współdzielone*) (*ang. dynamic libraries, shared libraries*) – dołączane do aplikacji na etapie jej uruchomienia. W systemie operacyjnym Windows **biblioteki dynamiczne** mają rozszerzenie **.dll** (z *ang. dynamic-link library*), natomiast w systemach z rodziny Unix rozszerzenie **.so** (z *ang. shared object*). Pojedyncza instancja **biblioteki dynamicznej** może być współdzielona przez wiele aplikacji jed-

nocześnie (kod *biblioteki* nie jest zawierany w kodzie wykonywalnym aplikacji).

Zbliżonym pojęciem do *biblioteki programistycznej* jest *framework*, znany również pod pojęciem *platformy programistycznej*. Różnica między *biblioteką* a *frameworkiem* polega na tym, że *framework* wykorzystuje wzorzec *odwróconego sterowania* (ang. *inversion of control, IoC*). W klasycznym podejściu program wywołuje określone funkcje biblioteczne. W przypadku *odwróconego sterowania* to *framework* przekazuje sterowanie do kodu aplikacji i decyduje o wykonaniu kodu użytkownika.

Aby zaimplementować *bibliotekę programistyczną* i wykorzystać ją w kodzie aplikacji należy:

1. zadeklarować *interfejs* biblioteki
2. zaimplementować zadeklarowane funkcje biblioteczne
3. skompilować kod do postaci *biblioteki statycznej* bądź *biblioteki dynamicznej*
4. załączyć *interfejs* biblioteki w kodzie aplikacji
5. dołączyć *bibliotekę* do aplikacji na etapie konsolidacji kodu lub wykonania programu

Interfejs programistyczny (ang. *application programming interface, API*) określa w jaki sposób zachodzi komunikacja między programami lub, jak w tym przypadku, między programem a *biblioteką*. Wprowadzenie *interfejsu programistycznego* umożliwia rozdzielenie deklaracji funkcji bibliotecznych od ich implementacji. Przykładem może być *biblioteka standardowa*, która posiada wiele niezależnych implementacji, ale jeden, ściśle zdefiniowany *interfejs*, wykorzystywany przez aplikacje języka *C* (lub *C++*). Dzięki temu możliwe jest tworzenie *bibliotek* dla różnych systemów operacyjnych, architektur procesorów czy języków programowania, jak również zmiana zasady działania aplikacji przez wymianę implementacji *biblioteki* z zachowaniem tego samego *interfejsu programistycznego* (nie ma

konieczności modyfikacji kodu źródłowego aplikacji). *Interfejs* powinien cechować się małym stopniem skomplikowania i być dobrze udokumentowany. Dobrze przygotowany *interfejs* jest wystarczający dla programisty, aby skorzystać z funkcji udostępnionych przez *bibliotekę*, bez oglądania ich implementacji (która często jest ukryta). Jako przykład może posłużyć prosta *biblioteka libmath* implementująca podstawowe operacje matematyczne. Jej *interfejs*, zawarty w pliku nagłówkowym *math_functions.h*, przedstawiono na listingu 1.

```
1 #pragma once
2
3 /**
4  * adding two double-precision floating-point
5  * numbers
6  * @param x first addent
7  * @param y second addent
8  * @return sum of addents x and y
9  */
10 double add(double x, double y);
11
12 /**
13  * subtracting two double-precision floating-point
14  * numbers
15  * @param x minuend
16  * @param y subtrahend
17  * @return difference of minuend and subtrahend
18  */
19 double subtract(double x, double y);
```

```
20 * multiplying two double-precision floating-point
    numbers
21 * @param x first factor
22 * @param y second factor
23 * @return multiplication of factors x and y
24 */
25 double multiply(double x, double y);
26
27 // Function pointer type of the illegal division
    operation handler
28 typedef double (*IllegalOperationHandler)();
29
30 /**
31 * dividing two double-precision floating-point
    numbers
32 * @param x dividend
33 * @param y divisor
34 * @param illegalOperationHandler handler to be
    called on division by zero
35 * @return value returned by the
    illegalOperationHandler invocation in case of
    division by zero or quotient of dividend and
    divisor otherwise
36 * The function invocation result is undefined if
    illegalOperationHandler is NULL
37 */
38 double divide(double x, double y,
    IllegalOperationHandler illegalOperationHandler);
39
40 /**
```

```

41 * modulus of a double-precision floating-point
    number
42 * @param x number for which the modulus is to be
    calculated
43 * @return modulus of x
44 */
45 double modulus(double x);

```

Listing 1. Interfejs biblioteki *libmath*

Implementacje funkcji bibliotecznych *libmath*, zawarte w pliku źródłowym *math_functions.c*, przedstawiono na listingu 2. Należy mieć na uwadze, że użytkownik *biblioteki libmath* może nie mieć dostępu do nieskompilowanych plików źródłowych. Plik nagłówkowy *interfejsu* (listing 1) powinien zawierać objaśnienia poszczególnych funkcji, a w razie bardziej skomplikowanych implementacji, również przykłady ich użycia w postaci komentarzy w kodzie *interfejsu*.

```

1 #include "math_functions.h"
2
3 double add(double x, double y) {
4     return x + y;
5 }
6
7 double subtract(double x, double y) {
8     return x - y;
9 }
10
11 double multiply(double x, double y) {
12     return x * y;
13 }
14

```

```
15 double divide(double x, double y,  
    IllegalOperationHandler illegalOperationHandler)  
    {  
16     if (y == 0)  
17         return illegalOperationHandler();  
18     else  
19         return x / y;  
20 }  
21  
22 double modulus(double x) {  
23     return x < 0 ? -x : x;  
24 }
```

Listing 2. Implementacja funkcji bibliotecznych *libmath*

Kompilacja plików *math_functions.h* oraz *math_functions.c* do postaci *biblioteki statycznej* (przy użyciu pakietu *GNU*) w systemie Windows i Unix odbywa się jednakowo. W pierwszym kroku należy skompilować pliki źródłowe do postaci pliku obiektowego (tu: *math_functions.o*). Można to zrobić wykorzystując flagę *-c* kompilatora *gcc* (kompilacja bez konsolidacji):

gcc -c math_functions.c

Jeżeli *biblioteka* zawiera więcej niż jeden plik źródłowy, to powyższy krok należy powtórzyć dla każdego z nich. W kolejnym etapie łączy się poszczególne pliki obiektowe do jednego pliku *biblioteki statycznej* (archiwum). W tym celu można posłużyć się programem *ar* (pakiet *GNU*). Składnia wywołania programu jest następująca

ar [flagi] [nazwa_biblioteki] [pliki_obiektowe]

Dla *biblioteki libmath*:

ar -cvru libmath.lib math_functions.o

[Uwaga: w przypadku systemu *Unix libmath.a*]

Zastosowane w przykładzie flagi określają odpowiednio:

- **c** – nie ostrzegaj, jeżeli biblioteka musiała zostać utworzona;
- **v** – zwiększ szczegółowość komunikatów;
- **r** – zamień lub dodaj nowe pliki do archiwum;
- **u** – zamień wyłącznie te pliki, które są nowsze niż obecna wersja archiwum.

Zawartość archiwum można wyświetlić za pomocą flagi **-t**:

ar -t libmath.lib

W przypadku *biblioteki dynamicznej*, w pierwszym kroku kompilacji należy dodatkowo uwzględnić flagę **-fPIC** (*position-independent code*), która określa, że działanie wygenerowanego kod maszynowego funkcji bibliotecznych nie jest zależne od konkretnego położenia w wirtualnej przestrzeni adresowej (cecha charakterystyczna *bibliotek dynamicznych*):

gcc -fPIC -c math_functions.c

Łączenie plików obiektowych można przeprowadzić przy pomocy kompilatora **gcc**, wykorzystując flagę **-shared**:

gcc -shared -o libmath.dll math_functions.o

[Uwaga: w przypadku systemu *Unix libmath.so*]

Aby skorzystać z funkcji *biblioteki libmath* w kodzie aplikacji, należy dołączyć nagłówek (*interfejs*) **math_functions.h** za pomocą dyrektywy **#include** (np. w pliku *main.c*). Przykład przedstawiono na listingu 3.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // Załącz API biblioteki libmath
5 #include <math_functions.h>
6
7 // IllegalOperationHandler
8 double onDivisionByZero() {
9     printf("Illegal operation\n");
10    return 0.0;
11 }
12
13 int main() {
14     double dividend, divisor;
15     printf("Put dividend: ");
16     scanf("%lf", &dividend);
17     printf("Put divisor: ");
18     scanf("%lf", &divisor);
19     // Użycie funkcji divide() z biblioteki libmath
20     printf("Quotient equals: %lf", divide(dividend,
21     divisor, onDivisionByZero));
22     return EXIT_SUCCESS;
23 }
```

Listing 3. Użycie funkcji bibliotecznych *libmath* w kodzie aplikacji

Próba kompilacji programu z listingu 3. zakończy się następującym błędem:

```
gcc -o app main.c
main.c:(.text+0x88): undefined reference to 'divide'
main.c:(.text+0x88): relocation truncated to fit: R_X86_64_PC32
against undefined symbol 'divide'
collect2: error: ld returned 1 exit status
```

Pomimo, że kompilator poprawnie dołączył nagłówek *math_functions.h*, to próba odnalezienia definicji funkcji *divide()*, wywołanej w funkcji *main()*, zakończyła się błędem (**undefined symbol 'divide'**). Nazwa *ld* wskazuje na błąd konsolidatora.

Poprawna kompilacja aplikacji może wyglądać następująco:

- jeżeli plik *biblioteki* położony jest w tym samym katalogu, co pliki źródłowe aplikacji:

```
gcc -o app main.c libmath.lib
gcc -o app main.c libmath.dll
```

- jeżeli plik *biblioteki* położony jest w innym katalogu niż pliki źródłowe aplikacji, należy skorzystać z flagi **-L**:

```
gcc -o app main.c libmath.lib -L[ścieżka_do_biblioteki]
gcc -o app main.c libmath.dll -L[ścieżka_do_biblioteki]
```

Można zauważyć, że rozmiar pliku wykonywalnego (*app.exe*) jest większy w przypadku dołączania *biblioteki statycznej* niż *biblioteki dynamicznej* – funkcje *biblioteki dynamicznej* zostaną dołączone na etapie uruchomienia aplikacji.

Stosowanie *bibliotek statycznych* i *dynamicznych* ma swoje wady i zalety:

-
- **biblioteki dynamiczne** są dołączane do aplikacji na etapie jej uruchomienia, dzięki czemu kod **biblioteki** nie jest zawierany w kodzie wynikowym programu, zmniejszając rozmiar pliku wykonywalnego. Pojedyncza instancja **biblioteki dynamicznej** może być współdzielona przez wiele aplikacji jednocześnie. Wykorzystanie **bibliotek dynamicznych** wiąże się z dodatkowym narzutem czasowym wynikającym z wczytania i wykonania funkcji bibliotecznych
 - **biblioteki statyczne** są zawierane w pliku wykonywalnym, zwiększając jego rozmiar. Dzięki temu, że **biblioteka statyczna** dołączana jest na etapie konsolidacji, może zachodzić optymalizacja rozmiaru pliku wykonywalnego przez dołączanie wyłącznie definicji funkcji bibliotecznych użytych w kodzie programu (a nie wszystkich dostępnych funkcji). **Biblioteki statyczne** nie posiadają dodatkowego narzutu czasowego wynikającego z wykonywania funkcji bibliotecznych

Warto również wspomnieć, że pliki **.dll** (*Windows*) i **.so** (*Unix*), choć koncepcyjnie podobne, są obsługiwane nieco inaczej. Mechanizm obsługi **bibliotek dynamicznych** w systemie Windows jest istotnie ograniczony w porównaniu z systemami Unix pod kątem obsługi offsetów symboli **biblioteki dynamicznej** (problem z odczytywaniem adresów funkcji i zmiennych bibliotecznych). Do poprawnego działania **biblioteki** wymagane jest dekladowanie symboli z użyciem atrybutu `__declspec(dllexport)`, np.:

```
1 __declspec(dllexport) void func();
```

Niektóre środowiska, jak **MinGW**, poza kompilatorami języków *C* i *C++* dostarczają również oprogramowanie stanowiące warstwę pośrednią między systemami Windows i Unix. Umożliwia ona automatyczne „tłumaczenie” kodu **bibliotek dynamicznych .so** na kod wspierany przez system operacyjny Windows (**.dll**). Dzięki temu kod **biblioteki** nie musi być jawnie dekladowany przy użyciu atrybutów `__declspec(dllexport)`, jeżeli posługujemy się kompilatorami **gcc** lub **g++**.

2.2. Biblioteka standardowa

Zarówno język *C*, jak i *C++*, posiadają swoje ***biblioteki standardowe***, które stanowią integralną część **standardu ISO** (*ang. International Organization for Standardization*). **API biblioteki standardowej** zawiera się w szeregu plików nagłówkowych. Pliki nagłówkowe ***biblioteki standardowej języka C*** posiadają rozszerzenie *.h*, np. *stdio.h*, *stdlib.h*, *math.h*. Konwencja nazewnictwa nagłówków ***biblioteki standardowej języka C++*** odrzuca jakiekolwiek rozszerzenia plików, np. *iostream*, *memory*, *string*, a w przypadku nagłówków zaadaptowanych z ***biblioteki standardowej języka C*** stosowany jest przedrostek *c*, np. *cmath*, *ctime*. **Biblioteka standardowa** posiada jeden ściśle zdefiniowany ***interfejs***, ale wiele implementacji (*GNU libc*, *BSD libc*, itp.) i może być zarówno ***biblioteką statyczną***, jak i ***dynamiczną***.

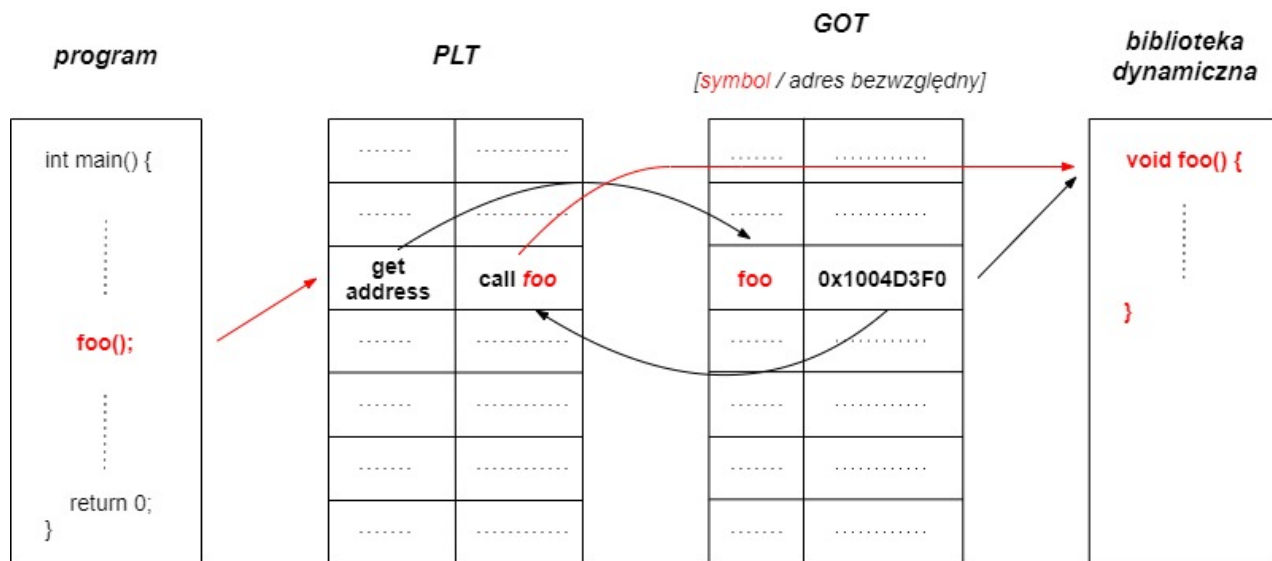
Celem istnienia ***biblioteki standardowej*** jest zapewnienie programistom sprawdzonych i zoptymalizowanych rozwiązań podstawowych algorytmów, struktur danych czy funkcji operujących na pamięci. **Biblioteka standardowa** jest automatycznie dołączana do programu podczas jego kompilacji. Aby przeprowadzić kompilację programu bez konsolidacji ***biblioteki standardowej*** należy (w przypadku kompilatorów *gcc* i *g++*) skorzystać z flagi ***-nostdlib***:

```
gcc main.c -nostdlib -o app
```

2.3. Funkcje biblioteczne a pamięć programu

Kod ***bibliotek statycznych*** zawierany jest w wynikowym kodzie wykonywalnym aplikacji. W związku z tym, kod funkcji bibliotecznych położony jest razem z kodem funkcji programu w segmencie ***text*** (*code*). W przypadku ***bibliotek dynamicznych*** sprawa jest nieco bardziej skomplikowana. Kompilacja z użyciem flagi ***-fPIC*** wyklucza odwoływanie się do bezwzględnych adresów funkcji bibliotecznych. Problem ten rozwiązywany jest przez zastosowanie tzw. ***globalnej tablicy offsetów*** (*ang. global offset table, GOT*),

w której mapowane są *symbole* funkcji bibliotecznych na ich adresy bezwzględne. Adresy te nie są znane na etapie kompilacji programu, dlatego w momencie jego uruchomienia wywoływany jest **konsolidator dynamiczny** systemu operacyjnego (*ang. dynamic linker*), którego zadaniem jest wypełnienie komórek tablicy **GOT**. **Konsolidator dynamiczny** jest w stanie obliczyć adresy bezwzględne funkcji bibliotecznych, ponieważ zna adresy wszystkich zaalokowanych segmentów pamięci. Adresy funkcji **bibliotek dynamicznych** mogą zostać umieszczone w różnych segmentach pamięci, ale są niezmiennie w ramach pojedynczego wykonania programu. Odwołanie się do funkcji bibliotecznej zachodzi z wykorzystaniem **tablicy powiązań** (*ang. procedure linkage table, PLT*). W momencie wywołania funkcji sterowanie przekazywane jest do odpowiedniej komórki tablicy **PLT**. Tam zachodzi odczytanie bezwzględnego adresu funkcji z tablicy **GOT** i wykonanie zawartych pod nim instrukcji [2]. Na rys. 2.1. zilustrowano schematyczną zasadę działania mechanizmu tablic **PLT/GOT**.



Rys. 2.1. Schemat działania mechanizmu tablic **PLT/GOT**

W celu szczegółowej analizy pamięci programu należy w pierwszej kolejności wygenerować plik *.map*. Można to osiągnąć przekazując do konsolidatora flagę *-Map*. Przykładowe wywołanie:

```
gcc -o app main.c libmath.lib -Xlinker -Map=output.map
```

poskutkuje wygenerowaniem pliku *output.map* zawierającego układ pamięci programu. Opcja *-Xlinker* określa, że następne argumenty wywołania mają zostać przekazane do konsolidatora (*linkera*). Wygenerowany plik zawiera sekcję *Memory Configuration*, w której znajdują się symbole programu w formacie:

Name	Origin	Length	Attributes,
------	--------	--------	-------------

gdzie:

- **Name** – *Nazwa segmentu pamięci;*
- **Origin** – *Adres początkowy;*
- **Length** – *Rozmiar;*
- **Attributes** – *Atrybuty.*

W przypadku konsolidacji *statycznej biblioteki libmath* przez aplikację z listingu 3., plik *output.map* będzie zawierał informacje o położeniu funkcji bibliotecznych w segmencie pamięci *.text*:

1	.text	0x0000000100401140	0xe0	libmath.lib(math_functions.o)
2		0x0000000100401140		add
3		0x000000010040115a		subtract
4		0x0000000100401174		multiply
5		0x000000010040118e		divide
6		0x00000001004011e5		modulus

Dołączony kod *biblioteki* zwiększa rozmiar aplikacji o 224 B (0xe0). Wywołana w funkcji *main()* funkcja biblioteczna *divide()* leży pod adresem 0x000000010040118e i zajmuje maksymalnie 87 B (0x57; różnica adresów funkcji *divide()* i *modulus()*). W przypadku konsolidacji *biblioteki dynamicznej* plik *output.map* zawiera tylko jedną funkcję *biblioteki libmath* – *divide()*. Tylko ta funkcja jest wykorzystywana przez aplikację:

1	.text	0x0000000100401790	0x8 d000001.o
2		0x0000000100401790	divide

Z analizy pliku *output.map* wynika, że również w tym przypadku leży ona w segmencie *.text*, ale pod adresem 0x0000000100401790 i zajmuje 8 B. Ta „anomalia” wymaga komentarza. Jeżeli implementacja funkcji w obu przypadkach była taka sama, to skąd wzięła się różnica w rozmiarze kodu funkcji w pamięci komputera? Aby to sprawdzić należy posłużyć się programem do analizy plików obiektowych, np. *GNU objdump*. Wywołanie:

objdump -CS app.exe >"C:\User\dump.txt"

przeprowadzi *deasemblację*¹ programu *app.exe* i zapisze rezultat do pliku tekstowego *dump.txt*, leżącego pod ścieżką C:\User\. Zastosowane w przykładzie flagi określają odpowiednio:

- *C* – przeprowadź dekodowanie udekorowanych symboli (patrz: Ćw. 5, Dekorowanie nazw);
- *S* – połącz kod źródłowy z kodem asemblera (dla łatwiejszego nawigowania w utworzonym pliku).

W przypadku *biblioteki statycznej* plik *dump.txt* zawiera kod asemblera funkcji *divide()*:

¹tłumaczenie kodu maszynowego na kod asemblera


```

1 0000000010040118e <divide>:
2     10040118e: 55                push    %rbp
3     10040118f: 48 89 e5          mov     %rsp,%rbp
4     100401192: 48 83 ec 30       sub     $0x30,%rsp
5     100401196: f2 0f 11 45 10    movsd   %xmm0,0x10(%
    rbp)
6     10040119b: f2 0f 11 4d 18    movsd   %xmm1,0x18(%
    rbp)
7     1004011a0: 4c 89 45 20       mov     %r8,0x20(%
    rbp)
8     1004011a4: 66 0f ef c0       pxor    %xmm0,%xmm0
9     1004011a8: 66 0f 2e 45 18    ucomisd 0x18(%rbp)
    ,%xmm0
10    1004011ad: 7a 18            jp      1004011c7 <
    divide+0x39>
11    1004011af: 66 0f ef c0       pxor    %xmm0,%xmm0
12    1004011b3: 66 0f 2e 45 18    ucomisd 0x18(%rbp)
    ,%xmm0
13    1004011b8: 75 0d            jne     1004011c7 <
    divide+0x39>
14    1004011ba: 48 8b 45 20       mov     0x20(%rbp),%
    rax
15    1004011be: ff d0            callq   *%rax
16    1004011c0: 66 48 0f 7e c0    movq    %xmm0,%rax
17    1004011c5: eb 0f            jmp     1004011d6 <
    divide+0x48>
18    1004011c7: f2 0f 10 45 10    movsd   0x10(%rbp),%
    xmm0
19    1004011cc: f2 0f 5e 45 18    divsd   0x18(%rbp),%
    xmm0
20    1004011d1: 66 48 0f 7e c0    movq    %xmm0,%rax

```

```

21 1004011d6: 48 89 45 f8      mov    %rax,-0x8(%
    rbp)
22 1004011da: f2 0f 10 45 f8    movsd  -0x8(%rbp),%
    xmm0
23 1004011df: 48 83 c4 30      add     $0x30,%rsp
24 1004011e3: 5d              pop     %rbp
25 1004011e4: c3             retq

```

Warto zwrócić uwagę, że adres funkcji *divide()* (0x000000010040118e) jest zgodny z adresem zawartym w pliku *output.map*. Widać teraz również, że rozmiar pamięci który zajmuje funkcja wynosi dokładnie 87 B. Dla *biblioteki dynamicznej* kod funkcji *divide()* zawarty w pliku *dump.txt* różni się znacząco:

```

1 00000000100401790 <divide>:
2   100401790: ff 25 ea 69 00 00    jmpq   *0x69ea(%
    rip)      # 100408180 <__imp_divide>
3   100401796: 90                  nop
4   100401797: 90                  nop
5   100401798: 0f 1f 84 00 00 00 00 nopl   0x0(%rax
    ,%rax,1)
6   10040179f: 00

```

Pierwsze polecenie asemblera, jakie wykonywane jest w ramach funkcji to *jmpq *0x69ea(%rip)*. Jest to instrukcja skoku do adresu zapisanego pośrednio jako suma stałej i wartości zapisanej pod zmienną *rip*: **0x69ea + %rip**. Pod tym adresem znajduje się symbol o identyfikatorze *__imp_divide*. Symbol ten nie ma więcej wystąpień w pliku *dump.txt*, można go jednak odszukać przenosząc się ponownie do pliku *output.map*:

```

1 .idata$5      0x00000000100408180      0x8 d000001.o
2               0x00000000100408180      __imp_divide

```

Segment **.idata** to segment pamięci zawierający listę symboli importowanych z **bibliotek dynamicznych** (*ang. imports data*), w tym importowanych funkcji (tu: **divide()**). Podczas wywołania funkcji **divide()** przez program z listingu 3., w miejsce symbolu **_imp_divide** podstawiany jest adres bezwzględny funkcji znajdującej się w instancji **biblioteki dynamicznej**, obliczony przez mechanizm **PLT/GOT**. Jest to właściwa implementacja funkcji **divide()**. Funkcja **divide()** o rozmiarze 8 B, znajdująca się w pamięci aplikacji pod adresem 0x0000000100401790, stanowi jedynie adapter, służący do wywołania kodu funkcji bibliotecznej.

3. Program ćwiczenia

Zadanie 1. Dany jest interfejs **math_utils.h** biblioteki dynamicznej **libmath**. Biblioteka udostępnia dwie funkcje:

- **int gcd(int x, int y)** – wyznaczającą największy wspólny dzielnik dwóch liczb całkowitych;
- **int lcm(int x, int y)** – wyznaczającą najmniejszą wspólną wielokrotność dwóch liczb całkowitych.

W ramach zadania:

- umieścić implementacje funkcji **gcd()** oraz **lcm()** w pliku **math_utils.c**, a następnie skompiluj otrzymany kod źródłowy do postaci **biblioteki dynamicznej**;
- napisz aplikację, która pobierze z klawiatury dwie liczby całkowite, a następnie, wykorzystując bibliotekę **libmath**, wyznaczy NWD i NWW tych liczb.

Zadanie 2. Dany jest interfejs (**sort.h**) oraz implementacja (**sort.cpp**) biblioteki **libsort**. Biblioteka udostępnia trzy funkcje realizujące odpowied-

nie algorytmy sortowania: ***bubbleSort()***, ***quickSort()*** oraz ***mergeSort()***.
W ramach zadania:

- skompiluj kod źródłowy biblioteki ***libsort*** do postaci **biblioteki statycznej**;
- napisz aplikację, która korzystając z biblioteki ***libsort***, posortuje rosnąco trzy tysiącelementowe tablice liczb zmiennoprzecinkowych podwójnej precyzji. Każdą z tablic posortuj za pomocą innej funkcji z biblioteki ***libsort***. Tablice zainicjalizuj wykorzystując generator liczb pseudolosowych (patrz: Ćw. 3);
- zbadaj położenie adresów funkcji ***biblioteki statycznej libsort*** wirtualnej przestrzeni adresowej (położenie względem segmentów pamięci).

Zadanie 3. Skompiluj plik ***sort.cpp*** z **Zadania 2.** do postaci **biblioteki dynamicznej**, a następnie:

- porównaj rozmiar plików wykonywalnych obu aplikacji;
- porównaj czas wykonania funkcji sortujących dla obu aplikacji (patrz: Ćw. 5) [*Uwaga: aby wielokrotnie wylosować ten sam zbiór liczb, zainicjalizuj PRNG za pomocą stałej wartości, np. `srand(0)`*], w tym celu rozpatrz następujące warunki:
 - rozmiar sortowanego zbioru liczb równy odpowiednio {10, 100, 1000, 10000, 100000} elementów;
 - sortowanie w kolejności rosnącej dla zbioru nieposortowanego oraz dla zbioru wstępnie posortowanego w kolejności malejącej (najgorszy przypadek);
 - wyznacz statystykę czasu wykonania funkcji ze 100 powtórzeń dla pojedynczego przypadku testowego.

Otrzymane wyniki przedstaw na wykresach korzystając z dowolnego arkusza kalkulacyjnego.

4. Dodatek

4.1. Biblioteki dynamiczne jako wtyczki

Wtyczkami (*ang. plug-ins*) nazywa się moduły programistyczne wczytywane w trakcie działania programu w celu rozszerzenia bądź zmiany jego funkcjonalności. W językach *C/C++* takimi modułami mogą być **biblioteki dynamiczne**. W zależności od systemu operacyjnego, w ramach którego ma być uruchamiany program, do wczytywania bibliotek mogą posłużyć odpowiednio funkcje **LoadLibraryA()** (*Windows, Win32 API, windows.h*) oraz **dlopen()** (*Unix, dlfcn.h*).

Program obsługujący **bibliotekę libmath** z listingu 1. w systemie operacyjnym *Windows* przedstawiono na listingu 4. (zawartość pliku *main.c*). Funkcja **LoadLibraryA()** jako argument przyjmuje nazwę (lub ścieżkę do) wczytywanego **modułu DLL** i zwraca uchwyt (wskaźnik) typu **HINSTANCE**. Jeżeli moduł został wczytany poprawnie, to zwrócony wskaźnik ma wartość różną od **NULL**. Funkcja **GetProcAddress()** zwraca adres, pod którym znajduje się określony symbol z wczytanego modułu. Na listingu 4. pobierany jest adres do funkcji **multiply** z **biblioteki libmath**. Istotne jest, aby symbol, przekazany do funkcji **GetProcAddress()** za pomocą łańcucha znaków, był w pełni zgodny z **interfejsem biblioteki**. Zwrócony adres należy rzutować na typ wskaźnika funkcyjnego odpowiadającego nagłówkowi funkcji biblioteczej. Jeżeli adres jest różny od **NULL**, to symbol został pobrany poprawnie i można wywołać określoną funkcję biblioteczną wykorzystując wskaźnik funkcyjny. Po zakończeniu pracy z modułem należy go zwolnić wywołując funkcję **FreeLibrary()**, przyjmującą uchwyt zwrócony z funkcji **LoadLibraryA()**. Funkcja **FreeLibrary()** usuwa moduł z przestrzeni adresowej programu i zwraca wartość różną od 0 w przypadku powodzenia. Aby korzystać z opisanych funkcji należy zaimportować nagłówek **windows.h**.

```
1 #include <stdio.h>
2 #include <windows.h>
3
4 #include "math_functions.h"
5
6 // Typ wskaźnika funkcyjnego zgodny z nagłówkiem
   funkcji multiply z biblioteki libmath
7 typedef double (*FcnPtr)(double, double);
8
9 int main() {
10     // Wczytaj modul biblioteki libmath.dll
11     HINSTANCE library = LoadLibraryA(TEXT("libmath.
   dll"));
12
13     // Jeżeli poprawnie wczytano modul biblioteki
14     if (library != NULL) {
15         // Pobierz adres do funkcji multiply z
   biblioteki libmath
16         FcnPtr handle = (FcnPtr) GetProcAddress(library,
   "multiply");
17
18         // Jeżeli pobrany adres jest poprawny
19         if (handle != NULL) {
20             // Wywołaj funkcję biblioteczną przez wskaźnik
   funkcyjny
21             printf("3.78 * 7.12 = %f\n", handle(3.78,
   7.12));
22         } else
23             printf("Failed to get the address of the
   library function\n");
24     }
```

```

25     // Zwolnij modul biblioteki libmath.dll
26     if (FreeLibrary(library) == 0)
27         printf("Failed to release the DLL module\n");
28 } else
29     printf("Failed to load the DLL module\n");
30
31 return 0;
32 }

```

Listing 4. Użycie funkcji bibliotecznych *libmath* w postaci wtyczki w systemie Windows

Pracując w systemach *Unix* można wczytywać moduły **bibliotek dynamicznych** w czasie wykonania programu wykorzystując funkcję *dlopen()* z **biblioteki libdl**. Program obsługujący **bibliotekę libmath** z listingu 1. przedstawiono na listingu 5. Funkcja *dlopen()* jako argumenty przyjmuje nazwę (lub ścieżkę do) wczytywanego **modułu biblioteki** oraz flagi konfigurujące sposób obsługi modułu. Na listingu 5. zastosowano flagę **RTLD_LAZY**, która określa, że wiązane będą wyłącznie symbole biblioteczne wywoływane w kodzie aplikacji [1]. Funkcja *dlopen()* zwraca wskaźnik typu *void **, którego wartość jest różna od **NULL**, jeżeli moduł został wczytany poprawnie. Funkcja *dlsym()* stanowi odpowiednik funkcji *GetProcAddress()* z systemu *Windows*. Weryfikacja poprawności pobranego adresu do symbolu z **modułu biblioteki** odbywa się dodatkowo przez sprawdzenie ostatniego błędu **biblioteki libdl**. Służy do tego funkcja *dlerror()*, która zwraca łańcuch znaków zawierający opis ostatniego błędu lub wartość **NULL** w przypadku braku błędów. Aby zwolnić wczytany **moduł biblioteki** należy posłużyć się funkcją *dlclose()*, zwracającą wartość równą 0 w przypadku powodzenia. Aby korzystać z opisanych funkcji należy zaimportować nagłówek **dlfcn.h**.

```
1 #include <stdio.h>
2 #include <dlfcn.h>
3
4 #include "math_functions.h"
5
6 // Typ wskaźnika funkcyjnego zgodny z nagłówkiem
   funkcji multiply z biblioteki libmath
7 typedef double (*FcnPtr)(double, double);
8
9 int main() {
10     // Wczytaj modul biblioteki libmath.dll
11     void * library = dlopen("libmath.so", RTLD_LAZY)
12     ;
13
14     // Jeżeli poprawnie wczytano modul biblioteki
15     if (library != NULL) {
16         // Wyczyść dotychczasowe flagi błędów
17         dlerror();
18
19         // Pobierz adres do funkcji multiply z
20         biblioteki libmath
21         FcnPtr handle = (FcnPtr) dlsym(library, "
22         multiply");
23
24         // Jeżeli pobrany adres jest poprawny
25         if (handle != NULL && dlerror() == NULL) {
26             // Wywołaj funkcję biblioteczną przez wskaźnik
27             funkcyjny
28             printf("3.78 * 7.12 = %f\n", handle(3.78,
29             7.12));
30         } else
```



```

26     printf("Failed to get the address of the
    library function\n");
27
28     // Zwolnij modul biblioteki libmath.dll
29     if (dlclose(library) != 0)
30         printf("Failed to release the library module\n
    ");
31 } else
32     printf("Failed to load the library module\n");
33
34 return 0;
35 }

```

Listing 5. Użycie funkcji bibliotecznych *libmath* w postaci wtyczki w systemie Unix

W obu przypadkach **biblioteka *libmath*** została wstępnie skompilowana do postaci **biblioteki dynamicznej** (*libmath.dll* w systemie *Windows*, *libmath.so* w systemie *Unix*). Ponieważ plik biblioteki wczytywany jest dynamicznie w trakcie działania programu, to kompilację pliku wykonywalnego aplikacji można przeprowadzić wywołując:

`gcc -o app main.c` (w systemie *Windows*)

`gcc -o app main.c -ldl` (w systemie *Unix*; konsolidacja **biblioteki *libdl***)

Literatura

- [1] *dlopen(3)* - *Linux man page*. URL: <https://linux.die.net/man/3/dlopen>.
- [2] *Dynamic Linking*. URL: https://refspecs.linuxfoundation.org/ELF/zSeries/lzsabi0_zSeries/x2251.html.