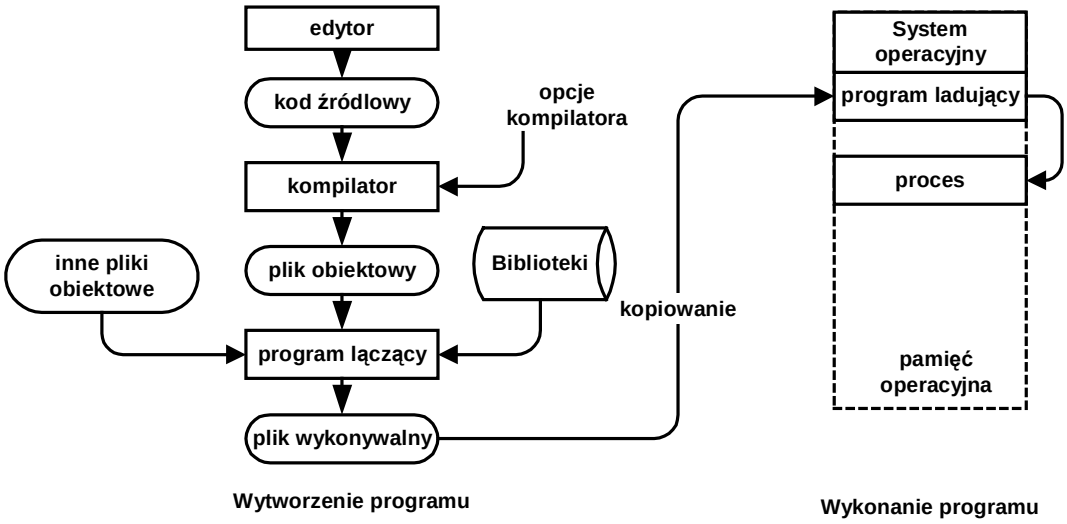
1. Kompilacja i uruchamianie programów

1.1 Jak kod źródłowy przekształca się w proces

- Kod aplikacji tworzony jest zazwyczaj w języku wysokiego poziomu np. C.
- Plik źródłowy przetwarzany jest przez kompilator do programu wykonywalnego.
- Program wykonywalny przekształcany jest w wykonujący się proces, co wykonywane jest przez program ładujący systemu operacyjnego (ang. *loader*).
- Program wykonywalny może się wykonać na tej samej maszynie na której został utworzony, lub, jest on przesyłany do systemu docelowego i tam dopiero wykonany (w systemach wbudowanych).



Rys. 1-1 Przebieg procesu wytworzenia i wykonania programu

Przetworzenie kodu źródłowego w wykonywany proces odbywa się w kilku etapach. Najważniejsze to:

- kompilacja,
- łączenie
- ładowanie programu.

1.1.1 Kompilacja

Celem kompilacji jest transformacja kodu źródłowego będącego zapisem algorytmu w języku wysokiego poziomu (który nie może być wykonany przez procesor) na kod maszynowy danego procesora.

Kompilacja przebiega w kilku etapach i prowadzi ona do wytworzenia pliku obiektowego.

Plik obiektowy zawiera kod maszynowy właściwy dla procesora na którym kod będzie wykonywany i informacje dodatkowe.

Typowy plik obiektowy składa się z takich części jak:

- nagłówek,
- kod maszynowy,
- dane,
- tablica symboli,
- informacje o relokacji,
- informacje dla programu uruchomieniowego (ang. *debugger*).

Na etapie kompilacji nie sposób określić pod jaki adres w pamięci należy załadować utworzony program, gdyż kompilator nie posiada informacji o stanie pamięci procesora w chwili wykonania programu.
Pliki obiektowe i wykonywalne zawierają tak zwaną tablicę relokacji (ang. *relocation table*).

Składa się ona z pozycji, z których każda zawiera wskaźnik do adresu w kodzie obiektowym, który musi być zmodyfikowany w procesie ładowania programu do pamięci operacyjnej.

W systemie Linux plik obiektowy jak i wykonywalny tworzony jest w tak zwanym formacie ELF (ang. *Executable and Linkable Format*).
Informacje o plikach w formacie ELF uzyskać można za pomocą narzędzi Linuksowych takich jak:

- **`readelf`**
- **`objdump`**,

Kompilator języka C - **`gcc`** .

1.1.2 Łączenie

Plik obiektowy zawiera:

- tłumaczenie kodu źródłowego na instrukcje kodu maszynowego
- dane na których te instrukcje operują

Nie jest jeszcze kompletnym programem gdyż nie zawiera:

- bibliotek
- innych segmentów programu.

Kompletny program wykonywalny powstanie na etapie łączenia. peracje łączenia wykonuje program nazywany konsolidatorem lub linkerem (ang. *linker*).

Konsolidator dołącza programu głównego inne pliki obiektowe i biblioteki w wyniku czego powstaje program wykonywalny. Jest on także w formacie ELF.

W systemie Linux rolę linkera pełni program `ld`.

1.1.3 Ładowanie programu

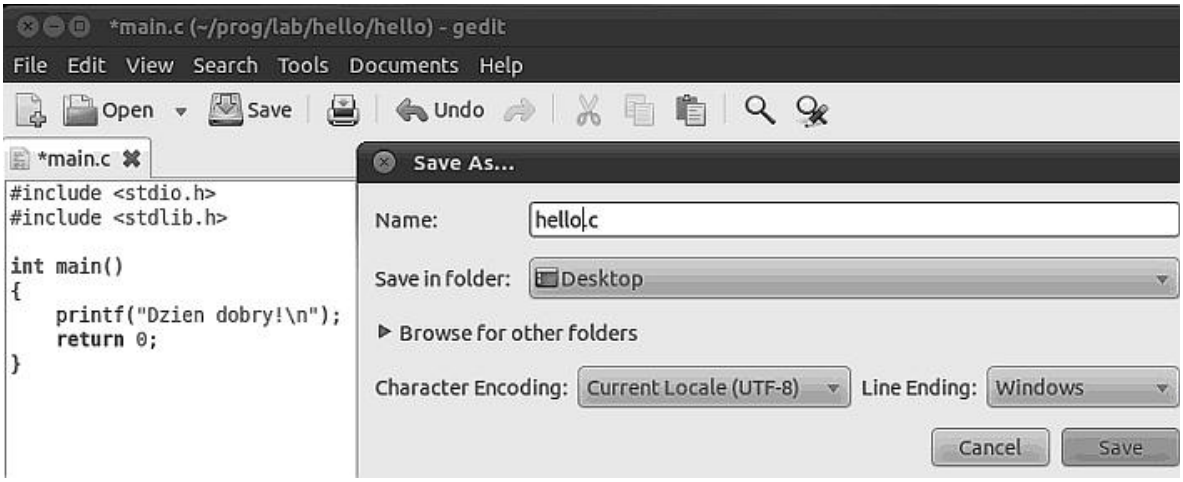
Kolejną czynnością która musi być wykonana jest utworzenie procesu na podstawie pliku wykonywalnego. Czynność tę wykonuje program ładujący (ang. *loader*). Funkcje programu ładującego to:

- Weryfikacja pozwoleń, wymagań na zasoby
- Skopiowanie segmentów programu do pamięci operacyjnej
- Skopiowanie argumentów linii poleceń na stos
- Inicjalizacja rejestrów procesora
- Przekazanie sterowania do punktu startowego programu

Po wykonaniu powyższych czynności program zostaje przekształcony w proces i przystępuje do wykonywania swojej funkcji.

1.2 Metoda elementarna – użycie edytor gedit i kompilatora gcc

Najprostszą metodą tworzenia i uruchamiania programów w systemie Linux jest użycie systemowego edytora `gedit` i kompilatora `gcc` uruchamianego w trybie wsadowym.



Przykład 1-1 Edycja programu hello.c

$gcc hello.c –o hello
$./hello

```
juka@debian6:~$ gcc hello.c -o hello
juka@debian6:~$ ./hello
Dzien dobry !
```

Przykład 1-1 Kompilacja i uruchomienie programu **hello.c**

Polecenie **file** pozwala na zbadanie rodzaju pliku

```
$file hello
hello: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), dynamically linked (uses shared
libs), for GNU/Linux 2.6.18, not stripped
```

Przykład 1-2 Ilustracja działania programu **file** dla pliku wykonywalnego

Program źródłowy można skompilować do postaci pliku obiektowego który następnie będzie konsolidowany. Robi się to korzystając z opcji –c kompilatora **gcc**.

gcc hello.c –c –o hello.o

Plik **hello.o** jest plikiem obiektowym. Gdy zastosujemy do tego pliku polecenie **file** otrzymamy:

```
$file hello
hello.o: ELF 32-bit LSB relocatable, Intel 80386, version 1
(SYSV), not stripped
```

Przykład 1-3 Ilustracja działania programu file dla pliku obiektowego

O pliku obiektowym można uzyskać różne informacje posługując się programem objdump . Gdy wykonamy go z opcją –x otrzymamy wiele informacji o zawartości pliku obiektowego hello.o

```
$objdump –x hello.o
hello.o:     file format elf32-i386
architecture: i386, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000001c  00000000  00000000  00000034  2**2
  1 .data         00000000  00000000  00000000  00000050  2**2
  2 .bss          00000000  00000000  00000000  00000050  2**2
  3 .rodata       0000000d  00000000  00000000  00000050  2**0
  4 .comment      0000001d  00000000  00000000  0000005d  2**0
  5 .note.GNU-stack 00000000  00000000  00000000  0000007a  2**0
…
SYMBOL TABLE:
…
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
…
```

Przykład 1-4 Ilustracja działania programu objdump dla pliku obiektowego

Z powyższego przykładu widać jakie informacje zawiera plik obiektowy. Są to nagłówki, segmenty, tablica symboli i dane do relokacji. Ważniejsze segmenty to:

- **`.text`** – segment kodu, zawiera instrukcje
- **`.data`** – segment danych zainicjowanych, dane którym nadano wartości początkowe
- **`.bss`** – segment danych nie zainicjowanych, dane którym nie nadano wartości początkowych
- **`.rodata`** – segment danych zawierających stałe (tylko do odczytu)
- **`.comment`** – komentarze
- **`.note.GNU-stack`** – informacja że potrzebny będzie stos

Za pomocą polecenia **objdump** z opcją –d można uzyskać kod assemblera zawarty w segmencie kodu co pokazuje poniższy przykład.

```
objdump –d hello.o
00000000 <main>:
   0:     55                      push   %ebp
   1:     89 e5                   mov    %esp,%ebp
   3:     83 e4 f0                and    $0xfffffff0,%esp
   6:     83 ec 10                sub    $0x10,%esp
   9:     c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  10:     e8 fc ff ff ff          call   11 <main+0x11>
  15:     b8 00 00 00 00          mov    $0x0,%eax
  1a:     c9                      leave
  1b:     c3                      ret
```

Przykład 1-5 Ilustracja działania programu **objdump** dla pliku obiektowego – disassemlacja segmentu kodu

W kolejnym etapie możemy dokonać konsolidacji pliku **hello.o**

```
gcc hello.o –o hello
```

W tym przykładzie budowanie programu wykonywalnego przebiegało w dwóch etapach.

- pierwszym utworzyliśmy plik obiektowy
- drugim dokonaliśmy jego konsolidacji otrzymując plik wykonywalny.

Wiele informacji o pliku wykonywalnym można uzyskać za pomocą programów:

- **file,**
- **size,**
- **readelf.**

Polecenie **size** pozwala na uzyskanie informacji o rozmiarach programu co pokazuje poniższy przykład.

```
$size hello
text    data     bss     dec     hex filename
916      264       8    1188     4a4 hello
```

Przykład 1-6 Ilustracja działania programu **size** dla pliku programu **hello**

Chodzi przy tym nie o rozmiar pliku wykonywalnego zapisanego na dysku ale o rozmiar programu umieszczonego w pamięci operacyjnej.

- **text** - wielkość segmentu kodu,
- **data** - wielkość segmentu danych zainicjowanych,
- **bss** - wielkość segmentu danych nie zainicjowanych
- **dec** - całkowitą wielkość pamięci zajmowaną przez program

Zestawienie narzędzi przydatnych w analizie programów:

Program	Opis	Przykład
objdump	Podaje zawartość plików obiektowych	**objdump –x hello.o**
readelf	Odczyt pliku w formacie ELF	**readelf –a hello**
size	Podaje ile pamięci zajmuje proces	**size hello**
file	Podaje typ pliku	**file hello**

Tabela 1-1 Zestawienie narzędzi do analizy plików programowych

1.3 Kompilator gcc

1.3.1 Wiele plików źródłowych

W licznych przypadkach program źródłowy składa się z wielu plików. Pliki składowe mogą być kompilowane oddzielnie a potem łączone. Załóżmy że kod źródłowy składa się z plików: **`main.c`** i **`pierwszy.c`**. Kompilujemy je oddzielnie poprzez polecenia:

```
$gcc –c main.c
$gcc –c pierwszy.c
```

W wyniku kompilacji utworzone zostaną dwa pliki obiektowe: **`main.o pierwszy.o`**.

Do zbudowania pliku wykonywalnego potrzebny jest jeszcze proces konsolidacji która to wykonywana jest przez program nazywany linkerem (W Linuksie nazywa się **`ld`**). Nie jest raczej wywoływany wprost lecz wywołuje go program **`gcc`**.

$gcc –o main pierwszy.o main.o

1.3.2 Pliki nagłówkowe

Pliki nagłówkowe są także plikami źródłowymi zawierającymi deklaracje typów i funkcji. Potrzebne są po to aby kompilator mógł sprawdzić prawidłowość użycia funkcji i zmiennych które zaimplementowane są w innych plikach. Wiele problemów z kompilacją ma swoje źródło w tym że kompilator nie wie gdzie położone są pliki nagłówkowe.

```
#include <stdio.h>
```

Linuks przechowuje pliki nagłówkowe w katalogu **`/usr/include`**. Gdyby plik nagłówkowy o nazwie **`pierwszy.h`** był w innym katalogu powiedzmy **`/home/juka/include`** należałoby poinformować o tym kompilator jak poniżej.

```
$gcc –c –I/home/juka/include pierwszy.c
```

Jeżeli plik nagłówkowy jest w tym samym katalogu co plik źródłowy to umieszczamy jego nazwę w podwójnym cudzysłowie.

```
#include ” pierwszy.h”
```

1.3.3 Preprocesor języka C

Zanim właściwy kompilator przystąpi do pracy, plik źródłowy przetwarzany jest wstępnie przez program zwany preprocesorem. Informacje dla preprocesora, nazywane dyrektywami, poprzedzane są znakiem krzyżyka #.
Preprocesor można uruchomić bezpośrednio, nazywa się **cpp**. Na przykład:

```
$cpp hello.c
```

Można też wywołać kompilator **gcc** z opcją –E, na przykład:

```
$gcc -E hello.c
```

Preprocesor wyróżnia trzy rodzaje dyrektyw:

- Pliki nagłówkowe
- Makrodefinicje
- Dyrektywy warunkowe

Dyrektywa **#include plik** nakazuje preprocesorowi włączyć do kodu cały **plik** wymieniony po **#include**.

Makrodefinicja nakazuje zastąpienie jednego łańcucha znaków innym łańcuchem. Na przykład

```
#define SUMA(a,b)  (a + b)
```

Napotkane w programie napisy **SUMA(2,3)** zostaną zastąpione przez napis **(2 + 3).**
Makrodefinicje mogą być podane także w linii poleceń kompilatora poprzez użycie opcji **-D,** np.:

```
$gcc main.c –DDEBUG=1
```

Działanie tej opcji będzie takie same jak pojawienie się w kodzie linii:

```
#define DEBUG 1
```

Dyrektywy warunkowe pozwalają na włączenie/wyłączenie pewnych fragmentów kodu za pomocą dyrektyw:

```
#ifdef, #if, #endif.
```

`#ifdef MAKRO`
`  tekst`
`#endif`
	Gdy MAKRO zostało wcześniej zdefiniowane, to tekst zostanie włączony

`#if warunek`
`  tekst`
`#endif`
	Gdy warunek ma wartość większą od zera, to tekst zostanie włączony

Fragment kodu pomiędzy liniami **#ifdef DEBUG** a **#endif** zostanie włączony tylko wtedy, gdy zdefiniowane będzie makro **DEBUG.** Można to osiągnąć albo poprzez odkomentowanie trzeciej linii poniższego przykładu lub poprzez użycie opcji –D kompilatora.

```
#include <stdio.h>
#include <stdlib.h>
// #define DEBUG 1

int main(int argc, char *argv[])
{
    #ifdef DEBUG
     printf("argc: %d argv[0]: %s\n",argc,argv[0]);
    #endif
    printf("Dzien dobry\n");
    return 0;
}
```

Przykład 1-7 Kompilacja warunkowa program hello2.c

```
$gcc hello2.c –o hello2 –DDEBUG=1
```

Wykonanie programu:

```
$./hello2
argc: 1 argv[0]: ./hello2
Dzien dobry
```

Gdy w linii kompilacji pominiemy opcję –D wynik działania programu nie będzie zawierał informacji o parametrach funkcji main.

```
$gcc hello2.c –o hello2
$./hello2
Dzien dobry
```

1.4 Biblioteki

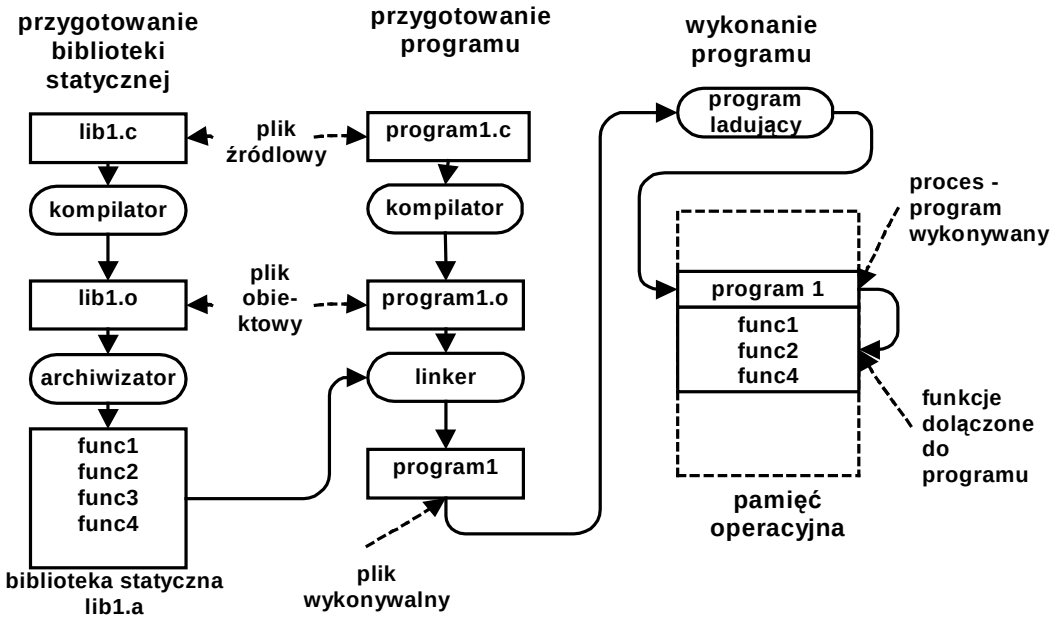
Programy wykonują wiele typowych czynności których samodzielne programowanie byłoby niecelowe. Czynności te, typowo realizowane są przez napisane wcześniej funkcje. Funkcje zgrupowane są w bibliotekach, które są nieodłącznym elementem systemu.
Wyróżniamy dwa rodzaje bibliotek:

- Biblioteki statyczne
- Biblioteki współdzielone

1.4.1 Biblioteki statyczne

Biblioteka statyczna składa się z funkcji i danych na których funkcje te operują. Bibliotekę tworzy się kompilując zestawy funkcji, zawarte w jednym lub wielu plikach źródłowych, do postaci plików obiektowych. Pliki te następnie łączone są w archiwum za pomocą polecenia **`ar`**.

Gdy program łączący buduje plik wykonywalny i gdy jest wskazanie by korzystał z bibliotek statycznych, to przeszukuje on bibliotekę i dołącza do pliku wykonywalnego moduły zawierające potrzebne funkcje.



Rys. 1-2 Tworzenie i wykorzystanie biblioteki statycznej

Przykład - dwa programy **`pierwszy.c`** i **`drugi.c`** korzystają z funkcji **`pisz(char * text)`** która umieszczona jest w pliku **`wspolny.c`** .

```
#include "wspolny.h"
int main(void)
{
  pisz("program 1");
  return 0;
}
```

Kod 1-1 Plik programu **pierwszy.c**

```
#include "wspolny.h"
int main(void)
{
  pisz("program 2");
  return 0;
}
```

Kod 1-2 Plik programu **drugi.c**

```
#include <stdio.h>
void pisz(char * tekst) {
  printf("%s\n",tekst);
}
```

Kod 1-3 Plik biblioteki **wspolny.c**

```
void pisz(char * tekst);
```

Kod 1-4 Plik nagłówkowy **wspolny.h**

Standardowy sposób utworzenia pliku wykonywalnego **pierwszy** podany jest poniżej.

```
gcc –c pierwszy.c –o pierwszy.o
gcc –c wspolny.c –o wspolny
gcc –o pierwszy pierwszy.o wspolny.o
```

Pokażemy jak utworzyć bibliotekę statyczną zawierającą funkcję **pisz** która to funkcja wykorzystywana jest w programie **pierwszy.c i drugi.c.** Bibliotekę tworzymy w dwóch krokach:

- Krok1 - utworzenie pliku obiektowego **wspolny.o** zawierającego funkcję pisz za pomocą kompilatora gcc. Użyta będzie opcja –c nakazująca tylko kompilację bez tworzenia pliku wynikowego
- Krok 2 - utworzenie archiwum za pomocą programu archiwizatora ar

Aby utworzyć bibliotekę o nazwie **libwspolny.a** należy:

```
$gcc –c wspolny.c –o wspolny.o
$ar rcsv libwspolny.a wspolny.o
```

Ważne jest aby nazwa biblioteki zaczynała się od liter lib.
Można sprawdzić zawartość biblioteki za pomocą polecenia:
nm nazwa_biblioteki.

```
$nm libwspolny.a
libwspolny.o:
00000000 T pisz
         U puts
```

Aby uzyskać plik wykonywalny o nazwie pierwszy wykonujemy łączenie programu informując że dołączamy bibliotekę statyczną libwspolny.a

$gcc pierwszy.c –o pierwszy –L. -lwspolny

W powyższym poleceniu opcja –L. informuje kompilator że należy szukać biblioteki w katalogu bieżącym.

Należy zwrócić uwagę że po literze **–l** nie wpisujemy całej nazwy biblioteki ale część bez przedrostka **lib**. Brakujący przedrostek **lib** kompilator doda sam.

$./pierwszy

Program pierwszy

Częstym źródłem problemów jest to że program łączący nie może znaleźć właściwych bibliotek. Standardowo biblioteki umieszczone są w katalogu **/lib** i **/usr/lib**. Gdy biblioteki umieszczone są w innym miejscu należy poinformować o tym program łączący. Gdyby biblioteka **libwspolny.a** umieszczona była w katalogu **/home/juka/lib** należało by użyć następującej linii kompilacyjnej:

$gcc pierwszy.c –o pierwszy –L/home/juka/lib –lwspolny

1.4.2 Biblioteki współdzielone

1.4.2.1 Zasada działania

Zastosowanie bibliotek statycznych ma tak skutek że:

- Kod występujących w nich funkcji występuje wielokrotnie w różnych programach co powoduje niepotrzebne straty przestrzeni systemu plików
- W przypadku znalezienia w bibliotece błędu, należy przekompilować wszystkie programy

Powstał pomysł by biblioteki umieścić w ogólnie znanym miejscu, a wtedy wiele programów mogłoby z nich korzystać.

Aby posługiwać się bibliotekami współdzielonymi należy posiadać informację:

- Jak sprawdzić jakich bibliotek potrzebuje dany program
- Jak program szuka bibliotek współdzielonych
- Jak łączyć program z bibliotekami współdzielonymi
- Jak tworzyć biblioteki współdzielone

1.4.2.2 Wykorzystanie bibliotek współdzielonych

Aby program mógł odnaleźć bibliotekę współdzieloną musi być ona umieszczona w dobrze zdefiniowanym miejscu. Zgodnie z zaleceniem FHS (ang. *Filesystem Hierarchy Standard*) biblioteki współdzielone powinny być umieszczone w katalogu **`/usr/lib`** lub **`/usr/local/lib`**. Biblioteki współdzielone mają rozszerzenie **`so`**, przykładową biblioteką jest plik **`/lib/ldlinux.so.2`**. Za pomocą programu **`ldd`** można uzyskać informacje jakich bibliotek współdzielonych używa dany program.

```
$ldd hello
    linux-gate.so.1 =>  (0xb7721000)
    libc.so.6 => /lib/i686/cmov/libc.so.6
(0xb75c5000)
    /lib/ld-linux.so.2 (0xb7722000)
```

Przykład 1-8 Uzyskanie informacji o bibliotekach dzielonych za pomocą polecenia **`ldd`**

W celu zachowania elastyczności, programy nie zawierają informacji o bezwzględnym położeniu bibliotek współdzielonych a tylko ich nazwy. Odnajdowaniem bibliotek współdzielonych zajmuje się konsolidator dynamiczny (ang. *runtime dynamic linker*). W systemie Linux jest on widoczny w powyższym przykładzie jako **`ld-linux.so.2`**.

Stosunkowo częstą przyczyną błędów jest niemożność odnalezienia właściwej biblioteki współdzielonej. Konsolidator dynamiczny poszukuje bibliotek w następujący sposób:

1. Sprawdza czy istnieje zmienna środowiska LD_LIBRARY_PATH. Gdy tak poszukuje biblioteki w wskazanej przez tę zmienna ścieżce.
2. Sprawdza czy lokalizacja biblioteki umieszczona jest w systemowym schowku (ang. *cache*) /**etc/ld.so.cache**
3. Odczytuje plik **/etc/ld.so.conf** gdzie są nazwy plików z bibliotekami dzielonymi. (dla przykładu zawiera on treść: **include /etc/ld.so.conf.d/*.conf).** W przykładzie są to pliki: **i486-linux-gnu.conf, libc.conf, vmware-tools-libraries.conf.**
4. Pliki zawierają znane systemowi położenia bibliotek współdzielonych, np. plik **libc.conf** zawiera wpis **/usr/local/lib**

Gdyby zostały wprowadzone zmiany w powyższych plikach konfiguracyjnych, należy zaktualizować schowek przez polecenie:

#ldconfig –v

Gdy tworzymy program który ma używać niestandardowej biblioteki współdzielonej należy poinformować o tym program łączący. Powiedzmy że w programie **pierwszy** mamy użyć biblioteki: **/home/juka/lib/libwspolny.so.**

Jak utworzyć tę bibliotekę pokazane zostanie dalej.
gcc –o pierwszy pierwszy.c –Wl,-rpath=/home/juka/lib –L/home/juka/lib -lwspolny

Można teraz za pomocą polecenia **ldd** sprawdzić z jakich bibliotek korzysta program **pierwszy**.

```
$ldd pierwszy
linux-gate.so.1 =>  (0xb76ea000)
libwspolny.so => /home/juka/lib/libwspolny.so (0xb76e6000)
libc.so.6 => /lib/i686/cmov/libc.so.6 (0xb758c000)
/lib/ld-linux.so.2 (0xb76eb000)
```

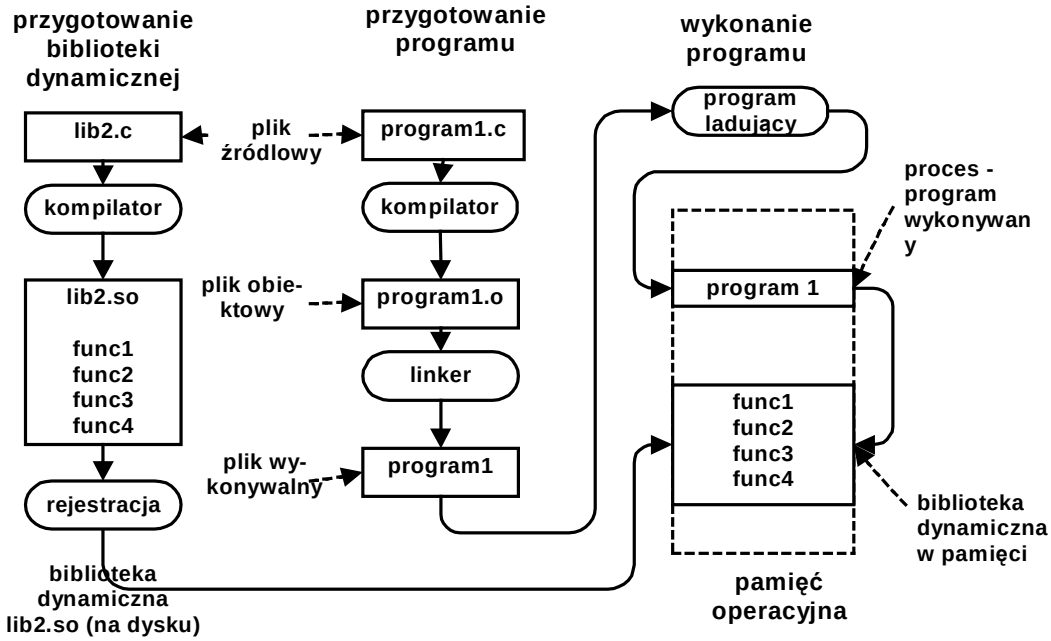
Przykład 1-9 Testowanie bibliotek używanych przez program `pierwszy` za pomocą narzędzia **ldd**

1.4.3 Tworzenie biblioteki współdzielonej

Aby utworzyć bibliotekę współdzieloną trzeba:

- napisać kod źródłowy funkcji wchodzących w skład biblioteki,
- skompilować do postaci biblioteki współdzielonej używając do tego odpowiednich opcji kompilatora (–shared –fpic).
- zainstalować bibliotekę czyli poinformować system o nazwie biblioteki i o lokalizacji pliku ją zawierającego.

W procesie konsolidacji programu wykonywalnego biblioteki nie są dołączane do pliku wynikowego, ale są tylko rejestrowane to znaczy program łączący wpisuje do pliku wykonywalnego informację o używanej bibliotece, tak aby było możliwe, załadowanie biblioteki w czasie wykonania programu.



Rys. 1-3 Tworzenie i wykorzystanie biblioteki współdzielonej

Pokażemy jak utworzyć bibliotekę współdzieloną. Plik **`wspolny.c`** zawiera kod funkcji **`pisz(char * tekst)`** wykorzystywanej w programach **`pierwszy`** i **`drugi`**.

```
$gcc –shared –fpic –o libwspolny.so wspolny.c
```

Powyższe polecenie zleca utworzenie biblioteki współdzielonej o nazwie **`libwspolny.so`** z pliku **`wspolny.c`** .

Dalej kopiujemy bibliotekę do katalogu **`/home/juka/lib`**. Następnie tworzymy plik wykonywalny dla programu **`pierwszy`** pisząc polecenie:

```
$gcc –o pierwszy pierwszy.c –Wl,-rpath=/home/juka/lib
–L/home/juka/lib -lwspolny
```

Polecenie:

- nakazuje utworzenie pliku wykonywalnego **pierwszy** z pliku **pierwszy.c,**
- użycie biblioteki współdzielonej **libwspolny.so** umieszczonej w katalogu **/home/juka/lib**.

Za pomocą polecenia **ldd** możemy sprawdzić jakich bibliotek dynamicznych używa program pierwszy.

```
$ldd pierwszy
linux-gate.so.1 =>  (0xb76ea000)
libwspolny.so => /home/juka/lib/libwspolny.so (0xb76e6000)
libc.so.6 => /lib/i686/cmov/libc.so.6 (0xb758c000)
/lib/ld-linux.so.2 (0xb76eb000)
```

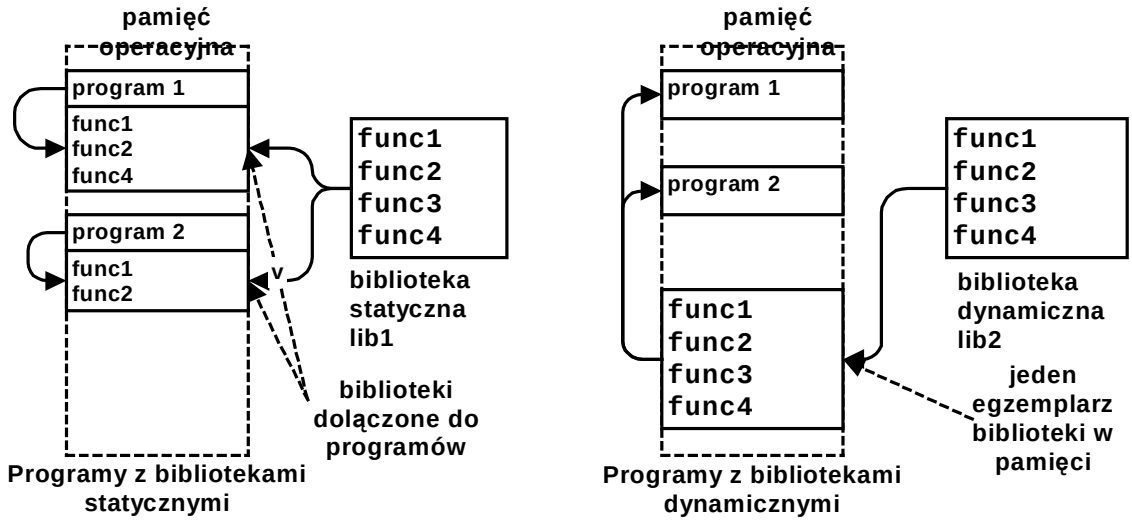
Przykład 1-10 Testowanie bibliotek używanych przez program **pierwszy** za pomocą narzędzia **ldd**

Program	Opis	Przykład
ldconfig	Instalacja biblioteki współdzielonej	**ldconfig –p** (testowanie jakie biblioteki współdzielone są zainstalowane)
ldd	Podaj z jakich bibliotek dynamicznych korzysta program	**ldd hello**
nm	Podaj zawartość pliku biblioteki	**nm library.a**
readelf	Odczyt pliku w formacie ELF	**readelf –a /lib/libutil.so.1**

Tabela 1-2 Zestawienie narzędzi do analizy bibliotek

1.4.4 Biblioteki statyczne i współdzielone – porównanie

Gdy program wykonywalny korzysta z bibliotek statycznych, to zawiera on w sobie wszystkie potrzebne funkcje biblioteczne. Podejście to ma tak zalety jak i wady. Wadą jest to że jeżeli w komputerze wykonuje się kilkadziesiąt procesów korzystających z tych samych bibliotek to występujące w nich moduły wielokrotnie się powielają co prowadzi do niepotrzebnej straty pamięci.



Rys. 1-4 Ilustracja działania biblioteki statycznej i współdzielonej

Biblioteka statyczna lib1 zawiera funkcje func1, func2, func3 i func4. Program 1 po konsolidacji zawiera funkcje func1, func2 i func4, a program 2 zawiera funkcje func1 i func2. Widać że funkcje func1 i func2 dublują się co powoduje utratę pamięci.

Ważną cechą biblioteki współdzielonej jest fakt że wystarczy gdy w pamięci operacyjnej będzie tylko jedna jej kopia.

Tak więc zastosowanie bibliotek dynamicznych powoduje zwykle oszczędność pamięci. Zaletą biblioteki współdzielonej jest także większa łatwość aktualizacji.

Załóżmy że w bibliotece wykryjemy błąd.

- Gdy jest to biblioteka statyczna, to w celu naprawienia błędu, należy dokonać ponownego łączenia wszystkich plików wykonywalnych.
- W przypadku użycia biblioteki współdzielonej poprawiamy tylko samą bibliotekę.

Biblioteki współdzielone mają jednak także poważne wady.

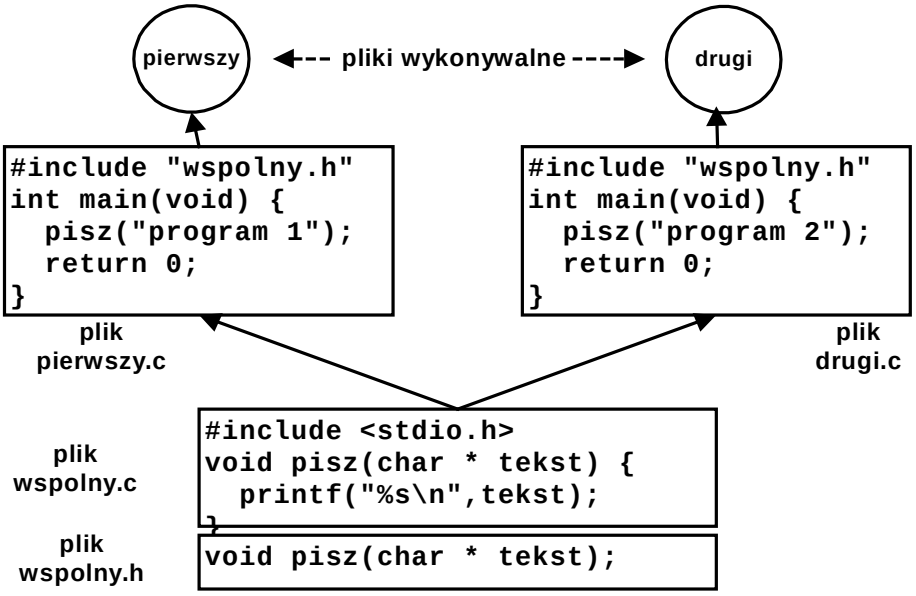
- Trudność określenia czy dany program wykona się na innych komputerach niż ten na którym został opracowany i przetestowany.
- Trzeba utrzymywać wiele wersji bibliotek gdyż pewne programy mogą korzystać ze starej wersji

1.5 Uruchamianie programów za pomocą narzędzia make

W praktyce programistycznej typowa jest sytuacja gdy aplikacja składa się z pewnej liczby programów wykonywalnych zawierających jednak pewne wspólne elementy (stałe, zmienne, funkcje).

Narzędzie **make** powszechnie stosowane w tworzeniu złożonych aplikacji.

Przykładowa aplikacja składa się z dwóch programów: **pierwszy.c** i **drugi.c**. Każdy z programów wypisuje na konsoli swoją nazwę i w tym celu korzysta z funkcji **void pisz(char * tekst)** zdefiniowanej w pliku **wspolny.c** a jej prototyp zawarty jest w pliku **wspolny.h**. Sytuację pokazuje poniższy rysunek.



Rys. 1-1 Aplikacja składająca się z dwóch programów

Aby skompilować aplikację należy:

```
$gcc pierwszy.c wspolny.c –o pierwszy
$gcc drugi.c wspolny.c –o drugi
```

Analogiczny efekt osiągnąć można tworząc plik definicji **makefile** dla narzędzia **make** a następnie pisząc z konsoli polecenie **make**.

Po wpisaniu polecenia **make** system szuka w folderze bieżącym pliku o nazwie **Makefile** a następnie **makefile** po czym go przetwarza.

```
# Plik makefile dla aplikacji skladajacej się z dwoch programow
all: pierwszy drugi
pierwszy: pierwszy.c wspolny.c wspolny.h
    gcc -o pierwszy pierwszy.c wspolny.c
drugi: drugi.c wspolny.c wspolny.h
    gcc -o drugi drugi.c wspolny.c
```

Przykład 1-2 Plik **makefile** dla aplikacji składającej się z dwóch plików

Wyniki działania polecenia **make:**

```
$ ls
drugi.c makefile pierwszy.c wspolny.c wspolny.h
$make
gcc –o pierwszy pierwszy.c wspolny.c
gcc –o drugi drugi.c wspolny.c
$ls
drugi drugi.c makefile pierwszy pierwszy.c wspolny.c
wspolny.h
```

Przykład 1-3 Działanie polecenia make

Plik definicji **makefile** składa się z **zależności** i **reguł**.

Zależność podaje jaki cel ma być osiągnięty (zwykle jest to nazwa pliku który ma być utworzony) i od jakich innych plików zależy. Na podstawie zależności program **make** określa jakie pliki są potrzebne do kompilacji, sprawdza czy ich kompilacja jest aktualna - jeśli tak, to pozostawia bez zmian, jeśli nie, sam kompiluje to co jest potrzebne zgodnie z poleceniem.

W prostym wariancie reguła składa się z nazwy celu (może to być nazwa pliku wynikowego lub akcji którą należy przeprowadzić) a po dwukropku listy plików od których dany cel zależy. Jeżeli program **make** stwierdzi że plik wynikowy jest starszy od któregoś z plików od którego zależy dokonywana jest jego kompilacja zgodnie z regułą zawartą w kolejnej linii.

cel: $\quad$ **plik**$_1$ **plik**$_2$ **... plik**$_n$
polecenia

Linia polecenia zaczyna się niewidocznym znakiem tabulacji. Zastąpienie znaku tabulacji spacjami spowoduje błędne działanie programu.

```
pierwszy: pierwszy.c wspolny.c
```

Informuje ona system że plik **pierwszy** zależy od plików **pierwszy.c wspolny.c** jakakolwiek zmiana w tych plikach spowoduje konieczność powtórnego tworzenia pliku **pierwszy**.
Reguły mówią jak taki plik utworzyć. W tym przykładzie aby utworzyć plik wykonywalny **pierwszy** należy uruchomić kompilator z parametrami jak poniżej.

```
gcc -o pierwszy pierwszy.c wspolny.c
```

```
$touch wspolny.c
$make
gcc –o pierwszy pierwszy.c wspolny.c
gcc –o drugi drugi.c wspolny.c
```

Przykład 1-4 Działanie polecenia make – rekompilacja programów pierwszy i drugi

```
$touch pierwszy.c
$make
gcc –o pierwszy pierwszy.c wspolny.c
```

Przykład 1-5 Działanie polecenia make – rekompilacja programu pierwszy

W plikach **makefile** umieszczać można linie komentarza poprzez umieszczenie na pierwszej pozycji takiej linii znaku #.

W linii określającej cel zależność może być pominięta. Tak jest w poniższym przykładzie gdzie cel **archiw** nie zawiera żadnej zależności. Natomiast akcja definiuje wykonanie archiwizacji plików źródłowych. Gdy program **make** zostanie wywołany z parametrem będącym nazwą pewnego celu można spowodować wykonanie reguły odpowiadające temu celowi. Do pliku **makefile** dodać można regułę o nazwie **archiw** wykonania archiwizacji plików źródłowych.
Wpisanie polecenia: **make archiw** spowoduje utworzenie archiwum plików źródłowych i zapisanie ich w pliku **prace.tgz**.

```
all: pierwszy drugi
pierwszy: pierwszy.c wspolny.c wspolny.h
    gcc -o pierwszy pierwszy.c wspolny.c
drugi: drugi.c wspolny.c wspolny.h
    gcc -o drugi drugi.c wspolny.c
clean:
    rm *.o pierwszy drugi
archiw:
    tar -cvf prace.tar *.c *.h makefile
    gzip prace.tar
    mv prace.tar.gz prace.tgz
```

Przykład 1-6 Plik `make` z opcją archiwizacji plików źródłowych

1.5.1 Argumenty polecenia make

Program **`make`** może być wywołany z parametrami lub bez. Wywołany bez argumentów powoduje realizację pierwszego celu.

Przykłady wywołania programu z argumentami

`make cel` - powoduje realizację podanego w argumencie celu

`make –f plik` - powoduje przetwarzanie pliku

`make –n` - wypisuje działania ale ich nie wykonuje

1.5.2 Wbudowane makra i zmienne

System make posiada wbudowane makra. Przykłady niektórych pokazane są poniżej.

CFLAGS - opcje kompilatora języka C
CC - nazwa kompilatora języka C, domyślnie cc
CXXFLAGS - opcje kompilatora języka C

```
all: pierwszy drugi
pierwszy: pierwszy.c wspolny.c wspolny.h
    $(CC) -o pierwszy $(CFLAGS) pierwszy.c wspolny.c
drugi: drugi.c wspolny.c wspolny.h
    $(CC) -o drugi $(CFLAGS) drugi.c wspolny.c
```

Przykład 1-11 Przykład użycia wbudowanych makr

1.5.3 Makrodefinicje użytkownika

W plikach makefile można stosować makrodefinicje którym przypisuje się pewne wartości.
Odwołanie do zmiennej **nazwa** ma postać: $(**nazwa**). Postępowanie takie jest stosowane gdy w pliku **makefile** powtarzają się pewne napisy, na przykład nazwy plików.

```
all: pierwszy drugi
ZRODLA1= pierwszy.c wspolny.c wspolny.h
ZRODLA2= drugi.c  wspolny.c wspolny.h
pierwszy: $(ZRODLA1)
    $(CC) -o pierwszy $(CFLAGS) $(ZRODLA1)
drugi:  $(ZRODLA1)
    $(CC) -o drugi $(CFLAGS) $(ZRODLA2)
```

Przykład 1-7 Plik **make**– ilustracja użycia makrodefinicji

1.5.4 Typowe cele kompilacji

Znaczna liczba plików **makefile** zawiera typowe cele kompilacji, które realizują części procesu przygotowania programu. Typowe cele kompilacji dane są poniżej:

- **all** – jest to pierwszy cel kompilacji zdefiniowany w pliku makefile.
- **install**- akcja wykonywana w ramach tego celu ma spowodować skopiowanie plików wykonywalnych w ich właściwe miejsce przeznaczenia.
- **clean** - usunięcie plików pośrednich i wykonywalnych
- **test** - sprawdzenie czy utworzone programy działają poprawnie

1.6 Uruchamianie programów za pomocą narzędzia gdb

Często zdarza się że uruchamiany program nie zachowuje się w przewidywany przez nas sposób. Wówczas należy uzyskać dodatkowe informacje na temat:

- Ścieżki wykonania programu
- Wartości zmiennych a ogólniej zawartości pamięci związanej z programem

Informacje takie uzyskać można na dwa sposoby:

- Umieścić w kodzie programu dodatkowe instrukcje wyprowadzania informacji o przebiegu wykonania i wartości zmiennych.
- Użyć programu uruchomieniowego (ang. *Debugger*)

Gdy używamy pierwszej metody, dodatkowe informacje o przebiegu wykonania programu są zwykle wypisywane na konsoli za pomocą instrukcji **`printf`** lub też zapisywane do pliku.

W bardziej skomplikowanych przypadkach, wygodniej jest użyć programu uruchomieniowego. Program taki daje następujące możliwości:

- Uruchomienie programu i ustawienie dowolnych warunków jego wykonania (np. argumentów, zmiennych otoczenia, itd)
- Doprowadzenie do zatrzymania programu w określonych warunkach.
- Sprawdzenie stanu zatrzymanego programu (np. wartości zmiennych, zawartość rejestrów, pamięci, stosu)
- Zmiana stanu programu (np. wartości zmiennych) i ponowne wznowienie programu.

Szeroko używanym programem uruchomieniowym jest **gdb** (ang. *gnu debugger*) który jest częścią projektu GNU Richarda Stallmana. Może on być użyty do uruchamiania programów napisanych w językach C, C++, assembler, Ada , Fortran, Modula-2 i częściowo OpenCL.

Program działa w trybie tekstowym, jednak większość środowisk graficznych IDE takich jak Eclipse czy CodeBlocks potrafi się komunikować z `gdb` co umożliwia pracę w trybie okienkowym. Istnieją też środowiska graficzne specjalnie zaprojektowane do współpracy z **gdb** jak chociażby DDD (ang. *Data Display Debugger*).

1.6.1 Kompilacja programu

Aby możliwe było uruchamianie programu z użyciem **gdb** testowany program należy skompilować z kluczem: **-g**.
Użycie tego klucza powoduje że do pliku obiektowego z programem dołączona zostanie informacja o typach zmiennych i funkcji oraz zależność pomiędzy numerami linii programu a fragmentami kodu binarnego.
Aby skorzystać z debuggera program **test.c** należy skompilować następująco:

```
$gcc test.c -o test -g
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
     int i,j;
     puts("Witamy w Lab PRW");
      system("hostname");
     for(i=0;i<10;i++) {
        j=i+10;
        printf("Krok  %d\n",i);
        sleep(1);
     }
     printf("Koniec\n");
     return EXIT_SUCCESS;
}
```

Przykład 1-12 Program test.c

1.6.2 Uruchomienie i zakończenie

Program **gdb** uruchamia się w następujący sposób:

```
gdb [opcje] [prog [obraz-pam lub pid]]
gdb [opcje] --args prog [argumenty_prog ...]
```

gdzie:

opcje	Opcje programu które można uzyskać pisząc: gdb --h
prog	Nazwa pliku wykonywalnego
obraz-pam	Obraz pamięci utworzony przy awaryjnym zakończeniu programu
pid	Pid procesu do którego chcemy się dołączyć
args	Argumenty programu

Najprostszy sposób uruchomienia debuggera gdb:

```
$gdb prog
```

Można też dołączyć się do już działającego programu. W tym celu po nazwie programu należy podać jego pid co pokazuje Tabela 1-3.

Program gdb może też służyć do analizy przyczyny awaryjnego zakończenia programu. Gdy proces jest kończony, na skutek otrzymania jednego z pewnych sygnałów, system operacyjny tworzy plik zawierający obraz pamięci procesu. Obraz ten może być analizowany przez gdb w celu znalezienia przyczyny awaryjnego zakończenia procesu.

$gdb prog core.

Program gdb pozwala na podanie argumentów uruchamianego programu. Gdy program prog należy uruchomić z argumentami a1 a2 ... an to wtedy gdb należy uruchomić z opcją --arg jak następuje:

$gdb --arg prog a1 a2 ... an.

gdb prog	Zwykłe uruchomienie **gdb** dla programu zawartego w pliku **prog**
gdb prog core	Uruchomienie **gdb** dla programu z pliku **prog**. Obraz pamięci znajduje się w pliku core.
gdb prog pid	Dołączenie się do działającego programu, oprócz nazwy podajemy też jego **pid**
gdb --args prog argumenty	Uruchomienie **gdb** dla programu z argumentami
gdb –help	Uzyskiwanie pomocy

Tabela 1-3 Różne sposoby uruchomienia programu gdb

Program **gdb** kończy się wpisując polecenie **quit**, skrót **q** lub też kombinację klawiszy **Ctrl+d.**

Możemy uruchomić program gdb w celu testowania podanego w programu test. W tym celu piszemy polecenie:

$gdb test

Uzyskiwanie pomocy

Wpisując polecenie **help** uzyskujemy zestawienie kategorii poleceń, wpisując polecenie **help all** uzyskujemy zestawienie wszystkich poleceń.

Listowanie programu źródłowego

Uzyskanie fragmentu kodu źródłowego następuje przez użycie polecenia list. Polecenie to występować może w różnych wariantach co pokazuje Tabela 1-4.

list	Listowanie fragmentu kodu źródłowego począwszy od bieżącej pozycji
list nr	Listowanie fragmentu kodu źródłowego w pobliżu linii o numerze nr
list pocz, kon	Listowanie fragmentu kodu źródłowego od linii pocz do linii kon
list +	Listowanie nastepnego fragmentu kodu źródłowego (od pozycji bieżącej)
list -	Listowanie poprzedniego fragmentu kodu źródłowego (od pozycji bieżącej)

Tabela 1-4 Polecenia listowania fragmentu kodu źródłowego

```
(gdb) list
1
2       #include <stdio.h>
3       #include <stdlib.h>
4       #include <unistd.h>
5
6       int main(void) {
7               int i,j;
8               puts("Witamy w Lab PRW");
9               for(i=0;i<20;i++) {
10                 j=i+10;
(gdb)
```

Ekran 1-1 Listowanie kodu źródłowego

Zatrzymywanie procesu

Testowanie programów polega zwykle na próbie ich wykonania i zatrzymania, po czym następuje zbadanie stanu programu. Momenty zatrzymania określane przez tak zwane punkty zatrzymania (ang. *breakpoint*). Są trzy metody zdefiniowania punktu zatrzymania:

- Wskazanie określonej linii w kodzie programu lub też nazwy fukcji. Jest to zwykły punkt zatrzymania.
- Zatrzymanie programu gdy zmieni się wartość zdefiniowanego przez nas wyrażenia (ang. *watchpoint*).
- Zatrzymanie programu gdy zajdzie określone zdarzenie (ang. catchpoint) jak wystąpienie wyjątku czy załadowanie określonej biblioteki.

Tworzonym punktom zatrzymania nadawane są kolejne numery począwszy od 1. Po utworzeniu mogą one być aktywowane (ang *enable*), dezaktywowane (ang. *disable*) i kasowane (ang. *delete*). Najprostszym sposobem ustawienia punktu zatrzymania jest użycie polecenia: break nr_lini. Polecenie:

`info break`

powoduje wypisanie ustawionych punktów wstrzymania..

```
(gdb) break 10
Breakpoint 1 at 0x8048453: file test.c, line 10.
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x08048453 in main at test.c:10
(gdb)
```

Ekran 1-2 Ustawienie punktu zatrzymania

Punkty wstrzymania można kasować za pomocą polecenia:

`clear nr_linii.`

Polecenie **`break`** występuje w wielu wariantach. Można ustawić zatrzymanie procesu gdy spełniony jest pewien warunek.

`break nr_linii if warunek`

Skutkiem będzie zatrzymanie procesu w danej linii gdy warunek będzie spełniony. Np.

`break 10 if i > 5`

```
(gdb) break 10 if i > 5
Breakpoint 2 at 0x8048453: file test.c, line 10.
(gdb) i b
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x08048453 in main at test.c:10
        stop only if i > 5
(gdb)
```

Ekran 1-3 Ustawienie warunkowego punktu zatrzymania

Uruchamianie procesu

Jeżeli w programie ustawiono punkty zatrzymania to można go uruchomić. Wykonuje się to poprzez polecenie **`run`**.

```
(gdb) run
Starting program: /home/juka/prog/beagle/test
Witamy w Lab PRW
Krok  0
Krok  1
Krok  2
Krok  3
Krok  4
Krok  5

Breakpoint 1, main () at test.c:10
10                  j=i+10;
```

Ekran 1-4 Wykonanie programu do punktu zatrzymania

Listowanie programu	**list [numer linii]**	**skrót**
Uruchomienie programu	**run [argumenty]**	
Ustawienie punktu zatrzymania	**break nr_linii** **break nazwa_funkcji** **break nr_linii if warunek**	**b**
Ustawienie pułapki, program zatrzyma się gdy zmieni się wartość obserwowanej zmiennej	**watch nazwa_zmiennej**	**w**
Listowanie punktów zatrzymania	**info break**	
Kasowanie punktu zatrzymania	**clear nr_linii**	
Wyprowadzenie wartości zmiennych	**print nazwa_zmienne**	**p**
Kontynuacja do następnego punktu wstrzymania	**continue**	**c**
Wykonanie następnej instrukcji, nie wchodzimy do funkcji	**next**	**n**
Wykonanie następnej instrukcji, wejście do funkcji	**step**	**s**
Uzyskanie pomocy	**help**	**h**
Ustawienie wartości zmiennej **var** na **wart**	**set var=wart**	
Zakończenie pracy programu	**quit**	**q**

Tabela 1-5 Najczęściej używane polecenia programu uruchomieniowego gdb