7. Środowisko programistyczne systemu Linux

Wstęp

W wykładzie 7 przedstawione są podstawowe informacje o środowisku programistycznym systemu Linux. Składają się na nie głównie narzędzia opracowane w ramach projektu GNU, co wynika z faktu że sam Linux jest produktem projektu GNU.

Przedstawione tu informacje mają charakter uniwersalny i mogą być stosowane przez programistów tworzących oprogramowanie dla systemu UNIX.

7.1. Narzędzia programistyczne

Linux jest blisko związany z projektem GNU. Z tego względu środowisko programistyczne składa się głównie z narzędzi stworzonych właśnie w ramach projektu GNU. Narzędzia te są standardowymi składnikami systemu Linux. Można je również instalować *on-line* za pośrednictwem Internetu, gdyż są one dostępne w postaci standardowych pakietów typu RPM lub DEB.

Tak więc, w skład środowiska programistycznego GNU wchodzą między innymi:

- generatory analizatorów składniowych i leksykalnych: **bison** i **flex,**
- kompilatory języków, np.: **C/C++**, **Java**, **Fortran**, **Common Lisp**, **Smalltalk**, **Pascal**, **Prolog**,
- środowisko do tworzenia interferjsów graficznych **GTK+**,
- narzędzia dla WWW, np. narzędzia i biblioteki do tworzenia aplikacji CGI ,
- własna wersja programu **make**,
- narzędzia do automatycznej konfiguracji pakietów **autoconf** i **automake** (radykalnie ułatwiają kompilację i instalację pakietów GNU na różnych platformach systemowych)
- program śledzący (ang. *debugger*) **gdb** oraz graficzne środowisko **DDD** ułatwiające pracę z **gdb** lub innymi programami śledzącymi.

Ponadto dostępne są pakiety nie będące częścią projektu GNU, ale dystrybuowane na zasadach licencji GPL, wśród których można znaleźć:

- kompilatory języków **Ada**, **Algol 68**, **Modula-2**,
- system zarządzania wersjami **CVS** (zastępuje podobne oprogramowanie takie jak RCS lub SCCS),
- język **Perl** (zastępuje takie narzędzia jak **sed**, **awk** i skrypty w języku powłoki).

Oprócz narzędzi z pakietów GNU dostępnych jest także szereg innych produktów, w tym także stworzonych przez światowych potentatów na rynku oprogramowania.

Narzędzia stworzone w ramach projektu GNU można oczywiście wykorzystywać w innych systemach niż Linux. Dostępne są ich teksty źródłowe, a na najbardziej popularne platformy (np. SUN Solaris dla platform SPARC lub Intel, HP-UX) gotowe do instalacji pakiety binarne. Wszystkie produkty powstające w ramach projektu GNU objęte są licencję GPL.

Autorzy tego podręcznika zalecają zapoznanie się z treścią licencji GPL każdemu, kto zamierza uczynić użytek z oprogramowania dystrybuowanego jej na warunkach. Oryginalna (w języku angielskim) treść licencji GPL jest dołączana do każdego narzędzia którego dotyczy. Można ją też znaleźć na głównym serwerze projektu GNU - **www.gnu.org**. Nieoficjalne tłumaczenie GPL na język polski można znaleźć na serwerze **www.linux.org.pl**.

7.2. Tworzenie programów

Język C jest językiem programowania ogólnego przeznaczenia, który jest szczególnie związany z systemami uniksowymi (przez "systemy uniksowe" rozumiemy tu różne warianty systemu UNIX, takie jak np. UNIX System V, SUN Solaris czy HP-UX). Wynika to z faktu, że język C został opracowany specjalnie dla potrzeb tego systemu. System operacyjny UNIX, kompilatory C i niemal wszystkie programy usługowe zostały napisane w C (lub C++). W starszych wersjach systemu UNIX (np. w SunOS 4.x) kompilator języka C standardowo wchodził w skład programów użytkowych dostarczanych przez producenta i nosił nazwę **cc**. Powodem tego była budowa systemu, która w przypadku zmiany konfiguracji jądra wymagała jego rekompilacji. Obecnie, nowoczesne wersje systemów uniksowych nie wymagają takich operacji i w związku z tym kompilator języka C nie jest standardowym elementem systemu.

Również w przypadku systemu Linux występują bardzo silne związki z językiem C/C++. Stąd też przy omawianiu środowiska programistycznego Linuksa ograniczymy się do kompilatora tego języka. Szczegółowe informacje o kompilatorach innych języków można znaleźć w dokumentacji.

Etapy kompilacji programu w języku C

Proces tworzenia binarnego kodu wykonywalnego, zwany zwyczajowo kompilacją, składa się z kilku odrębnych etapów. Są to:

prekompilacja	stworzenie ostatecznego tekstu źródłowego programu poprzez włączenie plików wskazanych dyrektywą preprocesora **#include**, wykonanie podstawie makrodefinicji **#define** i opcjonalne usunięcie komentarzy,
kompilacja właściwa	wyszukanie tokenów (słowa kluczowe, operatory) i przekształcenie ich na wewnętrzną reprezentację; reprezentacja wewnętrzna jest następnie przekształcana na kod asemblera,
optymalizacja kodu asemblera	opcjonalna modyfikacja kodu asemblera w celu zwiększenia jego efektywności (np.: zmiana sposobu obliczania adresów względnych, eliminacja nieużywanych fragmentów kodu, optymalizacja przydziału rejestrów),
asemblacja	przetworzenie kodu asemblera w relokowalny kod w języku maszynowym, który umieszczany jest w pliku obiektowym (ang. *object file*); etap ten wykonywany jest przez program **as** (systemowy) lub **gas** (z pakietu GNU),
konsolidacja	konsolidator (ang. *link editor*), dokonuje szeregu operacji w celu stworzenia pliku binarnego z kodem wykonywalnym w tym : • przeszukanie standardowego zestaw bibliotek oraz bibliotek wskazanych w linii wywołania w celu włączenia do programu kodu funkcji niezdefiniowanych w modułach stworzonych przez użytkownika, • przypisanie kodu maszynowego do ustalonych adresów, • utworzenie wykonywalnego pliku binarnego w formacie **ELF** (ang. **E**xecutable and **L**inking **F**ormat) Konsolidacja wykonywana jest przez program **ld**.

Funkcja main() w programach dla systemów uniksowych

Program wykonywany w środowisku systemu Linux/Unix otrzymuje od procesu, który go wywołał dwa zestawy danych: **argumenty** oraz **środowisko**. Dla programów stworzonych w języku C są one dostępne w postaci tablic zawierających wskaźniki, z których wszystkie oprócz ostatniego wskazują na napisy zakończone bajtami zerowymi. Ostatni wskaźnik ma zawsze wartość NULL. Dla innych języków, Pascal czy FORTRAN, przyjęto inne konwencje.

Tak więc, prototyp funkcji **main()** dla programu tworzonego dla systemu Unix jest następujący:

```
int main (int argc, char *argv[], char *envp[]);
```

Pierwszy parametr przechowuje liczbę argumentów przekazanych w wierszu poleceń, a drugi jest tablicą wskaźników do napisów będących tymi argumentami. Argumenty te mogą być całkowicie dowolnymi napisami.

Trzeci argument funkcji main() to wskaźnik na tablicę napisów tworzących środowisko. W przypadku środowiska istnieje wymaganie, aby każdy z napisów miał postać **zmienna=wartość**.

W celu uzyskania dostępu do środowiska można też wykorzystać globalną zmienną **environ**, która jest zadeklarowana w następujący sposób:

```
extern char *environ[];
```

Zmienna ta jest tablicą wskaźników do każdego elementu środowiska, czyli napisu o postaci **zmienna=wartość**. Deklaracja zmiennej **environ** zamieszczona jest w pliku nagłówkowym **<unistd.h>**. W związku z tym, programista musi jedynie włączyć ten plik do swojego programu dyrektywą #include.

Przykład

Poniżej pokazano przykład prostego programu wypisującego wszystkie napisy składające się na definicję środowiska:

```
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
 int i;
 for (i=0; environ[i]!=NULL; i++)
  printf("%s\n", environ[i]);
 exit(0);
}
```

Dostęp do środowiska możliwy jest także za pośrednictwem następujących funkcji systemowych systemu Linux:

```
const char *getenv(const char *nazwa);

int putenv(const char *napis);

int setenv(const char *nazwa, const char *wartość, int zastąpienie);
```

W przypadku funkcji **getenv()** jedynym argumentem jest nazwa zmiennej środowiska, której wartość nas interesuje. Jeśli zmienna taka istnieje to otrzymamy wskaźnik do napisu stanowiącego jej wartości. Jeśli zmienna nie istnieje to wynikiem będzie NULL.

Funkcja **putenv()** umożliwia programiście zdefiniowanie wartości zmiennej środowiska lub też zmianę wartości zmiennej już istniejącej w środowisku. Przekazywany tutaj argument jest napisem o postaci **zmienna=wartość.**

Funkcja **setenv()**, zapożyczona z systemów Unix linii BSD, służy do modyfikacji środowiska. Umożliwia ona jednak oddzielne przekazanie nazwy zmiennej środowiska oraz jej wartości, co jest wygodniejszym rozwiązaniem z punktu widzenia programisty. Ponadto trzeci argument pozwala decydować czy należy zmienić wartość już istniejącej zmiennej. Tak więc, jeśli **zastąpienie** ma wartość **0** to istniejąca zmienna nie będzie modyfikowana.

7.3. Kompilator języka C

Polecenie wywołania kompilatora ma następującą postać :

```
gcc [opcje] plik [plik]...
```

Do najczęściej używanych opcji kompilatora **gcc** należą:

-o nazwa - powoduje utworzenie programu wynikowego o nazwie podanej przez użytkownika,

-E - powoduje zatrzymanie po etapie prekompilacji, wyniki są wypisywane na ekran,

-S - powoduje zatrzymanie po etapie generowania kodu asemblera, wyniki są umieszczane w pliku z rozszerzeniem .s,

-c - powoduje zatrzymanie po etapie asemblacji, wyniki są umieszczane w pliku z rozszerzeniem .o,

-Dmakro - użycie opcji jest równoznaczne z umieszczeniem linii **#define** makro naśpoczątku pliku zawierającego tekst źródłowy,

-Umakro - użycie opcji jest równoznaczne z umieszczeniem linii **#undef** makro naśpoczątku pliku zawierającego tekst źródłowy,

-O - powoduje wykonanie optymalizacji,

-Opoziom - powoduje wykonanie bardziej złożonej optymalizacji, o zakresie wskazanym przez parametr poziom,

-g - powoduje włączanie do pliku wynikowego informacji (numery linii, typ i rozmiar identyfikatorów, tablica symboli) umożliwiających śledzenie wykonywania programu wynikowego,

-Ikatalog - powoduje włączenie katalogu **katalog** do zestawu katalogów, w których należy poszukiwać plików nagłówkowych,

-Lkatalog - powoduje włączenie katalogu **katalog** do ścieżki poszukiwań bibliotek; w linii wywołania opcja ta musi poprzedzać opcję -l (wyjaśnienie poniżej),

-lident - polecenie dla konsolidatora powodujące, że biblioteka **libident.a** jest przeszukiwana w celu znalezienia kodu funkcji zewnętrznych,

-pipe - powoduje, że kompilator gcc zamiast tworzyć pliki pośrednie do komunikacji pomiędzy programami wykonującymi kolejne etapy kompilacji wykorzystuje do tego celu łącza (ang. pipe); opcja nie działa gdy asembler nie może czytać danych ze strumienia wejściowego,

-ansi - tekst źródłowy musi być w 100% być zgodny z normą ANSI języka C.

-traditional - kompilator toleruje starsze konstrukcje języka C.

Konwencja tworzenia nazw plików

Zawartość argumentu pliki określana jest na podstawie rozszerzenia, zgodnie z następującą konwencją przedstawioną w Tabl. 7.1 (pokazano wybrane przykłady).

Tablica 7.1 Konwencja tworzenia nazw plików

Rozszerzenie	Zawartość
.c	tekst źródłowy w C
.i	kod źródłowy C po prekompilacji
.ii	kod źródłowy C++ po prekompilacji
.s	kod w języku asemblera
.h	plik nagłówkowy
.cc **.C** **.cp** **.cxx** **.cpp** **.c++**	tekst źródłowy w C++
.f **.fpp** **.FPP**	tekst źródłowy w FORTRAN-ie
.o	kod relokowalny (wynik asemblacji)
.so	biblioteka dzielona
.a	biblioteka statyczna

Program **gcc** sam rozpoznaje kod źrodłowy stworzony w języku C lub C++ i wywołuje odpowiedni kompilator. Jednak w przypadku języka C++ wskazane jest by użytkownik samodzielnie wywoływał odpowiedni kompilator, który nosi nazwę **g++**.

Przykłady użycia kompilatora języka C

Najprostszym sposobem użycia kompilatora jest wydanie polecenia kompilacji pojedynczego pliku zawierające kompletny tekst źródłowy programu, np. :

```
gcc main.c
```

Kompilator potraktuje wtedy zawartość pliku main.c jako tekst źródłowy w C (zgodnie z obowiązującą konwencją) i wykona wszystkie etapy kompilacji, aż do uzyskania kodu wykonywalnego, który zostanie umieszczony w pliku o standardowej nazwie **a.out**. Na etapie konsolidacji pod uwagę będzie wzięta tylko biblioteka **libc.a**, zawierająca kod standardowych funkcji języka C takich jak printf(), fopen(), itd. Wszystkie pliki pośrednie zostaną usunięte.

W bardziej skomplikowanym przypadku, gdy tekst źródłowy znajduje się nie w jednym lecz w kilku plikach, np.: main.c, data.c, input.c i output.c i dodatkowo programista użył funkcji matematycznych, polecenie kompilacji powinno wyglądać następująco :

```
gcc input.c output.c data.c main.c -lm
```

Kompilator dla każdego z plików wykona wszystkie etapy kompilacji, aż do uzyskania plików obiektowych main.o, data.o, input.o i output.o, a następnie dokona ich konsolidacji biorąc tym razem pod uwagę oprócz biblioteki **libc.a** także bibliotekę **libm.a** zawierającą kod funkcji takich jak sin(),

cos(), i.t.d. Program wykonywalny znajdzie się w pliku a.out. Podobnie jak w poprzednim przykładzie wszystkie pliki pośrednie oprócz plików obiektowych zostaną usunięte.

Argumentami wywołania kompilatora gcc mogą być różne typy plików i możliwa jest sytuacja, w której każdy z argumentów jest innego typu (uzyskany został przez zatrzymanie kompilacji po innym etapie), np.:

```
gcc main.o data.s input.i output.c -lm
```

Pliki, które w czasie prekompilacji mają być włączone do tekstu programu (przy pomocy dyrektywy #include) muszą znajdować się w bieżącym katalogu (dotyczy to plików nagłówkowych stworzonych przez użytkownika) lub w standardowym katalogu instalacyjnym kompilatora (standardowe pliki nagłówkowe kompilatora. np. stdio.h). Jeśli programista chce umieścić własne pliki nagłówkowe w innym katalogu niż pliki z tekstem źródłowym, to by uczynić je dostępnymi dla kompilatora, musi w linii wywołania użyć opcji **-Ikatalog**, gdzie **katalog** jest nazwą tego katalogu (względną lub bezwzględną). Przykładowo, jeśli pliki nagłówkowe zostały umieszczone w katalogu ../headers to wywołanie kompilatora ma postać:

```
gcc -I../headers input.c output.c data.c main.c
```

Podobnie jest w przypadku bibliotek. Konsolidator oczekuje, że biblioteki znajdują się w standardowym katalogu instalacyjnym. Jeśli użytkownik korzysta z innych bibliotek (np. stworzonych samodzielnie) to musi poinformować konsolidator przy pomocy opcji -Ldir, gdzie one się znajdują. Informacja ta musi poprzedzić opcję -l. Przykładowo, jeśli dodatkowa biblioteka nosi nazwę libusux.a i znajduje się w katalogu ../libs, to wywołanie kompilatora powinno mieć postać:

```
gcc main.c data.c input.c output.c -L../lib -lusux
```

Należy zwrócić uwagę na regułę nadawania bibliotekom nazw. Nakazuje ona, by nazwa bibioteki miała postać **libident.a**, gdzie pole **ident** może mieć długość od 1 do 7 znaków. W linii wywołania kompilatora, po opcji -l, podawana jest tylko część ident zamiast całej nazwy biblioteki, np.: -lm dla biblioteki libm.a.

7.4. Biblioteki

Biblioteki statyczne

Biblioteką statyczną (ang. *library archive*) jest archiwum plików obiektowych zawierające nagłówek, w którym znajdują się informacje m.in. o nazwach i położeniu wewnątrz archiwum poszczególnych obiektów oraz tablicę symboli biblioteki. Taka organizacja biblioteki zwiększa efektywność jej przeszukiwania podczas konsolidacji. Wraz z systemem dostarczane są jego standardowe biblioteki takie jak **libc.a** czy **libm.a**. Oprócz nich istnieją biblioteki zawierające zestaw funkcji przeznaczonych do specjalnych zastosowań jak np. libcurses.a umożliwiająca programiście programową obsługę terminali. Niezależnie od bibliotek dostępnych w danym systemie każdy użytkownik może tworzyć własne biblioteki, które mogą być następnie używane w taki sam sposób jak systemowe. Do tego celu używany jest program **ar**. Zasady posługiwania się tym programem są bardzo podobne do sposobu korzystania z programu tar. Wywołanie **ar** ma następującą składnię :

```
ar [opcje] archiwum plik1 plikn...
```

Argument **archiwum** jest nazwą biblioteki, a **plik1**, **plikn** są plikami obiektowymi, z których należy stworzyć bibliotekę, lub które należy wyekstrahować lub usunąć z biblioteki. Najczęściej używane opcje to:

-d - usuwanie z archiwum wskazanego plik,

-q - dodawanie pliku na koniec archiwum (UWAGA! nie sprawdza czy dany plik jest już w archiwum),

-r - zamiana (lub dodanie) wskazanego pliku w archiwum; jeśli pliku nie ma w archiwum to następuje jego dodanie, a w przypadku gdy archiwum nie istnieje - jego utworzenie,

-u - opcja używana razem z **-r** powoduje, że zamiana następuje tylko wtedy, gdy data modyfikacji pliku w archiwum jest wcześniejsza niż data modyfikacji pliku podanego jako argumentu,

-s - ponowne utworzenie tablicy symboli biblioteki; umożliwia odtworzenie tablicy symboli po jej usunięciu programem **strip**,

-t - wypisanie zawartości archiwum; zwykle używana razem z opcją **-v**,

-x - ekstrakcja wskazanego lub wszystkich plików z archiwum (nie niszczy archiwum),

-v - wyświetlanie bardziej szczegółowych informacji podczas działania.

Przykładowo, jeśli programista chce utworzyć bibliotekę libusux.a z następujących plików: funkcja1.o, funkcja2.o i funkcja3.o, to polecenie może mieć postać:

```
ar -rv libusux.a funkcja1.o funkcja1.o funkcja1.o
```

lub

```
ar -ruv libusux.a funkcja1.o funkcja1.o funkcja1.o
```

Jeśli biblioteka libusux.a nie istniała wcześniej, to w obu przypadkach zostanie utworzona. Opcja **-r** stanowi zabezpieczenie przed sytuacją, w której do istniejącej już biblioteki dopisywane byłyby za każdym razem nowe wersje plików funkcja1.o, funkcja2.o i funkcja3.o. Sprawdzenie zawartości biblioteki możliwe poprzez wydanie następujące polenia:

```
ar -tv libusux.a
```

Biblioteki dzielone

W systemie Linux oprócz bibliotek statycznych można tworzyć biblioteki dzielone (ang. *shared library*). Zastosowanie bibliotek te ma szereg zalet w porównaniu z bibliotekami statycznymi, gdyż:

1. system dzieli kod wykonywalny biblioteki pomiędzy wszystkie procesy z niej korzystające, dzięki czemu zmniejsza się zajętość pamięć systemu,
2. kod biblioteki dzielonej nie jest kopiowany do plików wykonywalnych, a więc tylko jedna kopia kodu znajduje się na dysku,
3. wykrycie błędu w bibliotece dzielonej wymaga wymiany tylko tej biblioteki bez konieczności rekompilacji wykorzystających ją programów.

Pewną wadą bibliotek dzielonych jest bardziej złożony proces ich tworzenia oraz konieczność tzw. rejestrowania biblioteki dzielonej w systemie. Ponadto, modyfikowanie biblioteki dzielonej wymaga od programisty rozwiązania problemu tzw. zarządzania zgodnością wersji. Zagadnienia te wykraczają poza zakres tego podręcznika. Szersze informacje na temat tworzenia bibliotek dzielonych można znaleźć w literaturze [2].

7.5. Program śledzący

W pakiecie GNU dostępny jest program śledzący **gdb** (nazywany też czasem uruchomieniowym). Jego wywołanie jest następujące:

```
gdb [opcje] program [core]
```

lub

```
gdb [opcje] program [process_ID]
```

Program **gdb** umożliwia krokowe wykonywanie programu, ustawianie pułapek, śledzenie wartości zmiennych i wyrażeń oraz inne, typowe dla programów uruchomieniowych, operacje. Oczywiście uruchamiany program (a dokładnie każdy plik zawierający tekst źrodłowy) musi być skompilowany w taki sposób, aby kod wykonywalny zawierał informacje niezbędne dla programu uruchomieniowego (w przypadku kompilatora języka C jest to opcja **-g**). Po wywołaniu program **gdb** komunikuje się z użytkownikiem za pomocą prostego interpretera poleceń podobnego w działaniu do np. powłoki bash. Jedną z dostępnych komend jest polecenie **help**, która pozwala uzyskać informacje o dostępnych poleceniach i ich składni w aktualnej wersji programu. Wybrany zestaw poleceń programu **gdb** zaprezentowany jest w Tabl. 7.2.

Tablica 7.2 Wybrane polecenia programu gdb

Polecenie (skrót polecenia)	Opis
attach (at)	Dołącza **gdb** do działającego procesu. Jedynym argumentem jest PID procesu, do którego chcemy się dołączyć. Polececenie powoduje zatrzymanie działającego procesu, przerywając funkcję sleep() lub dowolną inną przerywalną funkcję systemową.
backtrace (bt)	Wypisuje zawartość stosu.
break (b)	Ustawia pułapkę (*ang. breakpoint*), którą można określić podając jako argument nazwę funkcji, numer wiersza kodu w bieżącym pliku, parę nazwa_pliku:numer_wiersza lub dowolny adres komórki pamięci. Każdej pułapce jest nadawany unikalny numer referencyjny.
clear	Usuwa pułapkę. Przyjmuje takie same argumenty jak break.
condition	Zmienia pułapkę o podanym numerze referencyjnym w taki sposób, że przerwanie następuje tylko w przypadku spełnienia podanego warunku.
continue (cont)	Wznawia wykonywanie programu po zatrzymaniu na pułapce.
delete	Usuwa pułapkę o podanym numerze referencyjnym.
detach	Odłącza gdb od aktualnie przyłączonego procesu.
display	Wyświetla wartość podanego wyrażenia przy każdym zatrzymaniu działania programu. Każde wyrażenie otrzymuje unikalny numer referencyjny.
help	Pomoc. Wywołane bez argumentów podaje listę dostępnych tematów. Wywołane z argumetem (np. nazwą polecenia) podaje informacje szczegółowe (np. o danym poleceniu), np. help set.

jump	Powoduje wykonanie skoku pod podany adres. Działanie jest kontynuowane od podanego adresu. Adres może być podany jako numer wiersza lub adres komórki pamięci.
list (l)	Wywołane bez argumetów wypisuje 10 wierszy kodu źródłowego otaczających bieżący adres. Kolejne wywołania wyświetlają kolejne 10 wierszy. Podanie argumentu [nazwa_pliku:]numer_wiersza powoduje wypisanie kodu we wskazanym pliku wokół wskazanego numeru wiersza. Podanie zakresu wierszy zamiast pojedynczego numeru powoduje wypisanie tych wierszy.
next (n)	Wykonuje program do następnego wiersza kodu źródłowego bieżącej funkcji. Nie "wchodzi" do wnętrza wywoływanych funkcji.
nexti	Przechodzi do następnej instrukcji języka maszynowego. Nie "wchodzi" do wnętrza wywoływanych funkcji.
print (p)	Wypisuje wartość wyrażenia w czytelnej postaci, tzn. jeśli wskazany obiekt jest np. napisem to wypisany będzie napis, jeśli obiekt jest np. strukturą to wypisane zostaną jej pola.
run (r)	Uruchamia od początku aktualnie śledzony program. Argumenty polecenia są argumentami wywołania programu. Argumety wywołania śledzonego programu można również zdefiować przy pomocy polecenia set args.
set	Umożliwia zmianę wartości zmiennych, np. set a = 5.0, lub ustawienie argumentów wywołania programu, np. set args 1 5 3.
step (s)	Wykonuje program instrukcja po instrukcji dopóki nie osiągnie nowego wiersza w kodzie źródłowym.
stepi	Wykonuje jedną instrukcję języka maszynowego. "Wchodzi" do wnętrza wywoływanych funkcji.
undisplay	Kończy wyświetlanie wyrażenia wskazanego przez numer referencyjny. Wywołane bez argumentu powoduje zakończenie wyświetlania wartości wszystkich zdefiniowanych wyrażeń.
whatis	Wypisuje typ wyrażenia podanego jako argument.

Przykład
Praktyczne wykorzystanie programu śledzącego **gdb** zostanie zademonstrowane na przykładzie prostego programu obliczania pierwiastków równania kwadratowego. Kod źródłowy programu przedstawiamy poniżej.

Program ten należy skompilować z opcją **-g** tak, aby w pliku wynikowym zawrzeć informacje wykorzystywane przez **gdb**:

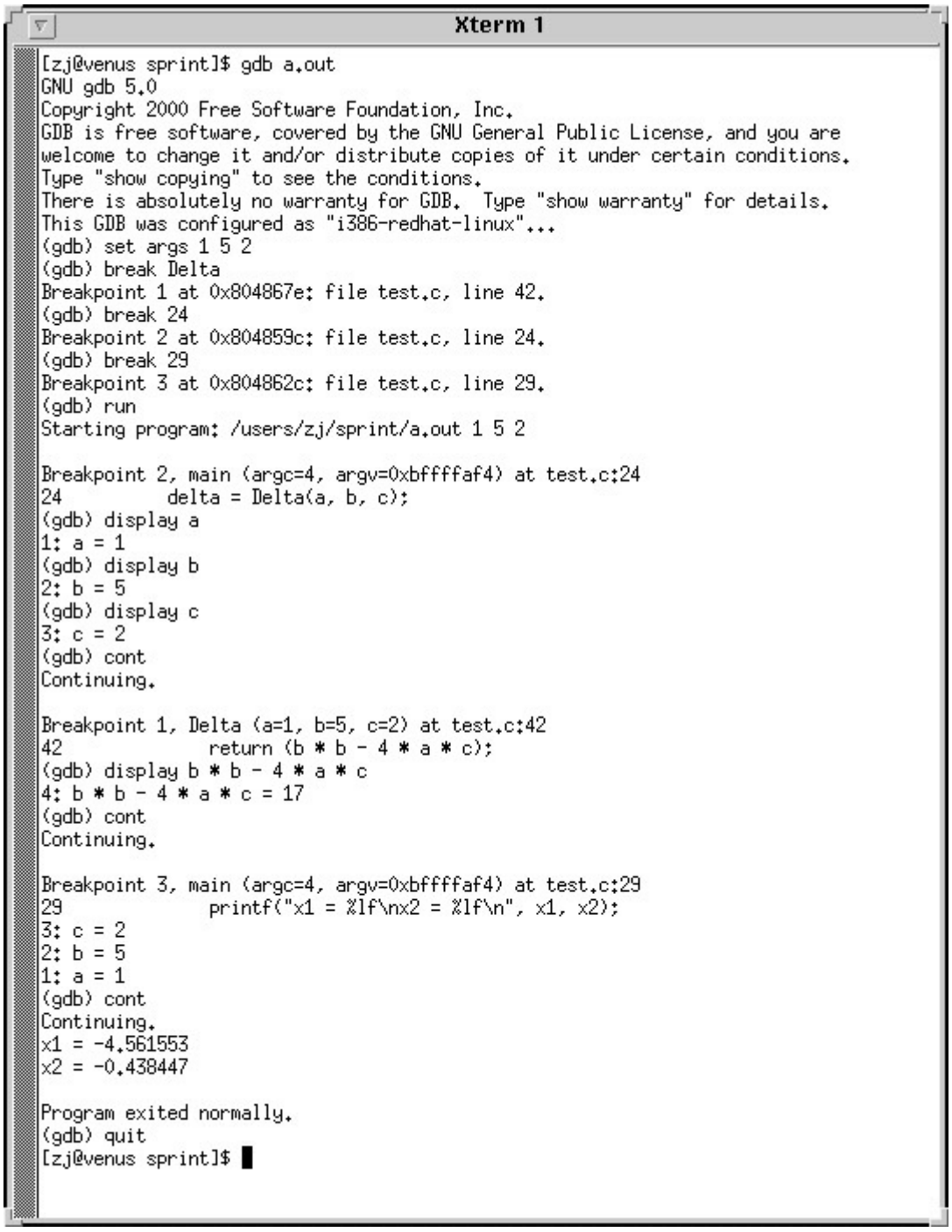
```
gcc -g rownanie.c -lm
```

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

/* prototypy*/
double Delta(double, double, double);
double Pierw(double, double, double, int);
double* PierwZesp(double, double, double, int);


void main(int argc, char* argv[])
{
  double a, b, c, delta;
  double x1, x2;

  if (argc != 4) {
    printf("Poprawna skladnia:\t%s a b c\n", argv[0]);
    exit(1);
  }
  sscanf(argv[1], "%lf", &a);
  sscanf(argv[2], "%lf", &b);
  sscanf(argv[3], "%lf", &c);
  delta = Delta(a, b, c);
  if (delta >= 0) {
    x1 = Pierw(a, b, delta, 1);
    x2 = Pierw(a, b, delta, 2);
    printf("x1 = %lf\nx2 = %lf\n", x1, x2);
  }
  else
    printf("Brak pierwiastkow rzeczywistych.\n");
  return;
  }


  double Delta(double a, double b, double c)
  {
    return (b * b - 4 * a * c);
  }

  double Pierw(double a, double b, double delta, int flaga)
  {
  if (flaga == 1)
    return (-b - sqrt(delta))/2*a;
  else if (flaga == 2)
    return (-b + sqrt(delta))/2*a;
  }
}
```

Następnie uruchamiamy program śledzący, a po pojawieniu się znaku zachęty wydajemy polecenia ustawiania pułapek, śledzenia zawartości zmiennych, itp. Przykładowa sesja pokazana jest na rys. 7.1. Jak łatwo zauważyć, użytkownik wykonał następujące czynności:

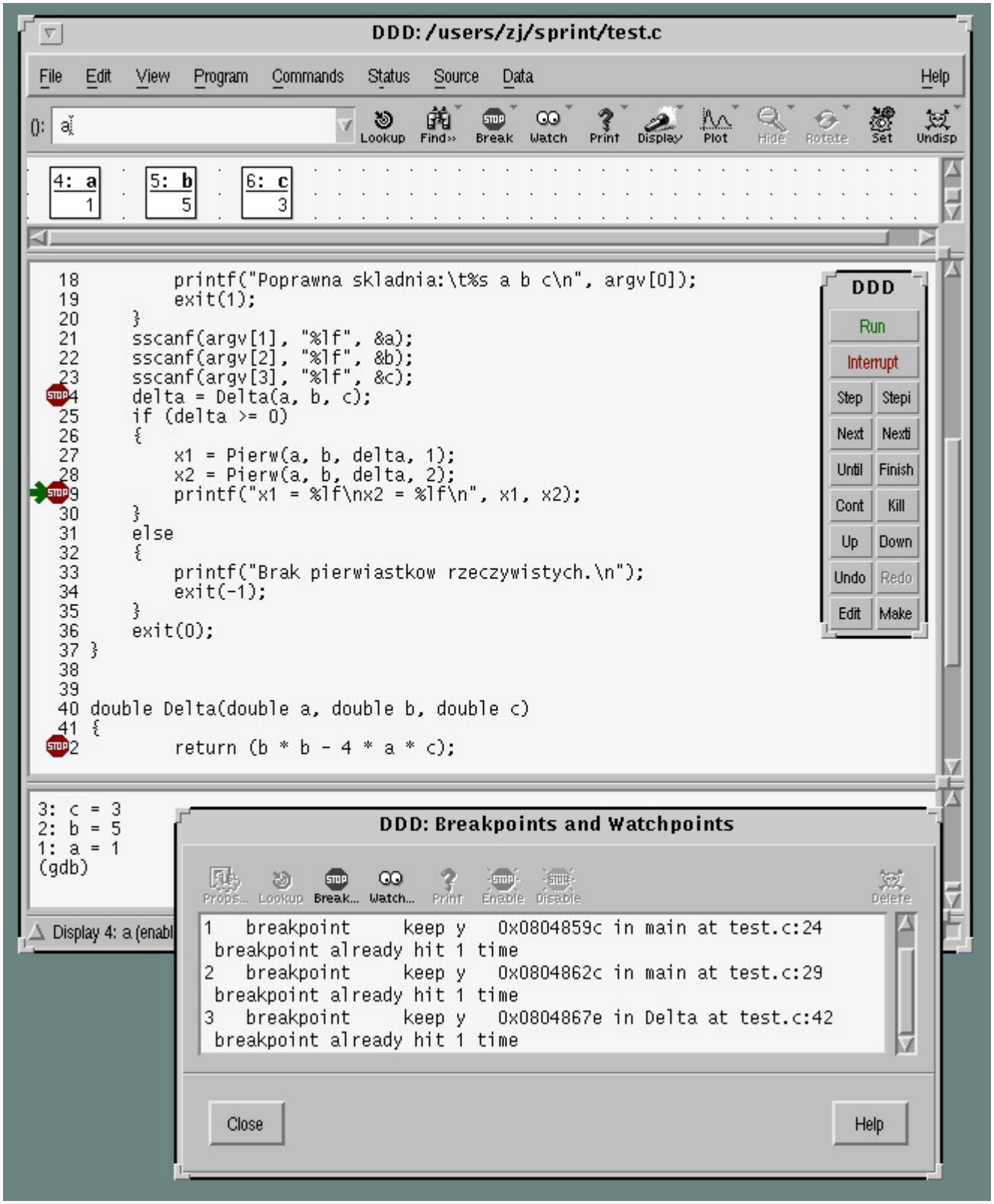
- zdefiniował argumenty wywołania programu (`set args 1 5 2`),
- ustawił pułapkę w pierwszym wierszu kodu funkcji Delta (`break Delta`),
- ustawił dwie kolejne pułapki w wierszach o numerach 24 i 29 (`break 24` i `break 29`),
- nakazał rozpoczęcie wykonywania programu (`run`),

- po zatrzymaniu programu na pierwszej pułapce ustawił śledzenie wartości zmiennych a, b i c (`display a, display b, display c`),
- nakazał wznowienie wykonywania programu (`cont`),
- po zatrzymaniu na następnej pułapce wewnątrz funkcji Delta ustawił śledzenie wyrażenia "b*b - 4*a*c" (`display b*b - 4*a*c`),
- dwukrotnie nakazał wznowienie wykonywania programu (`cont`),
- zakończył sesję wydając polecenie `quit`.



Rys. 7.1 Przykład zastosowania programu uruchomieniowego gdb

Śledzenie i wyszukiwanie błędów w złożonym programie przy pomocy samego tylko programu **gdb** nie jest zbyt wygodne ze względu na brak graficznego interfejsu użytkownika jaki zwykle posiadają zintegrowane środowiska dostarczane na platformy MS Windows czy też komercyjne oprogramowanie dla systemów typu Linux/Unix. Interfejsu takiego dostarczają oddzielne programy np. **xxgdb** lub program o nazwie **DDD** (produkt z pakietu GNU). Na rys. 7.2 pokazano przykładowe wykorzystanie środowiska **DDD**.



Rys. 7.2 Przykład zastosowania środowiska graficznego DDD do programu uruchomieniowego gdb

7.6. Program make

Program **make** jest standardowym narzędziem dostępnym w środowisku systemu Linux/Unix ułatwiającym programiście pracę nad tworzeniem programu. Podstawową funkcją programu **make** jest zarządzanie kompilacją zbioru tekstów źródłowych składających się na dany program przy zachowaniu minimalnego kosztu operacji. Oznacza to, że po zmodyfikowaniu części tekstów źródłowych, kompilacji zostaną poddane wyłącznie te moduły, które są od nich uzależnione. O tym, jakie polecenia należy wykonać, aby z tekstów źródłowych otrzymać programy wykonywalne decydują stworzone przez użytkownika relacje zależności. Relacje te umieszcza się w pliku sterującym (*ang. makefile*). Plik sterujący powinien znajdować się w katalogu, który zawiera teksty źródłowe danego programu. Uruchomienie procesu kompilacji następuje poprzez wywołanie programu **make**. Program odczytuje wtedy całą zawartość pliku sterującego i na podstawie jego treści, jak również na podstawie wbudowanych reguł transformacji, treści linii wywołania, wartości zmiennych środowiska, czasu systemowego oraz czasu modyfikacji plików tworzy wynikowy ciąg poleceń prowadzący do uzyskania żądanego celu. Następnie polecenia te są wykonane, przy czym wykonanie każdego z nich poprzedzone jest wypisaniem jego treści.

Parametry wywołania

Wywołanie programu **make** ma następującą postać:

```
make [opcje] [makrodefinicje] [-f plik_sterujący] [cel]
```

Najczęściej stosowane opcje to:

-d - włącza tryb szczegółowego śledzenia,

-f plik_sterujący - umożliwia stosowanie innych niż standardowe nazw plików sterujących,

-n - powoduje wypisanie poleceń na ekran zamiast ich wykonania,

-p - powoduje wypisanie makrodefinicji i reguł transformacji,

-s - wyłącza wypisywanie treści polecenia przed jego wykonaniem,

-i - powoduje ignorowanie błędów kompilacji (stosować z ostrożnością!).

W najprostszym przypadku linia polecenia zawiera tylko słowo **make**. Program próbuje wtedy odczytać polecenia z pliku sterującego o jednej z nazw: **makefile**, **Makefile** lub **MakeFile**. Jeśli w bieżącym katalogu nie ma pliku o takiej nazwie to zgłaszany jest błąd. Jeśli plik sterujący nosi inną nazwę to należy użyć wywołania **make -f plik_sterujący**. W linii wywołania można zdefiniować nowe lub zmienić istniejące już makrodefinicje, np.: *make "CC=gcc"*. Można też polecić osiągnięcie innego celu niż domyślny, np.: *make all, make clean lub make install*.

Plik sterujący

Plik sterujący zawiera definicje **relacji zależności**, które mówią w jaki sposób i z jakich elementów należy stworzyć cel (program, bibliotekę, lub plik obiektowy) i wskazują pliki, których zmiany implikują wykonanie powtórnej kompilacji poszczególnych celów. Plik sterujący może również zawierać zdefiniowane przez programistę **reguły transformacji**.

Relacje zależności

Program make, stosując **wbudowane reguły transformacji**, potrafi samodzielnie wykonać proste sekwencje poleceń, ale potrzebuje wskazówek programisty by utworzyć bardziej skomplikowane cele, takie jak program wykonywalny. Programista dokonuje tego poprzez umieszczenie w pliku sterującym definicji określających, z jakich elementów należy tworzyć program wynikowy i jak te

elementy zależą od innych obiektów np. plików nagłówkowych. Wszystkie relacje zależności pomiędzy obiektami opierają się na porównywaniu czasu ostatniej modyfikacji plików oraz na sprawdzaniu czy dane pliki istnieją. Ogólna postać definicji, jaką można umieścić w pliku sterującym, jest następująca:

```
cel1 [cel2...] :[:] [lista_obiektów_odniesienia]
[<TAB> polecenia] [#komentarz]
[<TAB> polecenia] [#komentarz]
```

gdzie <TAB> oznacza znak tabulacji .

Każde polecenie umieszczone w definicji wykonywane jest w oddzielnej kopii powłoki. Standardowo jest to powłoka **sh**. Jeśli użytkownik chce wywoływać inny rodzaj powłoki, to musi przedefiniować zmienną SHELL. Złożone polecenia powinny być zawarte w pojedynczym wierszu, gdyż tylko wtedy wykonane będą przez tą samą kopię powłoki. Elementarne składniki polecenia należy umieścić obok siebie, oddzielając je średnikami, a jeśli cały wiersz jest zbyt długi, to należy rozbić go na kilka stosując znak kontynuacji- ****, jak pokazano na poniższych przykładach:

```
prog : source/main.o source/input.o source/output.o
     cd source; $(CC) -o prog main.o input.o output.o

prog : source/main.o source/input.o source/output.o
    cd source; \
    $(CC) -o prog main.o input.o output.o
```

Jeśli użytkownik chce zapobiec wypisywaniu treści polecenia podczas jego wykonywania to może umieścić @ jako pierwszy znak w poleceniu.Poniższy przykład pokazuje typową definicję relacji zależności:

```
prog : main.o input.o output.o
      $(CC) -o prog main.o input.o output.o
```

Mówi ona, że cel prog zależy od trzech obiektów odniesienia: main.o, input.o, output.o. Jeśli którykolwiek z tych obiektów ulegnie zmianie (tzn. zostanie na nowo skompilowany), to cel prog należy utworzyć ponownie. Jeżeli obiekt odniesienia nie istnieje,to zostanie stworzony przy zastosowaniu wbudowanej reguły transformacji (lub innej definicji podanej przez programistę w pliku sterującym).W drugim wierszu definicji umieszczone jest polecenie, które należy wykonać by zbudować cel prog. W tym przypadku, należy wywołać kompilator z opcją zmieniającą nazwę pliku wynikowego na nazwę celu i wykonać konsolidację plików main.o, input.o, output.o z biblioteką standardową.
Programista nie musi umieszczać w pliku sterującym instrukcji powodujących ponowne tworzenie plików obiektowych w przypadku zmiany tekstów źródłowych, gdyż jest to wykonywane automatycznie. Musi natomiast poinformować program make o wszystkich plikach włączanych do tekstów źródłowych dyrektywą #include, aby make mógł zareagować na zmiany zawartości tych plików (wyjątkiem od tej zasady są standardowe pliki nagłówkowe kompilatora, które nigdy nie ulegają zmianom). Informacja ta podawana jest w następującej postaci:

```
main.o : global.h
main.o input.o : czytaj.h
output.o main.o : global.h data.h matrix.h
```

Definicje te mówią, że w przypadku zmiany jakiegokolwiek pliku (plików) po prawej stronie "**:**" należy od nowa stworzyć plik (pliki) wymienione po lewej stronie "**:**", stosując wbudowane reguły transformacji. Program make inaczej interpretuje definicje zawierające separatory "**:**" oraz "**::**". W przypadku definicji zawierającej "**:**" cel jest budowany jeśli:

1. dowolny obiekt odniesienia jest "młodszy" od celu,
2. cel nie istnieje.

Poniższy przykład, pokazuje użycie separatora"**::**". Dwie definicje celu **a** umieszczone w jednym pliku sterującym możliwiają wykonania kompilacji "warunkowej":

```
a :: a.sh
      cp a.sh a
a :: a.c
      cc -o a a.c
```

Definicja pierwsza aktywna jest, gdy w bieżącym katalogu znajduje się plik a.sh, a druga jeśli plik a.c. W przypadku, gdy istnieją oba pliki, wykonana może być zarówno jedna z definicji lub też obie zależnie od tego czy cel jest "starszy" od a.sh czy też od a.c, czy jednocześnie od a.sh i a.c. W następnym przykładzie użycia definicji z "**::**" cel ma nazwę pokrywającą się z nazwą katalogu zawierającego jego teksty źródłowe (sytuacja często spotykana w praktyce). Gdyby użyto definicji z pojedynczym "**::**" to cel nigdy nie zostałby utworzony, gdyż istnieje już w bieżącym katalogu obiekt o tej samej nazwie (to, że jest katalogiem nie ma znaczenia dla programu make).

```
program ::
      cd program; cc program.o -o program
```

Kolejność umieszczenia definicji w pliku sterującym nie wpływa na kolejność wykonywania poleceń przez program make. To, jakie czynności i w jakiej kolejność należy wykonać określa sam program make na podstawie wewnętrznej struktury danych (utworzonej w wyniku analizy całej treści pliku sterującego) i czasu ostatniej modyfikacji plików. Wyjątkiem od tej zasady jest określenie, który z celów zdefiniowanych w pliku sterującym będzie realizowany domyślnie, czyli po uruchomieniu programu make bez podania nazwy celu.

Za cel domyślny przyjmowany jest pierwszy cel zdefiniowany w pliku sterującym.

Makrodefinicje

Makrodefinicja jest to zmienna używana w celu sparametryzowania reguł, dzięki czemu stają się one bardziej przejrzyste i łatwiejsze do modyfikacji. Deklaracja makrodefinicji to linia zawierająca znak równości i nie zaczynająca się od kropki ani tabulatora, np.:

```
OBJECTS = main.o data.o input.o output.o
HDRS = ../headers ../includes
CFLAGS= -g -DAUX -I$(HDRS)
CC = gcc
LIBS = -lusux
```

W celu odwołania się do zawartości makrodefinicji używa się konstrukcji **$(nazwa_makrodef)** lub **${nazwa_makrodef}**. Jeśli nazwa makrodefinicji zawiera tylko jeden znak to nawiasy można pominąć. Stosowanie makrodefinicji daje możliwość łatwego wprowadzania modyfikacji takich jak zmiana opcji kompilatora, a także przystosowania pliku sterującego do nowego środowiska, np.: zmiana ścieżek dostępu do plików nagłówkowych. Istnieje zestaw predefiniowanych makrodefinicji, do których programista może się odwoływać w pliku sterującym. Wartości tych makrodefinicji mogą być zmienione, przy czym nowe wartości widoczne są tylko w konkretnym pliku sterującym. Niektóre z nich zostały pokazane w tablicy 7.2.

Tablica 7.3 Predefiniowane makrodefinicje

Makrodefinicja	Wartość predefiniowana	Objaśnienie
AR	ar	program zarządzający bibliotekami
AS	as	asembler
ASFLAGS		opcje programu AS
CC	cc	kompilator języka C
CFLAGS		opcje programu CC
LD	ld	konsolidator
LDFLAGS		opcje programu LD

Makrodefinicje dynamiczne

Istnieje również zbiór wbudowanych makrodefinicji, których wartości są określane w trakcie wykonywania poleceń zapisanych w definicjach (tzw. makrodefinicje dynamiczne). Stosowanie tych makrodefinicji znacznie ułatwia tworzenie definicji i reguł transformacji.W definicjach można stosować następujące makra :

$@** - aktualnie tworzony cel. W poniższym przykładzie **$@ przyjmuje wartość **prog**:

```
prog : main.o input.o output.o
	$(CC) -o $@ main.o input.o output.o
```

$?** - lista nieaktualnych obiektów odniesienia w stosunku do bieżącego celu. W poniższym przykładzie **$? jest listą tych plików obiektowych spośród wszystkich z $(LIB_OBJ), które zostały zmodyfikowane po ostatnim utworzeniu biblioteki libusux.a:

```
libusux.a : $(LIB_OBJ)
	$(AR) -rv $@ $?
```

$ < - nieaktualny obiekt odniesienia powodujący wywołanie reguły, np. plik źródłowy wywołujący wbudowaną regułę transformacji .c.o:

```
.c.o:
	$(CC) $(CFLAGS) -c $(<)
```

$ * - wspólny prefix celu i obiektu odniesienia, np.:

```
c.o:
	$(CC) $(CFLAGS) -c $*.c
```

Do powyższych makrodefinicji można zastosować modyfikatory **D** (ang. directory) oraz **F** (ang. file), umożliwiające odzyskanie z nazwy celu części opisującej katalog i części będącej właściwą nazwą, np:

$(@D) - część "katalogowa" nazwy bieżącego celu

$(@F) - część "plikowa" nazwy bieżącego celu

Istnieją również makrodefinicje, które umieszczać wolno wyłącznie na liście obiektów odniesienia (po ":" lub "::"). Są to :

$$@ - kolejny cel (obiekt z lewej strony znaku :), np.:

```
prog1 prog2 prog3 prog4 : $$@.c
     $(CC) -o $@ $?
```

Definicja ta powoduje wykonanie dla każdego z celów prog1, prog2, prog3, prog4 następującej sekwencji operacji :

1. ustawienie bieżącego celu na prog*n,*
2. ustawienie bieżącego obiektu odniesienia na prog*n*.c,
3. wywołanie kompilatora jeśli prog*n*.c jest "młodszy" niż prog*n* i utworzenie celu prog*n*.

W chwili wywołania program make odczytuje zawartość zmiennych środowiska i dodaje je do zbioru makrodefinicji. Jeżeli nazwy makrodefinicji użytych wewnątrz pliku sterującego pokrywają się z nazwami użytymi w linii wywołania i/lub z nazwami zmiennych środowiska, to ostateczna wartość makrodefinicji wynika z następującej hierarchii (w kolejności od najwyższego do najniższego priorytetu):

1. wartość nadana w linii wywołania,
2. wartość zdefiniowana w pliku sterującym,
3. wartość zmiennej środowiska,
4. predefiniowana wartość makrodefinicji.

Reguły transformacji

Program make posiada wewnętrzną tablicę **wbudowanych reguł transformacji**, które są wykorzystywane podczas kompilacji. Reguły te określają w jaki sposób z plików zawierających teksty źródłowe utworzyć pliki obiektowe lub programy wynikowe. Umożliwiające programowi make zarządzanie kompilacją tekstów źródłowych np. automatyczne tworzenie plików obiektowych z plików zawierających teksty źrodłowe w C, assemblerze, itp. Typy plików określane są na podstawie przyrostków (rozszerzeń nazw) w sposób opisany w segmencie 2 lekcji 5 (opis kompilatora gcc).

Programista może umieścić w pliku sterującym własne reguły transformacji. Reguły zdefiniowane w pliku sterującym przysłaniają reguły wbudowane.

Istnieją dwa typy **reguł transformacji**:

1. dwuprzyrostkowe
2. jednoprzyrostkowe.

Reguły transformacji dwuprzyrostkowe definiują sekwencję poleceń jakie trzeba wykonać, aby z pliku o typie określonym przez pierwszy przyrostek uzyskać plik o typie określonym przez drugi przyrostek. Typowym przykładem jest reguła definiująca w jaki sposób uzyskać plik obiektowy z pliku zawierającego kod źródłowy w C:

```
.c.o:
      $(CC) $(CFLAGS) -c $(<)
```

Reguła ta mówi, że należy wywołać kompilator języka C z opcją zatrzymującą kompilację po etapie asemblacji, podając jako argument plik z tekstem źródłowym w C. Podobną postać mają reguły określające transformacje plików zawierających teksty źródłowe dla asemblera :

```
.s.o:
      $(AS) $(ASFLAGS) -o $(@) $(<)
```

Reguły transformacji jednoprzyrostkowe określają sekwencję poleceń jakie należy wykonać, aby z pliku o typie określonym przez przyrostek uzyskać plik bez przyrostka czyli np.: program wynikowy. Typowy przykład to reguła definiująca sposób uzyskania programu, przyjmując za wejście plik z tekstem źródłowym w C :

```
.c:
	$(CC) $(CFLAGS) $(LDFLAGS) -o $(@) $(<)
```

Predefiniowane cele

Predefiniowane cele są w rzeczywistości wbudowanymi regułami, które są uaktywniane poprzez włączenie ich do pliku sterującego. Włączenie ich modyfikuje standardowy sposób działania programu make. Cele te są przedstawione w Tabl. 7.3.

Tablica 7.4 Predefiniowane cele

Cel	Objaśnienie
.DEFAULT:	Umieszczenie celu powoduje, że w przypadku, gdy brak jest definicji opisującej sposób zbudowania celu (a cel musi być osiągnięty) to stosowane są polecenia umieszczone w definicji celu .DEFAULT.
.IGNORE:	Umieszczenie tego celu jest równoznaczne z wywołaniem programu make z opcją -i.
.MAKESTOP [n]:	Umieszczenie tego celu powoduje zignorowanie całej treści pliku sterującego. Opcjonalnie można podać kod zakończenia (argument n), który standardowo ma wartość 0. Użycie celu .MAKESTOP umożliwia zaniechanie wykonywania wybranego pliku sterującego w celu szybkiego przejścia przez wielopoziomową strukturę wywołań programu make.
.PRECIOUS:	Umieszczenie tego celu zmienia standardowe zachowanie programu make w sytuacji przerwania jego pracy, powoduje zaniechanie usuwania utworzonych do tego czasu programów i plików obiektowych.
.SILENT:	Umieszczenie tego celu jest równoznaczne z wywołaniem programu make z opcją -s.

Instrukcja include

W pliku sterującym można umieścić instrukcję **include** (lub **Include**) o następującej składni:

```
include nazwa_pliku
```

Słowo **include** musi być umieszczone na początku linii zaczynając od pierwszej kolumny. Plik wskazany przez instrukcję **include** jest włączany do treści bieżącego pliku sterującego, po etapie wykonania podstawień makrodefinicji w pliku bieżącym. Jeśli nie jest możliwe odczytanie zawartości wskazanego pliku to program make przerywa pracę.

Bibliografia

[1] Glass G., Ables K.: Linux dla programistów i użytkowników, Wydawnictwo Helion 2007 (rozdział 11)

[2] Johnson M.K., Troan E.W.: Oprogramowanie użytkowe w systemie Linux, WNT 2000

[3] Kernighan B.W., Ritchie D.M.: Język ANSI C, WNT 1994,

Nowe pojęcia, definicje i wyrażenia

Termin	Objaśnienie
biblioteka dzielona	umożliwia dynamiczną konsolidację programu wykonywalnego, kod biblioteki jest dzielony pomiędzy wszystkie procesy z niej korzystające
biblioteka statyczna	archiwum plików obiektowych używanych podczas statycznej konsolidacji programu wykonywalnego; kod zawarty w bibliotekach statycznych jest dołączany do każdego programu wykonywalnego oddzielnie
makrodefinicja	zmienna używana w plikach sterujących programu **make** w celu sparametryzowania relacji zależności i reguł transformacji, dzięki czemu stają się one bardziej przejrzyste i łatwiejsze do modyfikacji; makrodefinicje mogą być predefiniowane (np. CC przechowująca nazwę kompilatora) lub też mogą być tworzone przez programistę (np. OBJECTS przechowująca listę plików obiektowych tworzących program wynikowy lub bibliotekę)
reguła transformacji dwuprzyrostkowa	definiuje sekwencję poleceń jakie program **make** musi wykonać, aby z pliku o typie określonym przez pierwszy przyrostek uzyskać plik o typie określonym przez drugi przyrostek (np. reguła .c.o definiuje sposób tworzenia pliku obiektowego z pliku zawierającego tekst źródłowy w C)
reguła transformacji jednoprzyrostkowa	definiuje sekwencję poleceń jakie program **make** musi wykonać, aby z pliku o typie określonym przez przyrostek uzyskać plik bez przyrostka czyli np.: program wynikowy
reguła transformacji wbudowana	predefiniowana reguła transformacji programu **make** (jedno- lub dwuprzyrostkowa); zestaw reguł wbudowanych umożliwia programowi make zarządzanie kompilacją tekstów źródłowych; reguły wbudowane mogą zostać przedefiniowane przez programistę w pliku sterującym programu **make**
relacja zależności	określa z jakich elementów i w jaki sposób program **make** będzie tworzyć program wynikowy i jak te elementy zależą od innych obiektów np. plików nagłówkowych

Zadania do wykładu 7

Zadanie 1

Rozpakuj archiwum **przyklad.tar.gz** i obejrzyj zawarte w nim teksty źródłowe.

Wykorzystując kompilator gcc wykonaj następujące czynności:

1. przenieś prywatne pliki nagłówkowe (*.h) do innego katalogu niż pliki źródłowe *.c i następnie skompiluj wszystkie pliki źródłowe i utwórz z nich program wynikowy o nazwie **prog**,
2. wykorzystując odpowiednie opcje zatrzymaj kompilację na wcześniejszych etapach i obejrzyj pliki pośrednie stworzone przez kompilator,
3. porównaj rozmiar pliku wynikowego otrzymanego po kompilacji z włączoną i wyłączoną optymalizacją,
4. znajdź w kodzie źródłowym makro sterujące procesem prekompilacji i wykorzystując odpowiednią opcję programu **gcc** wykonaj punkt 1 w dwóch wersjach.

Zadanie 2

Posługując się programem ar wykonać operacje:

1. zbudować własną bibliotekę statyczną **libsop.a** z wybranych plików obiektowych,
2. wykorzystać stworzoną bibliotekę do utworzenia programu z zadania 1.

Zadanie 3

Przy pomocy programu **nm** obejrzyj tablicę symboli programu **prog** i biblioteki **libsop.a**. Usuń tablice symboli z obu tych plików (poleceniem **strip**) i spróbuj ponownie utworzyć program (patrz punkt 2. z zadania 2).Wyjaśnić ewentualne różnice w działaniu kompilatora.

Zadanie 4

Używając programów **gcc** i **gdb** wykonaj następujące polecenia:

1. utwórz program **prog** w taki sposób aby umożliwić śledzenie jego pracy za pomocą **gdb**,
2. obejrzyj kod źródłowy przy pomocy gdb, wybierz miejsca i ustaw kilku pułapek,
3. używając odpowieniego polecenia programu **gdb** podaj argumenty wywołania programu,
4. ustaw tryb śledzenia dla wybranej zmiennej,
5. uruchom program,
6. w trakcie krokowego wykonywania programu zmień wartość zmiennej z punktu 4.

Zapoznaj się z obsługą programu DDD (o ile jest zainstalowany). Wykorzystując środowisko graficzne należy powtórzyć powyższe czynności.