Tworzenie oprogramowania

- Język C
- Budowa programu napisanego w języku C
  - podział na pliki z definicjami funkcji,
  - korzystanie z bibliotek systemowych i własnych
  - automatyzacja kompilacji za pomocą make
  - dzielenie projektu na części: katalogi zawierające pliki źródłowe, pliki nagłówkowe, pliki wykonywalne, pliki biblioteczne
- Narzędzia programisty w środowisku GNU/Linux
  - edytory (vi, vim, mcedit, emacs,kwrite, kate, anjuta, ...)
  - kompilator: gcc
  - inne użyteczne narzędzia:
    - przeszukiwanie dokumentacji (man, apropos, whatis, info, pinfo, GNOME yelp, ...)
    - wyszukiwanie tekstu (grep i wyrażenia regularne)
    - porównywanie plików (diff, patch)
    - automatyczne tworzenie dokumentacji: (doxygene, upiększcze kodu – indent, ...)
    - debuger (gdb – tekstowy, trzeba odpowiednio skompilować program, graficzne: kdbg, ddd, ...)

Podręcznik systemowy – polecenie man

- Sekcje
  - 2 – system calls
  - 3 – libraries
  - 4 – special, device, files (np. `/dev/random`, `/dev/urandom`)
  - 5 – file formats

- Użyteczne polecenia

```
$ whereis printf
printf: /usr/bin/printf /usr/include/printf.h
/usr/share/man/man1/printf.1.gz /usr/share/man/man3/printf.3.gz
/usr/share/man/man1p/printf.1p.gz /usr/share/man/man3p/printf.3p.gz

$ man 3 printf
PRINTF(3)   Linux Programmer's Manual                          PRINTF(3)

NAME
printf,  fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf
- formatted output conversion

SYNOPSIS
  #include <stdio.h>

  int printf(const char *format, ...);
  int fprintf(FILE *stream, const char *format, ...);
  int sprintf(char *str, const char *format, ...);
  int snprintf(char *str, size_t size, const char *format, ...);
...

DESCRIPTION
The functions in the printf family produce output according to a format  as
described  below. The  functions  printf and vprintf write output to
stdout, the standard output stream; fprintf and vfprintf write output  to
the  given  output  stream;  sprintf,  snprintf,  vsprintf  and vsnprintf
write to the character string str.

$ whatis printf
printf               (1)  - format and print data
printf               (3)  - formatted output conversion
printf [builtins]    (1)  - bash built-in commands, see bash(1)

$ apropos printf
fprintf [printf]     (3)  - formatted output conversion
printf               (1)  - format and print data
printf               (3)  - formatted output conversion
printf [builtins]    (1)  - bash built-in commands, see bash(1)
snprintf [printf]    (3)  - formatted output conversion
sprintf [printf]     (3)  - formatted output conversion
...
$ info printf

$ info --apropos printf
```

Kompilacja programu

- Kompilator **`gcc`**

  Przykład wywołania:

  ```
  $ gcc file1.c file2.c file3.c –o prog
  ```

- Pełny opis opcji: `man gcc`
- Ważniejsze opcje kompilatora gcc

  - `-o filename` : określa nazwę pliku wynikowego, zawierającego program gotowy do wykonania
  - `-c` : kompiluj tylko do postaci obiektów (.o), nie łącz
  - `-D symbol` : definiuj symbol w preprocesorze
  - `-I path` : gdzie szukać plików nagłówkowych (domyślnie – aktualny katalog i katalogi standardowych bibliotek)
  - `-L path` : określa dodatkowy katalog przeglądany w poszukiwaniu bibliotek
  - `-l libname` : określa dołączane biblioteki (np. `–lm` oznacza bibliotekę `libm.so` lub `libm.a`, w takiej kolejności)
  - `-static` : dołączaj tylko biblioteki statyczne
  - `-O n` : określa stopień optymalizacji kodu wynikowego
  - `-pedantic` : : wyświetl wszystkie ostrzeżenia i błędy wymagane przez standard ANSI
  - `-Wall` : włącz wszystkie użyteczne ostrzeżenia
  - `-v` : wyświetl komunikaty o przebiegu kompilacji
  - `-std` : określa używany standard języka C (np. C99)

Kompilacja programu za pomocą `make`

- Zasada: plik Makefile zawiera cele (ang. *targets*), które chcemy osiągnąć i reguły (ang. *rules*) wyjaśniające jak to zrobić. Definiuje się również zależności (ang. *dependents*, *prerequisities*), określające kiedy należy ponownie zbudować dany cel.

- Przykład 1

```
$ cat p1.c
#include <stdio.h>
int main()
{
  printf("Witaj\n");
  return 0;
}

$ cat Makefile
p1:     p1.c
        gcc -Wall p1.c -o p1

$ make
gcc -Wall p1.c -o p1

$ ./p1
Witaj
```

- Przykład 2

```
CC=gcc
CFLAGS=-Wall -O2
SOURCES=p1.c
OUTFILE=p1
all:
        $(CC) $(CFLAGS) $(SOURCES) -o $(OUTFILE)
```

- Przykład 3

```
CC=gcc
CFLAGS=-Wall -O2 -lm
SOURCES=p1.c funkcje.c
OUTFILE=p1
all:
        $(CC) $(CFLAGS) $(SOURCES) -o $(OUTFILE)
```

- Przykład 4

```
CC=gcc
CFLAGS=-Wall -O2
SOURCES=prog1.c mylib1.c
OUTFILE=prog1
SRCDIR=./src
INCDIR=./include
LDFLAGS= -lm

all:
   $(CC) $(CFLAGS) $(SRCDIR)/$(SOURCES) $(LDFLAGS) -I$(INCDIR) \
      -o $(OUTFILE)
```

- Przykład 5

```
$ ls -R .
.:
include  Makefile  src

./include:
func.h

./src:
func.c  main.c

$ cat Makefile
CC=gcc
CFLAGS=-Wall -O2
SOURCES=main.c func.c
#to po prostu main.o func.o, tylko krócej
OBJECTS=$(SOURCES:.c=.o)
OUTFILE=prog1
SRCDIR=./src
INCDIR=./include
LDFLAGS= -lm

all: $(OBJECTS)
        $(CC) $(LDFLAGS) -o $(OUTFILE) $(OBJECTS)
main.o:
        $(CC) -c $(CFLAGS) -I$(INCDIR) $(SRCDIR)/main.c
func.o:
        $(CC) -c $(CFLAGS) -I$(INCDIR) $(SRCDIR)/func.c
clean:
        rm -f $(OBJECTS) $(OUTFILE)

$ make
gcc -c -Wall -O2 -I./include ./src/main.c
gcc -c -Wall -O2 -I./include ./src/func.c
gcc -lm  -o prog1 main.o func.o


$ make clean
rm -f main.o func.o     prog1
```

Biblioteki

- Zbiór wstępnie skompilowanych funkcji przechowywanych w postaci plików binarnych
- Ogólny cel: wielokrotne wykorzystywanie kodu
- Standardowe biblioteki przechowywane są zazwyczaj w `/lib` i `/usr/lib`

```
$ ls -c /lib | more
      libnss_winbind.so.2
      libnss_wins.so.2
      libblkid.so.1
      libblkid.so.1.0
      libcom_err.so.2
      libcom_err.so.2.1
      ...

$ ls -c /usr/lib
      libecpg.a
      libecpg.so
      libfwcompiler.so.6.3
      librpm-4.3.so
      libselinux.so
      libbind.so.3.0.8
      libfwbuilder.so.6.3
      ...
```

- Rodzaje bibliotek:
  - biblioteka statyczna - zbiór plików obiektowych umieszczonych w jednym archiwum; linker przeszukuje archiwum i dołącza pliki obiektów do wynikowego programu
  - biblioteka współdzielona – pliki obiektów są połączone w jeden plik, współużywany przez wiele programów; zmiana biblioteki nie wymaga ponownej kompilacji każdego programu, który z niej korzysta; większa złożoność; możliwość kolizji wersji; może być dynamicznie ładowana w czasie wykonywania programu – w programie należy to zaznaczyć za pomocą odpowiednich funkcji (`dlopen()`,...)

- Nazwa biblioteki:
  - rozpoczyna się od `lib`
  - kończy się rozszerzeniem wskazującym rodzaj biblioteki
  - .a – statyczna
  - .so – współdzielona

- Korzystanie z biblioteki zainstalowanej w systemie:

  ```
  gcc prog.c –o prog –lm
  ```

  dołączy:

  ```
  /usr/lib/libm.so lub  /usr/lib/libm.a
  ```

Domyślnie dołączona zostanie biblioteka współdzielona. Jeśli istnieją obydwie biblioteki i chcemy dołączyć bibliotekę statyczną, należy użyć opcji `–static`. Przykład:

```
gcc –static prog.c –o prog –lm
```

Tworzenie biblioteki statycznej

- Kompilacja i dołączenie do biblioteki

```
gcc –c file1.c file2.c file3.c
ar cru moja_biblioteka.a file1.o file2.o file3.o
```

o Biblioteka statyczna nazywana jest również archiwum.

Opcje:

r	dodaj plik do archiwum
c	utwórz archiwum, jeśli nie istnieje
ru	zastąp poprzednią wersję w archiwum jeśli jest starsza (*replace*)
s	zapisz indeks plików obiektowych do archiwum

- Przykład:

```
$ cat libhello.c
      #include <stdio.h>
      void print_hello(void) {
        printf("Witam w bibliotece.\n");
      }

$ cat libhello.h
      #ifndef M_HELLO
      #define M_HELLO
      void print_hello(void);

      #endif

$ gcc -c libhello.c

$ ar cru libhello.a libhello.o

$ cat program.c
      #include "libhello.h"

      int main(void) {
        print_hello();
        return 0;
      }
$ gcc program.c libhello.a -o program

lub

$ gcc program.c -L. –lhello -o program
```

- Inne użyteczne opcje polecenia ar:

-t	listowanie zawartości biblioteki
-d	kasowanie modułu z biblioteki
-xv	wyciągnij moduły z biblioteki z listowaniem nazw wyciąganych modułów

Tworzenie biblioteki współdzielonej

- Kompilacja z użyciem odpowiedniej opcji (kod niezależny od adresu, pod który będzie załadowany w programie):

```
gcc –fPIC –c libhello.c
```

- Utworzenie biblioteki (najprostsza metoda):

```
gcc -shared libhello.o -o libhello.so
```

- Kompilacja programu

```
gcc –L. -lhello program.c –o program
```

- Uruchamianie:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
./program
```

- Przykład:

```
$ cat libhello.c
      #include <stdio.h>
      void print_hello(void) {
        printf("Witam w bibliotece.\n");
      }
$ cat libhello.h
      #ifndef M_HELLO
      #define M_HELLO
      void print_hello(void);

      #endif
$ gcc –fPIC –c libhello.c
$ gcc -shared libhello.o -o libhello.so
$ cat program.c
      #include "libhello.h"

      int main(void) {
        print_hello();
        return 0;
      }
$ gcc program.c -L. -lhello -o program
```

Sprawdzenie, czy można uruchomić program:

```
$ ldd program
        linux-gate.so.1 =>  (0xffffe000)
        libhello.so => not found                          // Jeszcze nie!
        libc.so.6 => /lib/libc.so.6 (0xb7e68000)
        /lib/ld-linux.so.2 (0xb7fa7000)
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
$ ldd program
        linux-gate.so.1 =>  (0xffffe000)
        libhello.so => ./libhello.so (0xb7fac000)     // Już można
        libc.so.6 => /lib/libc.so.6 (0xb7e6e000)
        /lib/ld-linux.so.2 (0xb7faf000)
```