

# **PODSTAWY PROGRAMOWANIA W SYSTEMIE OPERACYJNYM UNIX / LINUX**

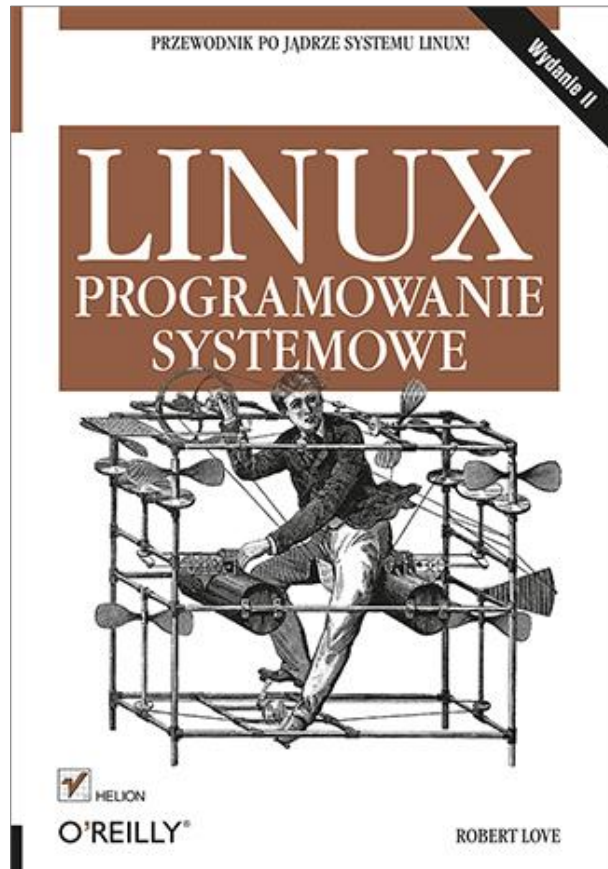
# Plan wykładu

- Wprowadzenie
- Interfejs wywołań systemowych
- Kompilowanie za pomocą GCC
- Wykorzystanie GNU Make
- Debugowanie za pomocą GDB
- Interakcja programu ze środowiskiem wykonania
- Tworzenie i używanie bibliotek

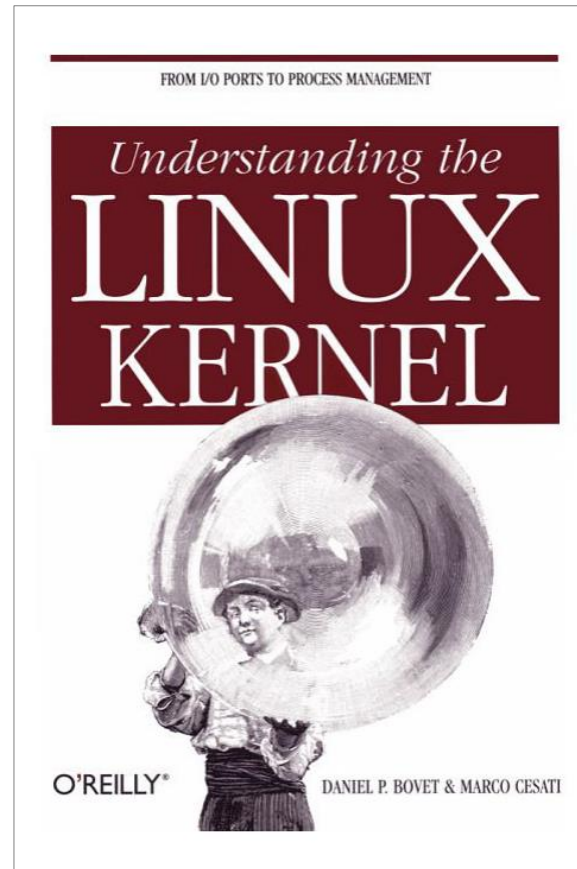
# Wprowadzenie

- **Wprowadzenie**
- Interfejs wywołań systemowych
- Kompilowanie za pomocą GCC
- Wykorzystanie GNU Make
- Debugowanie za pomocą GDB
- Interakcja programu ze środowiskiem wykonania
- Tworzenie i używanie bibliotek

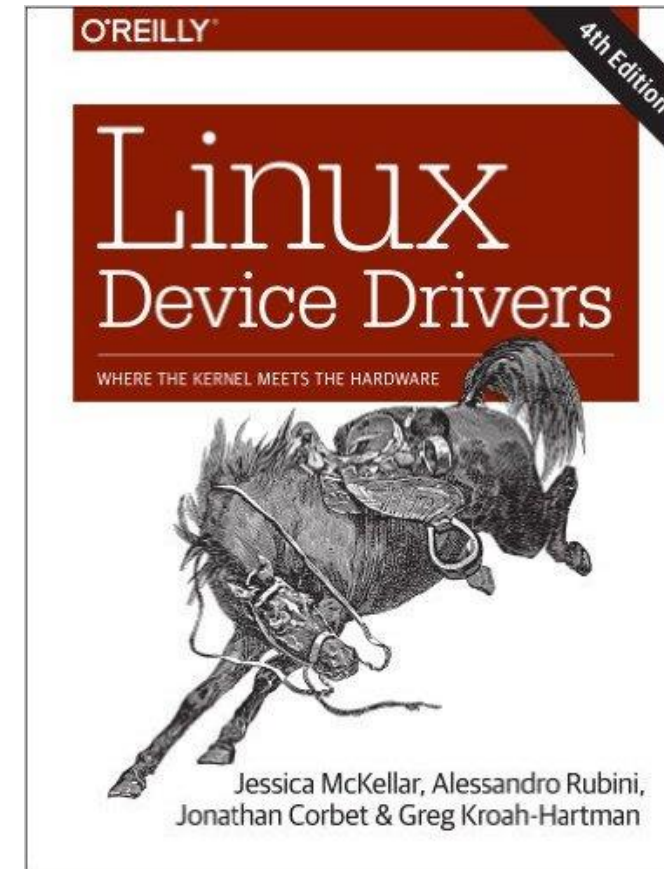
# Literatura 1/2



R. Love



D. Bovet, M. Cesati

J. McKellar, A. Rubini, J. Corbet,  
G. Kroah-Hartman

# Literatura 2/2

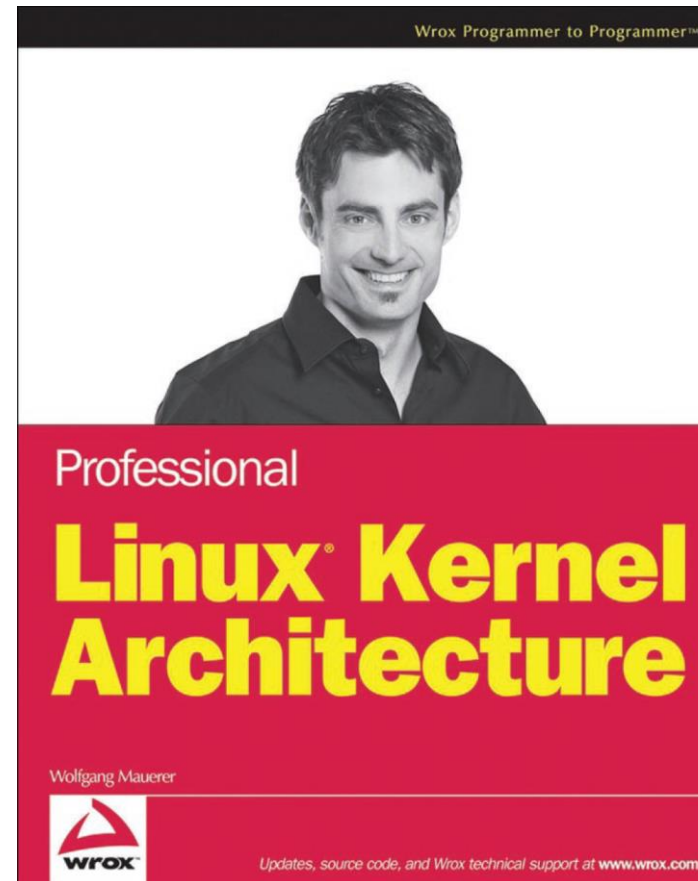
## THE LINUX PROGRAMMING INTERFACE

A Linux and UNIX\* System Programming Handbook

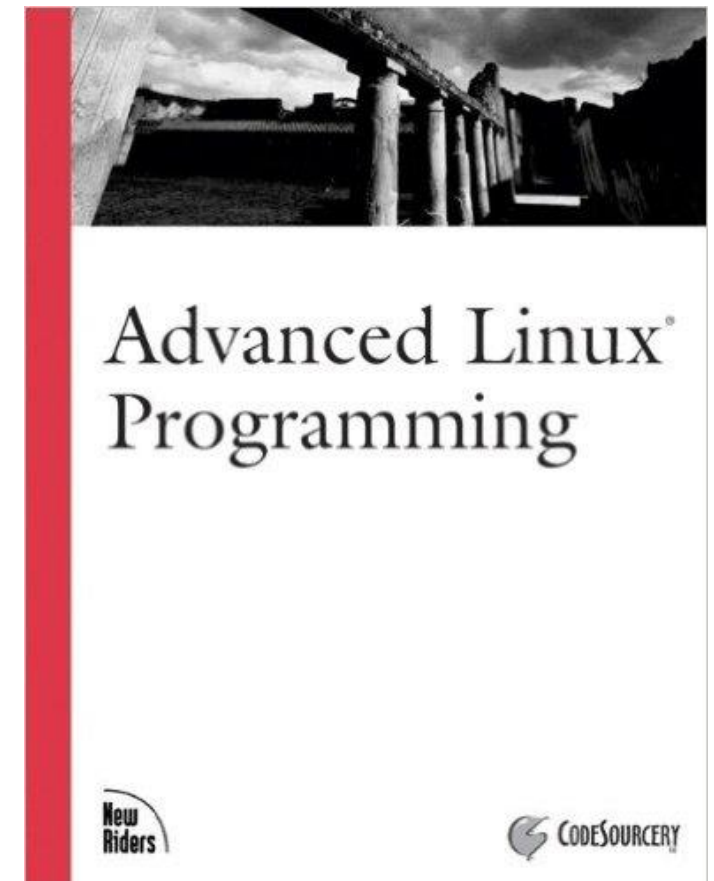
MICHAEL KERRISK



M. Kerrisk



W. Maurer

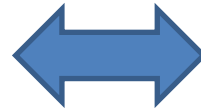


M. Mitchell, J. Oldham, A. Samuel

# Programowanie systemowe

## **Programowanie systemowe**

(usługi, elementy systemu, środowisko do wykonania innych programów itd.). Programowanie systemowe z założenia odbywa się bliżej sprzętu, chociaż niekoniecznie na poziomie jądra.



## **Programowanie aplikacyjne**

(programy bezpośrednio używane przez użytkownika, np. edytory, przeglądarki itp. – chociaż silnik przeglądarki może być z powodzeniem traktowany jako element oprogramowania systemowego)

Podział dosyć płynny. Programowanie systemowe wymaga lepszej znajomości działania systemów operacyjnych i sprzętu. Będziemy poruszać się na pograniczu przestrzeni jądra i użytkownika systemu Linux.

# UNIX - najważniejsze „wydania”

## Bell Labs firmy AT&T

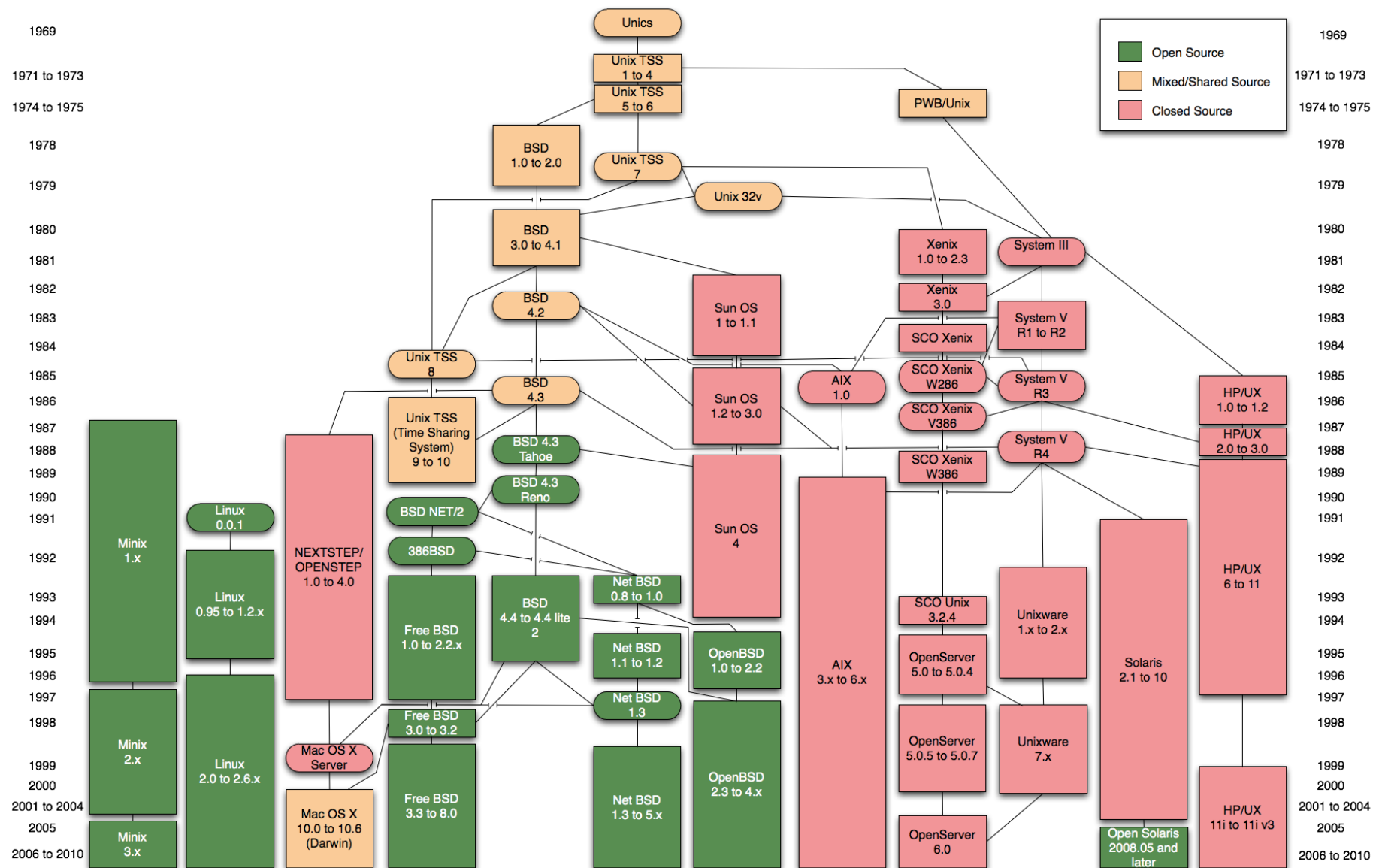
- 1969 – minikomputery PDP-7 i PDP-9 (DEC)
- 1971 – system przepisany na PDP-11/20
- V3: 1973 (potoki, przepisany w języku **C**)
- V6: 1975 (rozprowadzany nieodpłatnie na uczelniach)
- V7: 1979 (licencjonowany, przenośny, do czasu powstania POSIX standard *de facto*)

## Uniwersytet Berkeley

- 1975 - system **1BSD** (*Berkeley Software Distribution*)
- **3BSD**: 1979/1980 - system przepisany na VAX
- **4BSD / 4.1BSD**: 1980-1982 - na zamówienie DARPA dla ARPANET
- **4.2BSD**: 1983 - pierwszy system zawierający obsługę TCP/IP



PDP-11





# Interfejs a implementacja

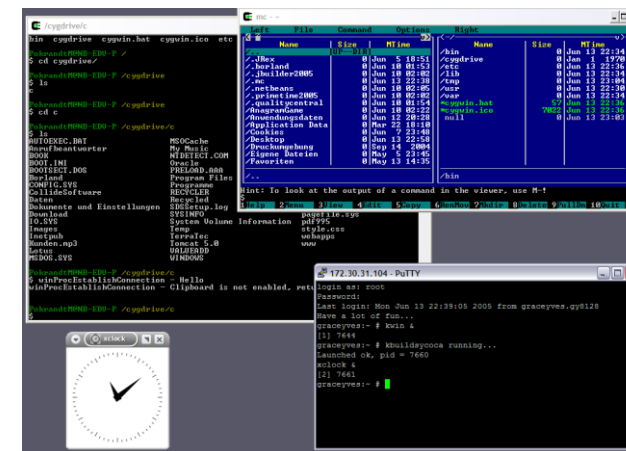
**Interfejs:** które żądania mogą być zgłaszane i jakiego typu odpowiedzi należy się podziwiać?

**Implementacja:** w jaki sposób udzielane są odpowiedzi na zgłaszane żądania

**BSD, HP-UX, Solaris i Tru64** mają więcej różnic funkcjonalnych niż **Linux** i np. **Unixware**

**Cygwin (GPL)** - implementacja standardu **POSIX** funkcji systemowych przeznaczona dla systemów Win32 oraz zestaw oprogramowania w większości przeniesionego z systemów typu **Unix** (w tym X.Org i KDE). Inne tego typu to np. **git bash**.

A jak się ma do tego **WSL2** ?



# Standaryzacja

**POSIX** (ang. *Portable Operating System Interface*) – próba standaryzacji różnych systemów UNIX (standard **IEEE 1003**). Współpraca: **IEEE**, **The Open Group**, IBM, Sun Microsystems, Hewlett-Packard, NEC, Fujitsu, Hitachi. Standard obejmuje m.in.:

1. interfejs programistyczny (API)
2. interfejs użytkownika, czyli polecenia systemowe takie jak między innymi: awk, echo, ed
3. właściwości powłoki systemu.

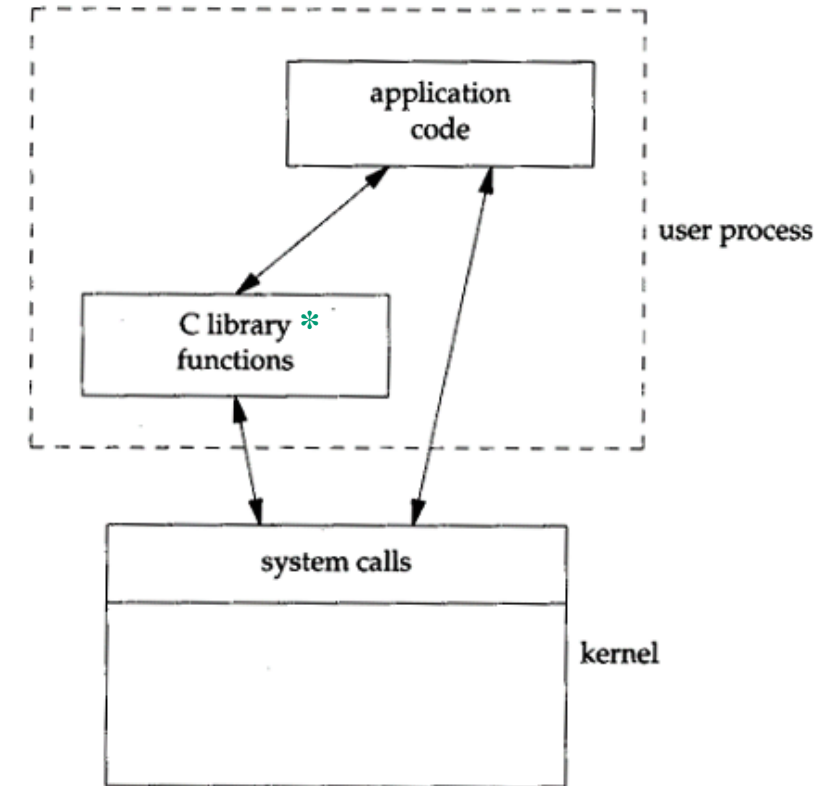
**Single UNIX Specification (SUS)** – inicjatywa **The Open Group** (poza IEEE), w dużej mierze zbieżna z POSIX. Dokumentacja ta zyskała bardzo dużą popularność dzięki bardzo wysokim cenom dokumentacji POSIX.

# ABI oraz API

- **ABI** (Application Binary Interface) – zbiór reguł komunikacji między programami, bibliotekami a systemem na poziomie **kodu skompilowanego** (np. sposób przekazywania argumentów w wywołaniach systemowych, użycie rejestrów, linkowanie, formaty plików wykonywalnych). ABI jest ściśle związane z architekturą.
- **API** (Application Programming Interface) – zbiór reguł komunikacji między programami (np. aplikacją a systemem) na poziomie **kodu źródłowego** (funkcje, struktury, klasy itp.).
- W systemach **UNIX**owych **ABI** ma mniejsze znaczenie niż **API**, np. format plików **ELF** nie gwarantuje wykonania programu przeniesionego między różnymi systemami.

# Wywołania systemowe

- Na **poziomie jądra** zaimplementowano szereg specjalnych procedur
- Program użytkownika wywołuje procedurę w **trybie jądra**, wykorzystując do tego celu **pułapkę** (w arch. 64 bitowych specjalny rozkaz).
- Procedura obsługi pułapki przełącza CPU w **tryb uprzywilejowany** i jądro wykonuje wywołanie systemowe
- CPU przechodzi z powrotem do **trybu użytkownika**
- Istnieje **API języka C** do wszystkich wywołań systemowych

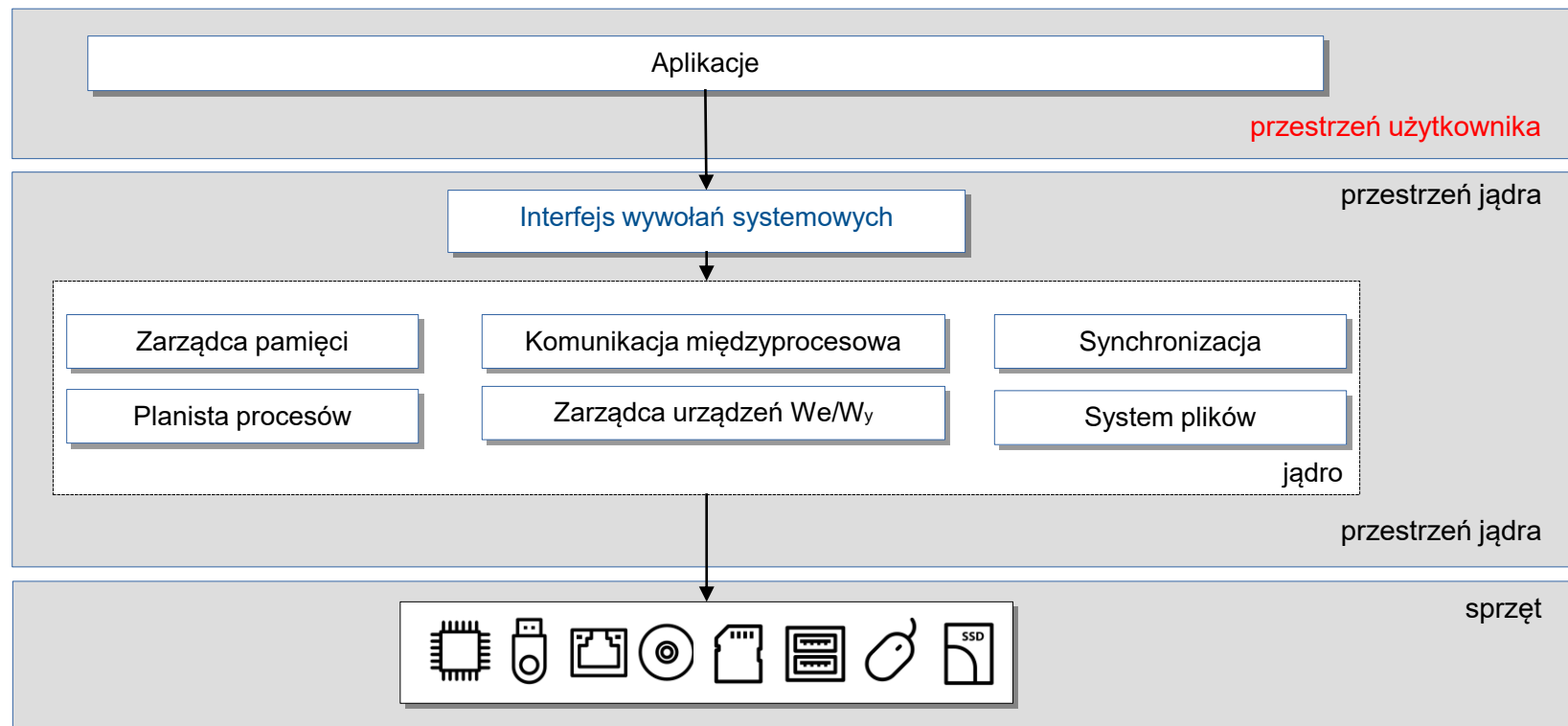


\* biblioteka standardowa (libc), w systemie Linux GNU C Library (glibc)

# Interfejs wywołań systemowych

- Wprowadzenie
- **Interfejs wywołań systemowych**
- Kompilowanie za pomocą GCC
- Wykorzystanie GNU Make
- Debugowanie za pomocą GDB
- Interakcja programu ze środowiskiem wykonania
- Tworzenie i używanie bibliotek

# Architektura monolityczna



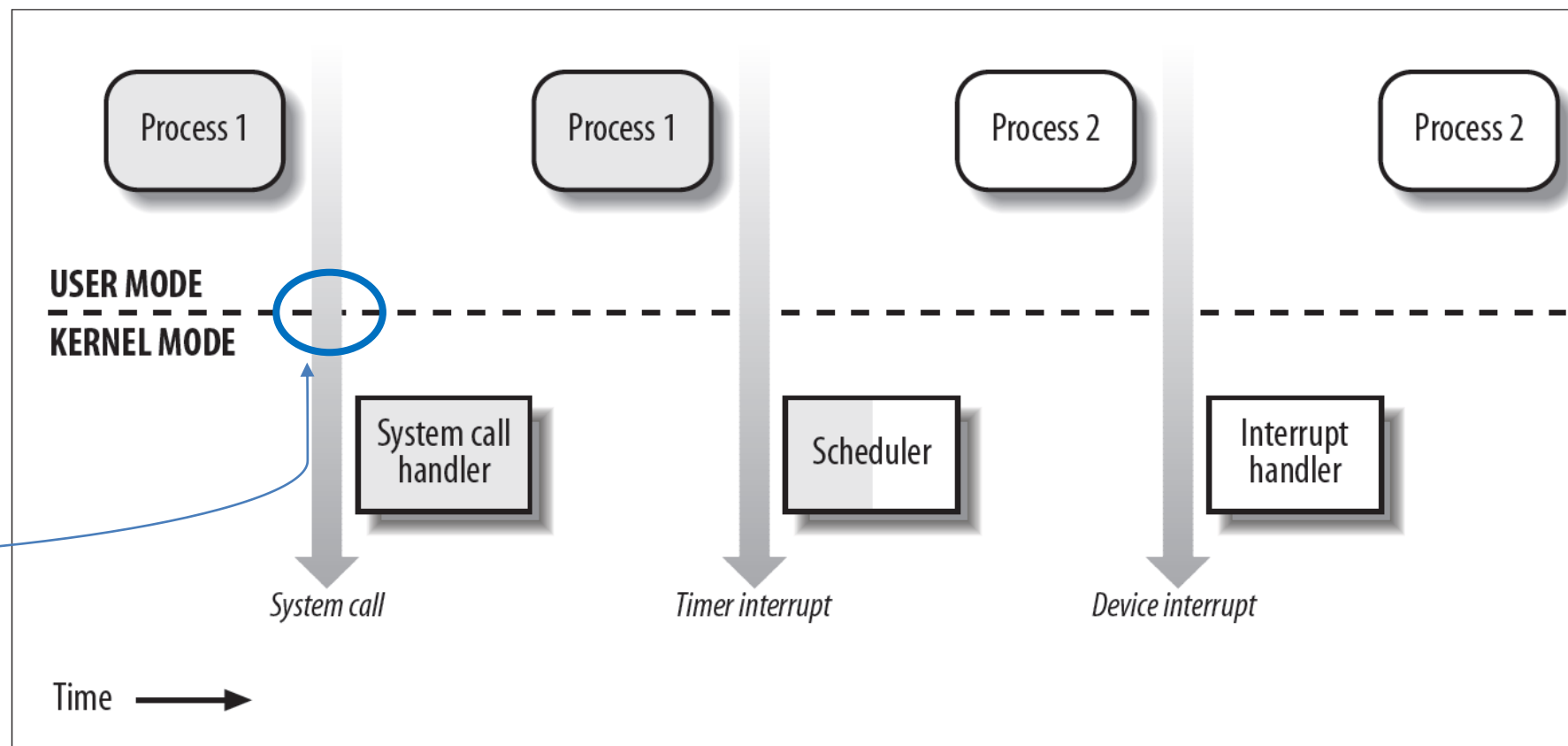
- **ochrona sprzętowa** (wsparcie ze strony procesora, ringi/poziomy uprzywilejowania)
- **dualny tryb pracy SO** (tryb użytkownika – tryb jądra)

- **system przerwań** (przełączanie między trybami)
- ochrona pamięci, operacji We/Wy, czasu procesora

# Tryb użytkownika vs tryb jądra 1/2

Część pracy jest wykonywana przez jądro na rzecz procesów użytkownika – dostęp do nich uzyskujemy m.in. wykorzystując **wywołania systemowe**.

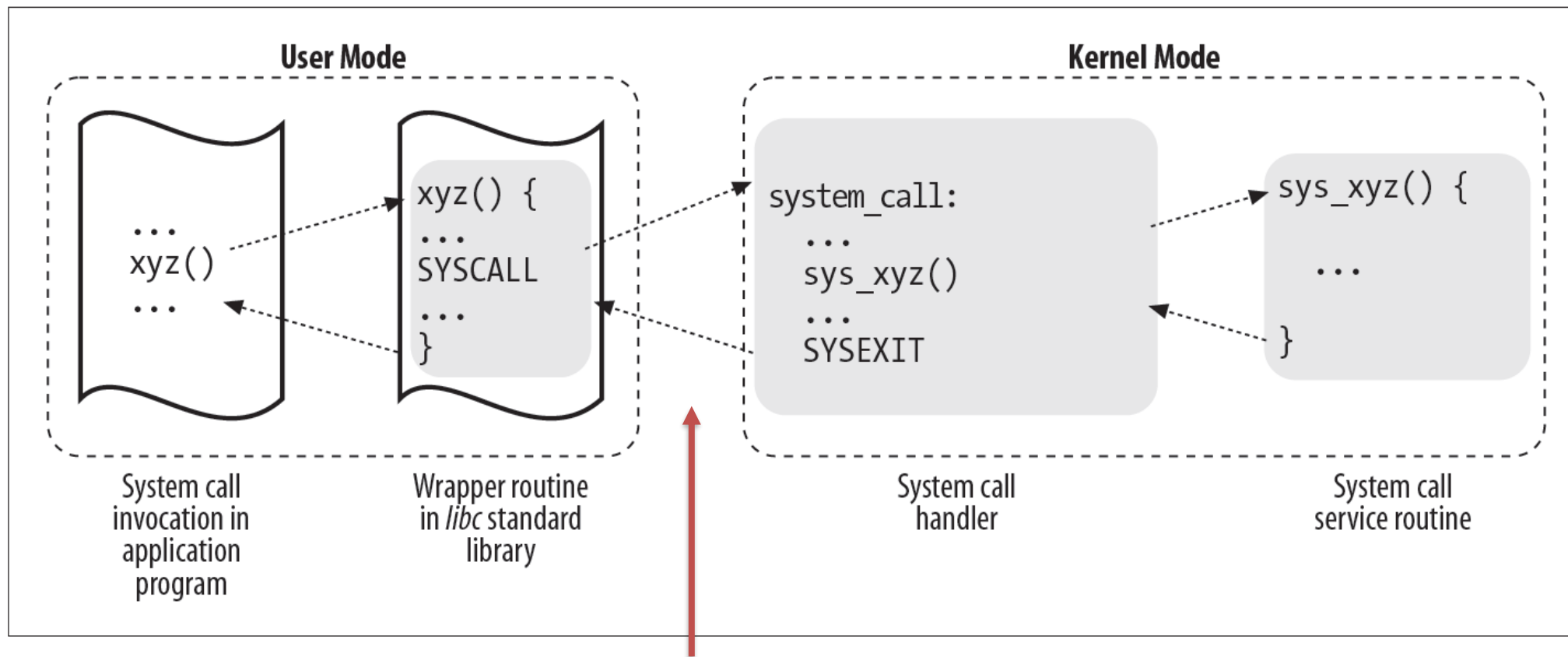
Lista wywołań:  
man syscalls



```
wmackow@jota-7273:~$ time find /usr > /dev/null 2> /dev/null  
  
real    0m13.552s  
user    0m3.620s  
sys     0m4.060s
```

## Tryb użytkownika vs tryb jądra 2/2

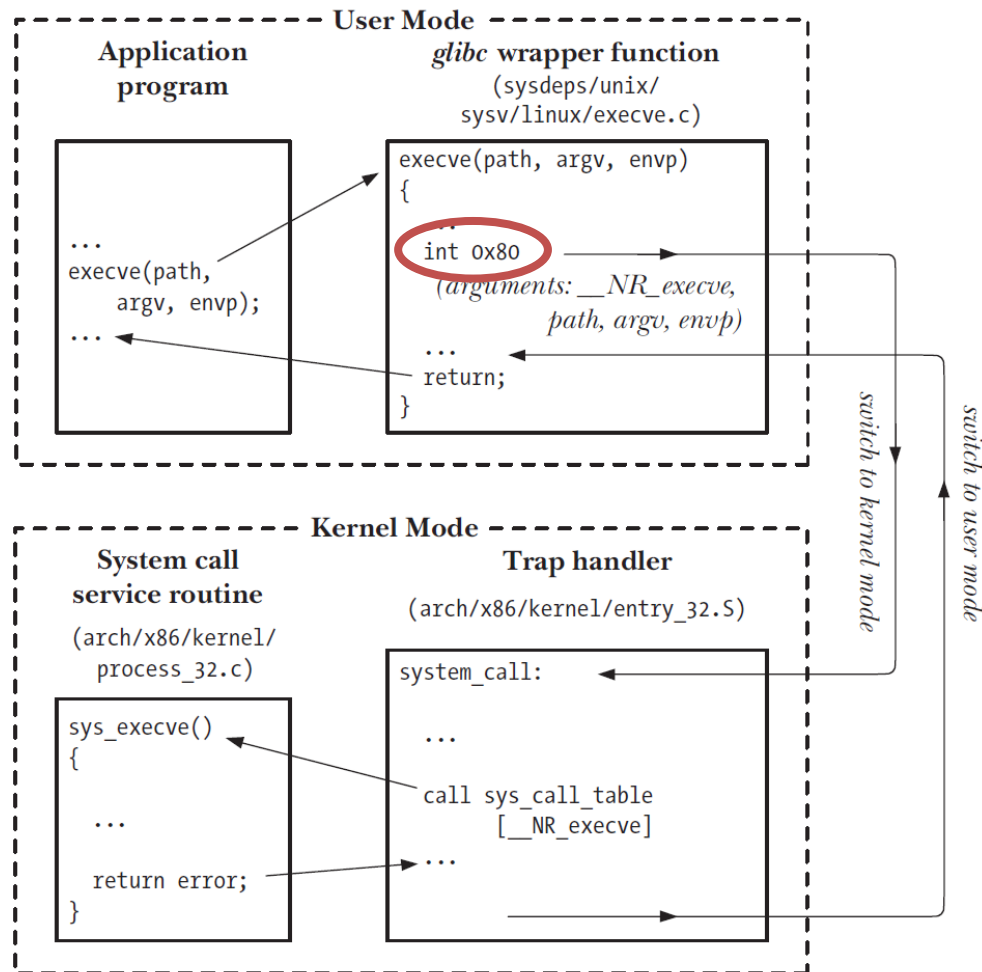
aplikacja > biblioteka systemowa > obsługa **sytem\_call** > obsługa wywołania



Jak sterowanie przekazywane jest do jądra?



# Interfejs wywołań systemowych 1/3



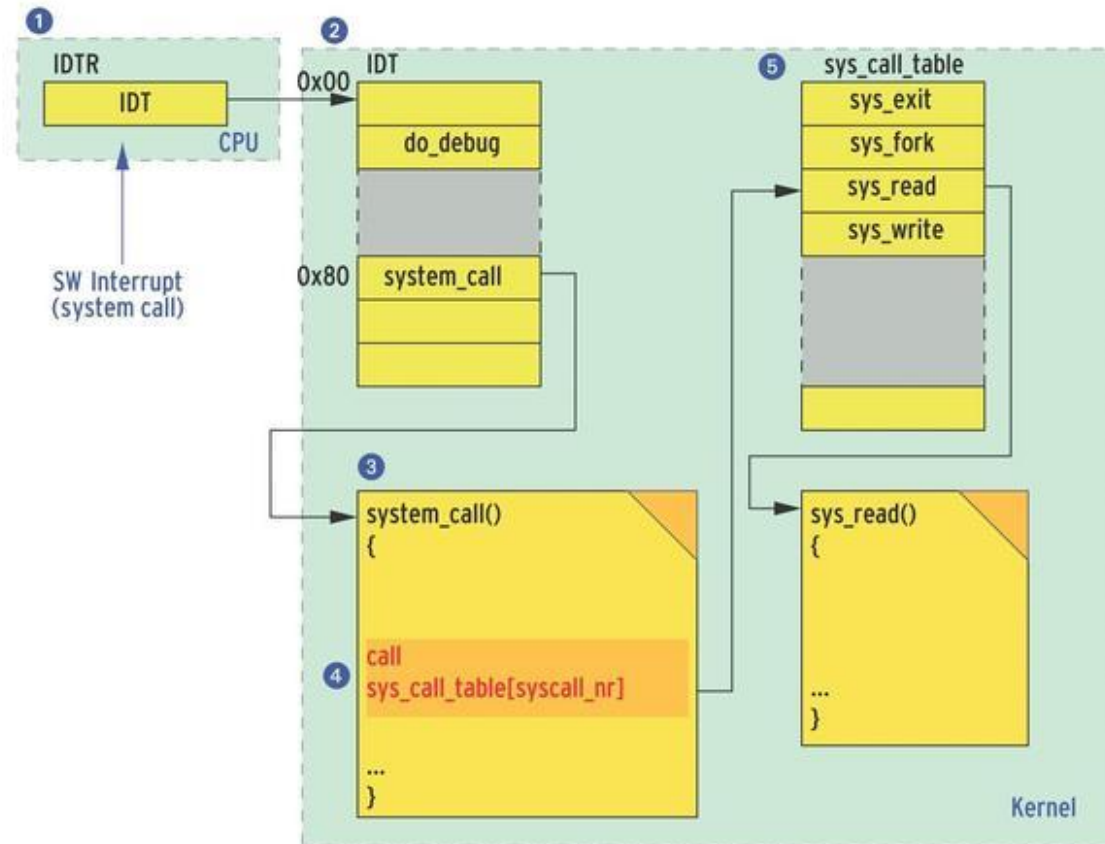
Metody uruchamiania wywołań systemowych:

1. wykorzystanie **funkcji systemowych** z biblioteki standardowej,
2. wywołanie funkcji systemowej **syscall** z numerem wywołania,
3. wywołanie przerwania **0x80**,
4. wykorzystanie techniki **fast syscall**:
  - rozkaz **syscall** (64-bitowy),
  - rozkaz **sysenter** (32-bitowy).

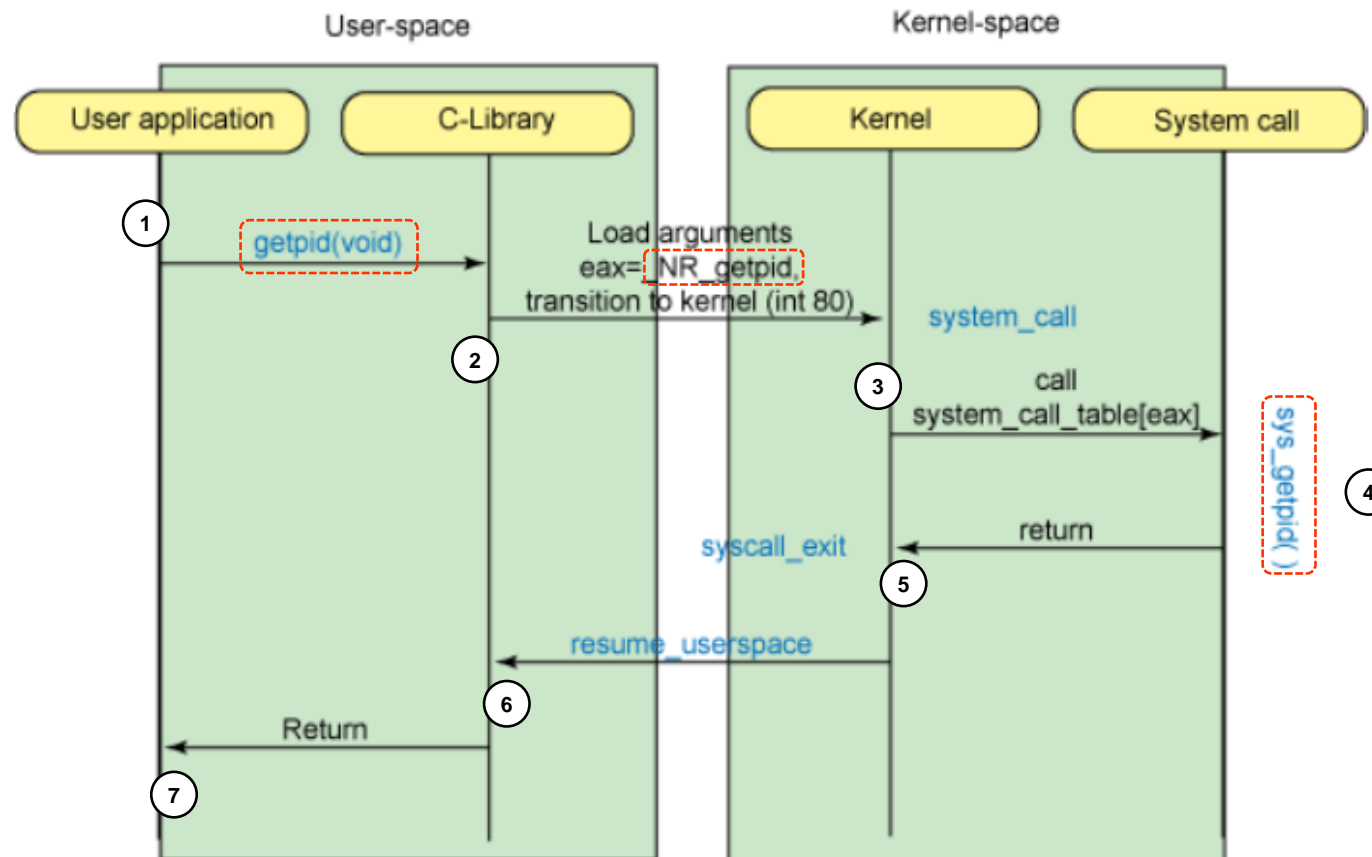
# Wywołanie przerwania **0x80** 1/2

IDT - Interrupt Descriptor Table  
(Tablica deskryptorów przerwania)

Tablica funkcji systemowych



# Wywołanie przerwania **0x80** 2/2



Uproszczony schemat wywołania `getpid`

# Przykład: wywołanie sys\_getpid 1/3

Wywołania systemowe można uruchamiać bezpośrednio przez funkcję **syscall** (lub powiązane z nim makra), jednak większość wywołań systemowych ma zdefiniowane funkcje opakowujące w C (biblioteka **glibc**).

*test-syscall.c*

```
#include <syscall.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>

int main(void)
{
    long ID1, ID2;
    ID1 = syscall(SYS_getpid);
    printf ("syscall(SYS_getpid)=%ld\n", ID1);

    ID2 = getpid();
    printf ("getpid()=%ld\n", ID2);
    return(0);
}
```

# Przykład: wywołanie sys\_getpid 2/3

Klasyczny syscall z przerwaniem **0x80** (dostępny w x86 i amd64).

```
test-int80.c

#include <stdio.h>

int main(void)
{
    long ID3;
    __asm__ __volatile__(
        "mov    $20, %%rax;"          // x86 : SYS_getpid = 20
        "int    $0x80;"
        "mov    %%rax, %0;" : "=r"(ID3)
    );
    printf ("getpid()=%ld\n", ID3);
    return(0);
}
```

Uwaga - normalnie nie odwołujemy się do wywołań w ten sposób, tylko korzystamy z funkcji systemowych!

# Przykład: wywołanie sys\_getpid 3/3

Fast syscall z bezpośrednim wywołaniem uprzywilejowanego kodu (dostępny w amd64).

```
test-fastSyscall.c

#include <stdio.h>

int main(void)
{
    long ID4;
    __asm__ __volatile__(
        "mov $39, %%rax;"           // amd64 : SYS_getpid = 39
        "syscall;"
        "mov %%rax, %0;" : "=r" (ID4)
    );
    printf ("getpid()=%ld\n", ID4);
    return(0);
}
```

Uwaga - normalnie nie odwołujemy się do wywołań w ten sposób, tylko korzystamy z funkcji systemowych!

# Kontrola błędów

W kontekście procesu przechowywana jest rozszerzona informacja o zakończeniu ostatnio wywołanej **funkcji systemowej**. Dostęp do tej informacji możemy uzyskać przez zmienną globalną **errno** po dołączenie nagłówka **<errno.h>**.

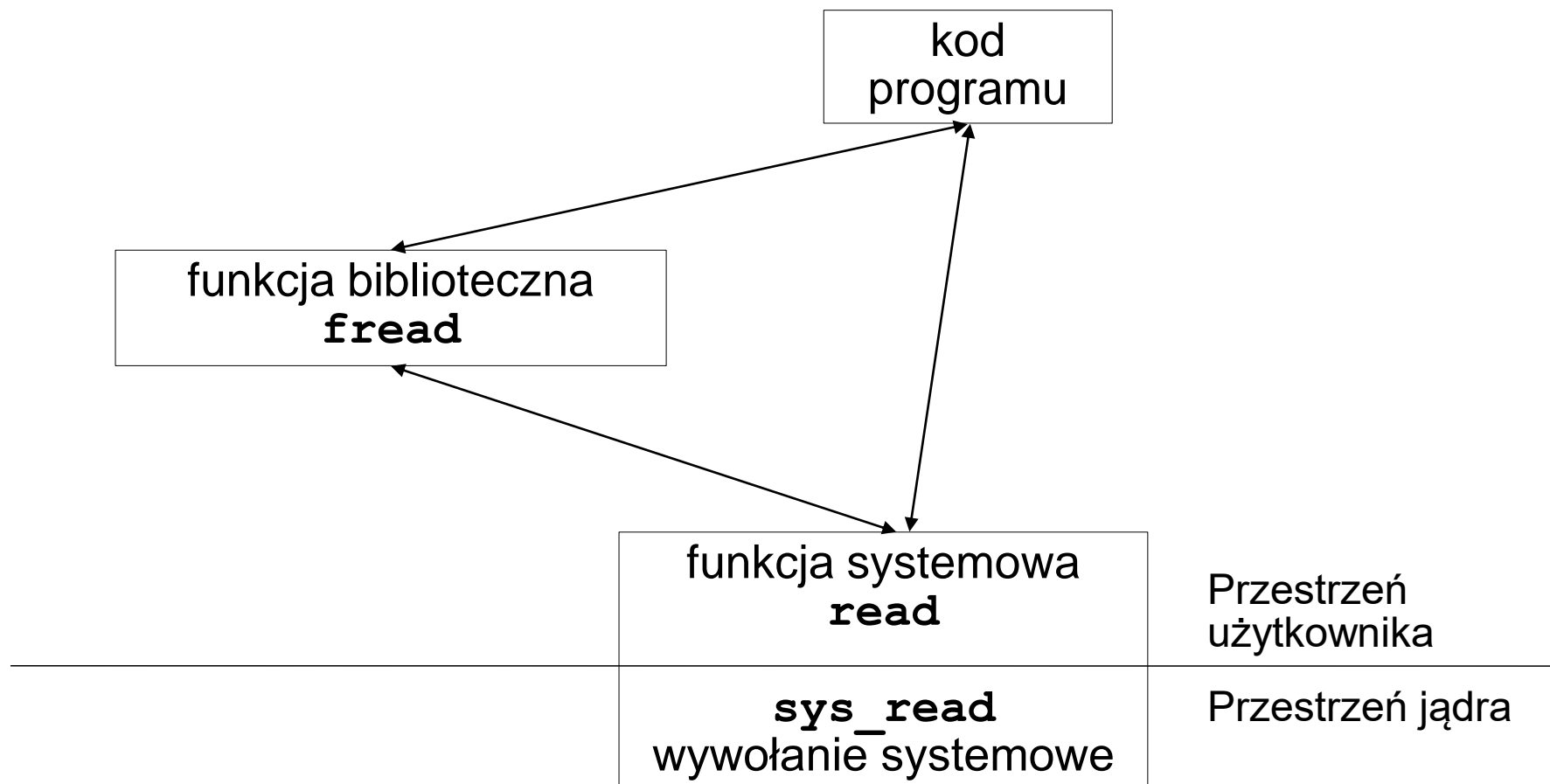
*test-err.c*

```
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>

int main()
{
    int des = open( "plik_ktorego.nie_ma", O_RDONLY);
    if( des == -1)
        printf( "errno=%d", errno);
    return 0;
}
```

Istnieje wiele funkcji formatujących i wyświetlających przechowywaną informację o błędzie, m.in.: **error** (3), **perror** (3), **strerror** (3), **strerror\_r** (3), itp.

# Funkcje biblioteczne a funkcje systemowe 1/2





# Funkcje biblioteczne a funkcje systemowe 2/2

systemowa

```
wmackow@jota-7273:~$ man 2 read
```

READ(2)

Linux Programmer's Manual

READ(2)

NAME

read - read from a file descriptor

SYNOPSIS

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

```
ssize_t read(int fd, void *buf, size_t count);
```

DESCRIPTION

read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

...

biblioteczna

```
wmackow@jota-7273:~$ man 3 fread
```

FREAD(3)

Linux Programmer's Manual

FREAD(3)

NAME

fread, fwrite - binary stream input/output

...

## Polecenie **strace** 1/2

**strace** to narzędzie do analizy kodu badające interakcję programu z jądrem systemu operacyjnego - śledzi wywołania systemowe programu przestrzeni użytkownika, wyświetla nazwy wywołań, wyświetla argumenty w postaci symbolicznej, wyświetla symboliczną nazwę błędu oraz odpowiadający jej napis (jeżeli któreś z wywołań zakończy się błędem). Dane te uzyskuje z jądra - program może być śledzony bez wsparcia dla debugowania.

### Opcje:

- **t** kiedy nastąpiło wywołanie systemowe,
- **T** czas spędzony w wywołaniu systemowym,
- **e** ograniczenie typu śledzonych wywołań, np. `-eopen` (oznacza `-e trace=open`) pozwala śledzić tylko wywołania systemowe `open`,
- **o** przekieruje wynik działania programu do pliku,
- **f** śledzenie nie tylko procesu macierzystego ale i dzieci,
- **p** możliwość "podłączenia" się do działającego w systemie procesu (**strace -p pid** )

# Polecenie **strace** 1/2

```
#include <unistd.h>

int main() {
    return getpid();
}
```

\$ gcc getpid.c -o getpid; **strace** ./getpid




```
...
getpid()                = 20104
exit_group(20104)        = ?
+++ exited with 136 +++
```

```
#include <stdio.h>

int main() {
    printf("abcd\n");
    return 0;
}
```

\$ gcc printf.c -o printf; **strace** ./printf



```
...
write(1, "abcd\n", 5)    = 5
exit_group(0)            = ?
+++ exited with 0 +++
```

# Kompilowanie za pomocą GCC

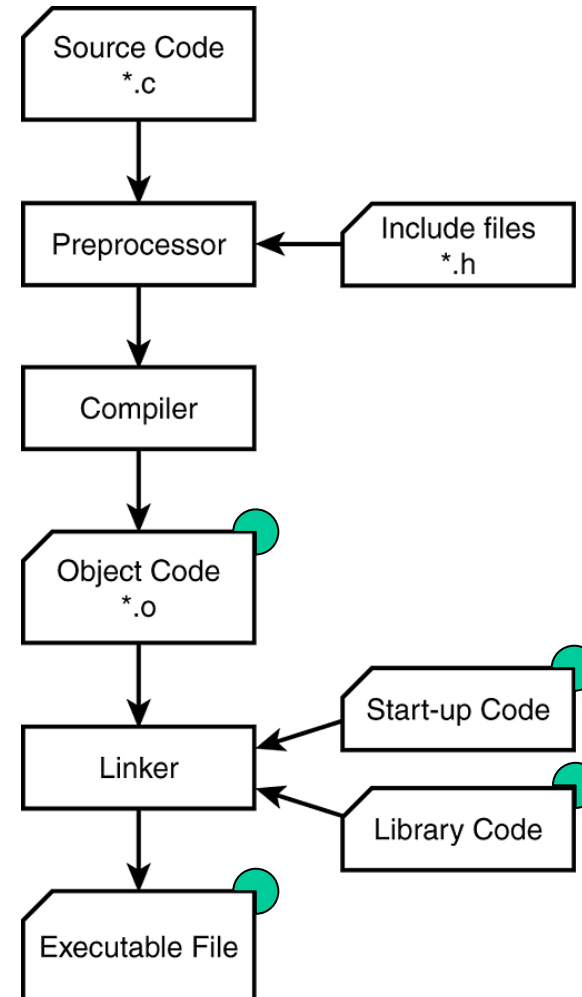
- Wprowadzenie
- Interfejs wywołań systemowych
- **Kompilowanie za pomocą GCC**
- Wykorzystanie GNU Make
- Debugowanie za pomocą GDB
- Interakcja programu ze środowiskiem wykonania
- Tworzenie i używanie bibliotek

# GCC

- **GCC** (ang. *GNU Compiler Collection*) jest zestawem kompilatorów kompilatory m.in. gcc, g++, g77 (fortran), itd.
- Składnia (dla C i C++)
  - **gcc** [opcje | pliki] – kompilator C
  - **g++** [opcje | pliki] – kompilator C++
- Etapy działania **GCC**
  - przetwarzanie wstępne (ang. *preprocessing*)
  - kompilacja (ang. *compilation*)
  - asemblacja (ang. *assembling*)
  - konsolidacja (ang. *linking*)

# GCC –podstawowe rozszerzenia plików

<b>.c</b>	źródła C
<b>.C</b>	źródła C++
<b>.cc</b>	źródła C++
<b>.cxx</b>	źródła C++
<b>.c++</b>	źródła C++
<b>.i</b>	preprocessed C
<b>.ii</b>	preprocessed C++
<b>.s</b>	źródła asemblera
<b>.S</b>	źródła asemblera
<b>.h</b>	pliki nagłówkowe
<b>.o</b>	pliki typu obiekt
<b>.a</b>	biblioteki
<b>.so</b>	biblioteki



# GCC – kompilacja jednego źródła

```
$ ls -l
```

```
-rw-r--r--  1 burek users      214  2007-02-22  20:04  prostokat.c
```

```
$ gcc prostokat.c
```

```
$ ls -l
```

```
-rwxr-xr-x  1 burek users  11337  2007-02-22  20:05  a.out
```

```
-rw-r--r--  1 burek users      214  2007-02-22  20:04  prostokat.c
```

```
$ ./a.out
```

```
obwod = 60, pole = 200
```

```
$ gcc prostokat.c -o prostokat
```

```
$ ls -l
```

```
-rwxr-xr-x  1 burek users  11337  2007-02-22  20:05  a.out
```

```
-rwxr-xr-x  1 burek users  11337  2007-02-22  20:11  prostokat
```

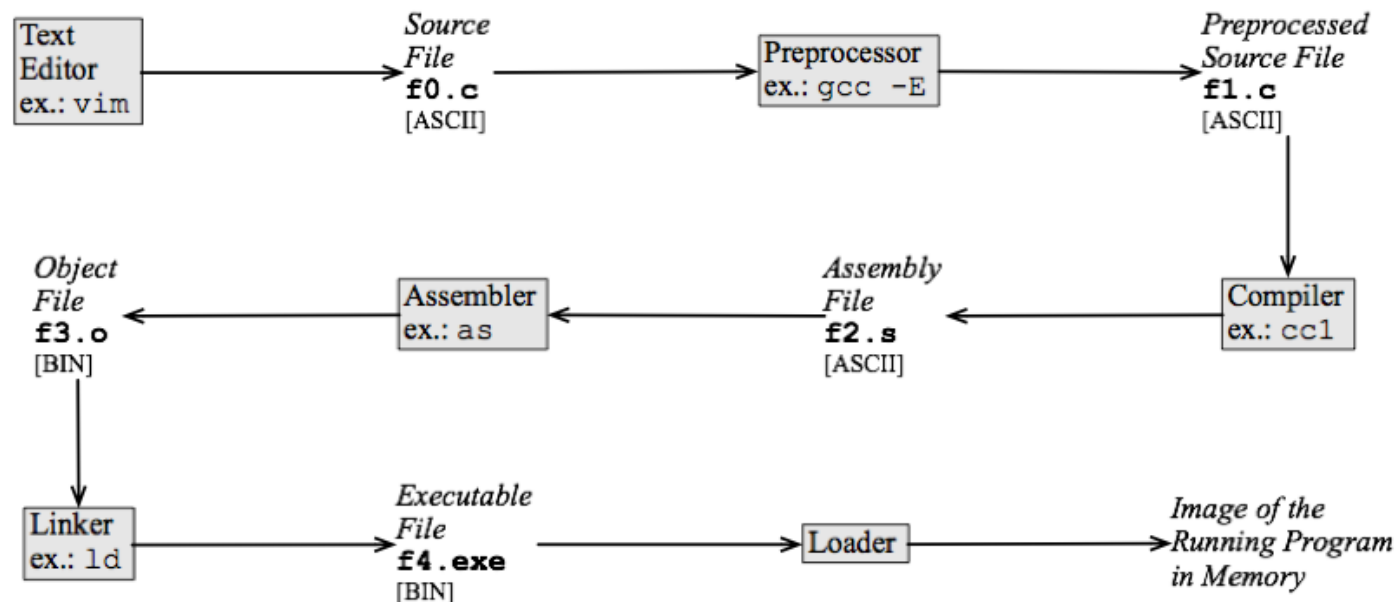
```
-rw-r--r--  1 burek users      214  2007-02-22  20:04  prostokat.c
```

```
$ ./prostokat
```

```
obwod = 60, pole = 200
```

# GCC – główne opcje

- E Tylko preprocesing, na wyjściu dostajemy pliki źródłowe przerobione przez preprocesor.
- S Zatrzymuj po poziomie kompilacji, nie asembluj, na wyjściu mamy plik źródłowym z kodem assemblera.
- c Zatrzymuj po poziomie kompilacji lub asemblacji, bez linkowania. Na wyjściu dostajemy pliki typu obiekt dla każdego pliku źródłowego.
- o *nazwa* Wskazanie nazwy pliku wynikowego dla danej operacji (domyślnie **a.out**).





# GCC – kompilacja wielu źródeł

W *prostokat.c* jest f-cja `main()`, w *lib.c* funkcje wywoływane z `main`.

---

```
$ gcc prostokat.c lib.c -o prostokat ; ls -l
```

```
-rw-r--r--  1 burek users      214  2007-02-22  20:07  lib.c
-rwxr-xr-x  1 burek users  11337  2007-02-22  20:08  prostokat
-rw-r--r--  1 burek users      214  2007-02-22  20:04  prostokat.c
```

---

```
$ gcc -c prostokat.c lib.c ; ls -l
```

```
-rw-r--r--  1 burek users      214  2007-02-22  20:07  lib.c
-rw-r--r--  1 burek users      214  2007-02-22  20:10  lib.o
-rw-r--r--  1 burek users      214  2007-02-22  20:04  prostokat.c
-rw-r--r--  1 burek users      214  2007-02-22  20:10  prostokat.o
```

---

```
$ gcc prostokat.o lib.o -o prostokat ; ls -l
```

```
-rw-r--r--  1 burek users      214  2007-02-22  20:07  lib.c
-rw-r--r--  1 burek users      214  2007-02-22  20:10  lib.o
-rwxr-xr-x  1 burek users  11337  2007-02-22  20:11  prostokat
-rw-r--r--  1 burek users      214  2007-02-22  20:04  prostokat.c
-rw-r--r--  1 burek users      214  2007-02-22  20:10  prostokat.o
```

# GCC – inne opcje 1/2

## Opcje preprocesora:

- `-D macro`                      Ustaw *macro* na 1.
  - `-D macro=defn`                Zdefiniuj makro *macro* jako *defn*.
  - `-U macro`                        Skasuj definicje makra *macro*.
- 

## Opcje debugera:

- `-g`                                Dodatkowe informacje dla debugera. Kompilacja z tą opcją pozwala na późniejsze debugowanie programu
  - `-ggdb`                            Dodatkowe informacje dla DBG (możliwość wykorzystania rozszerzeń GDB)
- 

## Opcje preprocesora:

- `-I dir`                        Dodaje katalog *dir* do listy katalogów przeszukiwanych ze względu na pliki nagłówkowe
- `-L dir`                        Dodaje katalog *dir* do listy katalogów przeszukiwanych ze względu na biblioteki (przełącznika `-l`)

# GCC – inne opcje 2/2

## Opcje linkera:

- `-llibrary` Użyj biblioteki *library* kiedy linkujesz. Uwaga! **gcc** automatycznie dodaje przedrostek *lib* i końcówkę *.a*, np.: `-lFOX` w celu załadowania *libFOX.a*. Patrz też `-L`.
  - `-nostdlib` Nie używaj standardowych bibliotek systemowych (tylko wskazane)
  - `-nostartfiles` Nie używaj standardowych systemowych plików startowych.
- 

## Opcje optymalizacji:

- `-o` Optymalizacja.
  - `-Olevel` Poziom optymalizacji: 0,1,2,3, jeśli 0, to brak optymalizacji.
- 

## Opcje ostrzeżeń:

- `-Wall` Wypisuje ostrzeżenia dla wszystkich sytuacji, które pretendują do konstrukcji, których używania się nie poleca i których użycie jest proste do uniknięcia, nawet w połączeniu z makrami.

# GCC – interpretacja komunikatów o błędach

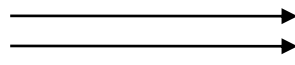
```
$ gcc prostokat.c
```

```
prostokat.c: In function `main':  
prostokat.c:10: error: `obwod' undeclared (first use in this function)  
prostokat.c:10: error: (Each undeclared identifier is reported only once  
prostokat.c:10: error: for each function it appears in.)  
prostokat.c:11: error: `pole' undeclared (first use in this function)  
prostokat.c:16:2: warning: no newline at end of file
```

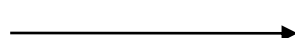
*prostokat.c*

```
#include <stdio.h>  
  
int main()  
{  
    int x, y;  
  
    x = 10;  
    y = 20;  
  
    obwod = 2*x + 2*y;  
    pole = x*y;  
  
    printf( " obwod = %d, pole = %d \n", obwod, pole);  
  
    return( 0);  
}
```

linia 10  
linia 11



linia 16



# Wykorzystanie GNU Make

- Wprowadzenie
- Interfejs wywołań systemowych
- Kompilowanie za pomocą GCC
- **Wykorzystanie GNU Make**
- Debugowanie za pomocą GDB
- Interakcja programu ze środowiskiem wykonania
- Tworzenie i używanie bibliotek

# Budowa prostego pliku **Makefile**

cel : zależności  
<Tab>reguła

## *Makefile*

```
outprognose : binarytree.o mainprog.o
    gcc -o outprognose binarytree.o mainprog.o

binarytree.o : binarytree.c
    gcc -c binarytree.c

mainprog.o : mainprog.c
    gcc -c mainprog.c
```

Wykonanie **make** dla nowych plików źródłowych (polecenie **make** szuka w aktualnym katalogu pliku *Makefile*)

```
$ make
gcc -c mainprog.c
gcc -c binarytree.c
gcc -o outprognose binarytree.o mainprog.o
```

Ponowne wykonanie **make** po modyfikacji *mainprog.c*

```
$ vi mainprog.c
$ make
gcc -c mainprog.c
gcc -o outprognose binarytree.o mainprog.o
```

# Kiedy **make** wykona regułę ?

- 1) Jeżeli plik celu **nie istnieje**
- 2) Jeżeli plik celu jest **starszy** niż któryś z plików określonych w zależnościach dla tego celu

W zależnościach możemy podać pliki, które nie są **jawnie** wykorzystane w regule tworzenia, ale których zmiany powinny pociągnąć za sobą ponowne tworzenie celu.

## *Makefile*

```
outprogname : binarytree.o mainprog.o
    gcc -o outprogname binarytree.o mainprog.o

binarytree.o : binarytree.c binarytree.h
    gcc -c binarytree.c

mainprog.o : mainprog.c binarytree.h
    gcc -c mainprog.c
```

# Użycie zmiennych w pliku Makefile

## *Makefile*

```
cc=gcc
cflags=-O3 -g
ldflags=-g

outprogname : binarytree.o mainprog.o
    ${cc} ${ldflags} -o outprogname binarytree.o mainprog.o

binarytree.o : binarytree.c
    ${cc} ${cflags} -c binarytree.c

mainprog.o : mainprog.c
    ${cc} ${cflags} -c mainprog.c
```

1.

```
$ make
gcc -O3 -g -c mainprog.c
gcc -O3 -g -c binarytree.c
gcc -g -o outprogname binarytree.o mainprog.o
```

2.

```
$ make cflags=-O2
gcc -O2 -c mainprog.c
gcc -O2 -c binarytree.c
gcc -g -o outprogname binarytree.o mainprog.o
```



# Makefile - zmienne wbudowane

`$@` - nazwa celu (`%` dla archiwów)  
`$<` - pierwszy wymagany  
`$^` - wszystkie wymagane, bez powtórzeń  
`$+` - wszystkie wymagane, z powtórzeniami  
`$?` - wszystkie wymagane nowsze niż cel

## *Makefile*

```
cc=gcc
cflags=-O3 -g
ldflags=-g

outprogname : binarytree.o mainprog.o
    ${cc} ${ldflags} -o $@ $^

binarytree.o : binarytree.c
    ${cc} ${cflags} -c $<

mainprog.o : mainprog.c
    ${cc} ${cflags} -c $<
```

```
$ make
gcc -O3 -g -c mainprog.c
gcc -O3 -g -c binarytree.c
gcc -g -o outprogname binarytree.o mainprog.o
```

# Makefile - użycie dopasowywania wzorców

Tworzenie wszystkich obiektów wymienionych we wcześniejszych zależnościach

```
Makefile
cc=gcc
cflags=-O3 -g
ldflags=-g

outprogname : binarytree.o mainprog.o
    ${cc} ${ldflags} -o $@ $^

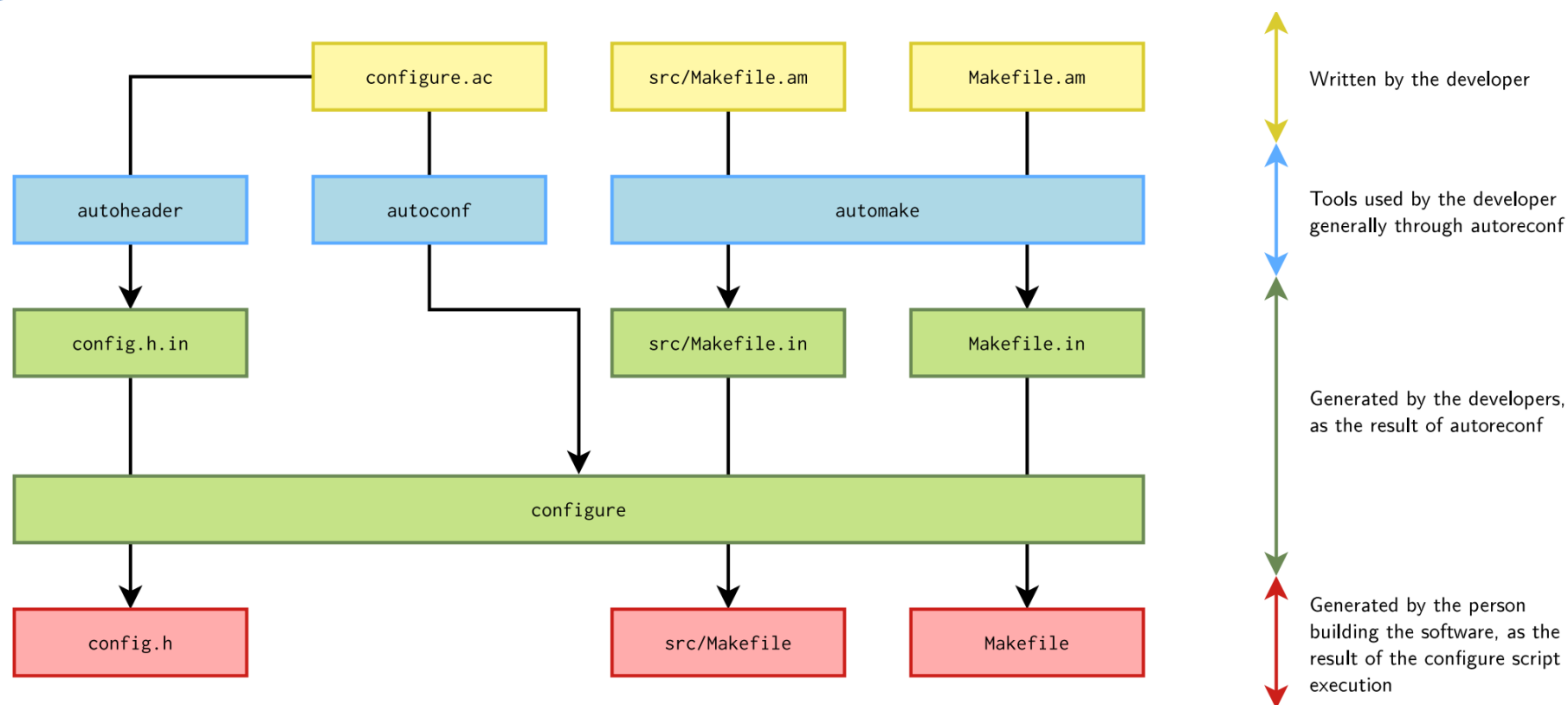
%.o : %.c
    ${cc} ${cflags} -c $<
```

```
$ make
gcc -O3 -g -c mainprog.c
gcc -O3 -g -c binarytree.c
gcc -g -o outprogname binarytree.o mainprog.o
```

# GNU Autotools 1/2

Autoconf  
Automake  
Libtool

Narzędzia z autotools, które pozwalają generować **skrypty instalacyjne** dla programów i bibliotek, które będziemy chcieli dystrybuować w postaci źródeł



# GNU Autotools 2/2

```
$ tree
```

```
.  
├── Makefile.am  
├── configure.ac  
└── src  
    ├── Makefile.am  
    └── main.c
```

```
1 directory, 4 files
```

```
$ autoreconf --verbose --install -force  
...
```

```
$ ./configure  
$ make  
$ src/hello  
Hello world!  
$ make install
```

## *Makefile.am*

```
SUBDIRS = src
```

## *configure.ac*

```
AC_INIT([PackageName], [1.0], [bug-report@address])  
AM_INIT_AUTOMAKE([-Wall -Werror foreign])  
AC_PROG_CC  
AC_CONFIG_HEADERS([config.h])  
AC_CONFIG_FILES([Makefile src/Makefile])  
AC_OUTPUT
```

## *src/Makefile.am*

```
bin_PROGRAMS = hello  
hello_SOURCES = main.c
```

## *src/main.c*

```
#include <stdio.h>  
int main() {  
    puts("Hello world!");  
    return 0;  
}
```

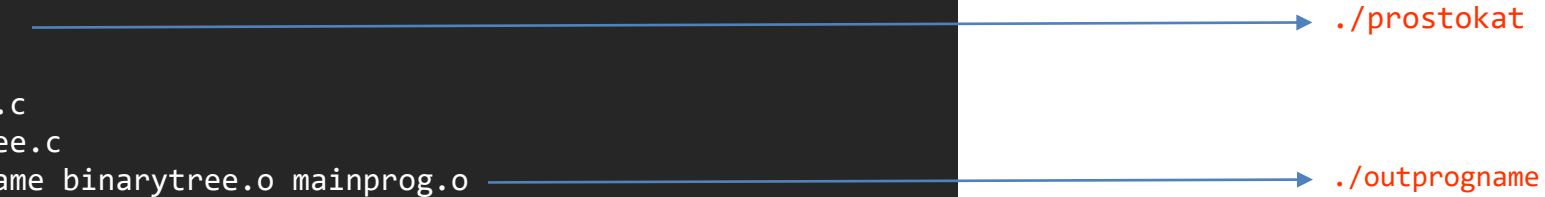
# Debugowanie za pomocą GDB

- Wprowadzenie
- Interfejs wywołań systemowych
- Kompilowanie za pomocą GCC
- Wykorzystanie GNU Make
- **Debugowanie za pomocą GDB**
- Interakcja programu ze środowiskiem wykonania
- Tworzenie i używanie bibliotek

# gdb - podstawy

**Kompilacja programu** - debugowany plik wykonywalny musi mieć dołączoną na etapie kompilacji i linkowania tablicę symboli (opcja `-g`)

```
$ gcc -g prostokat.c  
$ gcc -g -c mainprog.c  
$ gcc -g -c binarytree.c  
$ gcc -g -o outprognose binarytree.o mainprog.o
```



`./prostokat`  
`./outprognose`

## Uruchomienie gdb

bez argumentów

```
$ gdb
```

z nazwą programu


```
$ gdb outproganem
```

z nazwą programu i **pidem** procesu

```
$ gdb outproganem 12345
```

z nazwą programu i zrzutem pamięci

```
$ gdb qsort2 core.2957
```



```
Core was generated by `qsort2'.  
Program terminated with signal 7, Emulator trap.  
#0  0x2734 in qsort2 (l=93643, u=93864, strat=1)  
at qsort2.c:118  
118          do i++; while (i <= u && x[i] < t);  
(gdb) quit  
$
```

Wymuszenie na systemie tworzenia  
zrzutów pamięci w przypadku  
błędu: `$ ulimit -c unlimited`

# **gdb** – pliki i uruchomienie programu

**file** *[file]*

użyj plik *file* jako plik wykonywalny i tablicę symboli

**core** *[file]*

użyj plik *file* jako zrzut pamięci

**load** *[file]*

dołącz dynamicznie plik *file*

**info share**

wyświetl nazwy wszystkich aktualnie załadowanych  
bibliotek dzielonych

**run** *arglist*

uruchom program z listą argumentów *arglist*

**run**

uruchom program z aktualną listą argumentów

**set args** *arglist*

ustal listę argumentów *arglist* dla kolejnego

uruchomienia

**set args**

wyczyść listę argumentów dla kolejnego uruchomienia

**show args**

wyświetl listę argumentów dla kolejnego uruchomienia

**show env**

wyświetl wszystkie zmienne systemowe

## **gdb** - pułapki (*breakpoints*)

Użycie **b** jest równoważne użyciu **break**.

<b>b</b> [ <i>file</i> :] <i>line</i>	ustaw breakpoint w linii <i>line</i> [w pliku <i>file</i> ]
<b>b</b> [ <i>file</i> :] <i>func</i>	ustaw breakpoint na funkcji <i>func</i> [w pliku <i>file</i> ]
<b>b</b> ... <b>if</b> <i>expr</i>	przerwij warunkowo, jeżeli <i>expr</i> prawdziwe
<b>tbreak</b> ...	breakpoint tymczasowy (jednorazowy)
<b>rbreak</b> <i>regex</i>	przerywaj na wszystkich funkcjach dopasowanych do wyrażenia <i>regex</i>
<b>catch</b> <i>event</i>	przerwij po zdarzeniu <i>event</i> , (zdarzeniami mogą być <b>catch</b> , <b>throw</b> , <b>exec</b> , <b>fork</b> , <b>vfork</b> , <b>load</b> , <b>unload</b> )
<b>info break</b>	wyświetl zdefiniowane breakpointy
<b>clear</b> [ <i>file</i> :] <i>func</i>	usuń breakpoint z funkcji <i>func</i>
<b>clear</b> [ <i>file</i> :] <i>line</i>	usuń breakpoint z linii <i>line</i>
<b>delete</b> [ <i>n</i> ]	usuń wszystkie breakpointy [lub o numerze <i>n</i> ]



# gdb - wykonanie programu

<b>continue, c</b> [ <i>count</i> ]	kontynuuj wykonanie; jeżeli określono <i>count</i> to ignoruj <i>count</i> kolejnych breakpointów
<b>step, s</b> [ <i>count</i> ]	wykonaj pojedynczy krok (wchodzi do wnętrza funkcji); powtórz <i>count</i> razy
<b>next, n</b> [ <i>count</i> ]	wykonaj pojedynczy krok; powtórz <i>count</i> razy
<b>until</b> <i>location</i>	wykonaj aż do lokalizacji <i>location</i> (np. nr linii)
<b>finish</b>	wykonaj aż do zakończenia aktualnej f-cji i wyświetl zwróconą wartość
<b>jump</b> <i>line</i>	wznów wykonanie od linii <i>line</i>
<b>set var</b> <i>a=expr</i>	zapisz do zmiennej o nazwie <i>a</i> wartość <i>expr</i>
<b>backtrace</b>	wyświetl wszystkie ramki ze stosu (m.in. informacja o wywołanych funkcjach)

## gdb - wyświetlanie danych 1/2

**print, p** *[/f] [expr]*

wyświetl wartość wyrażenia *expr* (albo wyrażenia ostatnio wyświetlanego) zgodnie z formatem *f* ; może również posłużyć do wywołania f-cji C i wyświetlenia jej wyniku, wyświetlenia zawartości rejestru albo wskazanego obszaru pamięci

p abc

*wyświetl wartość zmiennej abc*

p /x abc

*wyświetl wartość zmiennej abc (hex)*

p /t abc

*wyświetl wartość zmiennej abc (binarnie)*

p &abc

*wyświetl adres zmiennej abc*

p (int)getpid()

*wyświetl wynik działania f-cji getpid*

p \$ebx

*wyświetl zawartość rejestru EBX*

**display** *[/f] [expr]*

wyświetlaj wartość wyrażenia *expr* zawsze gdy program się zatrzyma (zgodnie z formatem *f*)

**display**

wyświetl listę wszystkich obserwowanych zmiennych

**undisplay** *n*

usuń z listy obserwowanych zmiennych *n* -te wyrażenie

## **gdb** - wyświetlanie danych 2/2

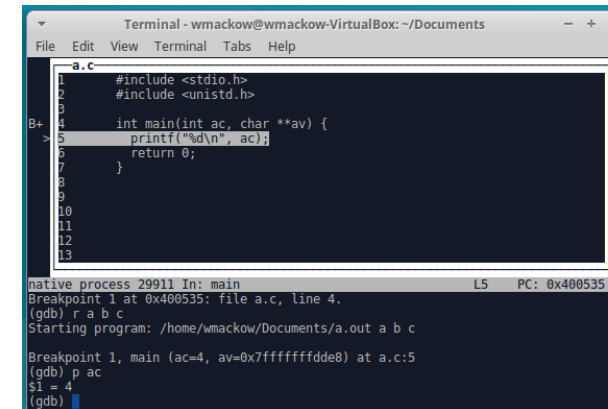
<b>x</b> [/l] [/f] [address expr]	wyświetl zawartość pamięci począwszy od podanego adresu zgodnie z formatem <i>f</i> może (format można poprzedzić liczbą elementów do wyświetlenia)
 p abc	 <i>wyświetl wartość zmiennej abc</i>
x /1d &abc	<i>wyświetl wartość zmiennej abc (dokładnie wyświetl zawartość pamięci od adresu &amp;abc)</i>
x /2x \$rip	<i>wyświetl fragment pamięci wskazywany przez adres z rejestru RIP (czyli kolejny rozkaz)</i>
 <b>info line</b> [nr]	 wyświetlaj informacje o położeniu w pamięci rozkazów odpowiadających wskazanej linii kodu
<b>disass</b> [func] [start] [, stop]	wyświetl zdeasemblowany fragment kodu (całą funkcję lub tylko podany zakres)
 disass main	
disass main+4, main+20	

# **gdb - sygnały i przeglądanie kodu**

<b>handle</b> <i>signal act</i>	ustal reakcję <i>act</i> gdb na sygnał <i>signal</i> ; możliwe reakcje: <b>print</b> (wyświetl informację o pojawieniu się sygnału), <b>noprint</b> (nie wyświetlaj), <b>stop</b> (zatrzymaj wykonanie), <b>nostop</b> (nie zatrzymuj), <b>pass</b> (pozwól debugowanemu programowi na odebranie sygnału), <b>nopass</b> (nie przekazuj sygnału do programu)
<b>info signals</b>	wyświetl tablicę obsługiwanych sygnałów GDB
<b>list</b>	wyświetl 10 kolejnych linii kodu
<b>list -</b>	wyświetl 10 poprzednich linii kodu
<b>list</b> [ <i>file</i> :] <i>num</i>	wyświetl 10 linii kodu wokół linii <i>num</i> w [pliku <i>file</i> ]
<b>list</b> [ <i>file</i> :] <i>func</i>	wyświetl 10 linii kodu wokół początku funkcji <i>func</i> w [pliku <i>file</i> ]
<b>list</b> <i>f</i> , <i>l</i>	wyświetl kod od linii <i>f</i> do linii <i>l</i>
<b>info sources</b>	wyświetl listę wszystkich plików źródłowych

# Interfejsy GDB

- Uruchomienie semigraficznego interfejsu: `gdb -tui`
- Alternatywna wersja z wygodniejszym interfejsem: `cgdb`
- Liczne „frontendy”, przykładowa lista:  
<https://sourceware.org/gdb/wiki/GDB%20Front%20Ends>

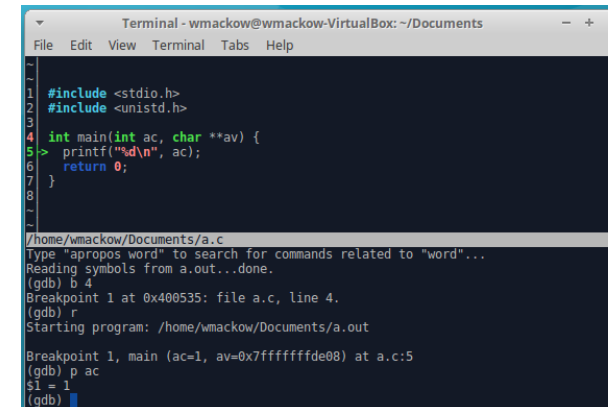


```
Terminal - wmackow@wmackow-VirtualBox: ~/Documents
File Edit View Terminal Tabs Help

a.c
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main(int ac, char **av) {
5     printf("%d\n", ac);
6     return 0;
7 }
8
9
10
11
12
13

native process 29911 in: main
Breakpoint 1 at 0x400535: file a.c, line 4.
(gdb) r a b c
Starting program: /home/wmackow/Documents/a.out a b c

Breakpoint 1, main (ac=4, av=0x7ffffffde8) at a.c:5
(gdb) p ac
$1 = 4
(gdb)
```



```
Terminal - wmackow@wmackow-VirtualBox: ~/Documents
File Edit View Terminal Tabs Help

1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main(int ac, char **av) {
5     printf("%d\n", ac);
6     return 0;
7 }
8
9
10
11
12
13

/home/wmackow/Documents/a.c
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...done.
(gdb) b 4
Breakpoint 1 at 0x400535: file a.c, line 4.
(gdb) r
Starting program: /home/wmackow/Documents/a.out

Breakpoint 1, main (ac=1, av=0x7ffffffde08) at a.c:5
(gdb) p ac
$1 = 1
(gdb)
```

# Inne narzędzia

- **Rats** – statyczna analiza kodu. Analizie poddawany jest kod źródłowy (jeden plik lub katalog), a nie program wynikowy.
  - uruchomienie **rats** `<path_to_source_dir>`
- **Valgrind** – kontrola pamięci (program zewnętrzny)
  - kompilacja programu z dołączeniem tablicy symboli dla gdb **-g** i zerową optymalizacją **-O0**
  - uruchomienie **valgrind** `<program>`
  - domyślnie ustawione narzędzie **memcheck**, można dodać opcję **--leak-check=yes**
- **Memtrace** – kontrola pamięci (biblioteka i program zewnętrzny), zdecydowanie mniej skuteczna niż Valgrind:
  - dołączenie pliku nagłówkowego `<mcheck.h>`
  - wywołanie na początku programu f-cji **mtrace()**;
  - określenie w zmiennych systemowych (bash) nazwy pliku, w którym **mtrace** umieści uzyskane informacje, np.:
    - `$export MALLOC_TRACE=memcheck.log`
  - analiza otrzymanego logu przy pomocy polecenia **mtrace**

# Valgrind (--tools)

- **memcheck** - wykrywa problemy z zarządzaniem pamięcią i jest przeznaczony głównie dla programów w C i C++. Kiedy program jest uruchamiany pod nadzorem Memcheck, wszystkie odczyty i zapisy pamięci są sprawdzane, a wywołania funkcji malloc/new/free/delete są przechwytywane. Memcheck uruchamia programy około 10--30x wolniej niż normalnie.
- **cachegrind** - profiler pamięci podręcznej. Przeprowadza on szczegółową symulację pamięci podręcznej L1, D1 i L2. Cachegrind uruchamia programy około 20-100x wolniej niż normalnie.
- **callgrind** - rozszerzenie Cachegrinda.
- **massif** - profiler sterty. Wykonuje szczegółowe profilowanie sterty poprzez robienie regularnych zrzutów sterty programu
- **helgrind** - debugger wątków, który znajduje wyścigi danych w programach wielowątkowych. Szuka on miejsc w pamięci, do których dostęp ma więcej niż jeden wątek, ale dla których nie można znaleźć żadnej konsekwentnie używanej blokad.
- **DRD** - narzędzie do wykrywania błędów w wielowątkowych programach w C i C++.

# Valgrind (memcheck)

**Valgrind** z opcją **memcheck** może m.in. wykryć, czy program:

- próbuje uzyskać dostęp do pamięci, do której nie powinien (obszary jeszcze nie zaalokowane, obszary, które zostały zwolnione, obszary poza końcem bloków sterty, niedostępne obszary stosu);
- używa niezainicjowanych zmiennych;
- doprowadza do wycieków pamięci;
- niepoprawnie zwalnia bloki sterty (np. dwukrotne zwolnienie);
- przekazuje nakładające się bloki pamięci źródłowej i docelowej do funkcji memcpy() i pokrewnych.

```
#include <stdlib.h>

int main()
{
    char *x = malloc(10);
    x[10] = 'a';
    return 0;
}
```

```
$ valgrind --tool=memcheck --leak-check=yes ./example
...
==9814== Invalid write of size 1
==9814== at 0x804841E: main (example.c:6)
==9814== Address 0x1BA3607A is 0 bytes after a block of size 10 alloc'd
==9814== at 0x1B900DD0: malloc (vg_replace_malloc.c:131)
==9814== by 0x804840F: main (example2.c:5)
```



# Valgrind (helgrind)

```
#include <pthread.h>

int var = 0;

void* child_fn ( void* arg ) {
    var++;
    return NULL;
}

int main ( void ) {
    pthread_t child;
    pthread_create(&child, NULL, child_fn, NULL);
    var++;
    pthread_join(child, NULL);
    return 0;
}
```

```
$ valgrind --tool=helgrind ./example
...
==25298== Thread #1 is the program's root thread
==25298==
==25298== ---Thread-Announcement-----
...
==25298==
==25298== Possible data race during read of size 4 at 0x10C014 by thread #1
==25298== Locks held: none
==25298==    at 0x1091E3: main (in /home/wmackow/Workspaces/PS/valgrind/hellgrind/main)
==25298==
==25298== This conflicts with a previous write of size 4 by thread #2
==25298== Locks held: none
==25298==    at 0x10919E: child_fn (in
/home/wmackow/Workspaces/PS/valgrind/hellgrind/main)
...
```

# Valgrind (callgrind) 1/2

```
#include <vector>
#include <iostream>

using namespace std;

int foo(vector<int> v) {
    int result = 0;
    for(auto x: v) {
        result += x;
    }
    return result % 1000;
}

int main() {
    vector<int> v;
    v.push_back(1);

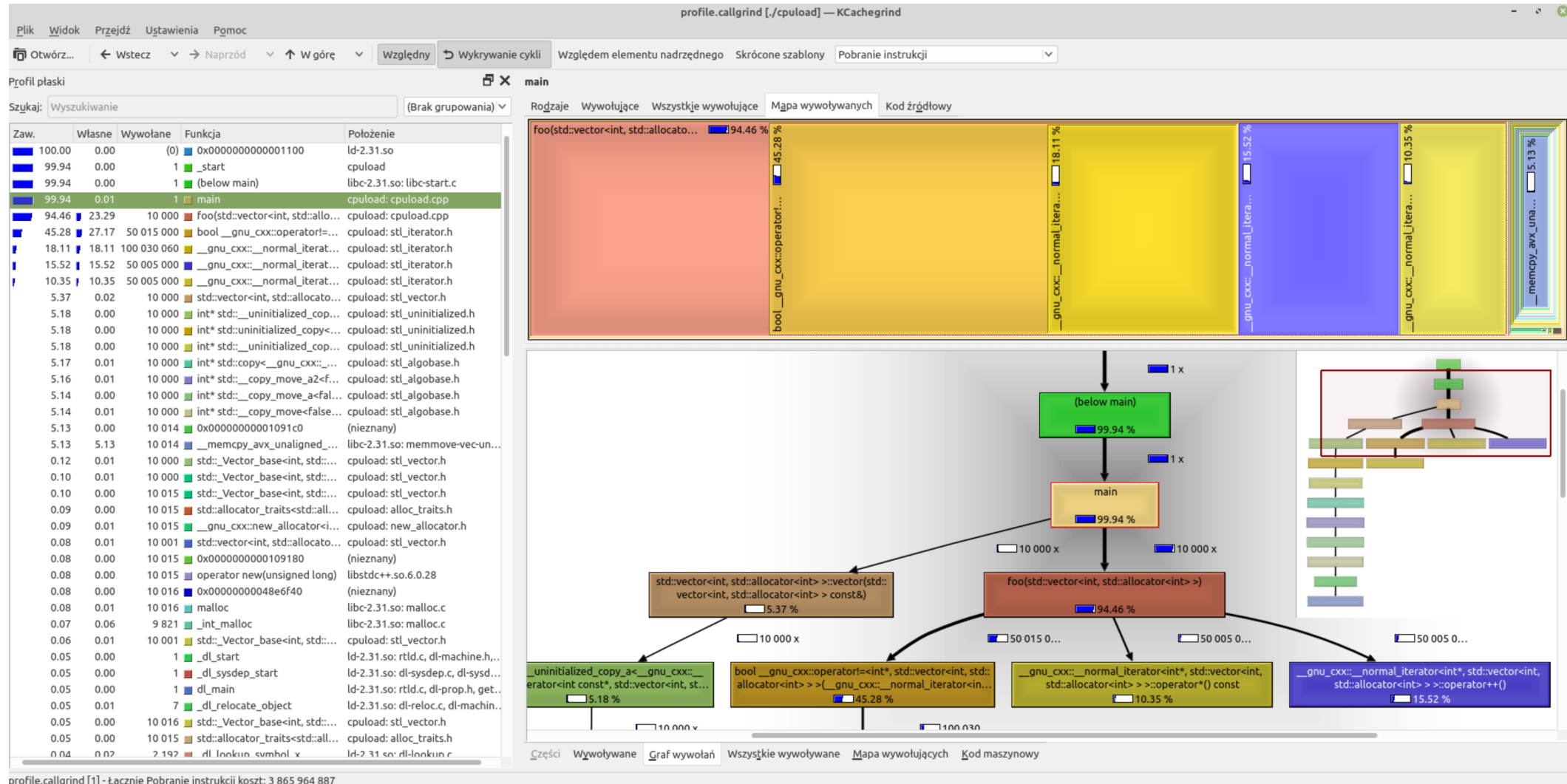
    int result = 0;
    for (int i=0; i<10000; i++) {
        result = foo(v);
        v.push_back(result);
    }
    cout << result << "\n";
    return 1;
}
```

```
$ g++ -std=c++11 cpuload.cpp -o cpuload
$ valgrind --tool=callgrind ./cpuload
$ less profile.callgrind
```

```
# callgrind format
version: 1
creator: callgrind-3.15.0
pid: 25321
cmd: ./cpuload
part: 1
...
```

```
$ kcache-grind profile.callgrind
```

# Valgrind (callgrind) 2/2 - KCachegrind



# gprof (profilowanie)

```
#include <vector>
#include <iostream>
// #include <gperftools/profiler.h>

using namespace std;

int foo(vector<int> v) {
    int result = 0;
    for(auto x: v) {
        result += x;
    }
    return result % 1000;
}

int main() {
    // ProfilerStart("profile.log");
    vector<int> v;
    v.push_back(1);

    int result = 0;
    for (int i=0; i<10000; i++) {
        result = foo(v);
        v.push_back(result);
    }
    // ProfilerStop();
    cout << result << "\n";
    return 1;
}
```

```
$ g++ -std=c++11 -pg cpuload.cpp -o cpuload
$ ./cpuload
$ gprof cpuload
```

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
30.04	0.12	0.12	50015000	0.00	0.00	bool __gnu_cxx::operator!= ...
<b>30.04</b>	<b>0.24</b>	<b>0.12</b>	<b>10000</b>	<b>12.02</b>	<b>39.05</b>	<b>foo(std::vector&lt;int, ...</b>
12.52	0.29	0.05	50005000	0.00	0.00	__gnu_cxx::__normal_iterator...
10.01	0.33	0.04	50005000	0.00	0.00	__gnu_cxx::__normal_iterator...
7.51	0.36	0.03	100030060	0.00	0.00	__gnu_cxx::__normal_iterator...
7.51	0.39	0.03	20030	1.50	1.50	__gnu_cxx::__normal_iterator...
2.50	0.40	0.01				main
...						

Inna alterantywa – [gperftools](#) od Google (do odkomentowania fragmenty kodu zaznaczone na niebiesko)

# Interakcja programu ze środowiskiem wykonania

- Wprowadzenie
- Interfejs wywołań systemowych
- Kompilowanie za pomocą GCC
- Wykorzystanie GNU Make
- Debugowanie za pomocą GDB
- **Interakcja programu ze środowiskiem wykonania**
- Tworzenie i używanie bibliotek

# Przekazywanie parametrów do programu: **argv, argc, environ**

Wykorzystanie argumentów wywołania i zmiennych środowiskowych. **argc** i **argv** to nazwy **zwyczajowe** (mogą być zastąpione dowolną inną nazwą). Zmienna **argc** przechowuje informację o ilości argumentów, z jakimi wywołany został program (nazwa programu liczona jest jako argument), zmienna **argv** to wskaźnik do tablicy argumentów (łańcuchów znaków). Pierwszym elementem tablicy (indeks 0) jest nazwa programu, ostatnim wartość **NULL**.

```
#include <stdio.h>

extern char** environ;

int main( int argc, char **argv) //(..., char *argv[])
{
    char **var;
    for( var = environ; *var != NULL; ++var)
        printf( "%s\n", *var);

    for( int i=0; i<argc; i++)
        printf( "%s\n", argv[ i]);

    return 0;
}
```

# getopt 1/3

```
#include <unistd.h>

extern char *optarg;
extern int optind, opterr, optopt;

int getopt(int argc, char * const argv[], const char *optstring);
```

- Kolejne wywołania funkcji „porządkują” **argumenty**, na początek przestawiane są opcje (argumenty rozpoczynające się od myślnika, opcje w formie „krótkiej”).
- Za pomocą **opstring** określamy m.in. dopuszczalne opcje oraz czy dana opcja musi mieć podaną wartość (wtedy w opstring po literze opcji stawiamy dwukropek ).
- Funkcję wywołujemy w pętli, jeżeli zwróci wartość -1 nie ma już więcej opcji do odczytania.
- Po zakończeniu czytania opcji mamy możliwość odczytania pozostałych argumentów bezpośrednio z **argv** rozpoczynając od indeksu **optind**.

# getopt 2/3

*opttest.c 1/2*

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

int main (int argc, char **argv) {
    int aflag = 0, bflag = 0, ret, index;
    char *cvalue = NULL;

    opterr = 0; //no default "invalid option" info

    while ((ret = getopt (argc, argv, "abc:")) != -1)
        switch (ret) {
            case 'a': aflag = 1; break;
            case 'b': bflag = 1; break;
            case 'c': cvalue = optarg; break;
            case '?':
                if (optopt == 'c')
                    fprintf (stderr, "Option -%c requires an
                        argument.\n", optopt);
                else
                    fprintf (stderr, "Unknown option `-%c'.\n",
                        optopt);
                return 1;
            default: abort ();
        }

    ...
}
```



# getopt 3/3

*opttest.c 2/2*

```
...  
    printf ("aflag = %d, bflag = %d, cvalue = %s\n", aflag, bflag, cvalue);  
  
    printf ("non-option arguments: ");  
    for (index = optind; index < argc; index++)  
        printf (" %s ", argv[index]);  
  
    printf ("\n");  
    for (index = 1; index < argc; index++)  
        printf ("argv[%d]:%s ", index, argv[index]);  
  
    return 0;  
}
```

```
$/opttest x -a -c 10 a b  
aflag = 1, bflag = 0, cvalue = 10  
non-option arguments: x a b  
argv[1]:-a argv[2]:-c argv[3]:10 argv[4]:x argv[5]:a argv[6]:b  
  
$/opttest -c  
Option -c requires an argument.  
  
$/opttest -d  
Unknown option '-d'.
```

# Standardowe wejście i wyjście 1/4

- strumień wejścia: **0**      **STDIN\_FILENO**      **stdin**      (np. **scanf**)
  - strumień wyjścia: **1**      **STDOUT\_FILENO**      **stdout**      (np. **printf**)
  - strumień błędów: **2**      **STDERR\_FILENO**      **stderr**
- 
- **stdout** jest buforowany, **fflush( stdout)** powoduje opróżnienie bufora
  - wstawienie znaku końca linii (np. niejawnie w **puts**, jawnie w **printf**) powoduje opróżnienie bufora
  - f-cje **read**, **write** czy **fprint**, które oczekują jako argumentu uchwytu do pliku, mogą obsłużyć standardowe strumienie, np.:

```
write( 2 /*stderr*/, "abc", 4)
write( STDERR_FILENO /*stderr*/, "abc", 4)
```

- przekierowanie wyjść do plików lub potoków przy wywołaniu programu (przekierowanie **stderr** do **stdout** musi być po przekierowaniu **stdout** do pliku, ale przed przekierowaniem do potoku):

```
$ program > output_file.txt 2>&1
$ program 2>&1 | filter
```

# Standardowe wejście i wyjście 2/4

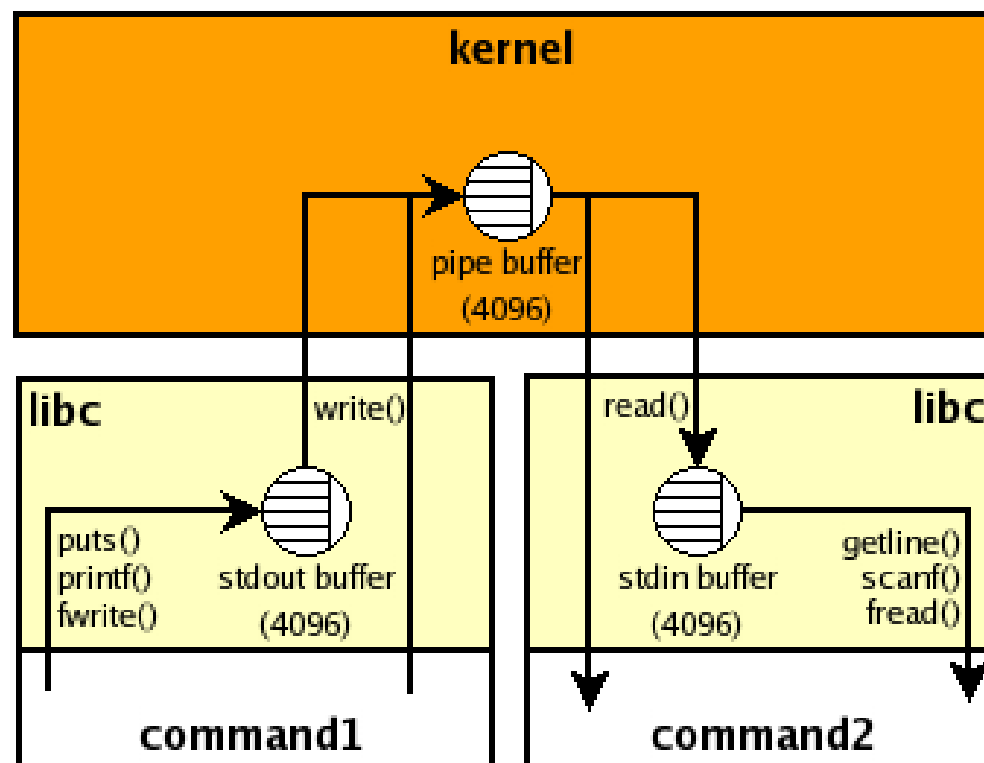
```
...
printf("1:printf ");
puts("2:puts ");
printf("3:printf \n");
printf("4:printf ");
fwrite("5:fwrite ", 1, 9, stdout);
write(STDOUT_FILENO, "6:write \n", 9);
puts("");
...
```

```
$. /stdoutBuff
1:printf 2:puts
3:printf
6:write
4:printf 5:fwrite
```

Buforowanie na poziomie biblioteki standardowej  
(write nie używa bufora)

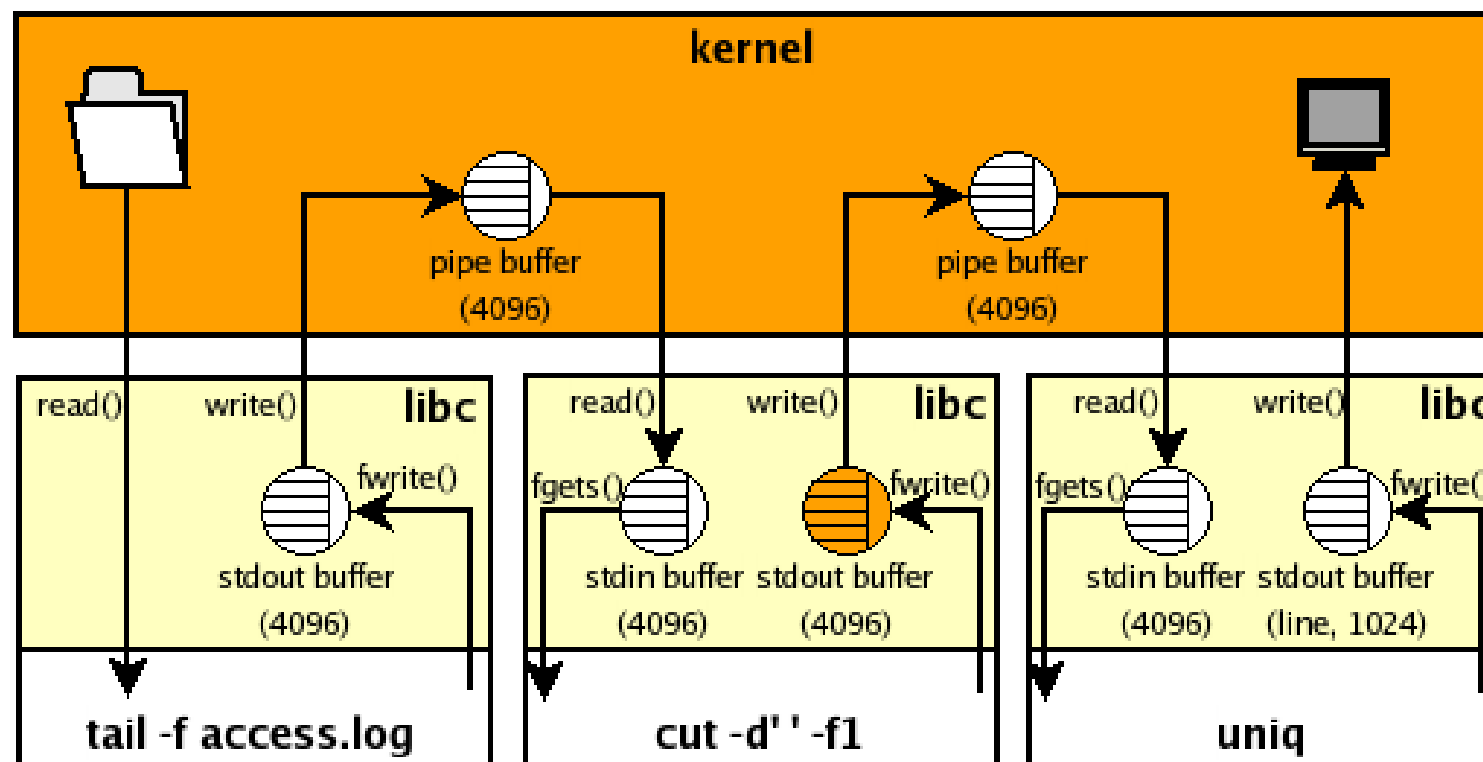
# Standardowe wejście i wyjście 3/4

```
$ command1 | command2
```



# Standardowe wejście i wyjście 4/4

```
$ tail -f access.log | cut -d' ' -f1 | uniq
```



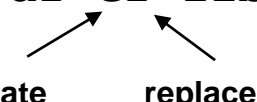
# Tworzenie i używanie bibliotek

- Wprowadzenie
- Interfejs wywołań systemowych
- Kompilowanie za pomocą GCC
- Wykorzystanie GNU Make
- Debugowanie za pomocą GDB
- Interakcja programu ze środowiskiem wykonania
- **Tworzenie i używanie bibliotek**

# Tworzenie biblioteki statycznej

Biblioteka statyczna jest plikiem archiwum (rozszerzenie \*.a) zawierającym zbiór obiektów (rozszerzenie \*.o), które powstały w wyniku kompilacji. Do tworzenia i zarządzania archiwami służy program **ar**.

```
$ ar cr libname.a obj1.o obj2.o obj3.o ...
```



create      replace

*libtest.h*

```
int add( int a, int b);  
void shrink( char *str);
```

*libtest.c*

```
#include "libtest.h"  
  
int add( int a, int b) { return a+b;}  
void shrink( char *str) { str[1]=0;};
```

```
$ gcc -c libtest.c  
$ ar cr libtest.a libtest.o
```

# Użycie biblioteki statycznej

Aby dołączyć bibliotekę statyczną:

- dołączamy w kodzie źródłowym programu odniesienie do pliku nagłówkowego biblioteki
- podczas konsolidacji programu używamy przełącznika `-l` podając nazwę biblioteki (nazwa bez `lib` i `.a`)
- opcjonalnie wykorzystujemy przełącznik `-L` aby wskazać katalogi, które mogą zawierać bibliotekę

**Uwaga:** kolejność wywołania przełączników ma znaczenie! Biblioteki należy dołączać na końcu linii poleceń, ponieważ linker przeszukuje je pod kątem wszystkich symboli, do których były odniesienia w przetworzonych wcześniej plikach i które nie zostały jeszcze zdefiniowane. Fragmenty kodu z bibliotek są kopiowane do pliku wynikowego.

*static.c*

```
#include "libtest.h"

int main()
{
    int c = add( 10, 20);
    return 0;
}
```

```
$ ls
libtest.a  static.c
$ gcc static.c -o static -L. -ltest
```



# Tworzenie biblioteki współdzielonej 1/2

Biblioteka współdzielona jest zbiorem obiektów o rozszerzeniu \*.so.#1.#2.#3 (numery #1, #2 i opcjonalne #3 oznaczają numery wersji – począwszy od najbardziej istotnego). Jest on linkowana przy pomocy `gcc` z opcjami `-shared` i `-fPIC` (opcja `-fPIC` również przy kompilacji obiektów). W kodzie biblioteki nie może być f-cji `main`, mogą być za to `_init()` i `_fini()` (po ustawieniu opcji `-nostartfiles`).

```
$ gcc -shared -fPIC -o libname.so.0.0.0 obj1.o obj2.o obj3.o ...
```

*libtest.h*

```
int add( int a, int b);  
void shrink( char *str);
```

*libtest.c*

```
#include "libtest.h"  
#include <stdio.h>  
  
int add( int a, int b) { return a+b;}  
void shrink( char *str) { str[1]=0;}  
_init() { printf( "connect ...\n");}  
_fini() { printf( "... free\n");}
```

```
$ gcc -c -fPIC libtest.c  
$ gcc -shared -nostartfiles -fPIC libtest.o -o libtest.so.0.1
```

Procesy używające biblioteki współdzielonej współdzielą kod funkcji, natomiast mają własne kopie danych (o ile były np. w postaci zmiennych globalnych deklarowane w bibliotece). Do danych używany jest mechanizm copy-on-write, znany również z niektórych implementacji f-cji fork.

## Tworzenie biblioteki współdzielonej 2/2

Funkcje `_init()` i `_fini()` traktowane są obecnie jako przestarzałe i potencjalnie niebezpieczne, w ich miejsce zaleca się użycie atrybutów **constructor** i **destructor** w następujący sposób:

```
void __attribute__((constructor)) my_init(void);  
void __attribute__((destructor)) my_fini(void);
```

gdzie `my_init` i `my_fini` są nazwami naszych funkcji inicjującej i czyszczącej.

Jeżeli używamy powyższych atrybutów to podczas kompilacji **gcc** nie możemy użyć przełączników:

- nostartfiles
- nostdlib

# Użycie biblioteki współdzielonej – łączenie 1/2

Aby dołączyć bibliotekę współdzieloną podczas konsolidacji wykonujemy **dokładnie** te same czynności co przy dołączaniu biblioteki statycznej. Opcjonalnie możemy w kod wynikowy wkompiłować informację o ścieżce, gdzie program będzie miał szukać bibliotekę podczas uruchamiania (opcja `-Wl, -rpath, path`). Kod programu nie zawiera kodu bibliotecznego, a jedynie odnośniki do niego. Biblioteka jest ładowana przy uruchamianiu programu. Każde podłączenie biblioteki powoduje wywołanie f-cji `_init()`, odłączenie - wywołanie f-cji `_fini()`.

*sshared.c*

```
#include "libtest.h"

int main()
{
    int c = add( 10, 20);
    return 0;
}
```

```
$ ls
libtest.so.0.1      sshared.c
$ gcc sshared.c -o sshared -L. -ltest -Wl,-rpath,.
```

# Użycie biblioteki współdzielonej – łączenie 2/2

Podczas kompilacji i konsolidacji **gcc**, jeżeli ma do wyboru wersję statyczną i wersję współdzieloną danej biblioteki, to domyślnie używa (łączy) bibliotekę w wersji współdzielonej (np. bibliotekę standardową). Jeżeli chcemy wymusić użycie bibliotek statycznych, to używamy przełącznika **-static**.

*main.c*

```
#include <stdio.h>

int main()
{
    puts("test");
    return 0;
}
```

```
$ gcc main.c -o mainShared
$ gcc -static main.c -o mainStatic
$ ls -l
razem 908
-rw-r--r-- 1 wmackow users      61 mar  5 12:16 main.c
-rwxr-xr-x 1 wmackow users  8600 mar  5 12:24 mainShared
-rwxr-xr-x 1 wmackow users 912720 mar  5 12:24 mainStatic
$ ./mainStatic
test
$ ./mainShared
test
```

# Użycie biblioteki współdzielonej – ładowanie 1/2

Bibliotek współdzielona może zostać załadowana dynamicznie podczas działania programu. Służą do tego funkcje biblioteki **dl**:

1) ładowanie biblioteki

```
void *dlopen(const char *file, int mode);
```

2) obsługa błędów

```
char *dlerror(void);
```

3) ładowanie symboli (m.in. funkcji)

```
void *dlsym(void *restrict handle, const char *restrict name);
```

4) zwolnienie biblioteki

```
int dlclose(void *handle);
```

Funkcje wymagają dołączenia do programu pliku nagłówkowego `<dlfcn.h>` oraz biblioteki **dl**. Każde załadowanie biblioteki powoduje wywołanie f-cji *inicjującej*, zwolnienie biblioteki - wywołanie f-cji *\_czyszczącej*.

## Użycie biblioteki współdzielonej – ładowanie 2/2

*dshared.c*

```
#include <dlfcn.h>

int ( *Add)( int, int);

int main()
{
    void *handle = dlopen( "./libtest.so.0.1", RTLD_LAZY);
    if( !handle)
        dLError();
    else
    {
        Add = dlsym( handle, "add");
        int c = Add( 10, 20);
        dlclose( handle);
    }
    return 0;
}
```

```
$ gcc dshared.c -o dshared -ldl
```

# gcc -nostartfiles

**-nostartfiles** - podczas linkowania nie używaj standardowych plików startowych. Standardowe biblioteki są używane normalnie.

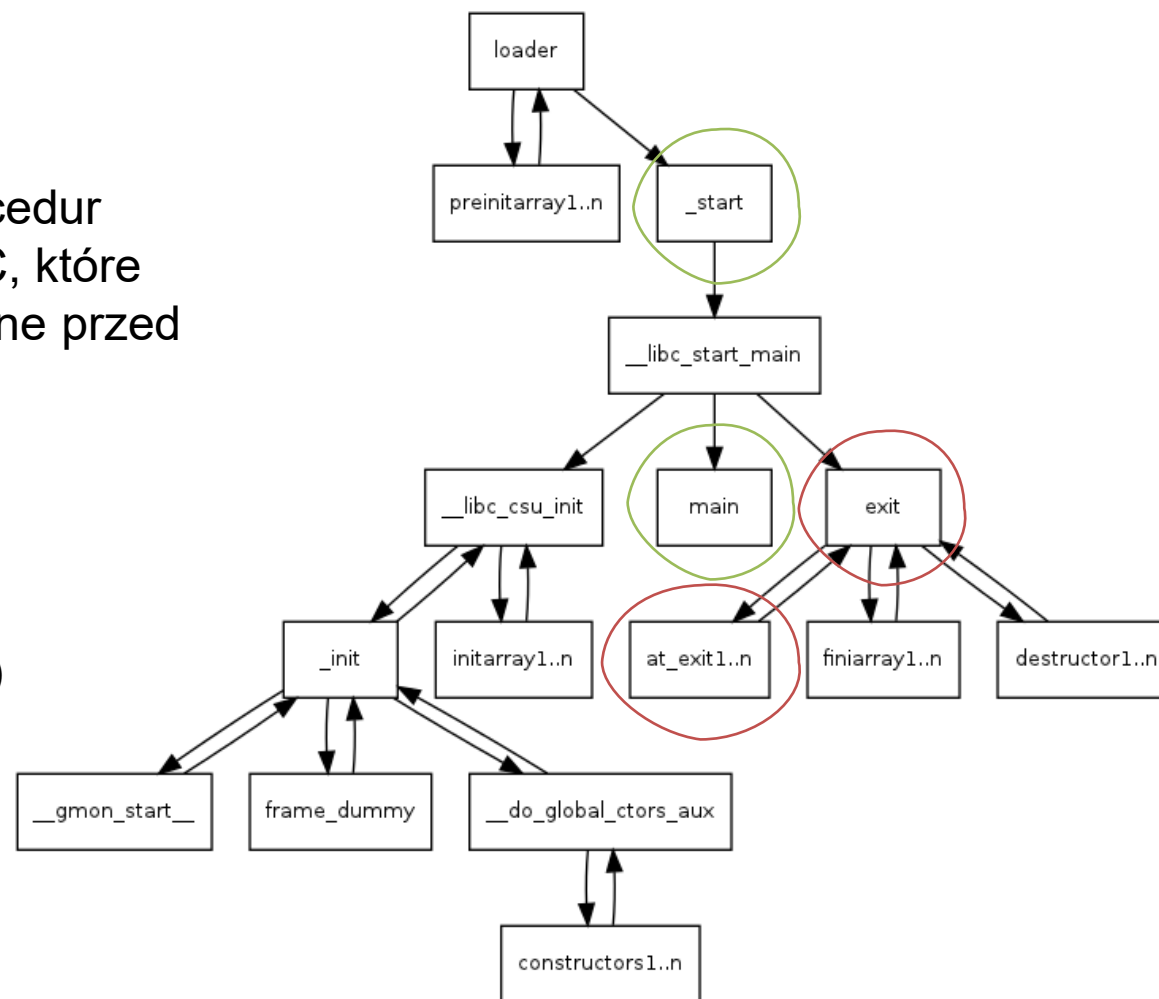
**-nodefaultlibs** - nie używaj standardowych bibliotek systemowych podczas linkowania (dołączane są jedynie biblioteki jawnie wskazane). Standardowe pliki startowe są używane normalnie

**-nostdlib** - nie używaj standardowych systemowych plików startowych ani bibliotek podczas linkowania. Do linkera nie będą przekazywane żadne pliki startowe i tylko te biblioteki, które zostaną jawnie podane.

# C Runtime

„crt0 (znany również jako c0) jest zestawem procedur startowych dołączonych do programu w języku C, które wykonują wszelkie prace inicjalizacyjne wymagane przed wywołaniem głównej funkcji programu.”

- pliki startowe: **crt0.o**, crt1.o, crtbegin.o, crtn.o
- **crt0.o** zawiera m.in. funkcję **\_start** (symbol będący domyślny punkt wejścia dla linkera **ld**)
- **main** jest pierwszą funkcją programu tylko pozornie





# CRT 1/2

Przykładowa, uproszczona funkcja **\_start**, którą zastąpimy funkcję domyślną.

*mycrt0.s*

```
.text

.globl _start

_start: # _start is the entry point known to the linker
    xor %ebp, %ebp          # effectively RBP := 0, mark the end of stack frames
    mov (%rsp), %edi        # get argc from the stack (implicitly zero-extended to 64-bit)
    lea 8(%rsp), %rsi       # take the address of argv from the stack
    lea 16(%rsp,%rdi,8), %rdx # take the address of envp from the stack
    xor %eax, %eax         # per ABI and compatibility with icc
    call main              # %edi, %rsi, %rdx are the three args (of which first two are C standard) to main

    mov %eax, %edi          # transfer the return of main to the first argument of _exit
    xor %eax, %eax         # per ABI and compatibility with icc
    call _exit             # terminate the program
```

```
$ gcc mycrt0.s -c
$ ls
mycrt0.o  mycrt0.s
```

# CRT 2/2

```
$ gcc main.c -c
$ ls
main.c  main.o  mycrt0.o  mycrt0.s

$ gcc main.o -o main1
$ gcc main.o -o main2 -nostartfiles
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to 0000000000400340
$ gcc main.o mycrt0.o -o main3 -nostartfiles

$ ./main1
abcd
$ ./main2
abcd
Naruszenie ochrony pamięci (zrzut pamięci)
$ ./main3
abcd

$ gdb main2
...
Reading symbols from main2...(no debugging symbols found)...done.
(gdb) disass main
Dump of assembler code for function main:
    0x0000000000400340 <+0>:    push    %rbp
    0x0000000000400341 <+1>:    mov     %rsp,%rbp
...
```

*main.c*

```
#include <stdio.h>

int main() {
    printf("abcd\n");
    return 0;
}
```

# gcc -e

*main.c*

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

int test1();
int test2();

void _start() {
    test1();
    exit(1);
}

void _another_start() {
    test2();
    exit(2);
}

int test1() {
    printf("test1\n");
    return 0;
}

int test2() {
    printf("test2\n");
    return 0;
}
```

Jawne wskazanie punktu wejścia (czyli odpowiednika funkcji **\_start** a nie **main**) przez użycie parametru **-e** w **gcc**

```
$ gcc main.c -o main1 -nostartfiles
$ gcc main.o -o main2 -nostartfiles -e _another_start

$ ./main1
test1
$ ./main2
test2
```