

PODSTAWY PROGRAMOWANIA W SYSTEMIE OPERACYJNYM UNIX / LINUX

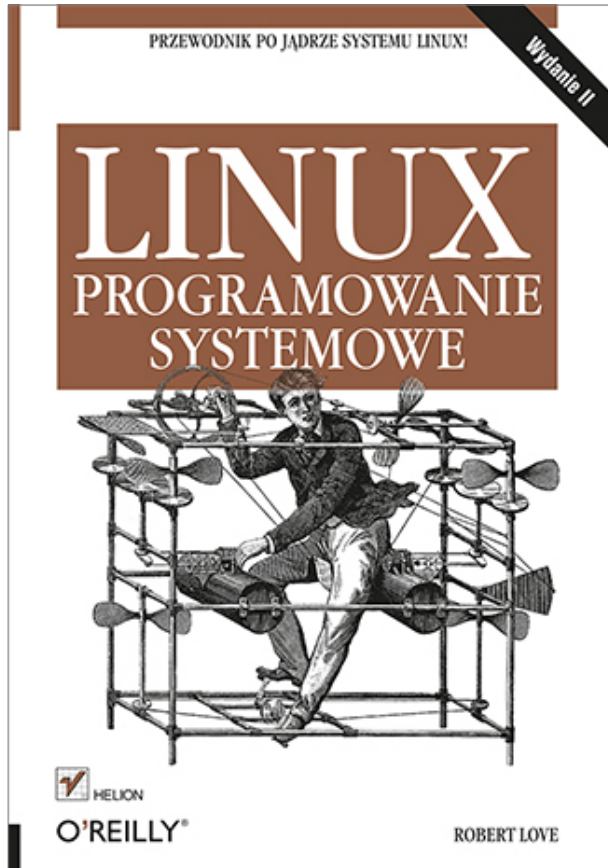
Plan wykładu

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

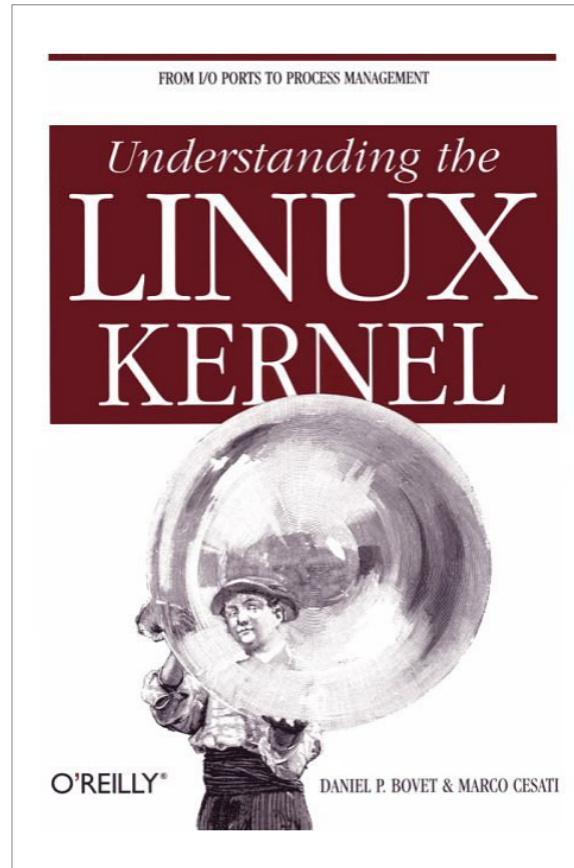
Wprowadzenie

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

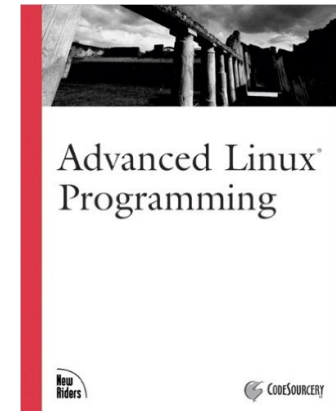
Literatura



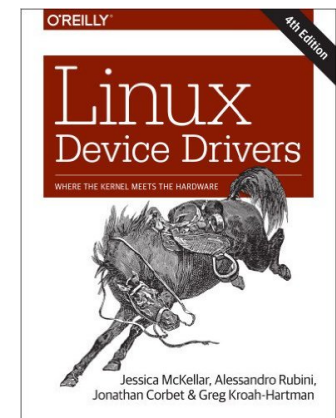
R. Love



D. Bovet, M. Cesati



M. Mitchell, J. Oldham, A. Samuel

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

UNIX - najważniejsze „wydania”

Bell Labs firmy AT&T

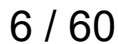
- 1969 – minikomputery PDP-7 i PDP-9 (DEC)
- 1971 – system przepisany na PDP-11/20
- V3: 1973 (potoki, przepisany w języku **C**)
- V6: 1975 (rozprowadzany nieodpłatnie na uczelniach)
- V7: 1979 (licencjonowany, przenośny, do czasu powstania POSIX standard *de facto*)

Uniwersytet Berkeley

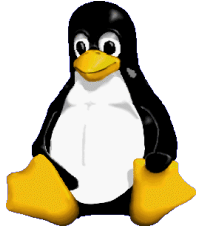
- 1975 - system **1BSD** (*Berkeley Software Distribution*)
- **3BSD**: 1979/1980 - system przepisany na VAX
- **4BSD / 4.1BSD**: 1980-1982 - na zamówienie DARPA dla ARPANET
- **4.2BSD**: 1983 - pierwszy system zawierający obsługę TCP/IP



PDP-11



Klony Unixa



Linux

- Napisany w 1991 r. przez Linusa Torvaldsa
 - Najbardziej popularna odmiana Unixa
 - Rozpowszechniany na zasadach licencji GNU
-



4.4BSD Lite (ostatnia wersja Berkeley, okrojona z kodu AT&T)

- **FreeBSD** (1993, dedykowany na PC) -> **TrustedBSD**
 - **NetBSD** (1993, zorientowany na sieci komputerowe)
 - **OpenBSD** (1996, zorientowany na bezpieczeństwo)
 - Wolny od licencyjnych
 - Rozwój mniej scentralizowany
-

Wiele komercyjnych: **AIX, Irix, Solaris, ...**

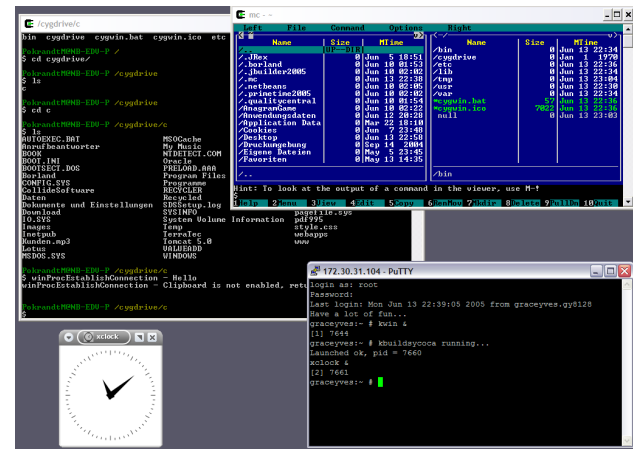
Interfejs a implementacja

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)



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.:

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

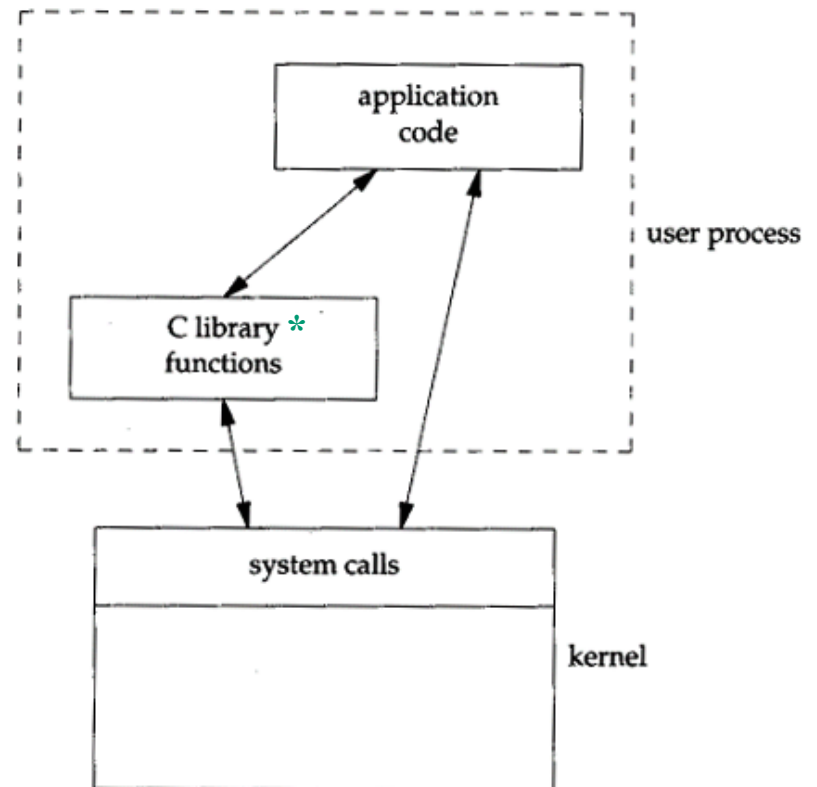
Single UNIX Specification (SUS) – inicjatywa **The Open Group** (poza IEEE), w dużej mierze zbieżna z POSIX. Dokumentacja ta zyskała bardzo dużą popularność dzięki bardzo wysokim cenom dokumentacji POSIX.

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**.
- 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

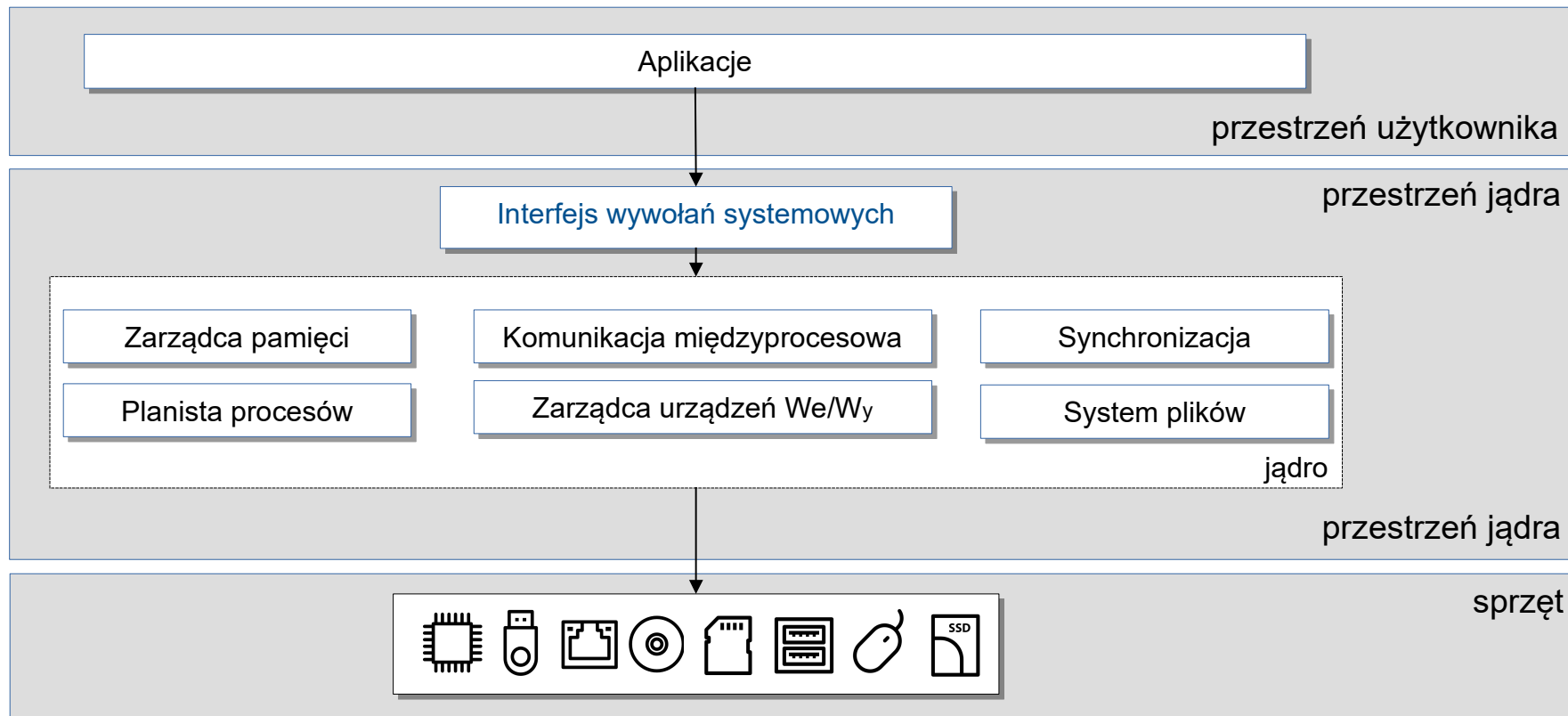


* biblioteka standardowa (libc), w systemie Linux GNU C Library (glibc)

Interfejs wywołań systemowych

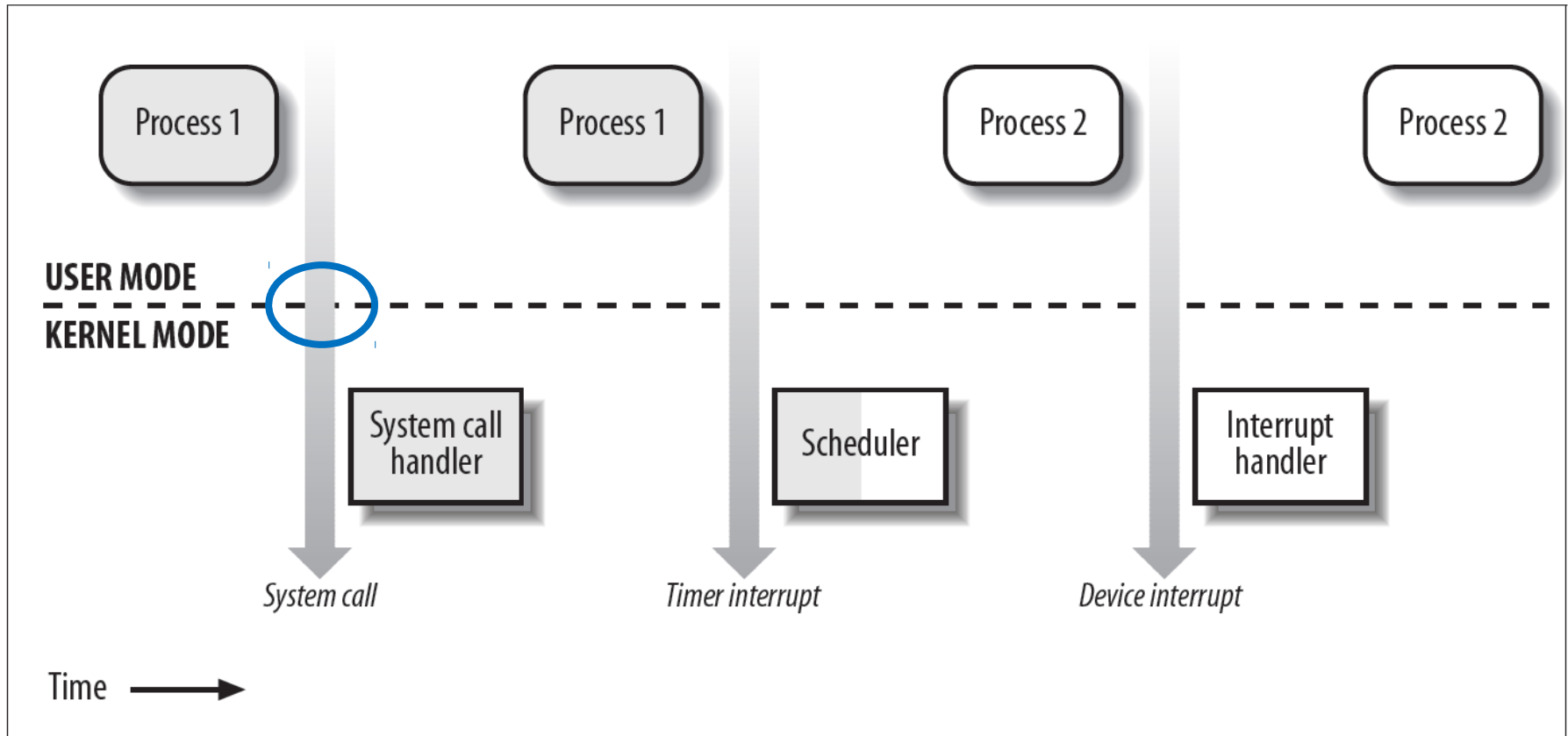
- Wprowadzenie
- **Interfejs wywołań systemowych**
- Kompilowanie za pomocą GCC
- Wykorzystanie GNU Make
- Debugowanie za pomocą GDB
- Interakcja programu ze środowiskiem wykonania
- Tworzenie i używanie bibliotek

Architektura monolityczna



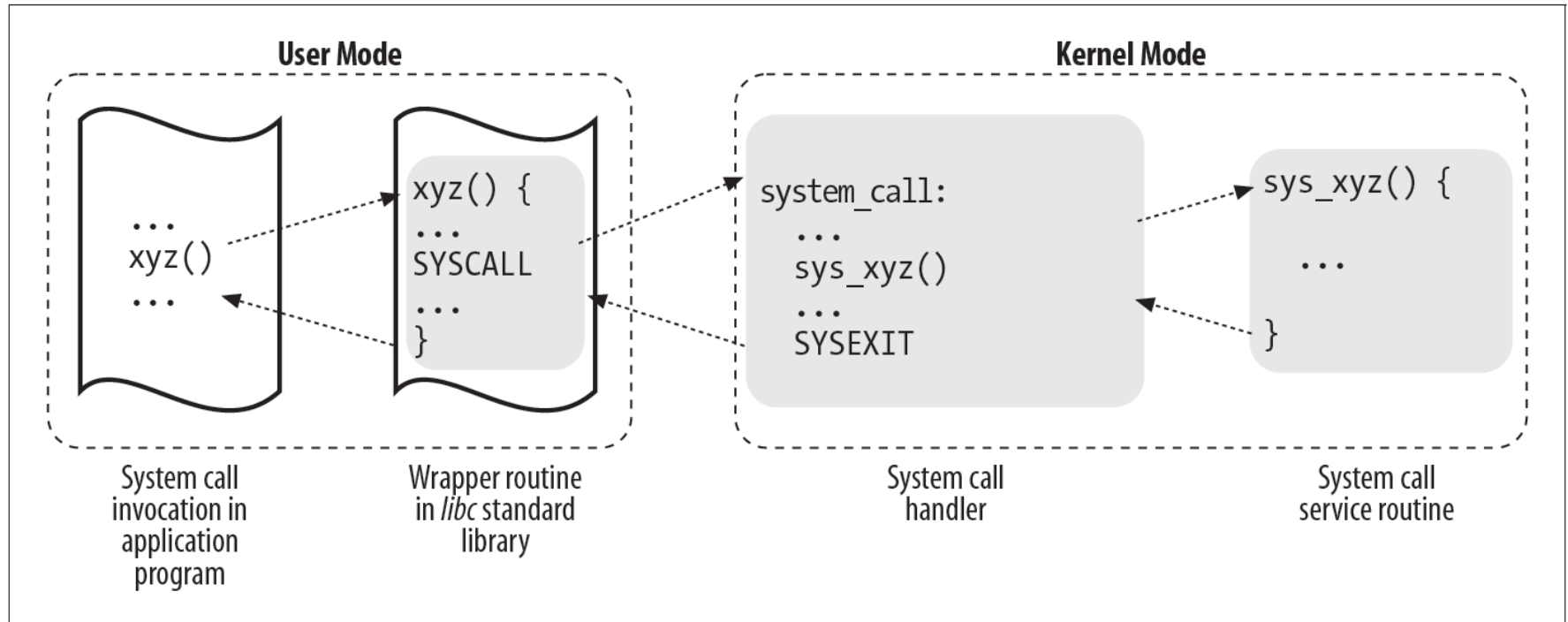
- ochrona sprzętowa (wsparcie ze strony procesora, ringi/poziomy uprzywilejowania)
- dualny tryb pracy SO (tryb użytkownika – tryb jądra)
- system przerwań (przełączanie między trybami)
- ochrona pamięci, operacji We/Wy, czasu procesora

Tryb użytkownika vs tryb jądra



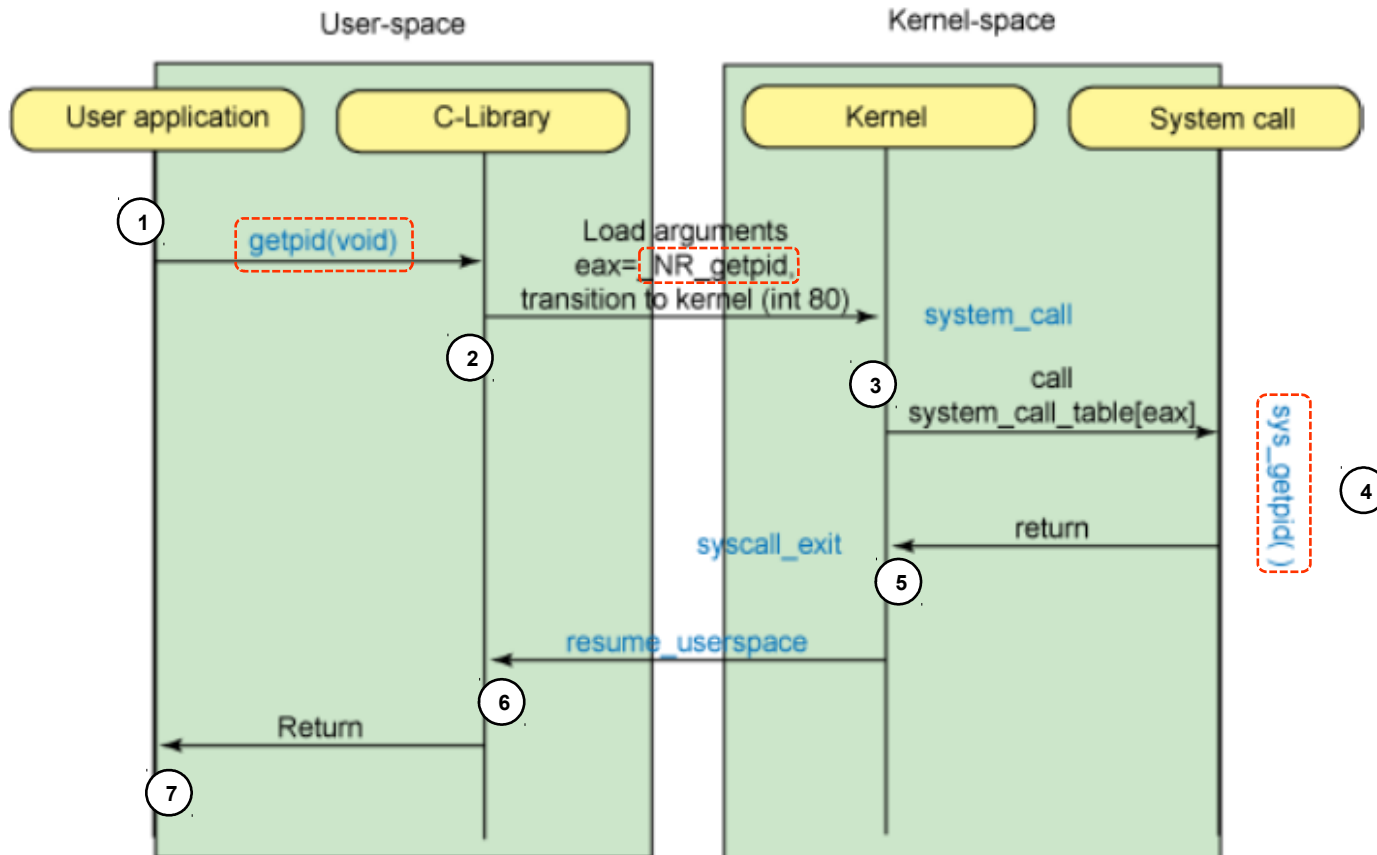
Część pracy jest wykonywana przez jądro na rzecz procesów użytkownika – dostęp do nich uzyskujemy m.in. wykorzystując **wywołania systemowe**.

Interfejs wywołań systemowych 1/3



aplikacja > biblioteka systemowa > obsługa **system_call** > obsługa wywołań

Interfejs wywołań systemowych 2/3



Uproszczony schemat wywołania `getpid`

Interfejs wywołań systemowych 3/3

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

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);
}
```

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>`.

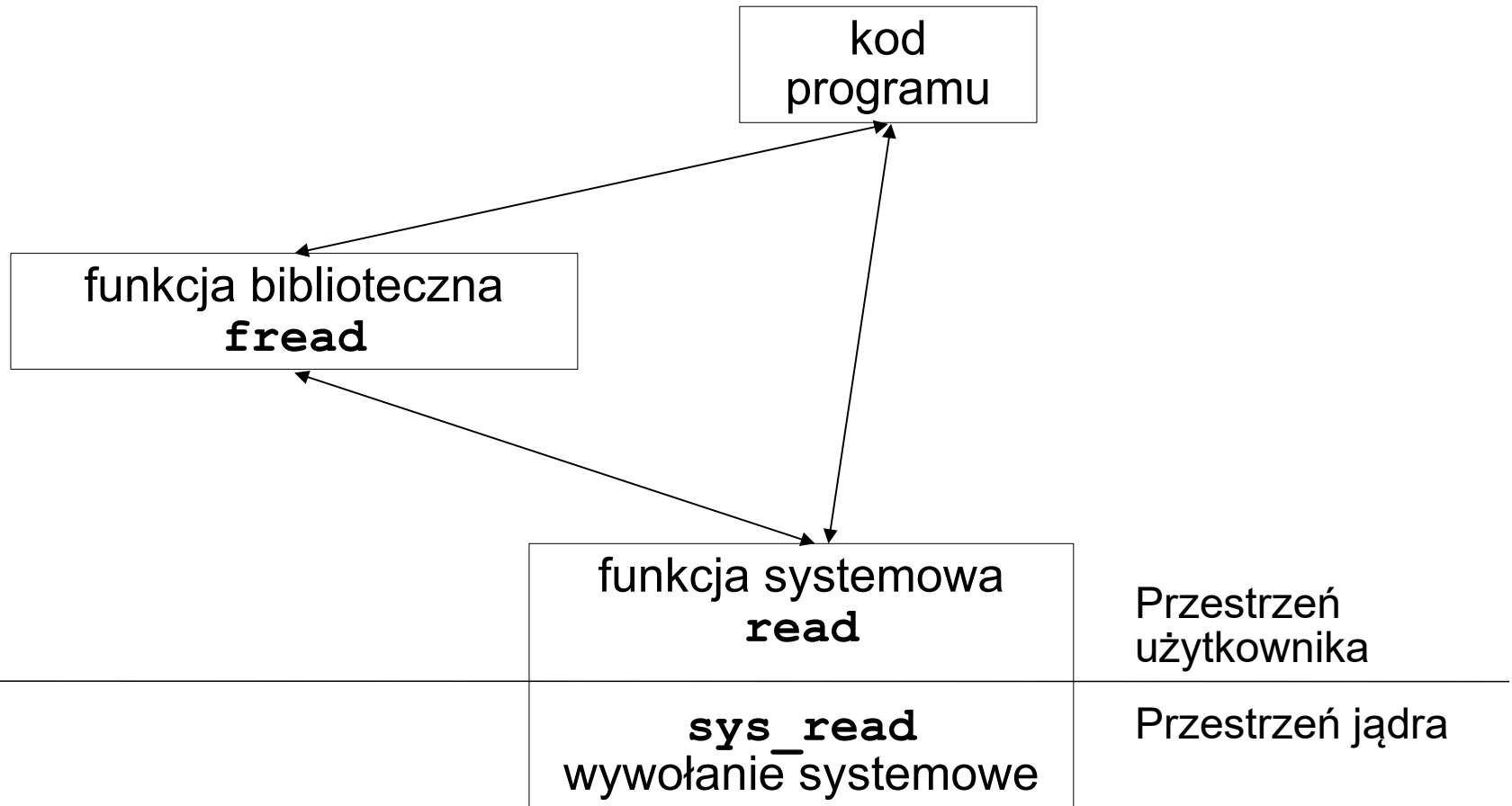
test-err.c

```
#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

...

Kompilowanie za pomocą GCC

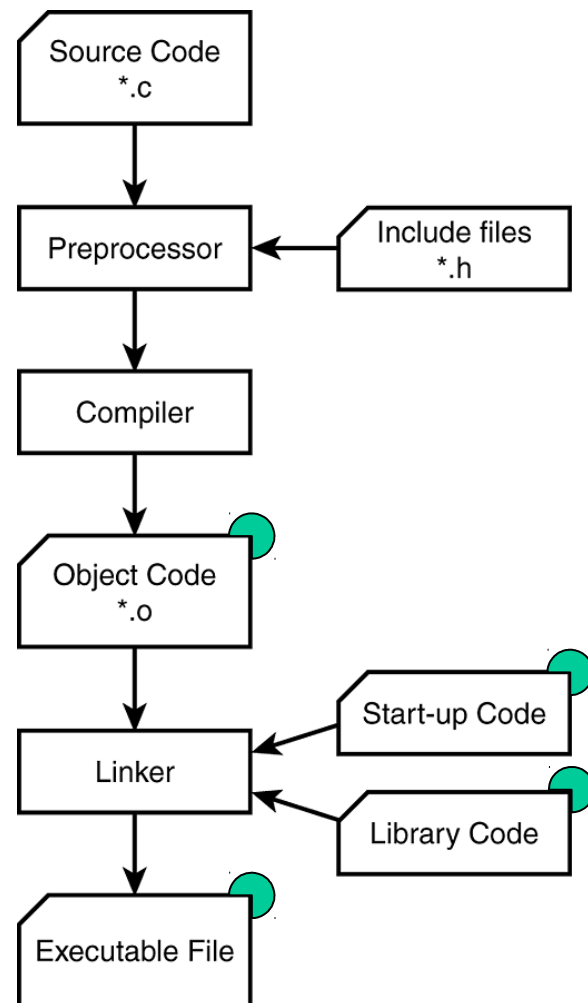
- Wprowadzenie
- Interfejs wywołań systemowych
- **Kompilowanie za pomocą GCC**
- Wykorzystanie GNU Make
- Debugowanie za pomocą GDB
- Interakcja programu ze środowiskiem wykonania
- Tworzenie i używanie bibliotek
-

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**
 - preprocesorowanie (ang. *preprocessing*)
 - kompilacja (ang. *compilation*)
 - asemblacja (ang. *assembling*)
 - konsolidacja (ang. *linking*)

GCC – rozpoznawane rozszerzenia plików

.c	źródła C
.C	źródła C++
.cc	źródła C++
.cxx	źródła C++
.c++	źródła C++
.i	preprocessed C
.ii	preprocessed C++
.s	źródła assemblera
.S	źródła assemblera
.h	pliki nagłówkowe
.o	pliki typu obiekt
.a	biblioteki
.so	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
```

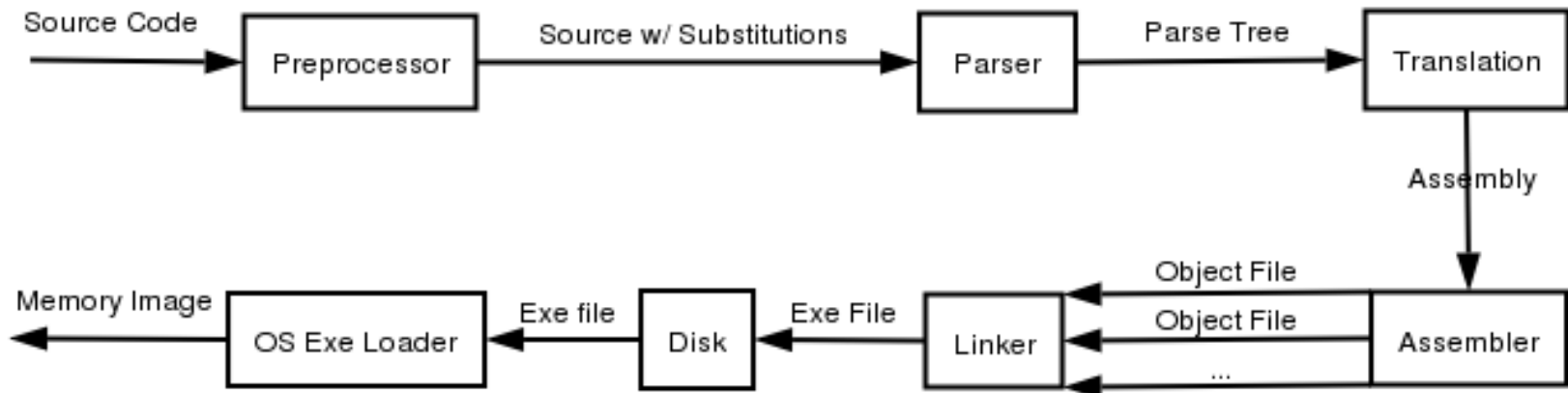
```
-rwxr-xr-x  1 burek users  11337   2007-02-22   20:05  a.out
-rwxr-xr-x  1 burek users  11337   2007-02-22   20:11  prostokat
-rw-r--r--  1 burek users    214   2007-02-22   20:04  prostokat.c
```

```
$ ./prostokat
```

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


GCC – główne opcje

- E Tylko preprocessing, 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 (przy kompilacji standardowo dostajemy 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 (include file).
- `-L dir` Dodaje katalog *dir* do listy katalogów przeszukiwanych przy użyciu przełącznika `-l` (patrz opcje linkera)

GCC – inne opcje 2/2

Opcje linkera:

- l*library*** 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 i startowych plików kiedy linkujesz. Używaj tylko wskazane.
-

Opcje optymalizacji:

- O** Optymalizacja.
- O*level*** 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;
```

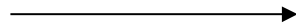
```
    pole = x*y;
```

```
    printf( " obwod = %d, pole = %d \n", obwod, pole);
```

```
    return( 0 );
```

```
}
```

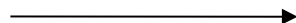
linia 10



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

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

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

```
$ vi mainprog.c
$ make
gcc -c mainprog.c
gcc -o outprogname 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
```

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 -----> ./prostokat
$ gcc -g -c mainprog.c
$ gcc -g -c binarytree.c
$ gcc -g -o outprogrname binarytree.o mainprog.o -----> ./outprogrname
```

Uruchomienie gdb

bez argumentów

```
$ gdb
```

z nazwą programu

```
$ gdb outprogrname
```

z nazwą programu i **pidem** procesu

```
$ gdb outprogrname 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 [<i>file</i>]	użyj plik <i>file</i> jako plik wykonywalny i tablicę symboli
core [<i>file</i>]	użyj plik <i>file</i> jako zrzut pamięci
load [<i>file</i>]	dołącz dynamicznie plik <i>file</i>
info share	wyświetl nazwy wszystkich aktualnie załadowanych bibliotek dzielonych

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

gdb - wykonanie programu

continue, c [*count*]

kontynuuj wykonanie; jeżeli określono *count* to ignoruj *count* kolejnych breakpointów

step, s [*count*]

wykonaj pojedynczy krok (wchodzi do wnętrza funkcji); powtórz *count* razy

next, n [*count*]

wykonaj pojedynczy krok; powtórz *count* razy

until *location*

wykonaj aż do lokalizacji *location* (np. nr linii)

finish

wykonaj aż do zakończenia aktualnej f-cji i wyświetl zwróconą wartość

jump *line*

wznów wykonanie od linii *line*

set *var=expr*

zapisz do zmiennej o nazwie *var* wartość *expr*

backtrace

wyświetl wszystkie ramki ze stosu (m.in. informacja o wywołanych funkcjach)

gdb - wyświetlanie danych

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

display [/f] [expr]

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

display

wyświetl listę wszystkich obserwowanych zmiennych

undisplay *n*

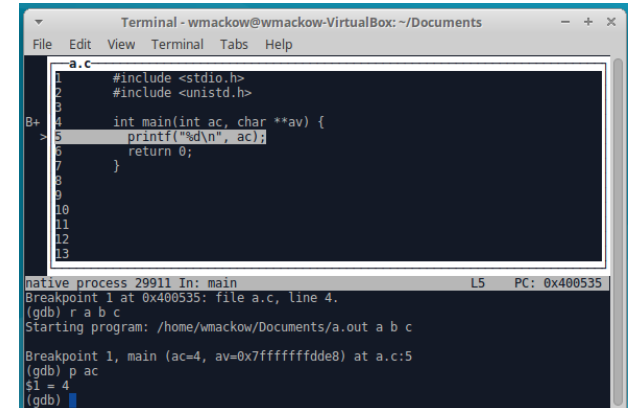
usuń z listy obserwowanych zmiennych *n*-te wyrażenie

gdb - sygnały i przeglądanie kodu

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

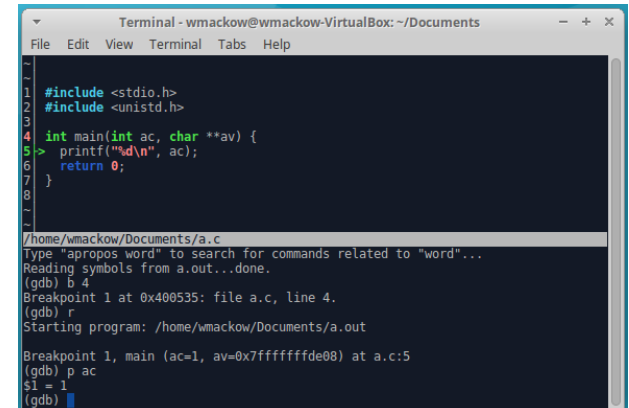


```
Terminal - wmackow@wmackow-VirtualBox: ~/Documents
File Edit View Terminal Tabs Help

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=0x7fffffffdde8) 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=0x7fffffffdde8) 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**

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**.

errtest.c

```
#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 (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_long, getopt_long_only -> #include <getopt.h>
```

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

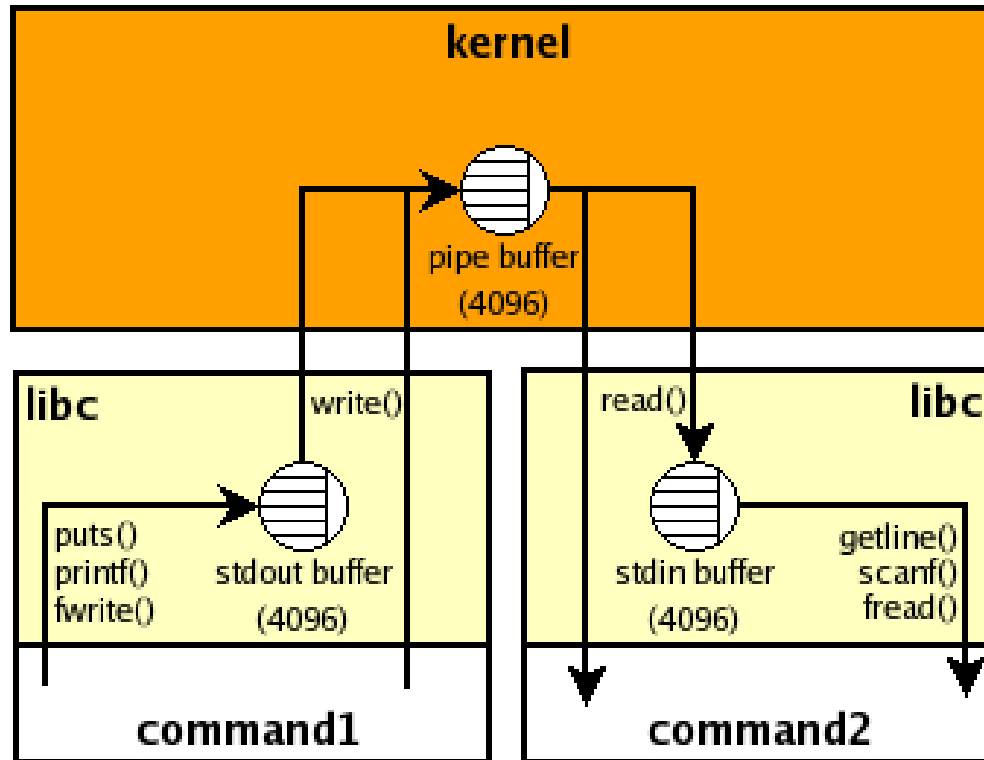
stdoutBuff.c

```
...  
    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  
$
```

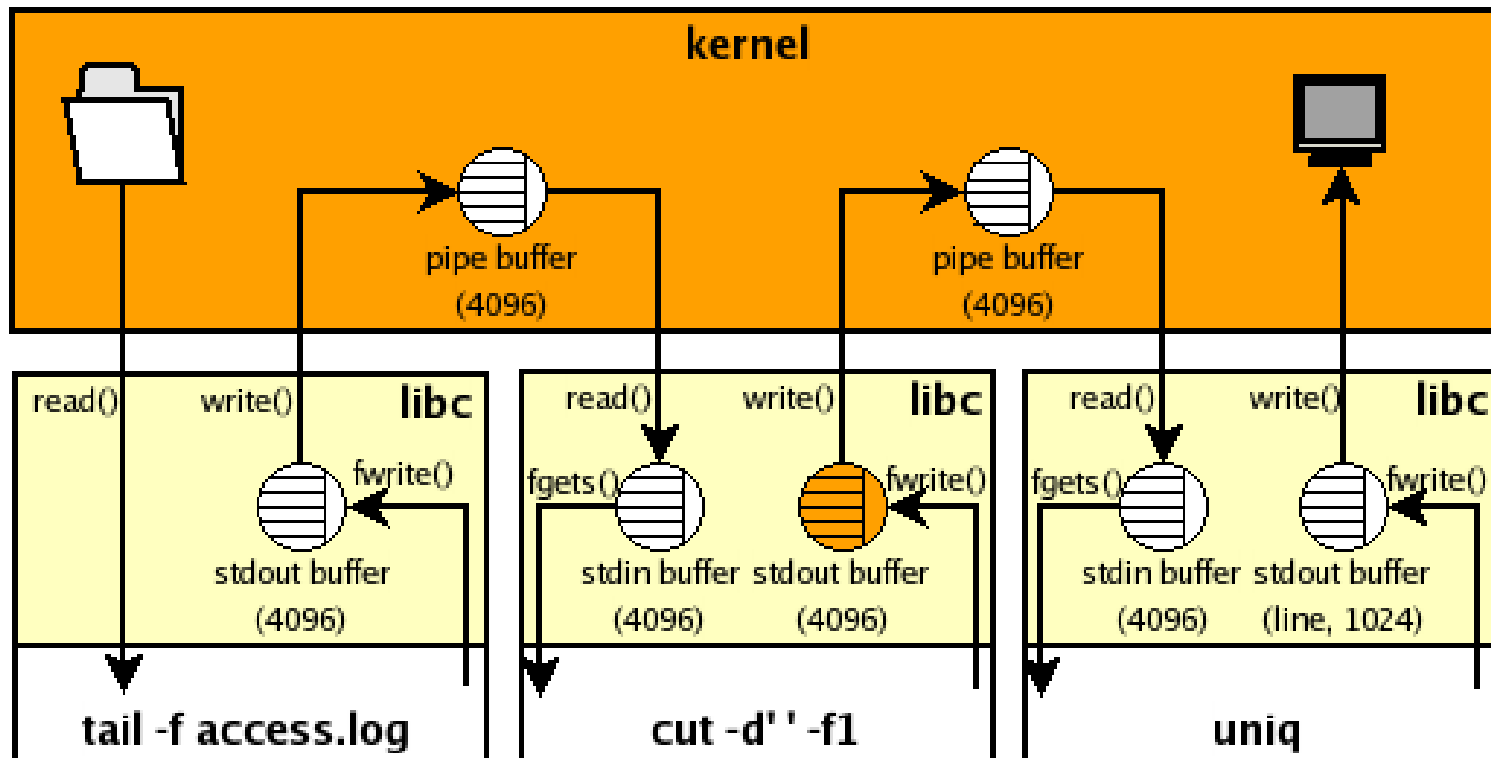
Standardowe wejście i wyjście 3/4

```
$ command1 | command2
```



Standardowe wejście i wyjście 4/4

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



Tworzenie i używanie bibliotek

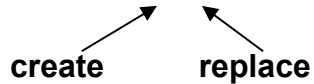
- Wprowadzenie
- Interfejs wywołań systemowych
- Kompilowanie za pomocą GCC
- Wykorzystanie GNU Make
- Debugowanie za pomocą GDB
- Interakcja programu ze środowiskiem wykonania
- **Tworzenie i używanie bibliotek**

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

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 -fPIC libtest.o \  
  -o libtest.so.0.1 -nostartfiles
```

Użycie biblioteki współdzielonej - łączenie

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 – ładowanie (1)

Bibliotek współdzielona może zostać załadowana dynamicznie podczas działania programu. Służą do tego f-cje 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 `_init()`, zwolnienie biblioteki - wywołanie f-cji `_fini()`.

Użycie biblioteki współdzielonej – ładowanie (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