

Oznacz jako wykonane

## Optymalizacja

(<https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>)

**-O**

Kompilator próbuje optymalizować kod i czas wykonania, bez wykonywania operacji znacząco zwiększających czas komplikacji

**-O1**

Możliwy dłuższy czas komplikacji, zużywa dużo pamięci dla dużych funkcji

**-O2**

Używane jeszcze więcej opcji komplikacji, które nie powodują zwiększenia pamięci programu, w porównaniu do -O zwiększa czas komplikacji i wydajność kodu, używa wszystkich flag używanych przez -O także dodatkowe flagi

**-O3**

Używa wszystkich flag używanych przez -O2, a także dodatkowe flagi

**-O0**

Ogranicza czas komplikacji, wartość domyślna

**-Os**

Optymalizacja rozmiaru używa wszystkich flagi -O2, które nie zwiększają kodu programu, używa też dodatkowych flag zmniejszających rozmiar kodu

## Pomiar czasu

### Funkcje czekania

- ```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```
- ```
#include <time.h>
int nanosleep(const struct timespec *req, struct timespec *rem);
```

```
struct timespec {
    time_t tv_sec; /* seconds */
    long tv_nsec; /* nanoseconds */
};
```

### Zegary POSIX

- Typ danych `clock_t` – reprezentuje takty zegara
- Typ danych `clockid_r` – reprezentuje określony zegar Posix
- Są 4 rodzaje zegarów – zalecany to `CLOCK_REALTIME` – ogólnosystemowy zegar czasu rzeczywistego

```
#include <time.h>
int clock_getres(clockid_t clk_id, struct timespec *res) – odczytuje rozdzielcość zegara wyspecyfikowanego w parametrze clk_id
```

```
int clock_gettime(clockid_t clk_id, struct timespec *tp) – pobranie wartości zegara
```

```
int clock_settime(clockid_t clk_id, const struct timespec *tp) - ustawienie wartości zegara
```

```
#include <sys/times.h>
```

```
clock_t times(struct tms *buffer);
```

Pola struktury tms

`tms_utime` – czas cpu wykonywania procesu w trybie użytkownika

`tms_stime` – czas cpu wykonywania procesu w trybie jądra

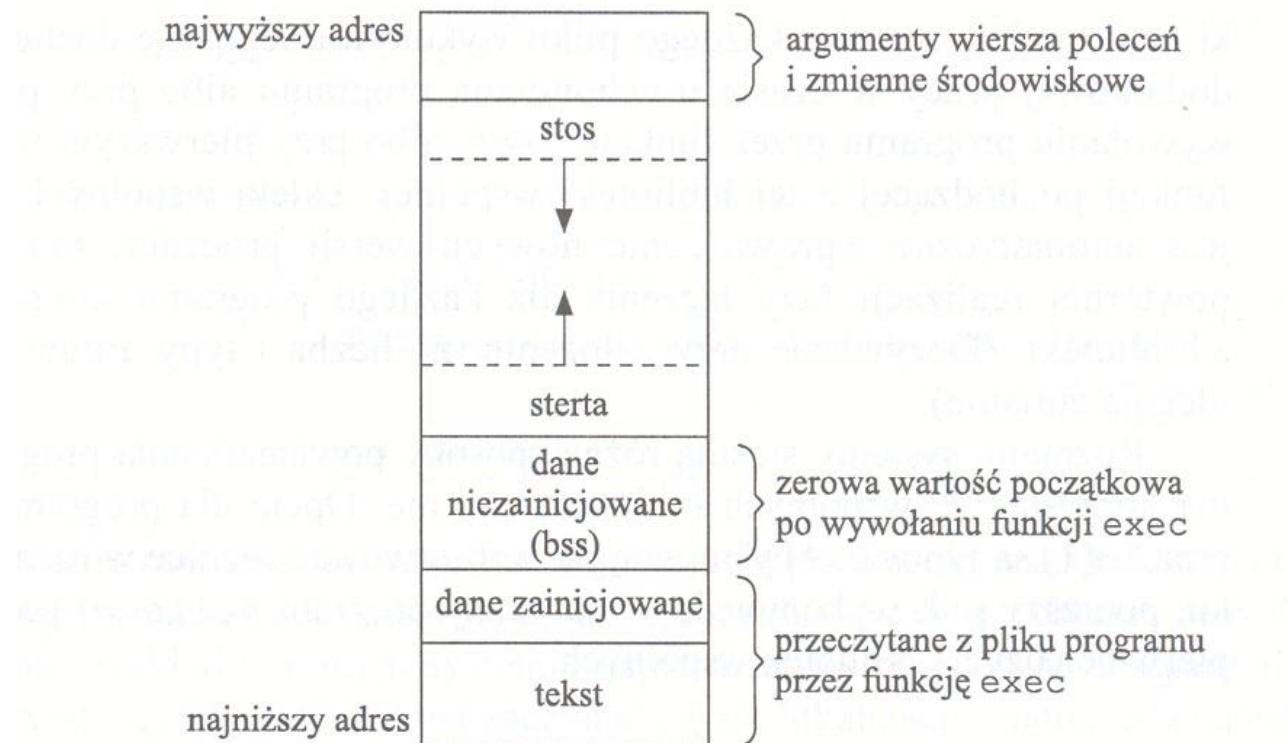
`tms_cutime` – suma czasów cpu wykonywania procesu i wszystkich jego potomków w trybie użytkownika

## Bibliografia

Robert Love, „Linux Kernel Development. A thorough guide to the design and implementation of the Linux kernel”, Novel Press, Third edition, 2010

## Zarządzanie pamięcią

Typowa organizacja pamięci w procesie



## Unix. Funkcje do alokacji pamięci dynamicznej w programach

Alokacja pamięci: (standard ANSI C)

- malloc – alokuje w pamięci wskazaną liczbę bajtów. Wartość początkowa zawartości pamięci nie jest określona
- calloc – alokuje przestrzeń dla określonej liczby obiektów o zadanym obszarze. Cały zarezerwowany obszar jest wypełniony bitami zerowymi
- realloc – zmienia rozmiar poprzednio zaalokowanego obszaru (zwiększa go lub zmniejsza). Jeśli rozmiar rośnie, może to oznaczać przesunięcie wcześniej zaalokowanego obszaru w inne miejsce, aby dodać wolną przestrzeń na jego końcu. W takiej sytuacji nie jest określona wartość początkowa fragmentu pamięci między końcem starego a końcem nowego obszaru.
- free – zwalnia pamięć wskazaną przez ptr

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

```
void * malloc(size_t size);
void * calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsize);
void free(void *ptr);
```

## Unix. Funkcje do zarządzania pamięcią.

- Funkcje alokujące są na ogół implementowane za pomocą funkcji systemowej sbrk(2), które rozszerza lub zwiększa stertę procesu
- Choć wywołanie funkcji sbrk może rozszerzyć lub zwiększyć pamięć procesu, to jednak większość wersji funkcji malloc i free nigdy nie zmniejsza rozmiaru pamięci procesu – zwalniana pamięć staje się dostępna dla kolejnych alokacji, ale nie powraca do jądra systemu – jest utrzymywana w puli, którą dysponuje funkcja malloc.
- Uwaga: większość implementacji alokuje nieco więcej pamięci, niż jest to wymagane, dodatkowy obszar jest używany do przechowywania specjalnych danych jak: rozmiar alokowanego bloku, wskaźnika do kolejnego bloku do alokacji.

## Inne funkcje mechanizmu alokacji: mallinfo

mallinfo – dostarcza charakterystyki mechanizmu alokacji:

```
struct mallinfo mallinfo(void)
unsigned long arena;//total space
unsigned long ordblks; //number of ordinary blocks
unsigned long smblks; //number of small blocks
unsigned long hblkhd; //space in holding block headres
unsigned long hblkls; //number of holding blocks
unsigned long usmblkls; //space in small blocks in use
unsigned long fsmblkls; //space in free small blocks
unsigned long uordblkls; //space in ordinary blocks in use
undigned long fordblks; //space in free ordinary blocks
unsigned lon keepcostl //space penalty if keep option is used
```

## Użycie obszarów pamięci

Przykładowy program:

```
int main(int argc, char * argv[])
{
    return 0;
}
```

Proces zawiera obszary odpowiadające sekcjom tekstu, danych i bss

Zakładając, że proces jest dynamicznie zlinkowany z biblioteką C, analogiczne trzy obszary pamięci istnieją także dla libc.so (biblioteka c) oraz dla ld.so (linkera dynamicznego)

Proces posiada także obszar pamięci odpowiadający za stos

Poniższe dane w pliku /proc/<pid>/maps przedstawiają obszary pamięci, mają postać:  
początek obszaru-koniec obszaru prawa dostępu duży:mały i węzeł plik

```
rlove@wolf:~$ cat /proc/1426/maps
00e80000-00faf000 r-xp 00000000 03:01 208530      /lib/tls/libc-2.5.1.so
00faf000-00fb2000 rw-p 0012f000 03:01 208530      /lib/tls/libc-2.5.1.so
00fb2000-00fb4000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 03:03 439029      /home/rlove/src/example
08049000-0804a000 rw-p 00000000 03:03 439029      /home/rlove/src/example
40000000-40015000 r-xp 00000000 03:01 80276       /lib/ld-2.5.1.so
40015000-40016000 rw-p 00015000 03:01 80276       /lib/ld-2.5.1.so
4001e000-4001f000 rw-p 00000000 00:00 0
bffffe000-c0000000 rwxp fffff000 00:00 0
```

Pierwsze trzy wiersza, to sekcja tekstu, danych i bss biblioteki C (libc.so)

Następne dwa wiersze to sekcje kodu i danych programu wykonywalnego

Następne trzy wiersze to sekcja tekstu, danych i bss linkera dynamicznego (ld.so)

Ostatni wiersz to obszar stosu

Cała przestrzeń adresowa zajmuje około 1340KB, ale tylko 40KB są zapisywane i prywatne

Jeśli obszar pamięci jest współdzielony lub niemodyfikowalny, jądro przechowuje tylko jedną jego kopię w pamięci

Dlatego biblioteka C potrzebuje tylko 1212KB pamięci fizycznej dla wszystkich procesów

Obszary pamięci bez zmapowanego pliku i o i-węźle 0 – są to strony zerowe (zero page): mapowania zawierające tylko zera

Przez zmapowanie strony zerowej na zapisywane obszary pamięci, obszar jest "inicjalizowany" zerami, co jest oczekiwane dla bss

## Bibliografia

Robert Love, „Linux Kernel Development. A thorough guide to the design and implementation of the Linux kernel”, Novel Press, Third edition, 2010

## Biblioteki

Co to są biblioteki?

Biblioteka jest zbiorem implementacji zachowań, opisanych w języku programowania, która ma dobrze zdefiniowany interfejs, przez który zachowania są wywoływanie [Wikipedia] "program library" jest plikiem zawierającym skompilowany kod i dane, które będą włączone potem do programu/programów, umożliwiając modularne programowanie, szybszą rekompilację i łatwiejsze aktualnienia [The Linux Documentation Project]

- Biblioteki można podzielić na trzy rodzaje: statyczne, współdzielone i dynamicznie ładowane
- Statyczne biblioteki są dołączane do programu wykonywalnego przed jego uruchomieniem
- Współdzielone biblioteki są ładowane w momencie uruchomienia programu i mogą być współdzielone z innymi programami
- Dynamicznie ładowane biblioteki są ładowane, gdy program wykonywalny się wykonuje.

### Biblioteki statyczne (Static Libraries)

- Biblioteki statyczne są zbiorami plików obiektowych. Zazwyczaj mają rozszerzenie ".a".
- Biblioteki statyczne pozwalają użytkownikom linkować się do plików obiektowych bez rekomplikacji kodu. Pozwalają także dystrybuować biblioteki bez rozpowszechniania kodu źródłowego.

Przykłady:

```
my_library.h
#pragma once
namespace my_library {
    extern "C" void my_library_function();
}

my_library.c
...
#include "my_library.h"
void my_library_function() {
    ...
}

main.c
#include "my_library.h"
int main() {
    my_library_function();
}
```

Przykład – kompilacja z plikami obiektowymi

```
$ gcc -c my_library.c
$ gcc -c main.c
$ gcc main.o my_library.o -o main
$ ./main
```

Przykłady – kompilacja jako biblioteka statyczna

```
$ gcc -c my_library.c
$ ar rcs libmy_library.a my_library.o
$ gcc -c main.c
$ gcc main.o libmy_library.a -o main
$ ./main
...
$ gcc main.o -l my_library -L ./ -o main
$ ./main
```

### Biblioteki współdzielone (Shared Libraries)

- Biblioteki współdzielone są ładowane gdy program jest ładowany. Wszystkie programy mogą współdzielić dostęp do współdzielonych bibliotek i będzie uaktualniona (upgraded) jeśli nowa wersja biblioteki zostanie zainstalowana
- Może być zainstalowanych wiele wersji bibliotek, by pozwolić programom ze specyficznymi potrzebami na używanie konkretnej wersji biblioteki
- Biblioteki te mają zazwyczaj rozszerzenie .so
- Biblioteki współdzielone używają specyficznej reguły nazewnictwa
- Każda biblioteka ma "soname" zaczynające się do prefiku "lib", po których następuje nazwa (name) biblioteki, po czym rozszerzenie ".so" ta następnie kropkę i wersję biblioteki (version numer).
- Każda biblioteka ma także nazwę rzeczywistą "real name" – nazwę pliku z kodem biblioteki. Rzeczywista nazwa "real name" obejmuje "soname", kropkę, version number i opcjonalnie kropkę i release number.

- Oba numery (version i release) umożliwiają wybór dokładnej wersji biblioteki.
- Dla jednej biblioteki współdzielonej system często ma wiele linków wskazujących na tę samą nazwę

Przykład:

```
soname
/usr/lib/libreadline.so.3
```

Linkowanie do realname:

```
/usr/lib/libreadline.so.3.0
```

Lub:

```
/usr/lib/libreadline.so
```

Linkowanie do:

```
/usr/lib/libreadline.so.3
```

Przykłady – Kompilowanie biblioteki współdzielonej

```
$ gcc -fPIC -c my_library.c
(-fPIC position independent code, potrzebny do kodu biblioteki współdzielonej)
```

```
$ gcc -shared -Wl,-soname,libmy_library.so.1 \
-o libmy_library.so.1.0.1 my_library.o -lc
```

```
$ ln -s libmy_library.so.1.0.1 libmy_library.so.1
$ ln -s libmy_library.so.1 libmy_library.so
```

Przykład – użycie biblioteki współdzielonej

```
$ gcc main.c -lmy_library -L ./ -Wl,-rpath,. -o main
$ ./main
```

- Problemem jest, że mamy stałą ścieżkę do biblioteki
- Biblioteka musi być w tym samym katalogu
- Położenie bibliotek współdzielonych może się różnić w zależności do systemu.
- \$LD\_LIBRARY\_PATH służy do ustawienia ścieżki poszukiwań bibliotek współdzielonych
- Kiedy program wymagający bibliotek współdzielonych jest uruchomiony, system poszukuje ich w katalogach podanych w \$LD\_LIBRARY\_PATH .
- Jeśli chcesz zainstalować swoją bibliotekę w systemie, możesz skopiować pliki .so do jednej ze standardowych katalogów - /usr/lib i wywołać ldconfig
- Każdy program używający biblioteki może teraz odwołać się do niej poprzez -lmy\_library i system znajdzie ją w /usr/lib
- Aby system podczas uruchamiania naszego programu zawsze (niezależnie od zmiennej LD\_LIBRARY\_PATH) szukał bibliotek w dodatkowym folderze, np. w bieżącym, należy podczas komplikacji dodać tę ścieżkę przy pomocy argumentu: -Wl,-rpath,.

### Biblioteki ładowane dynamicznie (Dynamically Loaded Libraries)

- Dynamicznie ładowane biblioteki są ładowane przez sam program z poziomu kodu źródłowego.
- Biblioteki są zbudowane jako standardowe obiekty lub biblioteki współdzielone, jedyną różnicą jest to, że biblioteki nie są ładowane podczas fazy linkowania przy komplikacji lub uruchomienia, ale w punkcie ustalonem przez programistę.
- Funkcje odpowiedzialne za operacje na bibliotekach ładowanych dynamicznie:

```
void* dlopen(const char *filename, int flag); - Otwiera bibliotekę, przygotowuje ją do użycia i zwraca wskaźnik/uchwyt na bibliotekę.
void* dlsym(void *handle, char *symbol); - Przegląda bibliotekę szukając specyficznego symbolu.
void dlclose(); - Zamyka bibliotekę .
```

```
#include <dlfcn.h>
int main() {
    void *handle = dlopen("libmy_library.so", RTLD_LAZY);
    if(!handle){/*error*/}
    void (*lib_fun)();
    lib_fun = (void (*)())dlsym(handle,"my_library_function");
```

```
if(dlerror() != NULL){/*error*/}
```

```
(*lib_fun)();
```

```
dlclose(handle);
```

```
}
```

## GNU Make

### Co to jest Make?

- Narzędzie generacji kodu wykonywalnego
- Uwzględnia modyfikacje plików źródłowych

### Możliwości

- Automatycznie określa, które pliki potrzebują uaktualnienia
- Nie ograniczony do szczególnego języka
- Użytkownik końcowy może zbudować i zainstalować pakiet bez znajomości szczegółów, jak to przeprowadzić

### Reguły (Rules) i Cele (Targets)

- Reguła (rule) w pliku makefile mówi użytkownikowi jak wykonywać serie poleceń by zbudować plik docelowy z plików źródłowych
- Określa listę zależności dla pliku docelowego

```
target: dependencies ...
```

```
    commands
```

```
    ...
```

### Budowa Procesu

- Kompilator pobiera pliki źródłowe i wyjściowe pliki obiektowe (object files).
- Linker pobiera pliki obiektowe i tworzy plik wykonywalny

### Prosty Makefile

```
all: hello
```

```
hello: main.o factorial.o hello.o
```

```
    gcc main.o factorial.o hello.o -o hello
```

```
main.o: main.c
```

```
    gcc -c main.c
```

```
factorial.o: factorial.c
```

```
    gcc -c factorial.c
```

```
hello.o: hello.cpp
```

```
    gcc -c hello.c
```

```
clean:
```

```
    rm *o hello
```

### Prosty Makefile ze zmiennymi

```
CC=gcc
```

```
CFLAGS=-c -Wall
```

```
all: hello
```

```
hello: main.o factorial.o hello.o
```

```
    $(CC) main.o factorial.o hello.o -o hello
```

```
main.o: main.c
```

```
    $(CC) $(CFLAGS) main.cpp
```

```
factorial.o: factorial.c
```

```
    $(CC) $(CFLAGS) factorial.c
```

```
hello.o: hello.c
```

```
    $(CC) $(CFLAGS) hello.c
```

```
clean:  
rm *o hello
```

## Użycie Makefile

Wykonanie makefile  
make

Konkretny cel (target) może być wykonany przez:  
make target\_label

Na przykład, wykonanie polecenia rm z poprzedniego slajdu:  
make clean

## GDB:GNU Debugger

### Co to jest GDB?

- GDB,GNU debuggery, które pozwalają na obserwację wykonania programu w danych punktach wykonania, lub co było robione w momencie, gdy nastąpiło załamanie się/błąd programu
- Debuggowany program może być napisany w różnych językach programowania - C, C++, Objective-C, Pascal (i wielu innych).

### Zastosowania GDB

- Uruchom program i wyspecyfikuj działania wpływające na jego zachowanie
- Zatrzymaj program, jeśli odpowiedni warunek jest spełniony
- Sprawdź, co się stało, gdy program się zatrzymał
- Zmodyfikuj program tak, że następuje poprawa efektów jednego z błędów i szukaj ewentualnych innych błędów

### Kompilacja programu w trybie do debugowania przy pomocy GDB

Normalna komplikacja programu

```
gcc [flags] <source files> -o <output file>
```

Kompilacja włączająca wsparcie dla debugowania poprzez dodanie symbolu -g  
gcc [other flags] -g <source files> -o <output file>

### Uruchamianie GDB

Po napisaniu „gdb” lub „gdb my\_prog.x”, debugger zgłosi się następująco:  
(gdb)

Jeżeli debuggowany program nie został wyspecyfikowany podczas uruchomienia gdb, można go wskazać używając polecenia „file”:  
(gdb) file my\_prog.x

### Uruchomienie programu

Po załadowaniu programu do debuggera, można program uruchomić w debuggerze przy pomocy polecenia „run”:  
(gdb) run

Jeśli nie wystąpią żadne błędy podczas wykonania, program zakończy się bez dodatkowych informacji. Jeśli pojawią się problemy, wyświetlona zostanie odpowiednia informacja, gdzie nastąpił błąd i program się załamał (gdzie nastąpił „crash”);

### Breakpoints

Można użyć polecenia break, aby zatrzymać program w wybranym punkcie:  
(gdb) break my\_file.c:7 - ustaw breakpoint w linii 7 w pliku my\_file.c

Można także ustawić breakpoint w wybranej funkcji:  
(gdb)break myfunc - myfunc jest nazwą funkcji w programie

Można także ustawić warunkowe breakpoints, które umożliwiają pominięcie niepotrzebnych kroków wykonania:  
(gdb) break file1.c:6 if i >= ARRAYSIZE - ustawia break point jeśli wartość i jest większa lub równa wartości ARRAYSIZE

### Kontynuacja po Breakpoint

Po ustawieniu breakpoint, można uruchomić program (run) ponownie. Możliwe są także następujące działania:

- Można napisać continue by dostać się do następnego breakpoint'a  
(gdb)continue
- Można napisać step by wykonać pojedynczy wiersz  
(gdb)step

- Można napisać next, by znów wykonać pojedynczy krok, ale nie wykonuje on wszystkich wierszy podprocedury;  
(gdb)next

Przykład: różnica między next i step

Mamy część kodu i przyjmujemy, że znajdujemy się w linii foo() znajdującej się w funkcji int main.

```
void foo() {
    for ( int i = 0; i < v.size(); i++ ) {
        ...
    }
}

int main() {
    foo();
}
```

Jeżeli napiszemy next, przemieścimy się do następnej linii, którą jest „}” w funkcji int main().

Jeżeli napiszemy step; przemieścimy się do pętli for wewnętrz funkcji foo().

## Zmienne

Jeżeli chcemy obejrzeć wartości zmiennych podczas debugowania, można użyć polecenia print;

```
(gdb) print my_var
(gdb) print/x my_var
```

Inne użyteczne polecenia

- backtrace – wyświetli ścieżkę stosu wywoływanej funkcji
- finish – wykonuj się do zakończenia aktualnej funkcji
- delete – usuń podany breakpoint
- info breakpoints - wyświetli informacje o wszystkich ustawionych breakpoints
- help – można używać wraz z nazwą wybranego polecenia lub bez niej, powoduje wyświetlenie listy dostępnych poleceń lub opis działania wybranego polecenia

## Bibliografia

University of Maryland Computer Science Department Tutorials

Tutorialspoint website

Official website of GNU Operating System

Ostatnia modyfikacja: środa, 8 marca 2023, 09:58



Platforma obsługiwana przez:  
[Centrum e-Learningu i Innowacyjnej Dydaktyki AGH](#)  
[Centrum Rozwiązań Informatycznych AGH](#)

Pobierz aplikację mobilną



Oznacz jako wykonane

## System plików, operacje na plikach

Unixowy system plików jest hierarchicznym uporządkowaniem katalogów i plików.

**Plik** – struktura danych zapisana na dysku i identyfikowana za pomocą nazwy.

Ogólny schemat operacji na plikach obejmuje:

- otwarcie pliku (przygotowujące do zapisywania lub odczytywania informacji, skojarzenia zmiennej plikowej z plikiem),
- wykonanie operacji zapisu lub odczytu danych,
- zamknięcie pliku (przerwanie skojarzenia pomiędzy zmienną plikową i plikiem).

## Funkcje systemowe

Funkcje systemowe operujące na plikach w systemie UNIX oparte są na pojęciu deskryptora. Deskryptor to nieujemna zmienna typu int przypisana w danym procesie do danego pliku, unikalna w obrębie tego procesu. Każdy proces dostaje 3 domyślne deskryptory:

- standardowe wejście (0)
- standardowe wyjście (1)
- standardowe wyjście diagnostyczne (2)

Korzystanie z funkcji systemowych do obsługi plików wymaga dołączenia bibliotek: <fcntl.h> <unistd.h> <sys/types.h> <sys/stat.h>

### Otwieranie i tworzenie plików

Funkcje systemowe odpowiadające za otwieranie i tworzenie plików to **open** oraz **creat**:

```
int open(const char *pathname, int flags[, mode_t mode]);
int creat(const char *pathname, mode_t mode);
```

Lista możliwych flag dla funkcji **open**:

|          |                                   |
|----------|-----------------------------------|
| O_RDONLY | Otwiera plik do odczytu           |
| O_WRONLY | Otwiera plik do zapisu            |
| O_RDWR   | Otwiera plik do zapisu i odczytu. |

Powyższe flagi można łączyć bitowym OR z poniższymi:

|          |  |
|----------|--|
| O_CREAT  | Utworzenie pliku, jeżeli nie istnieje.   |
| O_TRUNC  | Obcięcie pliku, jeśli plik istnieje i otwierany jest w trybie O_WRONLY lub O_RDWR  |
| O_EXCL   | Powoduje zgłoszenie błędu jeśli plik już istnieje i otwierany jest z flagą O_CREAT |
| O_APPEND | Operacje pisania odbywają się na końcu pliku                                       |

### Odczyt i zapis pliku

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

- próbuje wczytać podaną liczbę bajtów (count) z pliku o podanym deskryptorze (fd) do podanego bufora (buf); bieżąca pozycja w pliku przesuwa się o tyle, ile bajtów przeczytano,
- read() zwraca ilość bajtów naprawdę przeczytanych (zawartą wartość może być mniejsza od nbytes !)
- gdy "bieżąca pozycja" przekroczy koniec pliku, to read() zwraca 0

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

- zapis zawartości bufora do pliku, argumenty analogiczne do read.

### Ustawianie pozycji w pliku

```
long lseek(int fd, off_t offset, int whence);
```

Argumenty:

fd – deskryptor do pliku na którym operujemy

offset – nowa pozycja w pliku

whence – parametr służący interpretacji drugiego parametru. Musi być to liczba równa 0, 1 lub 2

Parametr whence funkcji lseek przyjmuje jedną z wartości:

- SEEK\_SET – początek pliku
- SEEK\_END – koniec pliku
- SEEK\_CUR – aktualna pozycja wskaźnika

Na podstawie tej wartości wylicza nową pozycję wskaźnika po przesunięciu o offset

**Wyniki:** W przypadku powodzenia funkcja zwraca nowa pozycje w pliku, w przeciwnym wypadku wartość mniejsza od zera.

## Zamykanie pliku

```
int close(int fd);
```

### Przykład 1

Program kopiujący znak po znaku z wykorzystaniem funkcji niskopoziomowych

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    char c;
    int we,wy;
    we=open("we", O_RDONLY);
    wy=open("wy",O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR);
    while(read(we,&c,1)==1)
        write(wy,&c,1);

}
```

### Przykład 2

Program kopiujący blokami o rozmiarze 1024B z wykorzystaniem funkcji niskopoziomowych

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    char blok[1024];
    int we, wy;
    int liczyt;
    we=open("we", O_RDONLY);
    wy=open("wy",O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR);
    while((liczyt=read(we,blok,sizeof(blok)))>0)
        write(wy,blok,liczyt);

}
```

## Funkcje biblioteki standardowej C

Funkcje z biblioteki standardowej C mapują funkcje systemowe. Zaletą tego rozwiązania jest przenośność kodu, na innym systemie funkcje z C zaimplementowane będą jako inne funkcje systemowe jednak na poziomie języka działanie będzie jednakowe w każdym wypadku.

### Otwarcie pliku

Aby otworzyć plik używamy funkcji **fopen**:

```
FILE * fopen ( const char * filename, const char * mode );
```

Atrybuty z jakimi można otworzyć plik:

|   |  |
|---|--|
| r | Otwiera plik do odczytu                              |
| w | Otwiera plik do zapisu (kasuje ewentualny poprzedni) |

|         |  |
|---------|--|
| a       | Otwiera plik do zapisu. Nie kasuje poprzedniego pliku i ustawia wskaźnik na końcu. |
| r+      | Otwiera plik do zapisu i odczytu. Plik musi istnieć.                               |
| w+      | Otwiera plik do zapisu i odczytu. Jeśli plik istniał to nadpisuje.                 |
| a+      | Otwiera plik do odczytu i dopisywania. Nie można pisać wcześniej niż na końcu.     |
| [rwa+]b | Otwiera plik jako binarny nie tekstowy.  |

## Zapis i odczyt pliku

```
size_t fread ( void * ptr, size_t size, size_t count, FILE * file)
```

Argumenty:

- ptr - wskaźnik na tablicę
- size - rozmiar elementu tablicy
- count - liczba elementów do odczytu
- file - plik, na którym wykonywana jest operacja

Funkcja fread kopiuje count elementów z podanego pliku do tablicy. Kopiowanie kończy się w przypadku wystąpienia błędu, końca pliku lub po skopiowaniu podanej liczby elementów. Wskaźnik pliku jest przesuwany, tak by wskazywał pierwszy nieodczytany element.

Wartość zwracana:Liczba faktycznie wczytanych elementów.

```
size_t fwrite ( const void * ptr, size_t size, size_t count, FILE * file);
```

Argumenty:

- ptr - wskaźnik na tablicę
- size - rozmiar elementu tablicy
- count - liczba elementów do zapisu
- file - plik, na którym wykonywana jest operacja

Funkcja fwrite kopiuje count elementów z podanej tablicy do pliku. Kopiowanie kończy się w przypadku wystąpienia błędu lub po skopiowaniu podanej liczby elementów. Wskaźnik pliku jest przesuwany, tak by wskazywał pierwszy element po ostatnim zapisanym.

Wartość zwracana: Liczba faktycznie zapisanych elementów.

## Ustawianie pozycji w pliku

```
int fseek ( FILE * file, long int offset, int mode);
```

Funkcja fseek ustawia pozycję w pliku file na offset w zależności od wartości argumentu mode.

- mode = **SEEK\_SET** (0) - offset liczony jest od początku.
- mode = **SEEK\_CUR** (1) - offset przesuwany od aktualnej pozycji,
- mode = **SEEK\_END** (2) - przesuwany o offset od końca pliku (wskaźnik pliku jest przesuwany do pozycji będącej sumą rozmiaru pliku i parametru offset).

Zwraca: Zero gdy funkcja wykonała się pomyślnie, w przypadku błędu wartość niezerowa.

```
int fsetpos (FILE* file, fpos_t* pos);
```

Funkcja zmienia aktualną pozycję wskaźnika do pliku file na pos.

Zwraca: Zero gdy funkcja wykonała się pomyślnie, EOF w przypadku wystąpienia błędu

```
int fgetpos (FILE* file, fpos_t* pos);
```

Funkcja umieszcza w pos aktualną pozycję wskaźnika do pliku file.

Zwraca: Zero gdy funkcja wykonała się pomyślnie, EOF w przypadku wystąpienia błędu

## Zamykanie pliku

```
int fclose ( FILE * stream );
```

**Przykład 3**

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

```
int main ()
{
    char napis[20];
    FILE *plik=fopen("nazwa1.txt", "a+");
    if(plik)
    {
        fread(napis,1, 15,plik);
        printf("%s",napis);
        printf("\n");
        fwrite("Zdanie drugie.", 1, 14, plik);
        rename("nazwa1.txt","nazwa2.txt");
        fclose(plik);
    }
}
```

```
    }
    return 0;
}
```

## Katalogi

**Katalogi** są w systemach Unix traktowane prawie tak samo jak pliki. Jedyna, ale bardzo ważna różnica to hierarchia jaką tworzą katalogi. Katalogi porządkują system plików tworząc drzewo katalogów. Katalog jest kontenerem dla plików. Katalogami są również . i .. - są to łącza do bieżącego katalogu i jego przodka. Katalog / jest korzeniem.

## Operacje na katalogach

W bibliotece `dirent.h` istnieją następujące definicje:

- `DIR` – struktura reprezentująca strumień katalogowy
- `struct dirent` – struktura, która zawiera:
  - `ino_t d_ino` – numer i-węzła pliku
  - `char d_name[]` – nazwa pliku

```
DIR* opendir(const char* dirname)
```

Otwiera strumień do katalogu znajdującego się pod ścieżką `dirname`. Po prawidłowym wykonaniu, `opendir()` zwraca wskaźnik do obiektu typu `DIR`, inaczej zwraca `NULL`.

```
int closedir(DIR* dirp)
```

Zamyka strumień katalogowy `dirp`. Po prawidłowym wykonaniu, funkcja zwraca wartość 0, inaczej zwraca -1 i zapisuje kod błędu w zmiennej `errno`.

```
struct dirent* readdir(DIR* dirp)
```

Zwroci wskaźnik do struktury reprezentującej plik w obecnej pozycji w strumieniu `dirp` i awansuje pozycję na następny plik w kolejce. Zwrócony wskaźnik do obiektu `struct dirent` nie powinien być zwolniony. Jeśli nie ma już więcej plików w katalogu, wartość `NULL` jest zwrócona. Gdy wystąpi błąd, wartość `NULL` także jest zwrócona i powód jest zapisany w zmiennej `errno`.

```
void rewinddir(DIR* dirp)
```

Ustawia strumień katalogowy na początek.

```
void seekdir(DIR* dirp, long int loc)
```

Zmienia pozycję strumienia katalogowego.

```
int stat (const char *path, struct stat *buf);
```

 - pobranie statusu pliku

wejście: `path` - nazwa sprawdzanego pliku

`buf` - bufor na strukturę `stat`

`err` - Po sukcesie zwracane jest zero. Po błędzie -1 i ustawiane jest 'errno':

rezultat:

- EBADF - 'filedes' jest nieprawidłowy.
- ENOENT - Plik nie istnieje.

```
int lstat(const char *ścieżka, struct stat *statystyka);
```

 -identyczna jak `stat()`, lecz nie zwraca on statusu plików, wskazywanych przez linki, a status samego linku.

### Struktura stat

```
struct stat
{
    dev_t          st_dev;      /* urządzenie */
    ino_t          st_ino;      /* inode */
    umode_t        st_mode;     /* ochrona */
    nlink_t        st_nlink;    /* liczba hardlinków */
    uid_t          st_uid;      /* ID użytkownika właściwego */
    gid_t          st_gid;      /* ID grupy właściwego */
    dev_t          st_rdev;     /* typ urządzenia (jeśli urządzenie inode) */
    off_t          st_size;     /* całkowity rozmiar w bajtach */
    unsigned long  st_blksize;  /* wielkość bloku dla I/O systemu plików */
    unsigned long  st_blocks;   /* ilość zaalokowanych bloków */
    time_t         st_atime;    /* czas ostatniego dostępu */
    time_t         st_mtime;    /* czas ostatniej modyfikacji */
    time_t         st_ctime;    /* czas ostatniej zmiany */
```

**int mkdir (const char \*path, mode\_t mode);** - tworzenie katalogu z uprawnieniami podanymi w mode

**int rmdir (const char \*path);** - usuwanie katalogu

**int chdir (const char \*path);** - argument path staje się nowym katalogiem bieżącym dla programu.

**char \*getcwd (char \*folder\_name, ssize\_t size);** - funkcja wpisuje do **folder\_name** bieżący katalog roboczy o rozmiarze **size**.

**int chmod (const char \*path, mode\_t new);** - zmiana uprawnień do pliku.

**int chown (const char \*path, uid\_t id\_właściciela, gid\_t id\_grupy);** - zmiana właściciela.

**int link (const char \*path, const char \*nowa);** – stworzenie twardego linku do pliku. Usunięcie łącza – funkcja **unlink**.

**int nftw(const char \*dir, int(\*fn) (), int nopen, int flags)**

**dir** - katalog główny drzewa do przeglądnięcia

**fn** - funkcja wywoływana dla każdego przeglądanego elementu w drzewie

wejście: **nopenfd** - maksymalna ilość otwieranych przez funkcję deskryptorów

**flags** - znaczniki definiujące zachowanie funkcji

rezultat: **err** - w przypadku powodzenia zwracana jest wartość 0, w przypadku błędu zwracana jest wartość -1.

opis: przegląd drzewa katalogów. Funkcja ftw() przechodzi przez drzewo katalogów startując z określonego katalogu 'dir'. Dla każdej znalezionej pozycji w drzewie, wywołuje funkcję 'fn' z pełną nazwą ścieżki do pozycji, wskaźnik na strukturę otrzymaną z funkcji **stat(2)** dla tej pozycji oraz flagę 'flag' której wartość jest jedną z poniższych wartości:

- FTW\_F - pozycja jest normalnym plikiem
- FTW\_D - pozycja jest katalogiem
- FTW\_DNR - pozycja jest katalogiem który nie może być czytany
- FTW\_SL - pozycja jest linkiem symbolicznym
- FTW\_NS - operacja stat nie powiodła się na pozycji która nie jest linkiem symbolicznym

Ostatnia modyfikacja: środa, 1 marca 2023, 08:40



Platforma obsługiwana przez:  
[Centrum e-Learningu i Innowacyjnej Dydaktyki AGH](#)  
[Centrum Rozwiązań Informatycznych AGH](#)

Pobierz aplikację mobilną 

Wybierz język



Oznacz jako wykonane

**Proces** jest pojedynczą instancją wykonującego się programu. Możemy w nim wyróżnić:

- **segment kodu** - zawiera kod binarny aktualnie wykonywanego programu. Znajduje się w nim kod zaimplementowanych przez nas funkcji oraz funkcji dołączanych z bibliotek. Zapisane w tym segmencie adresy funkcji pozwalają na ich lokalizację.
- **segment danych** - zawiera zainicjalizowane zmienne globalne zdefiniowane w programie. Adres segmentu danych można ustalić na podstawie adresu zmiennej globalnej.
- **segment BSS - Block Started by Symbol** - zawiera niezainicjalizowane zmienne globalne
- **segment stosu** - zmienne lokalne oraz adresy powrotu wykorzystywane podczas powrotu z wykonywanej funkcji. Ponieważ proces może pracować w trybie użytkownika lub trybie jądra, każdy z tych trybów ma do dyspozycji oddzielny stos.

Każdemu procesowi przydzielane są zasoby czas procesora, pamięć, dostęp do urządzeń we/wy oraz plików etc). Część tych zasobów jest do wyłącznej dyspozycji procesu, zaś część jest współdzielona z innymi procesami.

Na proces nakładane są pewne ograniczenia dotyczące zasobów systemowych, możemy do nich uzyskać dostęp następującymi funkcjami z **sys/resource.h**:

**int getrlimit (int resource, struct rlimit \*rlptr) Resource to jedno z makr określające rodzaj zasobu**

**int setrlimit (int resource, const struct rlimit \*rlptr)**

```
struct rlimit {
```

```
    rlim_t rlim_cur; //bieżące ograniczenie  
    rlim_t rlim_max; //maksymalne ograniczenie
```

```
}
```

### Identyfikatory procesów

Każdy proces w systemie UNIX ma przypisany unikalny identyfikator - **PID**. Jest to 16-bitowa, nieujemna liczba całkowita przypisywana do każdego procesu podczas jego tworzenia. Niektóre identyfikatory są odgórnie zarezerwowane dla specjalnych procesów w systemie, (swapper - 0, *init* -1 etc).

System UNIX pamięta także identyfikator procesu macierzystego - ta informacja jest zapisywana jako **PPID** (Parent PID).

Do każdego procesu przypisane są również (rzeczywiste) identyfikatory użytkownika (**UID**) oraz grupy (**GID**), określające kto dany proces utworzył. Istnieją również efektywne UID i GID przechowujące informacje o identyfikatorze właściciela oraz grupy właściciela programu.

Do pobrania informacji o identyfikatorach procesu możemy posłużyć się funkcjami z biblioteki unistd.h, takimi jak:

- **pid\_t getpid(void)** - zwraca PID procesu wywołującego funkcję
- **pid\_t getppid(void)** - zwraca PID procesu macierzystego
- **uid\_t getuid(void)** - zwraca rzeczywisty identyfikator użytkownika UID
- **uid\_t geteuid(void)** - zwraca efektywny identyfikator użytkownika UID
- **gid\_t getgid(void)** - zwraca rzeczywisty identyfikator grupy GID
- **gid\_t getegid(void)** - zwraca efektywny identyfikator grupy GID

Definicje niezbędnych typów znajdziemy w **sys/types.h**.

## Tworzenie procesów

W systemie Unix każdy proces, za wyjątkiem procesu o numerze 0 jest tworzony przez wykonanie przez inny proces funkcji **fork**. Proces ją wykonujący nazywa się **procesem macierzystym**, zaś nowoutworzony - **procesem potomnym**. Procesy, podobnie jak katalogi, tworzą drzewiastą strukturę hierarchiczną - każdy proces w systemie ma jeden proces macierzysty, lecz może mieć wiele procesów potomnych. Korzeniem takiego drzewa w systemie UNIX jest proces o PID równym 1, czyli *init*.

Mechanizm tworzenia procesu w systemach unixowych przedstawiono poniżej:

## Funkcje systemowe

### Funkcje fork oraz vfork

- `pid_t fork( void )`

W momencie jej wywołania tworzony jest nowy proces, będący potomnym dla tego, w którym właśnie została wywołana funkcja `fork`. Jest on kopią procesu macierzystego - otrzymuje duplikat obszaru danych, sterty i stosu (a więc nie współdziały danych). Funkcja `fork` jest wywoływana raz, lecz zwraca wartość dwukrotnie - proces potomny otrzymuje wartość 0, a proces macierzysty PID nowego procesu. Jest to konieczne nie tylko ze względu na możliwość rozróżnienia procesów w kodzie programu: proces macierzysty musi otrzymać PID nowego potomka, ponieważ nie istnieje żadna funkcja umożliwiająca wylistowanie wszystkich procesów potomnych. W przypadku procesu potomnego nie jest konieczne podawanie PID jego procesu macierzystego, ponieważ ten jest określony jednoznacznie (i można go wydobyć np. za pomocą funkcji `getppid`). Z kolei 0 jest bezpieczną wartością, ponieważ jest zarezerwowana dla procesu demona wymiany i nie ma możliwości utworzenia nowego procesu o takim PID.

Po wywołaniu fork'a oba procesy (macierzysty i potomny) kontynuują swoje działanie (od linii następnej po wywołaniu fork'a czyli efektem kodu):

```
#include <stdio.h>

main(){
    printf("Początek\n");
    fork();
    printf("Koniec\n");
}
```

Będzie:

```
Początek // z macierzystego przed wywołaniem fork'a
Koniec // z macierzystego lub potomnego po fork'u
Koniec // z macierzystego lub potomnego po fork'u
```

Powyzszy komentarz `// z macierzystego lub potomnego po fork'u` wynika z faktu że nie można przewidzieć, który z procesów będzie wykonywać swoje instrukcje jako pierwszy, dlatego w przypadku gdy wymaga się od nich współpracy, należy zastosować jakieś metody synchronizacji komunikacji międzyprocesowej.

vfork

- `pid_t vfork( void )`

Funkcji tej używa się w przypadku gdy głównym zadaniem nowego procesu jest wywołanie funkcji `exec`. `vfork` „odblokuję” proces macierzysty dopiero w momencie wywołania funkcji `exec` lub `exit`. Inną ważną cechą tej funkcji jest współdzielenie przestrzeni adresowej przez obydwa procesy.

Identyfikacja procesu macierzystego i potomnego

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

int main() {
    pid_t child_pid;
    printf("PID głownego programu: %d\n", (int)getpid());
    child_pid = fork();
    if(child_pid!=0) {
        printf("Proces rodzica: Proces rodzica ma pid:%d\n", (int)getpid());
        printf("Proces rodzica: Proces dziecka ma pid:%d\n", (int)child_pid);
    } else {
        printf("Proces dziecka: Proces rodzica ma pid:%d\n", (int)getppid());
        printf("Proces dziecka: Proces dziecka ma pid:%d\n", (int)getpid());
```

```
    }

    return 0;
}
```

**Przykładowy wynik działania programu:**

```
PID głównego programu: 2359
Proces rodzica: Proces rodzica ma pid:2359
Proces rodzica: Proces dziecka ma pid:2360
Proces dziecka: Proces rodzica ma pid:2359
Proces dziecka: Proces dziecka ma pid:2360
```

## Funkcje rodziny exec

Funkcje z rodziny exec służą do uruchomienia w ramach procesu innego programu.

W wyniku wywołania funkcji typu **exec** następuje reinitializacja segmentów kodu, danych i stosu procesu ale nie zmieniają się takie atrybuty procesu jak pid, ppid, tablica otwartych plików i kilka innych atrybutów z segmentu danych systemowych

- **int execl(char const \*path, char const \*arg0, ...)**

funkcja jako pierwszy argument przyjmuje ścieżkę do pliku, następnie są argumenty wywołania funkcji, gdzie arg0 jest nazwą programu

- **int execle(char const \*path, char const \*arg0, ..., char const \* const \*envp)**

podobnie jak execl, ale pozwala na podanie w ostatnim argumentem tablicy ze zmiennymi środowiskowymi

- **int execlp(char const \*file, char const \*arg0, ...)**

również przyjmuje listę argumentów ale, nie podajemy tutaj ścieżki do pliku, lecz samą jego nazwę, zmienna środowiskowa PATH zostanie przeszukana w celu lokalizowania pliku

- **int execv(char const \*path, char const \* const \* argv)**

analogicznie do execl, ale argumenty podawane są w tablicy

- **int execve(char const \*path, char const \* const \*argv, char const \* const \*envp)**

analogicznie do execle, również argumenty przekazujemy tutaj w tablicy tablic znakowych

- **int execvp(char const \*file, char const \* const \*argv)**

analogicznie do execlp, argumenty w tablicy

Różnice pomiędzy wywołaniami funkcji **exec** wynikają głównie z różnego sposobu budowy ich listy argumentów: w przypadku funkcji **execl** i **execvp** są one podane w postaci listy, a w przypadku funkcji **execv** i **execvp** jako tablica.

Zarówno lista argumentów, jak i tablica wskaźników musi być zakończona wartością NULL. Funkcja **execle** dodatkowo ustala środowisko wykonywanego procesu.

Funkcje **execlp** oraz **execvp** szukają pliku wykonywalnego na podstawie ścieżki przeszukiwania podanej w zmiennej środowiskowej PATH. Jeśli zmienna ta nie istnieje, przyjmowana jest domyślna ścieżka :/bin:/usr/bin.

Znaczenie poszczególnych literek w nazwach funkcji z rodziny exec:

- l oznacza, że argumenty wywołania programu są w postaci listy napisów zakończonej zerem (**NULL**)
- v oznacza, że argumenty wywołania programu są w postaci tablicy napisów (tak jak argument **argv** funkcji **main**)
- p oznacza, że plik z programem do wykonania musi się znajdować na ścieżce przeszukiwania ze zmiennej środowiskowej **PATH**
- e oznacza, że środowisko jest przekazywane ręcznie jako ostatni argument

Wartością zwracaną funkcji typu **exec** jest **status**, przy czym jest ona zwracana tylko wtedy, gdy funkcja

zakończy się niepoprawnie, będzie to zatem wartość -1.

PRZYKŁADY

```
execl("../bin/ls", "ls", "-l", null)
```

```
execlp("ls", "ls", "-l", null)
```

```
char* const av[]={"ls", "-l", null}
```

```
execv("../bin/ls", av)
```

```
char* const av[]={"ls", "-l", null}
```

```
execvp("ls", av)
```

Funkcje exec **nie tworzą nowego procesu**, tak jak w przypadku funkcji fork. Należy pamiętać, że jeśli w programie wywołamy funkcję exec, to kod znajdujący się dalej w programie nie zostanie wykonany, chyba że wystąpi błąd.

Przykład połączenia funkcji fork i exec

**main.c:**

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

int main() {
    pid_t child_pid;
    child_pid = fork();
    if(child_pid!=0) {
        printf("Ten napis został wyświetlony w programie 'main'!\n");
    } else {
        execvp("./child", NULL);
    }

    return 0;
}
```

**child.c:**

```
#include <stdio.h>

int main() {
    printf("Ten napis został wyświetlony przez program 'child'!\n");
    return 0;
}
```

Wynikiem działania programu jest:

```
Ten napis został wyświetlony w programie 'main'!
Ten napis został wyświetlony przez program 'child'!
```

## Funkcje wait oraz waitpid

Proces macierzysty może się dowiedzieć o sposobie zakończenia bezpośredniego potomka przez wywołanie funkcji systemowej *wait*. Jeśli wywołanie funkcji *wait* nastąpi przed zakończeniem potomka, przodek zostaje zawieszony w oczekiwaniu na to zakończenie. Jeżeli proces macierzysty zakończy działanie przed procesem potomnym, to proces potomny nazywany jest sierotą (ang. orphan) i jest „adoptowany” przez proces systemowy *init*, który staje się w ten sposób jego przodkiem. Jeżeli proces potomny zakończył działanie przed wywołaniem funkcji *wait* w procesie macierzystym, potomek pozostanie w stanie *zombi*. Zombi jest procesem, który zwalnia wszystkie zasoby (nie zajmuje pamięci, nie jest mu przydzielany procesor), zajmuje jedynie miejsce w tablicy procesów w jądrze systemu operacyjnego i zwalnia je dopiero w momencie wywołania funkcji *wait* przez proces macierzysty, lub w momencie zakończenia procesu macierzystego.

Aby pobrać stan zakończenia procesu potomnego należy użyć jednej z dwóch funkcji (plik nagłówkowy *sys/wait.h*):

```
· pid_t wait ( int *statloc )
```

```
· pid_t waitpid( pid_t pid, int *statloc, int options )
```

Wywołując *wait* lub *waitpid* proces może:

- ulec zablokowaniu (jeśli wszystkie procesy potomne ciągle pracują)
- natychmiast powrócić ze stanem zakończenia potomka (jeśli potomek zakończył pracę i oczekuje na pobranie jego stanu zakończenia)
- natychmiast powrócić z komunikatem awaryjnym (jeśli nie ma żadnych procesów potomnych)

Funkcja *wait* oczekuje na zakończenie dowolnego potomka (do tego czasu blokuje proces macierzysty). Funkcja *waitpid* jest bardziej elastyczna, posiada możliwość określenia konkretnego PID procesu, na który ma oczekiwania, a także dodatkowe opcje (np. nieblokowanie procesu w sytuacji gdy żaden proces potomny się nie zakończył). Argument *pid* należy interpretować w następujący sposób:

- *pid == -1* Oczekiwanie na dowolny proces potomny. W tej sytuacji funkcja *waitpid* jest równoważna funkcji *wait*.
- *pid > 0* Oczekiwanie na proces o identyfikatorze równym *pid*.

- pid == 0 Oczekiwanie na każdego potomka, którego identyfikator grupy procesów jest równy identyfikatorowi grupy procesów w procesie wywołującym tę funkcję.
- pid < -1 Oczekiwanie na każdego potomka, którego identyfikator grupy procesów jest równy wartości absolutnej argumentu pid.

W obydwu przypadkach statloc to wskaźnik do miejsca w pamięci, gdzie zostanie przekazany status zakończenia procesu potomnego (można go zignorować, przekazując wartość NULL).

## Kończenie procesów

Istnieje kilka możliwych sposobów na zakończenie procesu:

- zakończenie normalne
  - wywołanie instrukcji **return** w funkcji **main**
  - wywołanie funkcji **exit** - biblioteka **stdlib**
  - wywołanie funkcji **\_exit** - biblioteka **unistd**
- zakończenie awaryjne
  - wywołanie funkcji **abort** - generuje sygnał SIGABORT
  - **odebranie sygnału**

## Funkcje exit i \_exit

```
void exit( int status )
```

```
void _exit( int status )
```

Funkcja `_exit` natychmiast kończy działanie programu i powoduje powrót do jądra systemu. Funkcja `exit` natomiast, dokonuje pewnych operacji porządkowych - kończy działanie procesu, który ją wykonał i powoduje przekazanie w odpowiednie miejsce tablicy procesów wartości `status`, która może zostać odebrana i zinterpretowana przez proces macierzysty. Jeśli proces macierzysty został zakończony, a istnieją procesy potomne - to wykonanie ich nie jest zakłócone, ale ich identyfikator procesu macierzystego wszystkich procesów potomnych otrzyma wartość 1 będącą identyfikatorem procesu `init` (proces potomny staje się sierotą (ang. orphan) i jest „adoptowany” przez proces systemowy `init`). Funkcja `exit` zdefiniowana jest w pliku `stdlib.h`.

## Polecenie kill

Polecenie `kill` przesyła sygnał do wskazanego procesu w systemie. Standardowo wywołanie programu powoduje wysyłanie sygnału nakazującego procesowi zakończenie pracy. Proces zapisuje wtedy swoje wewnętrzne dane i kończy pracę. Kill może przesyłać procesom różnego rodzaju sygnały. Są to na przykład:

- SIGTERM – programowe zamknięcie procesu (15, domyślny sygnał)
- SIGKILL – unicestwienie procesu, powoduje utratę wszystkich zawartych w nim danych (9)
- SIGSTOP – zatrzymanie procesu bez utraty danych
- SIGCONT – wznowienie zatrzymanego procesu

Czasami może zdarzyć się sytuacja, iż proces nie chce się zamknąć sygnałem SIGTERM, bo jest przez coś blokowany. Wtedy definitelywnie możemy go unicestwić sygnałem SIGKILL, lecz spowoduje to utratę danych wewnętrznych procesu.

Ostatnia modyfikacja: poniedziałek, 27 lutego 2023, 13:37



# Sygnały

---

Sygnały to swego rodzaju programowe przerwania, to znaczy zdarzenia, które mogą nastąpić w dowolnym momencie działania procesu, niezależnie od tego, co dany proces aktualnie robi.

Sygnały są asynchroniczne. Umożliwiają komunikację między procesami.

Procesy mogą otrzymywać sygnały

- z jądra systemu,
- od samego siebie
- od innych procesów
- od użytkownika.

Sygnały są wysyłane:

- za pomocą funkcji systemowej kill
- za pomocą polecenia kill
- za pomocą klawiatury - tylko wybrane sygnały
- przez pewne sytuacje wyjątkowe wykrywane przez oprogramowanie systemowe
- przez pewne sytuacje wyjątkowe wykrywane przez sprzęt

Istnieją pewne ograniczenia - proces może wysyłać je tylko do procesów mających tego samego właściciela oraz należących do tej samej grupy (te same identyfikatory uid i gid). Bez ograniczeń może to czynić jedynie jądro i administrator. Ponadto jedynym procesem, który nie odbiera sygnałów jest init (PID = 1).

Sygnal jest **dostarczony** (ang. *delivered*) do procesu, gdy proces podejmuje akcję obsługi sygnału.

Proces może zareagować na otrzymany sygnal na różne sposoby:

- Zezwolenie na domyślną obsługę sygnału – najczęściej jest to zakończenie procesu i ewentualny zrzut zawartości segmentów pamięci na dysk (plik core).
- Zignorowanie sygnału – można zignorować wszystkie sygnały poza SIGKILL oraz SIGSTOP, co zapewnia, że proces zawsze można zakończyć.
- Przechwycenie sygnału – podjęcie akcji zdefiniowanej przez użytkownika
- Maskowanie – blokowanie sygnału tak, aby nie był dostarczany.

Działanie domyślne – czynności podejmowane przez jądro, gdy pojawi się sygnal. Są to:

- zakończenie (ang. *termination*)
- ignorowanie (ang. *ignoring*)
- zrzut pamięci (ang. *core dump*)
- zatrzymanie (ang. *stopped*)

W czasie pomiędzy wygenerowaniem sygnału a jego dostarczeniem, sygnal jest w stanie **oczekiwania** (ang. *pending*) na dostarczenie.

**Sygnal zablokowany** (ang. *blocked*) jest to sygnal, który nie może być dostarczony. Pozostaje w stanie oczekiwania.

**Maska sygnałów** (ang. *signal mask*) jest to zbiór sygnałów, które są dla procesu zablokowane

Sygnatom przypisane są nazwy, rozpoczynające się od liter "SIG", oraz numery. O ile proces nie został zakończony w wyniku przerwania, kontynuuje on swoje działanie od miejsca przerwania.

Dwa sygnały - SIGKILL i SIGSTOP - nie mogą zostać przechwycone ani zignorowane. Po otrzymaniu któregoś z nich proces musi wykonać akcję domyślną. Daje nam to możliwość bezwarunkowego zatrzymania/zakończenia dowolnego procesu, jeżeli zajdzie taka potrzeba. Nie powinno się ignorować również niektórych sygnałów związanych z błędami sprzętowymi. Jeśli chcemy przechwytywać jakiś sygnał, musimy zdefiniować funkcję która będzie wykonywana po otrzymaniu takiego sygnału, oraz powiadomić jądro za pomocą funkcji signal(), która to funkcja. Dzięki temu możemy np. sprawić że po naciśnięciu przez użytkownika `ctrl+c`, nasz program zdąży jeszcze zamknąć połączenia sieciowe czy pliki, usunąć pliki tymczasowe itd.

# Rodzaje sygnałów

---

Nazwy sygnałów są zdefiniowane w `signal.h`. Pełna lista znajduje się np [tutaj](#) lub w manualu polecenia `kill`.

| Nazwa          | Numer     | Znaczenie  | Czynność domyślna           |
|----------------|-----------|--|-----------------------------|
| SIGHUP         | 1         | Przerwanie łączności z terminaliem. Służy do zakończenia pracy wszystkich procesów w momencie zakończenia sesji w danym terminalu. | Zakończenie                 |
| SIGINT         | 2         | Terminalowe przerwanie (Ctrl+C)  | Zakończenie                 |
| SIGQUIT        | 3         | Terminalowe zakończenie (Ctrl+\)   | Zrzut pamięci i zakończenie |
| SIGILL         | 4         | Nielegalna instrukcja sprzętowa  | Zrzut pamięci i zakończenie |
| SIGABRT        | 6         | Przerwanie procesu. Wysyłany przez funkcję <code>abort()</code>  | Zrzut pamięci i zakończenie |
| SIGFPE         | 8         | Wyjątek arytmetyczny (np. dzielenie przez 0)   | Zrzut pamięci i zakończenie |
| <b>SIGKILL</b> | <b>9</b>  | <b>Unicestwienie (nie da się przechwycić ani zignorować)</b>   | <b>Zakończenie</b>          |
| SIGSEGV        | 11        | Niepoprawne wskazanie pamięci  | Zrzut pamięci i zakończenie |
| SIGPIPE        | 13        | Zapis do potoku zamkniętego z jednej strony (nikt nie czyta)   | Ignorowany                  |
| SIGALRM        | 14        | Pobudka (upłynął czas ustawiony funkcją <code>alarm()</code> lub <code>setitimer()</code> )  | Ignorowany                  |
| SIGTERM        | 15        | Zakończenie programowe (domyślny sygnał polecenia <code>kill</code> )  | Zakończenie                 |
| SIGCHLD        | 17        | Zakończenie procesu potomnego  | Ignorowany                  |
| <b>SIGSTOP</b> | <b>19</b> | <b>Zatrzymanie (nie da się przechwycić ani zignorować)</b>   | <b>Zatrzymanie</b>          |
| SIGCONT        | 18        | Kontynuacja wstrzymanego procesu   | Ignorowany                  |
| SIGTSTP        | 20        | Terminalowe zatrzymanie (Ctrl+Z lub Ctrl+Y)  | Zatrzymanie                 |
| SIGTTIN        | 21        | Czytanie z terminala przez proces drugoplanowy   | Zatrzymanie                 |
| SIGTTOU        | 22        | Pisanie do terminala przez proces drugoplanowy   | Zatrzymanie                 |
| SIGUSR1        | 10        | Sygnal zdefiniowany przez użytkownika  |                             |
| SIGUSR1        | 12        | Sygnal zdefiniowany przez użytkownika  |                             |

# Sygnały czasu rzeczywistego

---

## **SIGRTMIN, SIGRTMIN+n, SIGRTMAX**

Sygnały czasu rzeczywistego to rozszerzenie mechanizmu sygnałów. Systemy UNIX-owe wspierają 32 sygnały czasu rzeczywistego. Mają one numery od SIGRTMIN do SIGRTMAX. Do kolejnych sygnałów odwołujemy się za pomocą notacji SIGRTMIN+n, ponieważ ich numery różnią się w różnych systemach UNIXowych, choć zawsze są w tej samej kolejności.

Sygnały te nie posiadają predefiniowanego znaczenia; można je wykorzystać do celów określonych w danej aplikacji. Domyślnią akcją sygnału czasu rzeczywistego jest przerwanie procesu. Sygnały czasu rzeczywistego, w przeciwieństwie do standardowych sygnałów, są kolejkowane; przechowywane są w kolejce FIFO. Ponato mają priorytety, tzn. im mniejszy numer sygnału tym wcześniej zostanie dostarczony dany sygnał.

## Polecenia Unixa

---

**kill** - wysyła do procesu lub grupy procesów określony sygnał. Można mu przekazać zarówno numer sygnału, jak i jego nazwę.

**trap** - polecenie, służące do określenia reakcji procesu na dany sygnał.

# Wysyłanie sygnałów

---

```
kill(int pid, int SIGNAL);
```

Funkcja `kill` służy do przesyłania sygnału do wskazanego procesu w systemie. Wymaga dołączenia nagłówków **sys/types.h** and **signal.h**.

Argument `pid` określa identyfikator procesu-odbiorcy sygnału, natomiast `sig` jest numerem wysyłanego sygnału.

Jeśli:

|                          |  |
|--------------------------|--|
| <code>pid &gt; 0</code>  | sygnał jest wysyłany do procesu o identyfikatorze <code>pid</code>   |
| <code>pid = 0</code>     | sygnał jest wysyłany do wszystkich procesów w grupie procesu wysyłającego sygnał   |
| <code>pid = -1</code>    | sygnał jest wysyłany do wszystkich procesów w systemie, z wyjątkiem procesów specjalnych, na przykład procesu <code>init</code> ; nadal obowiązują ograniczenia związane z prawami |
| <code>pid &lt; -1</code> | sygnał jest wysyłany do wszystkich procesów we wskazanej grupie <code>-pid</code>  |

Funkcja `kill` zwraca 0, jeśli sygnał został pomyślnie wysłany, w przeciwnym wypadku zwraca -1 i ustawia kod błędu w zmiennej `errno`.

```
int raise( int signal);
```

Wysyła sygnał do bieżącego procesu; do tego celu w rzeczywistości wykorzystuje funkcję `kill()`. O ile nie zostanie przechwycenie lub zignorowanie sygnału, proces zostanie zakończony. Wywołanie `raise()` jest równoważne z wywołaniem `kill(getpid(), sig);`

```
int sigqueue(pid_t pid, int sig, const union sigval value)
```

Funkcja ta wysyła sygnał `sig` do procesu o danym `pid`. Jeśli przekazany `pid` jest równy 0 sygnał nie zostanie wysłany, natomiast nastąpi sprawdzenie ewentualnych błędów, które mogłyby nastąpić przy wysyłaniu.

Argument `sigval` może zawierać dodatkową wartość wyslaną wraz z sygnałem. Typ `sigval` zdefiniowany jest następująco:

```
union sigval {  
    int sival_int;  
    void *sival_ptr;  
}
```

# Odbieranie sygnałów – signal

---

```
void (*signal(int signo, void (*func)()))()
```

Funkcja ta służy do ustawienia przechwytywania danego sygnału.

Pierwszy argument to numer przechwytywanego sygnału, drugi - wskaźnik na funkcję, która wykona się w przypadku przechwycenia tego sygnału. Funkcja obsługi sygnału musi przyjmować odkładnie jeden argument (numer sygnału) i nic nie zwracać. Funkcja ta zwraca poprzedni, lub SIG\_ERR jeśli wystąpił błąd.

Zamiast wskaźnika na funkcję *func* można również przekazać jedną z dwóch zmiennych: SIG\_IGN, oznaczającą ignorowanie sygnału lub SIG\_DFL, oznaczającą domyślną reakcję na sygnał.

```
void (*signal (int signo, void (*func) (int))) ;
```

gdzie:

- *signo* - określa numer sygnału, dla którego definiowana jest obsługa.
- *handler* - nazwa funkcji obsługi sygnału zdefiniowanej przez użytkownika. Może on również przyjmować jedną z wartości:
  - - SIG\_IGN - oznacza, że sygnał będzie ignorowany
  - - SIG\_DFL - sygnał będzie obsługiwany w sposób domyślny, zdefiniowany w systemie.

Funkcja zwraca SIG\_ERR jeśli wystąpił błąd, lub adres poprzedniej funkcji obsługi sygnału.

## Przykład :

---

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

/* procedura obslugi sygnalu SIGINT */
void obslugaINT(int signum) {
    printf("Obsluga sygnalu SIGINT\n");
}

main() {
    /* zarejestrowanie obslugi sygnalu SIGINT */
    signal(SIGINT, obslugaINT)
    /* nieskonczona petla */
    while(1)
        sleep(100);
}
```

# Niezawodna obsługa sygnałów (ang. reliable signals)

---

Funkcje niezawodnej obsługi sygnałów muszą się charakteryzować:

- stałą (ang. persistent) obsługą sygnałów;
- blokowaniem tego samego sygnału podczas jego obsługi;
- jednokrotnym dostarczeniem sygnału do procesu po odblokowaniu
- możliwością maskowania (blokowania) sygnałów

Funkcją, która ma zapewniać niezawodną obsługę sygnałów jest sigaction.

# Odbieranie sygnałów – sigaction

---

```
int sigaction(int sig_no, const struct sigaction *act, struct sigaction *old_act);
```

Jest to rozszerzenie funkcji `signal` - służy zatem do zmiany dyspozycji sygnału.

Wykorzystywana tu struktura `sigaction` wygląda następująco:

```
struct sigaction{  
    void (*sa_handler)(); /* Wskaźnik do funkcji obsługi sygnału*/  
    sigset_t sa_mask; /* Maska sygnałów – czyli zbiór sygnałów blokowanych podczas obsługi  
                       bieżącego sygnału, sygnał przetwarzany jest blokowany domyślnie */  
    int sa_flags; /* Nadzoruje obsługę sygnału przez jądro*/  
};
```

*sa\_mask* zawiera zbiór sygnałów, które mają być zablokowane na czas wykonania tej funkcji. W ten sposób możemy się zabezpieczyć przed odebraniem jakiegoś sygnału (i w konsekwencji wykonaniem jego funkcji) w czasie, kiedy jeszcze wykonuje się funkcja obsługująca inny sygnał. W szczególności drugie obsłuzenie tego samego sygnału podczas obsługiwania pierwszego jego egzemplarza jest zawsze blokowane.

## Przykładowe użycie:

---

```
void au(int sig_no) {
    printf("Otrzymale signal %d.\n", sig_no);
}

int main() {
    struct sigaction act;
    act.sa_handler = au;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGINT, &act, NULL);

    while(1) {
        printf("Witaj.\n");
        sleep(3);
    }
    return 0;
}
```

# Czekanie na sygnały

---

`void pause();`

Zawiesza wywołujący proces aż do chwili otrzymania dowolnego sygnału. Jeśli sygnał jest ignorowany przez proces, to funkcja pause też go ignoruje. Najczęściej sygnałem, którego oczekuje pause jest sygnał pobudki SIGALARM.

`unsigned int sleep(unsigned int seconds);`

Usypia wywołujący ją proces na określoną w argumencie liczbę sekund. Funkcja zwraca 0 lub liczbę sekund, pozostających do zakończenia drzemki. Sprawia, że proces wywołujący ją jest zawieszany, dopóki nie zostanie wyzerowany licznik czasu określający czas pozostający do końca drzemki lub proces przechwyci sygnał, a procedura jego obsługi po zakończeniu pracy wykona return. . Funkcja zdefiniowana w bibliotece unistd.h.

`unsigned int alarm (unsigned int sec);`

Ustala czas, po jakim zostanie wygenerowany jednorazowo sygnał SIGALARM. Jeśli nie ignorujemy lub nie przechwytyujemy tego sygnału, to domyślną akcją jest zakończenie procesu. Funkcja zdefiniowana w bibliotece unistd.h.

# Zbiory sygnałów

---

Zbiory (zestawy) sygnałów definiuje się, aby pogrupować różne sygnały. Ma to umożliwić wykonywanie pewnych działań na całej grupie sygnałów. Zestaw sygnałów definiowany jest przez typ `sigset_t`. W tej strukturze każdemu sygnaliowi przypisany jest jeden bit (prawda/fałsz), mówiący, czy dany sygnał należy do danego zestawu czy nie.

```
int sigemptyset ( sigset_t* signal_set );
```

Inicjalizacja pustego zbioru sygnałów.

```
int sigfillset ( sigset_t* signal_set );
```

Inicjalizacja zestawu zawierającego wszystkie sygnały istniejące w systemie.

```
int sigaddset ( sigset_t* signal_set, int sig_no );
```

Dodawanie pojedynczego sygnału do zbioru.

```
int sigdelset ( sigset_t* signal_set, int sig_no );
```

Usunięcie pojedynczego sygnału z zestawu.

```
int sigismember ( sigset_t *signal_set, int sig_no );
```

Sprawdzenie, czy w zestawie znajduje się dany sygnał.

## Przykład

---

```
/* utworzenie zbioru dwóch sygnałów SIGINT i SIGQUIT */
sigset_t twosigs;
sigemptyset(&twosigs);
sigaddset(&twosigs, SIGINT);
sigaddset(&twosigs, SIGQUIT);
```

---

# Maskowanie – blokowanie sygnałów

---

Można poinformować jądro o tym iż nie chcemy, aby przekazywano sygnały bezpośrednio do danego procesu. Do tego celu wykorzystuje się zbiory sygnałów zwane maskami. Kiedy jądro usiłuje przekazać do procesu sygnał, który aktualnie jest blokowany, to zostaje on przechowyany do momentu jego odblokowania lub ustawienia ignorowania tego sygnału przez proces.

```
int sigprocmask(int how, const sigset_t *new_set, sigset_t *old_set);
```

Funkcja ustawiająca maskę dla aktualnego procesu.

Parametr *how* definiuje sposób uaktualnienia maski sygnałów. Może przyjmować następujące wartości:

- SIG\_BLOCK - nowa maska to połączenie maski starej i nowej (*new\_set* - zbiór sygnałów, które chcemy blokować).
- SIG\_UNBLOCK - maska podana jako argument to zbiór sygnałów, które chcemy odblokować.
- SIG\_SETMASK - nadpisujemy starą maskę nową.

Do parametru *old\_set* zostanie zapisana poprzednia maska.

## Przykłady wywołania

---

```
#include <signal.h>

sigset_t newmask; /* sygnały do blokowania */
sigset_t oldmask; /* aktualna maska sygnałów */
sigemptyset(&newmask); /* wyczyść zbiór blokowanych sygnałów */
sigaddset(&newmask, SIGINT); /* dodaj SIGINT do zbioru */
/* Dodaj do zbioru sygnałów zablokowanych */
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
    perror("Nie udało się zablokować sygnału");
/* tutaj chroniony kod */
if (sigprocmask(SIG_SETMASK, &newmask, NULL) < 0)
    perror("Nie udało się przywrócić maski sygnałów");
```

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

sigset_t oldmask, blockmask;
pid_t mychild;
sigfillset(&blockmask);
if (sigprocmask(SIG_SETMASK, &blockmask, &oldmask) == -1) {
    perror("Nie udało się zablokować wszystkich sygnałów");
    exit(1);
}
if ((mychild = fork()) == -1) {
    perror("Nie powołano procesu potomnego");
    exit(1);
} else if (mychild == 0) {
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) == -1) {
        perror("Proces potomny nie odtworzył maski sygnałów");
        exit(1);
    }
    /* .....kod procesu potomnego ..... */
} else {
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) == -1) {
        perror("Proces macierzysty nie odtworzył maski sygnałów ");
        exit(1);
    }
    /* ..... kod procesu macierzystego..... */
}
```

# Czekanie na określony sygnał

---

```
int sigpending(sigset_t *set);
```

Służy do odczytania listy sygnałów, które oczekuję na odblokowanie w danym procesie ((ang. *pending signals*)). Do zmiennej set zapisywany jest zestaw oczekujących sygnałów.

```
int sighold(cost sigset_t *set);
```

Służy do odebrania sygnału oczekującego

Tymczasowo zastępuje procesową maskę sygnałów na tę wskazaną parametrem set, a także wstrzymuje działanie procesu do momentu, kiedy nadjdzie odblokowany sygnał. Po obsłudze sygnału ponownie jest ustawiana maska sprzed wywołania sighold.

Zwraca -1 jeśli otrzymany sygnał nie powoduje zakończenia procesu.

# Dziedziczenie

---

- Po wykonaniu funkcji fork proces potomny dziedziczy po swoim przodku wartości maski sygnałów i ustalenia dotyczące obsługi sygnałów.
- Nieobsłużone sygnały procesu macierzystego są czyszczone.
- Po wykonaniu funkcji exec maska obsługi sygnałów i nieobsłużone sygnały są takie same jak w procesie, w którym wywołano funkcję exec.

Oznacz jako wykonane

## Potoki - materiały pomocnicze

### Potoki

Potoki stanowią jeden z podstawowych mechanizmów komunikacji międzyprocesowej w systemie Linux. Potoki umożliwiają wymianę danych pomiędzy procesami w sposób przypominający wykorzystanie pliku (np. dostęp do nich odbywa się poprzez deskryptory plików), natomiast unikają narzutu fizycznych operacji dyskowych, bowiem zapis i odczyt odbywają się do/z pamięci operacyjnej.

Istnieją dwa rodzaje potoków: potoki nienazwane, oraz potoki nazwane, często nazywane też kolejkami lub FIFO. Są one podobne pod względem interfejsu służącego do ich użytkowania, natomiast spełniają nieco różne role i posiadają inne ograniczenia.

### Potoki nienazwane

Potoki nienazwane są najstarszym mechanizmem komunikacji międzyprocesowej (IPC) w systemach unixowych. Zakres oferowanej przez nie funkcjonalności jest mocno ograniczony, jednak często wystarczający, co wraz ze stosunkowo prostym interfejsem sprawia, że pozostają często używane. Ze względu na opisany niżej sposób ich tworzenia, potoki nienazwane umożliwiają komunikację jedynie pomiędzy procesami posiadającymi wspólnego przodka.

Ponadto, potoki nienazwane umożliwiają komunikację tylko w jedną stronę (tzw. half-duplex) – intuicyjnie, potok ma dwa końce odpowiadające komunikującym się procesom i ustalony kierunek przepływu danych, nie można przesyłać danych "pod prąd". Standard Single UNIX Specification dopuszcza potoki dwukierunkowe (full duplex) jako rozszerzenie i niektóre systemy unixopodobne taką funkcjonalność oferują, natomiast nie jest to przenośne (np. Linux tego nie robi). Jeśli potrzebujemy przesyłać dane w obie strony, najlepiej użyć dwóch różnych potoków (albo innego mechanizmu IPC).

### Tworzenie potoku nienazwanego

Do utworzenia potoku nienazwanego służy funkcja pipe z <unistd.h> o sygnaturze

```
int pipe(int fd[2]);
```

Jako argument przyjmuje ona tablicę dwóch liczb całkowitych, do których po stworzeniu potoku zapisywane są deskryptory reprezentujące "wlot", czyli końcówkę do zapisu - fd[1], oraz "wyłot", czyli końcówkę do odczytu - fd[0]. Indeksy te są spójne z numeracją wejścia i wyjścia standardowego. Wartość zwracana standardowo informuje o sukcesie (0) lub porażce (-1). Wywołanie może zakończyć się porażką, jeśli przekażemy nieprawidłowy adres tablicy, lub proces przekroczy limit ilości otwartych deskryptorów.

Ponieważ dwa procesy mogą się komunikować przez łącze tylko w jedną stronę, więc żaden z nich nie wykorzysta obydwoj deskryptorów. Każdy z procesów powinien zamknąć nieużywany deskryptor łącza za pomocą funkcji **close()**.

### Korzystanie z potoku

Do przesyłania danych przez potok wykorzystywane są znane już funkcje read i write, do których jako deskryptory podać należy deskryptor odpowiedniego końca potoku. Domyślnie operacje odczytu i zapisu są blokujące - read czeka, aż w buforze potoku znajdą się jakieś dane (niekoniecznie tyle, ile zażądaliśmy), zaś write czeka, aż zapisane zostaną wszystkie przekazane do niego dane.

Operacje read i write na deskryptorach reprezentujących potoki (zarówno nienazwane, jak i nazwane) mają pewne specjalne zachowania:

- Operacja read na potoku, którego końcówka do zapisu nie ma żadnego otwartego deskryptora zwraca 0 (EOF) po przeczytaniu wszystkich danych
- Operacja write na potoku, którego końcówka do odczytu została zamknięta powoduje wysłanie do procesu piszącego sygnału SIGPIPE i zwrócenie z write wartości -1 plus ustawienie errno na EPIPE. Innymi słowy, nie można pisać do potoku, z którego nikt nie czyta.

Czytanie danych z łącza przebiega zgodnie z poniższymi regułami:

- jeżeli proces chce przeczytać mniejszą ilość danych niż jest w łączu, to proces przeczyta tyle danych ile chciał

- jeżeli proces chce przeczytać większa ilość danych niż jest w łączu, to proces przeczyta tyle danych ile jest w łączu
- jeżeli łącze jest puste, to jeśli została ustawiona flaga O\_NONBLOCK, zostanie zwrócony błąd EAGAIN, w przeciwnym przypadku proces zasypia dopóki łącze nie będzie puste i będzie można przeczytać dane. Jeśli nie ma pisarzy to w obydwu przypadkach zwracane jest 0

Pisanie danych do łącza przebiega zgodnie z poniższymi regułami:

- jeżeli proces chce zapisać mniejszą liczbę danych niż wynosi ilość wolnego miejsca, to zostanie zapisane do łącza tyle bajtów ile proces chciał
- jeżeli proces chce zapisać do łącza większą liczbę bajtów niż wynosi ilość wolnego miejsca, to jeśli została użyta flaga O\_NONBLOCK, zostanie zwrócona liczba bajtów, którą udało się zapisać (jeśli nie udało się nic zapisać, to będzie zwrócony błąd EAGAIN), w przeciwnym przypadku proces zostanie uśpiony, aż do momentu kiedy zwolni się miejsce w łączu
- jeżeli nie ma czytelników funkcja zwróci błąd EPIPE, a do procesu zostanie wysłany sygnał SIGPIPE (standardowa obsługa sygnału SIGPIPE jest zakończenie procesu)
- Jeżeli proces chce zapisać większą liczbę bajtów niż wynosi ilość wolnego miejsca i nie jest ustawiona flaga O\_NONBLOCK, to operacja pisania może nie być niepodzielna (po części danych od jednego procesu może zostać umieszczona część danych od drugiego procesu). W pozostałych przypadkach operacja pisania jest podzielna.

## Użycie potoku pomiędzy procesami

Powyższe informacje pozwalają nam stworzyć potok i przesyłać za jego pomocą dane w obrębie procesu, który go stworzył. By wykorzystać taki potok do komunikacji międzyprocesowej, skorzystamy z faktu, że proces potomny otrzymuje zduplikowane deskryptory plików procesu macierzystego. Stąd, jeśli w procesie A stworzymy potok poprzez wywołanie funkcji pipe, a następnie stworzymy proces potomny B przy użyciu funkcji fork, proces B będzie posiadał otwarte deskryptory obydwu końców potoku, tak samo jak proces A.

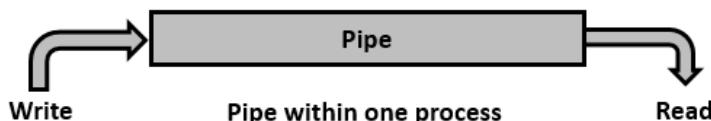
### Komunikacja rodzic - dziecko

W najprostszym wariantie mechanizm ten można wykorzystać do przesyłania danych pomiędzy rodzicem a dzieckiem. Założmy dla przykładu, że chcemy przesyłać dane z rodzica do dziecka. Możemy posłużyć się następującym kodem:

```
int fd[2];
pipe(fd);
pid_t pid = fork();
if (pid == 0) { // dziecko
    close(fd[1]);
    // read(fd[0], ...) – odczyt danych z potoku
} else { // rodzic
    close(fd[0]);
    // write(fd[1], ...) – zapis danych do potoku
}
```

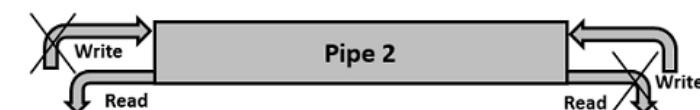
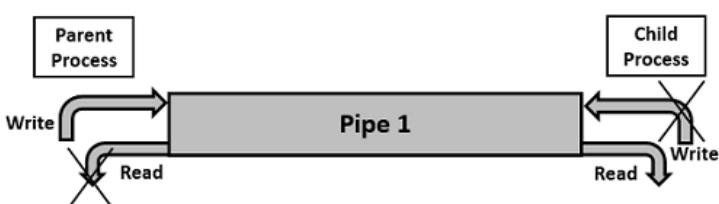
Ze względu na zachowanie funkcji read i write dla potoków opisane wyżej najlepiej po wywołaniu funkcji fork zamknąć w każdym procesie nieużywaną końówkę potoku (deskryptory są *duplicowane*, nie współdzielone, więc są niezależne dla rodzica i dziecka).

Zastosowanie łącza nienazwanego do jednokierunkowej komunikacji między procesami:



Jeden z procesów zamyka łączce do czytania a drugi do pisania. Uzyskuje się wtedy jednokierunkowe połączenie między dwoma procesami.

Zastosowanie łączy nienazwanych do dwukierunkowej komunikacji między procesami:



Dwukierunkowa komunikacja między procesami wymaga użycia dwóch łączy komunikacyjnych. Jeden z procesów pisze do pierwszego łącza i czyta z drugiego, a drugi proces postępuje odwrotnie. Obydwa procesy zamykają nieużywane deskryptory.

## Komunikacja dziecko - dziecko

Analogicznie możemy wykorzystać potoki do komunikacji pomiędzy wieloma procesami potomnymi:

```
int fd[2];
pipe(fd);
if (fork() == 0) { // dziecko 1 - pisarz
    close(fd[0]);
    // ...
} else if (fork() == 0) { // dziecko 2 - czytelnik
    close(fd[1]);
    // ...
}
```

## Przekierowanie wejścia i wyjścia standardowego

Powysze sposoby są wystarczające, jeśli chcemy komunikować się z procesami, które wykonują kod programu, w którym wywołujemy funkcję fork. Czasem chcemy jednak wywołać program zewnętrzny poprzez fork + exec i np. przekazać jakieś dane na jego wejście standardowe, lub stworzyć pipeline przetwarzający dane poprzez "przepuszczenie" danych przez kilka programów. Aby tego dokonać, możemy ustawić w stworzonych procesach wejście/wyjście standardowe na odpowiednie deskryptory potoku używając funkcji dup2:

```
int dup2(int oldfd, int newfd);
```

Jej działanie polega na skopiowaniu deskryptora oldfd na miejsce deskryptora o numerze newfd i w razie potrzeby uprzednim zamknięciu newfd (chyba, że oldfd i newfd są równe, wtedy wywołanie nie robi nic). Podmiana wejścia/wyjścia standardowego sprowadza się do skopiowania deskryptora potoku na miejsce STDIN\_FILENO / STDOUT\_FILENO. Przykładowy kod wywołania grep-a i podmiany jego wejścia standardowego na potok:

```
int fd[2];
pipe(fd);
pid_t pid = fork();
if (pid == 0) { // dziecko
    close(fd[1]);
    dup2(fd[0],STDIN_FILENO);
    execlp("grep", "grep","Ala", NULL);
} else { // rodzic
    close(fd[0]);
    // write(fd[1], ...) – przesłanie danych do grep-a
}
```

Analogicznie możemy łączyć wywołania programów w ciągi, jak robi to np. shell.

## Uproszczanie - popen

Powyszy przypadek użycia pojawia się na tyle często, że biblioteka standardowa udostępnia funkcje pomocnicze do jego obsługi w <stdio.h>:

```
FILE* popen(const char* command, const char* type);
int pclose(FILE* stream);
```

Funkcja popen tworzy potok, nowy proces, ustawia jego wejście lub wyjście standardowe na stosowną końcówkę potoku (zależnie od wartości argumentu type - "r" oznacza, że chcemy odczytać wyjście procesu, "w" że chcemy pisać na jego wejście) oraz uruchamia w procesie potomnym shell (/bin/sh) podając mu wartość command jako polecenie do zinterpretowania. Jeśli operacja ta się powiedzie, popen zwraca obiekt FILE\*, którego można używać z funkcjami wejścia/wyjścia biblioteki standardowej C.

Funkcja pclose oczekuje, aż tak proces powiązany z argumentem stream zakończy swoje działanie. W przypadku błędu zwraca -1, w przeciwnym wypadku status zakończenia tego procesu.

Analogiczne wywołanie grep-a przy pomocy popen:

```
FILE* grep_input = popen("grep Ala", "w");
// fputs(..., grep_input) – przesłanie danych do grep-a
pclose(grep_input);
```

## Potoki nazwane

Jednym z ograniczeń potoków nienazwanych jest to, że pozwalają na komunikację jedynie pomiędzy procesami posiadającymi wspólnego przodka (który musi stworzyć potok). Potoki nazwane omijają to ograniczenie poprzez wykorzystanie systemu plików do identyfikacji potoku - potok nazwany jest reprezentowany przez pewien plik na dysku. Dzięki temu wykorzystywać potok nazwany mogą dowolne procesy, bez

konieczności pokrewieństwa.

**Uwaga:** Należy pamiętać, że wciąż nie jest to to samo, co wykorzystanie zwykłego pliku do przekazywania danych pomiędzy procesami. Plik służy tu tylko do identyfikacji potoku nazwanego, operacje zapisu i odczytu wciąż operują tylko na buforze w pamięci. Plik reprezentujący potok nazwany nie jest plikiem regularnym (S\_IFREG), a plikiem specjalnym typu FIFO (S\_IFIFO).

## Tworzenie potoku nazwanego

Dostępne są dwa narzędzia do utworzenia pliku reprezentującego potok nazwany - mkfifo oraz mknod. Oba są dostępne zarówno jako program który można wywołać w terminalu, jak i jako wywołanie systemowe. Narzędzie mknod jest bardziej ogólne - potrafi tworzyć pliki specjalne różnych typów, mkfifo natomiast tworzy wyłącznie potoki nazwane.

### Tworzenie potoku w terminalu

Utworzyć potok o nazwie NAZWA możemy następującymi poleceniami:

```
mkfifo NAZWA  
mknod NAZWA p
```

Usunąć potok nazwany możemy tak, jak każdy inny plik:

```
rm NAZWA
```

### Tworzenie potoku w programie

Funkcje mkfifo i mknod wymagają dołączenia <sys/types.h> oraz <sys/stat.h>:

```
int mkfifo(const char *pathname, mode_t mode);  
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

Podobnie jak dla funkcji open, istnieje wariant z sufiksem \_at, który pozwala stworzyć potok nazwany o zadanej nazwie w folderze reprezentowanym przez deskryptor. Te warianty wymagają <fcntl.h>:

```
int mkfifoat(int dirfd, const char *pathname, mode_t mode);  
int mknodat(int dirfd, const char *pathname, mode_t mode, dev_t dev);
```

Argument mode działa jak w przypadku funkcji open, z tą różnicą że do funkcji typu mknod w celu stworzenia potoku nazwanego przekazać powinniśmy dodatkowo z-oryginalną flagę S\_IFIFO. Argument dev jest wówczas ignorowany, więc można przekazać np. 0.

## Używanie potoków nazwanych

Aby móc używać potoku nazwanego, musimy otworzyć plik go reprezentujący - open / fopen. Podobnie jak w przypadku potoków nienazwanych, odczyt i zapis odbywa się przy użyciu tych samych funkcji, co w przypadku zwykłych plików - read/write lub funkcje biblioteczne. Reguły są analogiczne do tych przy potoku nazwanym - nie można pisać do potoku, którego nikt nie czyta, przeczytanie wszystkiego z potoku do którego nikt już nie pisze daje efekt taki, jak napotkanie końca pliku.

Domyślnie otwieranie potoku nazwanego w trybie do odczytu blokuje do momentu, gdy jakiś inny proces otworzy potok w trybie do zapisu. Analogicznie w drugą stronę - otwieranie w trybie do zapisu blokuje do momentu, gdy ktoś inny otworzy w trybie do odczytu.

## Uwaga odnośnie wielu procesów (> 2)

W przypadku obu rodzajów potoków możliwe jest skonstruowanie sytuacji, w której wiele procesów pisze do jednego końca potoku i/lub wiele procesów czyta z drugiego końca potoku. W szczególności potok nazwany można próbować wykorzystać jako bufor do którego jedna grupa procesów wpisuje jakieś jednostki danych (np. wyniki obliczeń), a druga grupa je pojedynczo odczytuje i przetwarza.

Problemem w takiej sytuacji staje się atomiczność operacji wejścia/wyjścia. Jeśli procesy A i B próbują jednocześnie zapisywać dane, nie ma w ogólności gwarancji, że całość danych z procesu A zostanie zapisana przed całością danych z procesu B lub odwrotnie - może wystąpić sytuacja, w której zapisany do potoku zostanie kawałek danych procesu A, potem kawałek danych procesu B itd. Podobna sytuacja może mieć miejsce z odczytami.

W przypadku operacji zapisu, POSIX gwarantuje, że żądania zapisu danych wielkości nie większej, niż wartość stałej PIPE\_BUF (co najmniej 512) z pliku <limits.h> będą wykonane atomicznie. Wartości PIPE\_BUF dla różnych systemów można zobaczyć [tutaj](#) (dla Linuxa jest to 4096). Jeśli możemy zatem zagwarantować, że wszystkie procesy piszące do potoku nazwanego zachowują to ograniczenie, wieloprocesowe zapisy są bezpieczne.

W przypadku operacji odczytu nie ma tego rodzaju gwarancji, w związku z czym najlepiej unikać wielu procesów czytających z tego samego potoku nazwanego, a w razie potrzeby zaimplementowania tego rodzaju komunikacji wykorzystać inne mechanizmy.