

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

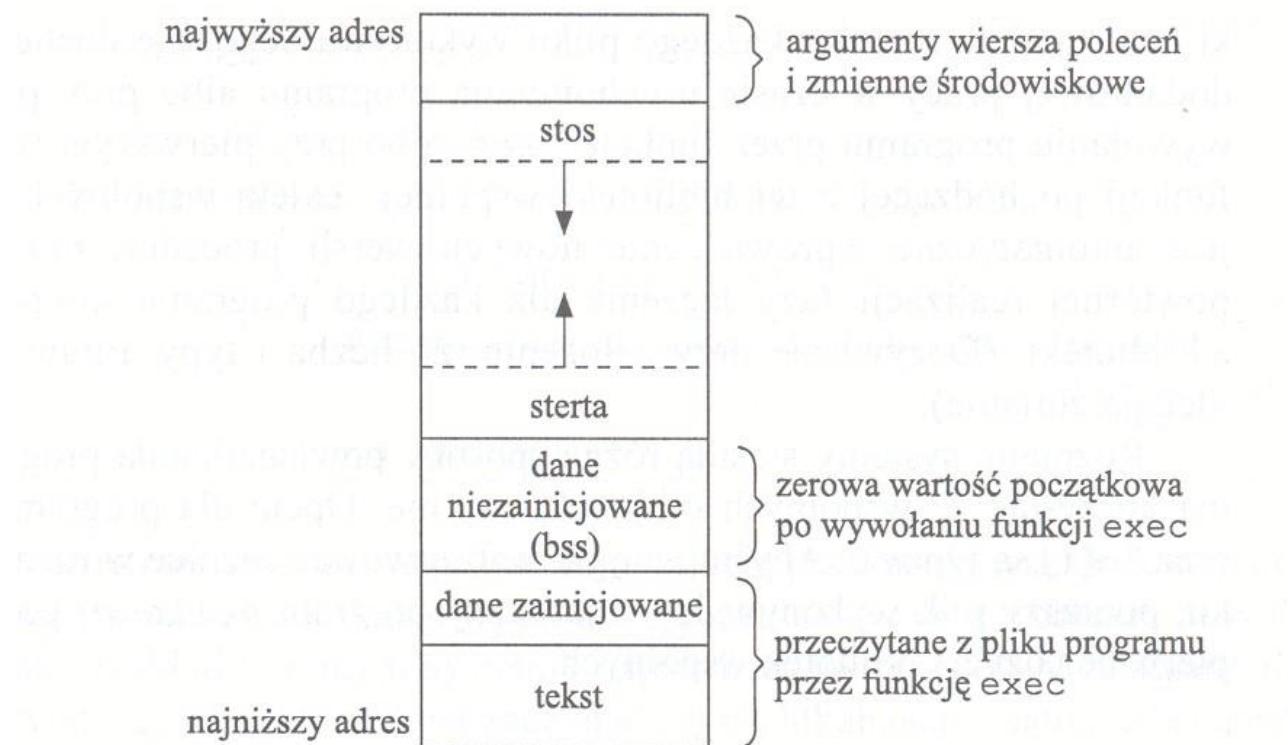
`tms_cutime` – suma czasów cpu wykonywania procesu i wszystkich jego potomków w trybie użytkownika

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 dąstępu 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
SIGKILL	9	Unicestwienie (nie da się przechwycić ani zignorować)	Zakończenie
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
SIGSTOP	19	Zatrzymanie (nie da się przechwycić ani zignorować)	Zatrzymanie
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 > 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 < -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.

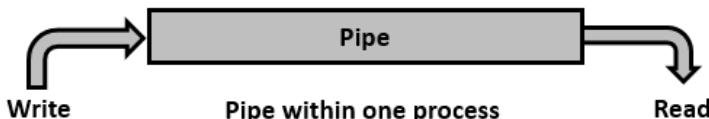
Komunikacja rodzic - dziecko

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
}
```

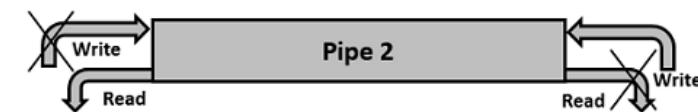
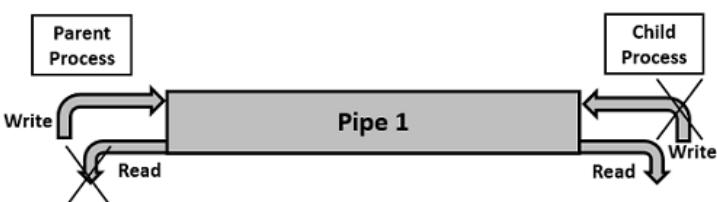
Ze względu na zachowanie funkcji read i write dla potoków opisane wyżej najlepiej po wywołaniu funkcji fork zamknąć w każdym procesie nieużywaną końówkę potoku (deskryptory są *duplicowane*, nie współdzielone, więc są niezależne dla rodzica i dziecka).

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.

Oznacz jako wykonane

Kolejki komunikatów - materiały pomocnicze

Mechanizmy IPC

Podobnie jak łącza, mechanizmy IPC (Inter Process Communication) jest grupą mechanizmów komunikacji i synchronizacji procesów działających w ramach tego samego systemu operacyjnego. Mechanizmy IPC obejmują:

- kolejki komunikatów — umożliwiają przekazywanie określonych porcji danych,
- pamięć współdzieloną — umożliwiają współdzielenie kilku procesom tego samego fragmentu wirtualnej przestrzeni adresowej,
- semafory — umożliwiają synchronizację procesów w dostępie do współdzielonych zasobów (np. do pamięci współdzielonej)

SYSTEM V

Wprowadzenie

Kolejki komunikatów to specjalne listy (kolejki) w jądrze, zawierające odpowiednio sformatowane dane i umożliwiające ich wymianę poprzez dowolne procesy w systemie. Istnieje możliwość umieszczania komunikatów w określonych kolejkach (z zachowaniem kolejności ich wysyłania przez procesy) oraz odbierania komunikatu na parę różnych sposobów (zależnie od typu, czasu przybycia itp.).

W systemie V kolejki komunikatów reprezentowane są przez strukturę msqid_ds.

Do utworzenia obiektu potrzebny jest unikalny **klucz** w postaci 32-bitowej liczby całkowitej. Klucz ten stanowi nazwę obiektu, która jednoznacznie go identyfikuje i pozwala procesom uzyskać dostęp do utworzonego obiektu. Każdy obiekt otrzymuje również swój identyfikator, ale jest on unikalny tylko w ramach jednego mechanizmu. Oznacza to, że może istnieć kolejka i zbiór semaforów o tym samym identyfikatorze.

Wartość klucza można ustawić dowolnie. Zalecane jest jednak używanie funkcji **ftok()** do generowania wartości kluczy. Nie gwarantuje ona sprawdzenie unikalności klucza, ale znacząco zwiększa takie prawdopodobieństwo.

key_t ftok(char *pathname, char proj);

gdzie:

pathname – nazwa ścieżkowa pliku,

proj – jednoliterowy identyfikator projektu.

Wszystkie tworzone obiekty IPC mają ustalone prawa dostępu na podobnych zasadach jak w przypadku plików. Prawa te ustawiane są w strukturze **ipc_perm** niezależnie dla każdego obiektu IPC.

Obiekty IPC pozostają w pamięci jądra systemu do momentu, gdy:

- jeden z procesów zleci jądru usunięcie obiektu z pamięci,
- nastąpi zamknięcie systemu.

Polecenia systemowe

Polecenie **ipcs** wyświetla informacje o wszystkich obiektach IPC istniejących w systemie, dokonując przy tym podziału na poszczególne mechanizmy. Wyświetlane informacje obejmują m.in. klucz, identyfikator obiektu, nazwę właściciela, prawa dostępu.

ipcs [-asmq] [-tclup]

ipcs [-smq] -i id

Wybór konkretnego mechanizmu umożliwiają opcje:

-s – semafory,

-m – pamięć dzielona,

-q – kolejki komunikatów,

-a – wszystkie mechanizmy (ustawienie domyślne).

Dodatkowo można podać identyfikator pojedyńczego obiektu **-i id**, aby otrzymać informacje tylko o nim.

Pozostałe opcje specyfikują format wyświetlanych informacji.

Dowolny obiekt IPC można usunąć posługując się poleceniem:

ipcrm [shm | msg | sem] id

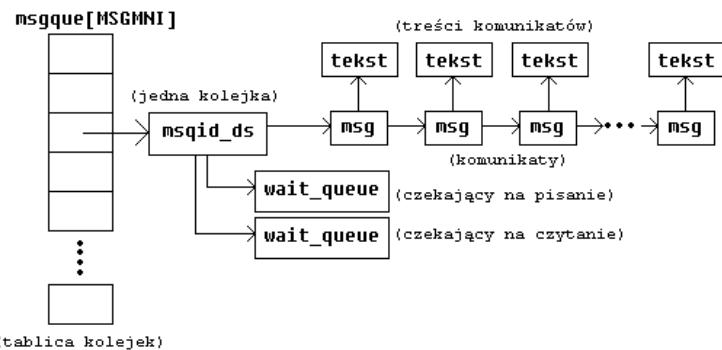
gdzie:

shm, msg, sem - specyfikacja mechanizmu, kolejno: pamięć dzielona, kolejka komunikatów, semafory,

id - identyfikator obiektu.

Struktury danych

Za każdą kolejką komunikatów odpowiada jedna struktura typu msqid_ds. Komunikaty danej kolejki przechowywane są na liście, której elementami są struktury typu msg - każda z nich posiada informacje o typie komunikatu, wskaźnik do następnej struktury msg oraz wskaźnik do miejsca w pamięci, gdzie przechowywana jest właściwa treść komunikatu. Dodatkowo, każdej kolejce komunikatów przydziela się dwie kolejki typu wait_queue, na których spią procesy zawieszone podczas wykonywania operacji czytania bądź pisania do danej kolejki. Poniższy rysunek przedstawia wyżej omówione zależności:



W pliku include/linux/msg.h zdefiniowane są ograniczenia na liczbę i wielkość kolejek oraz komunikatów w nich umieszczanych:

```
#define MSGMNI 128 /* <= 1K max # kolejek komunikatow */
#define MSGMAX 4056 /* <= 4056 max rozmiar komunikatu (w bajtach) */
#define MSGMNB 16384 /* ? max wielkosc kolejki (w bajtach) */
```

Struktura msqid_ds

Dokładna definicja struktury msqid_ds z pliku include/linux/msg.h:

```
/* jedna struktura msg dla każdej kolejki w systemie */
struct msqid_ds {
    struct ipc_perm    msg_perm;
    struct msg        *msg_first;   /* pierwszy komunikat w kolejce */
    struct msg        *msg_last;    /* ostatni komunikat w kolejce */
    __kernel_time_t   msg_stime;    /* czas ostatniego msgsnd */
    __kernel_time_t   msg_rtime;    /* czas ostatniego msgrcv */
    __kernel_time_t   msg_ctime;    /* czas ostatniej zmiany */
    struct wait_queue *wwait;
    struct wait_queue *rwait;
    unsigned short    msg_cbytes;   /* liczba bajtów w kolejce */
    unsigned short    msg_qnum;     /* liczba komunikatów w kolejce */
    unsigned short    msg_qbytes;   /* maksymalna liczba bajtów w kolejce */
    __kernel_ipc_pid_t msg_lspid;   /* pid ostatniego msgsnd */
    __kernel_ipc_pid_t msg_lrpid;   /* pid ostatniego receive*/
};
```

Dodatkowe wyjaśnienia:

msg_perm

Jest to instancja struktury **ipc_perm**, zdefiniowanej w pliku [linux/ipc.h](#). Zawiera informacje o prawach dostępu do danej kolejki oraz o jej założycielu.

wwait, rwait

Przydzielone danej kolejce komunikatów dwie kolejki typu **wait_queue**, na których spią procesy zawieszone podczas wykonywania operacji odpowiednio czytania oraz pisania w danej kolejce komunikatów.

Struktura msg

Dokładna definicja struktury msg z pliku [include/linux/msg.h](#):

```
/* jedna struktura msg dla kazdego komunikatu */
struct msg {
    struct msg *msg_next; /* nastepny komunikat w kolejce */
    long     msg_type;
    char    *msg_spot;
    time_t   msg_stime; /* czas wyslania tego komunikatu */
    short    msg_ts;    /* dlugosc wlasciwej tresci komunikatu */
};
```

Dodatkowe wyjaśnienia:

msg_type

Typ przechowywanego komunikatu. Wysyłanemu do kolejki komunikatowi nadawca przypisuje dodatnią liczbę naturalną, stającą się jego typem. Przy odbiorze komunikatu można zażądać komunikatów określonego typu (patrz opis funkcji msgrcv()).

msg_spot

Wskaźnik do miejsca w pamięci, gdzie przechowywana jest właściwa treść komunikatu. Na każdy komunikat przydzielane jest oddzielne miejsce w pamięci.

Funkcje i ich implementacja

Istnieją cztery funkcje systemowe do obsługi komunikatów:

msgget() uzyskanie identyfikatora kolejki komunikatów używanego przez pozostałe funkcje,

msgctl() ustawianie i pobieranie wartości parametrów związanych z kolejkami komunikatów oraz usuwanie kolejek,

msgsnd() wysłanie komunikatu,

msgrcv() odebranie komunikatu.

Funkcja msgget()

Funkcja służy do utworzenia nowej kolejki komunikatów lub uzyskania dostępu do istniejącej kolejki.

```
DEFINICJA: int msgget(key_t key, int msgflg)
WYNIK: identyfikator kolejki w przypadku sukcesu
       -1, gdy blad: errno = EACCESS (brak praw)
              EEXIST (kolejka o podanym kluczu istnieje,
                        wiec niemożliwe jest jej utworzenie)
              EIDRM (kolejka została w międzyczasie skasowana)
              ENOENT (kolejka nie istnieje),
              EIDRM (kolejka została w międzyczasie skasowana)
              ENOMEM (brak pamięci na kolejce)
              ENOSPC (liczba kolejek w systemie jest równa
                       maksymalnej)
```

Pierwszym argumentem funkcji jest wartość klucza, porównywana z istniejącymi wartościami kluczyc. Zwracana jest kolejka o podanym kluczu, przy czym flaga **IPC_CREAT** powoduje utworzenie kolejki w przypadku braku kolejki o podanym kluczu, zaś flaga **IPC_EXCL** użyta z **IPC_CREAT** powoduje błąd EEXIST, jeśli kolejka o podanym kluczu już istnieje. Wartość klucza równa **IPC_PRIVATE** zawsze powoduje utworzenie nowej kolejki.

W przypadku konieczności utworzenia nowej kolejki, alokowana jest nowa struktura typu msqid_ds.

Funkcja msgsnd()

Wysłanie komunikatu do kolejki.

```
DEFINICJA: int msgsnd(int msqid, struct msgbuf *msgp, int msgsz,
                      int msgflg)
WYNIK: 0 w przypadku sukcesu
       -1, gdy blad: errno = EAGAIN (pelna kolejka (IPC_NOWAIT))
              EACCES (brak praw zapisu)
             EFAULT (zły adres msgp)
              EIDRM (kolejka została w międzyczasie skasowana)
              EINTR (otrzymano sygnal podczas czekania)
              EINVAL (zły identyfikator kolejki, typ lub rozmiar
                       komunikatu)
              ENOMEM (brak pamięci na komunikat)
```

Pierwszym argumentem funkcji jest identyfikator kolejki. msgp jest wskaźnikiem do struktury typu **msgbuf**, zawierającej wysyłany komunikat. Struktura ta jest zdefiniowana w pliku linux/msg.h następująco:

```
/* message buffer for msgsnd and msgrcv calls */
struct msbuf {
    long mtype;          /* typ komunikatu */
    char mtext[1];       /* tresc komunikatu */
};
```

Jest to jedynie przykładowa postać tej struktury; programista może zdefiniować sobie a następnie wysyłać dowolną inną strukturę, pod warunkiem, że jej pierwszym polem będzie wartość typu long, zaś rozmiar nie będzie przekraczać wartości MSGMAX (=4096). Wartość msgsz w wywoaniu funkcji msgsnd jest równa rozmiarowi komunikatu (w bajtach), nie licząc typu komunikatu (sizeof(long)).

Flaga **IPC_NOWAIT** zapewnia, że w przypadku braku miejsca w kolejce funkcja natychmiast zwróci błąd EAGAIN.

Funkcja msgrecv()

Odebranie komunikatu z kolejki.

DEFINICJA:	int msgrecv(int msgqid, struct msbuf *msgp, int msgsz,
	long type, int msgflg)
WYNIK:	liczba bajtów skopiowanych do bufora w przypadku sukcesu
	-1, gdy błąd: errno = E2BIG (długość komunikatu większa od msgsz)
	EACCES (brak praw odczytu)
	EFAULT (zły adres msgp)
	EIDRM (kolejka została w międzyczasie skasowana)
	EINTR (otrzymano sygnał podczas czekania)
	EINVAL (zły identyfikator kolejki lub msgsz < 0)
	ENOMSG (brak komunikatu (IPC_NOWAIT))

Pierwszym argumentem funkcji jest identyfikator kolejki. msgp wskazuje na adres bufora, do którego ma być przekopiowany odbierany komunikat. msgsz to rozmiar owego bufora, z wyłączeniem pola mtype (sizeof(long)). mtype wskazuje na rodzaj komunikatu, który chcemy odebrać. Jądro przydzieli nam najstarszy komunikat zadanego typu, przy czym:

- jeśli mtype = 0, to otrzymamy najstarszy komunikat w kolejce
- jeśli mtype > 0, to otrzymamy komunikat odpowiedniego typu
- jeśli mtype < 0, to otrzymamy komunikat najmniejszego typu mniejszego od wartości absolutnej mtype
- jeśli msgflg jest ustawiona na **MSG_EXCEPT**, to otrzymamy dowolny komunikat o typie różnym od podanego

Ponadto, flaga **IPC_NOWAIT** w przypadku braku odpowiedniego komunikatu powoduje natychmiastowe wyjście z błędem, zaś **MSG_NOERROR** powoduje brak błędu w przypadku, gdy komunikat nie mieści się w buforze (zostaje przekopiowane tyle, ile się mieści).

Funkcja msgctl()

modyfikowanie oraz odczyt rozmaitych właściwości kolejki.

DEFINICJA:	int msgctl(int msgqid, int cmd, struct msqid_ds *buf)
WYNIK:	0 w przypadku sukcesu
	-1, gdy błąd: errno = EACCES (brak praw czytania (IPC_STAT))
	EFAULT (zły adres buf)
	EIDRM (kolejka została w międzyczasie skasowana)
	EINVAL (zły identyfikator kolejki lub msgsz < 0)
	EPERM (brak praw zapisu (IPC_SET lub IPC_RMID))

Dopuszczalne komendy to:

- IPC_STAT:** uzyskanie struktury msgid_ds odpowiadającej kolejce (zostaje ona skopiowana pod adres wskazywany przez buf)
- IPC_SET:** modyfikacja wartości struktury ipc_perm odpowiadającej kolejce
- IPC_RMID:** skasowanie kolejki

Działanie funkcji sprowadza się do przekopiowania odpowiednich wartości od lub do użytkownika, lub skasowania kolejki. Usunięcie kolejki wygląda następująco:

```
{
    msqid_ds.ipc_perm.seq+=1;      /* patrz opis struktury ipc_perm w rozdziale
                                    o cechach wspólnych mechanizmów IPC */
    msg_seq+=1;                  /* zwiększenie wartości globalnej zmiennej związanej z
                                    ipc_perm.seq - patrz tenże rozdział */
    aktualnienie statystyk;
    msgque[msgqid]=IPC_UNUSED;
    obudzenie czekających na pisanie lub czytanie do/z usuwanej kolejki;
    zwolnienie struktur przydzielonych kolejce;
}
```

POSIX

POSIX (ang. *Portable Operating System Interface for UNIX*) – przenośny interfejs dla systemu UNIX. Jest to zestaw standardów opracowany przez stowarzyszenie IEEE (Institute of Electrical and Electronics Engineers) w 1985 roku w celu zapewnienia kompatybilności pomiędzy różnymi wersjami i dystrybucjami systemów operacyjnych. Standard ten definiuje zarówno interfejs programistyczny (API), jak i powłokę systemową oraz interfejs użytkownika.

Kolejki komunikatów

Służą do wymiany komunikatów (ciągu danych o ustalonej długości i priorytecie) pomiędzy procesami. Kolejka to tak naprawdę lista, z której w czasie odczytu pobieramy najstarszy komunikat o najwyższym priorytecie (wg standardu POSIX). Pojedynczy komunikat zawiera priorytet (unsigned int), długość (size_t) oraz same dane, o ile długość jest większa niż 0 (char*).

Kolejki komunikatów tworzone są w określonym katalogu na dysku, np /DEV/mqueue.

Funkcje do obsługi kolejek komunikatów

Plik nagłówkowy

```
#include <mqueue.h>
```

Struktura struct mq_attr

```
struct mq_attr {  
    long mq_flags; /* sygnalizator kolejki: 0, O_NONBLOCK */  
    long mq_maxmsg; /* maksymalna liczba komunikatów w kolejce */  
    long mq_msgsize; /* maksymalny rozmiar komunikatu (w bajtach) */  
    long mq_curmsgs; /* liczba komunikatów w kolejce */  
};
```

Otwieranie kolejki

```
mqd_t mq_open(const char *name, int oflag [, mode_t mode, struct mq_attr *attr]);
```

Funkcja ta próbuje otworzyć kolejkę komunikatów (która tak naprawdę jest plikiem o nazwie name). Zwraca deskryptor kolejki, jeśli się powiedzie lub -1 w przypadku błędu.

Uwaga! Nazwa musi zaczynać się od znaku /

Parametr oflag ma analogiczne znaczenie, jak w przypadku otwierania plików (unixowymi metodami obsługi plików). Zatem akceptuje jedną z wartości: O_RDONLY, O_WRONLY, O_RDWR, którą można zsumować logicznie z wartościami: O_CREAT, O_EXCL oraz O_NONBLOCK (aby używać tych stałych należy dodać plik nagłówkowy fcntl.h).

Parametr mode specyfikujemy, gdy tworzymy nową kolejkę i określa on prawa dostępu do niej. Możemy podać wartość ósemkowo lub dowolną sumę logiczną stałych: S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, S_IWOTH (aby korzystać z tych stałych należy dodać plik nagłówkowy sys/stat.h).

Ostatnim parametrem jest attr. Jest to struktura określająca parametry kolejki. Jeśli nie podamy tego parametru lub podamy NULL, to ustawione zostaną parametry domyślne.

Zamykanie kolejki

```
int mq_close(mqd_t mqdes);
```

Funkcja ta zamyka kolejkę o deskryptorze mqdes. Zwraca 0 w przypadku sukcesu lub -1 w przypadku błędu.

Warto zauważyć, że funkcja ta **nie niszczy** kolejki, która dalej jest dostępna w systemie operacyjnym, ale jedynie ją zamyka.

Gdy proces się kończy, automatycznie zamykane są wszystkie jego kolejki.

Usuwanie kolejki

```
int mq_unlink(const char *name);
```

Usuwa z systemu kolejkę o nazwie name. Zwraca 0 w przypadku sukcesu lub -1 w przypadku błędu.

Kolejka zostanie usunięta dopiero po zamknięciu jej przez wszystkie podłączone procesy.

Odczytywanie parametrów kolejki

```
int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
```

Odczytuje parametry kolejki o deskryptorze mqdes i zapisuje je w miejscu wskazywanym przez attr. Zwraca 0 w przypadku sukcesu lub -1 w przypadku błędu.

Ustawianie parametrów kolejki

```
int mq_setattr(mqd_t mqdes, const struct mq_attr *attr, struct mq_attr *oattr);
```

Ustawia parametry kolejki o deskryptorze mqdes wskazywane przez attr. Jeśli oattr nie wskazuje na NULL, to zapisywane są w tym miejscu stare parametry kolejki. Zwraca 0 w przypadku sukcesu lub -1 w przypadku błędu.

Wysyłanie komunikatów

```
int mq_send(mqd_t mqdes, const char* ptr, size_t len, unsigned int prio);
```

Wysyła komunikat wskazywany przez ptr do kolejki o deskryptorze mqdes o długości len i priorytecie prio. Zwraca 0 w przypadku powodzenia lub -1 w przypadku błędu.

Priorytet nie może przekraczać MQ_PRIO_MAX!

Odbieranie komunikatów

```
ssize_t mq_receive(mqd_t mqdes, char *ptr, size_t len, unsigned int *priop);
```

Odbiera komunikat z kolejki o deskryptorze mqdes o długości len (co najmniej tyle, ile w polu mq_msgsize w strukturze struct mq_attr). Dane zapisuje do ptr, a priorytet do priop (o ile priop nie jest NULL). Zwraca liczbę odczytanych bajtów w przypadku powodzenia lub -1 w przypadku błędu.

Mechanizm powiadomień (ang. notifications)

Mechanizm powiadomień pozwala na asynchroniczne zawiadamianie procesu, że w pustej kolejce umieszczono komunikat. Może to się odbyć poprzez:

- wysłanie sygnału
- utworzenie wątku w celu wykonania określonej funkcji

Korzystanie z mechanizmu powiadomień

```
int mq_notify(mqd_t mqdes, const struct sigevent *notification);
```

Funkcja ta powoduje zarejestrowanie (gdy notification nie jest równe NULL) lub wyrejestrowanie (gdy notification jest NULL) mechanizmu powiadomień dla kolejki o deskryptorze mqdes. Zwraca 0 w przypadku powodzenia lub -1 w przypadku błędu. Struktura struct sigevent wygląda następująco:

```
struct sigevent {  
    int      sigev_notify; /* sygnał czy wątek: SIGEV_NONE, SIGEV_SIGNAL, SIGEV_THREAD */  
    int      sigev_signo;  /* numer sygnału (dla SIGEV_SIGNAL) */  
    union sigval sigev_value; /* przekazywane procedurze obsługi sygnału lub wątkowi */  
    /* dla SIGEV_THREAD występują jeszcze: */  
    void (*sigev_notify_function)(union sigval);  
    pthread_attr_t *sigev_notify_attributes;  
};  
  
union sigval {  
    int sival_int; /* wartość całkowitoliczbową */  
    void *sival_ptr; /* wskaźnik */  
};
```

Korzystając z mechanizmu powiadomień należy pamiętać, że:

- W jednym procesie możemy korzystać z powiadomień tylko z jednej kolejki.
- Rejestracja obowiązuje tylko na jedno powiadomienie. Po powiadomieniu trzeba zarejestrować się ponownie, gdyż rejestracja jest kasowana.
- Jeśli w pustej kolejce pojawi się komunikat, a jednocześnie proces oczekuje na rezultat funkcji mq_receive, to do procesu **nie zostanie** wysłane powiadomienie. Nie ma to większego sensu, gdyż proces i tak oczekuje na wiadomość.

Oznacz jako wykonane

Uwaga: podstawowe aspekty IPC Systemu V oraz POSIX, np. sposób tworzenia kluczy czy identyfikacji obiektów, zostały omówione w materiałach pomocniczych do Laboratorium 6 i nie są one ponownie omawiane w niniejszym dokumencie.

Semafora

Semafora są jednym z najważniejszych mechanizmów synchronizacji dostępu do współdzielonych zasobów. Można je rozumieć jako zmienne licznikowe dla których zdefiniowano dwie atomowe operacje:

- Dekrementacja semafora: powoduje zmniejszenie wartości semafora o 1, o ile jego aktualna wartość jest > 0 . Jeśli aktualna wartość semafora wynosi 0, operacja dekrementacji powoduje zablokowanie procesu który próbuje ją wykonać. Proces ten będzie zablokowany do momentu zwiększenia wartości semafora (przez inny proces). Po odblokowaniu proces zmniejszy wartość semafora o 1.
- Inkrementacja semafora: powoduje zwiększenie wartości semafora o 1. Jeśli semafor blokuje jeden lub więcej procesów, zwiększenie jego wartości spowoduje odblokowanie jednego oczekującego procesu.

Semafora w IPC systemu V.

W systemie V semantyka semaforów jest rozszerzona w stosunku do klasycznej ich definicji:

1. Semafora IPC systemu V można zmniejszać lub powiększać o wartości większe niż 1. Jednak po każdej operacji wartość semafora musi być ≥ 0 . Operacja, która narusza ten warunek jest blokowana. Blokada trwa do momentu, gdy wartość semafora pozwoli wykonać operację z zachowaniem warunku wartości końcowej ≥ 0 .
2. W systemie V można wykonać operacje na wielu semaforach równocześnie. Operacje te wykonywane są w sposób atomowy, tzn. wykonywane są wszystkie wskazane operacje (jeśli jest to możliwe) lub wszystkie wskazane operacje są blokowane (jeśli ze względu na aktualne wartości semaforów nie można wykonać wszystkich operacji na raz).
3. System V definiuje operację blokowania procesu do momentu, gdy semafor przyjmie wartość 0.

Operacje na semaforach systemu V są zdefiniowane w pliku nagłówkowym **sys/sem.h**. Dodatkowo warto również dołączyć pliki nagłówkowe **sys/ipc.h** i **sys/types.h**.

Aby stworzyć zbiór semaforów korzystamy z funkcji **semget**

```
int semget(key_t key, int nsems, int flag);
```

Funkcja ta zwraca identyfikator zbioru semaforów, który można następnie wykorzystać w innych funkcjach operujących na semaforach. Argument **key** wskazuje klucz zbioru semaforów, zaś w argumencie **flag** przekazywane są flagi modyfikujące proces tworzenia obiektu IPC (patrz materiały do Laboratorium 6). Liczba semaforów do utworzenia przekazywana jest w argumencie **nsems**. Funkcję **semget** można również wykorzystać do pobrania identyfikatora istniejącego już zbioru semaforów (np. utworzonego przez inny proces). Wówczas wartość **nsems** powinna wynosić 0. Bezpośrednio po utworzeniu, wartości semaforów są niezdefiniowane. Należy je zainicjalizować, np. za pomocą funkcji **semctl** (patrz poniżej).

Do wykonania operacji na zbiorze semaforów służy funkcja **semop**

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

Funkcja ta wykonuje operacje na zbiorze semaforów o identyfikatorze **semid**. Liczba operacji do wykonania jest zdefiniowana w argumencie **nsops**. Same operacje przekazywane są w tablicy **sops** (zawierającej **nsops** elementów). Każda operacja zdefiniowana jest jako struktura postaci:

```
struct sembuf
{
    unsigned short sem_num;
    short sem_op;
    short sem_flg;
};
```

gdzie:

- **sem_num** to numer semafora (w zbiorze) na którym należy wykonać operację,
- **sem_op** to operacje do wykonania,

- **sem_flg** to flagi operacji.

Wartość **sem_op < 0** oznacza operację zmniejszenia wartości semafora o **sem_op**. Wartość **sem_op > 0** oznacza operację zwiększenia wartości semafora o **sem_op**. Wartość **sem_op == 0** oznacza operację oczekiwania, aż wartość semafora będzie wynosić 0. Dla pola **sem_flg** zdefiniowano dwie istotne flagi: flaga **IPC_NOWAIT** oznacza, iż operacja nie powinna blokować procesu. Jeśli operacji nie można wykonać (ze względu na wartość semafora) i ustawiona jest flaga **IPC_NOWAIT**, funkcja **semop** zwróci błąd. Flaga **SEM_UNDO** oznacza, że w przypadku zakończenia procesu operacja wykonana na semaforze powinna zostać cofnięta.

Funkcja **semctl**, pozwala wykonać pewne dodatkowe operacje na zbiorze semaforów. Funkcja ta ma sygnature:

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

gdzie:

- **semid** to identyfikator zbioru semaforów,
- **semnum** to numer semafora w zbiorze,
- **cmd** to operacja do wykonania,
- **arg** to unia bitowa przekazująca pewne dodatkowe argumenty.

Najważniejsze operacje (cmd) do wykonania za pomocą funkcja **semctl** to:

- **SETVAL** ustawienie wartości semafora na liczbę przekazaną w polu **arg.val**
- **GETVAL** pobranie wartości semafora,
- **IPC_RMID** usunięcie zbioru semaforów z systemu.

Semafora w IPC POSIX.

Semafora POSIXa realizują klasyczną semantykę semaforów. Aby z nich korzystać do programu należy dołączyć pliki nagłówkowe: **semaphore.h**, **sys/stat.h** oraz **fcntl.h**. Ponadto program należy zlinkować z biblioteką **pthread**.

Do utworzenia semafora służy funkcja **sem_open**

```
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
```

Funkcja ta zwraca adres semafora lub **SEM_FAILED** w przypadku wystąpienia błędu. Argument **name** określa nazwę semafora, zaś argument **oflag** określa tryb otwarcia (patrz materiały do Laboratorium 6). Argument **value** określa początkową wartość semafora. Inkrementację semafora (o wartość 1) realizuje funkcja **sem_post**

```
int sem_post(sem_t *sem);
```

Funkcja ta przyjmuje wskaźnik na semafor i zwraca 0 w przypadku sukcesu oraz -1 w przypadku wystąpienia błędu. Dekrementacja semafora realizowana jest analogicznie zdefiniowaną funkcją **sem_wait**

```
int *sem_wait(sem_t *sem);
```

Funkcja ta posiada również wariant nieblokującym **sem_trywait**

```
int *sem_trywait(sem_t *sem);
```

Jeśli dekrementacja semafora nie jest możliwa (semafor ma wartość 0) funkcja **sem_trywait** nie blokuje procesu, lecz zwraca wartość -1 i ustawia zmienną **errno** na **EAGAIN**.

POSIX pozwala również odczytać aktualną wartość semafora. Służy do tego funkcja **sem_getvalue**

```
int sem_getvalue(sem_t *sem, int *valp);
```

Aktualna wartość semafora zapisywana jest pod adresem wskazywanym przez argument **valp**

Po zakończeniu pracy z semaforem należy go zamknąć. Służy do tego funkcja **sem_close**

```
int sem_close(sem_t *sem);
```

Semafor usuwamy za pomocą funkcji **sem_unlink**.

Funkcje **sem_getvalue** oraz **sem_close** zwracają 0 w przypadku sukcesu oraz -1 w przypadku wystąpienia błędu.

Pamięć wspólna

Co do zasady, każdy proces w systemie posiada odrębną przestrzeń adresową. Zmiany zawartości pamięci dokonywane przez jeden proces nie są widoczne w innych procesach. Mechanizm pamięci wspólnej stanowi wyjątek od tej zasady – umożliwia połączanie segmentu pamięci do przestrzeni adresowej wielu procesów. Pozwala to na komunikację między procesami bez konieczności dodatkowego kopiowania danych. Mechanizm ten nie zapewnia jednak żadnej synchronizacji dostępu do wspólnej pamięci. Z reguły konieczne więc jest wykorzystanie dodatkowych mechanizmów synchronizacji (np. semaforów) aby zapewnić prawidłową sekwencję odczytów/zapisów z/do pamięci wspólnej.

Pamięć wspólna w IPC systemu V.

W IPC systemu V operacje na pamięci wspólnej są zdefiniowane w pliku nagłówkowym `sys/shm.h`. Dodatkowo warto również dołączyć pliki nagłówkowe `sys/ipc.h` oraz `sys/types.h`.

Aby stworzyć segment pamięci wspólnej korzystamy z funkcji `shmget`

```
int shmget(key_t key, size_t size, int shmflg);
```

Funkcja ta zwraca identyfikator segmentu pamięci wspólnej. Argument `key` oznacza klucz segmentu pamięci wspólnej a argument `flag` przekazuje flagi modyfikujące proces tworzenia obiektu IPC (patrz materiały do Laboratorium 6). Rozmiar segmentu pamięci wspólnej przekazywany jest w argumencie `size`. Funkcja ta pozwala również uzyskać identyfikator istniejącego już segmentu pamięci wspólnej. Wówczas argument `size` powinien mieć wartość 0. Dołączenie segmentu pamięci wspólnej do przestrzeni adresowej procesu realizuje funkcja `shmat`

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Funkcja ta zwraca adres pod którym dołączono segment pamięci wspólnej. W przypadku błędu zwracana jest wartość `(void *) -1`. W argumencie `semid` przekazywany jest identyfikator segmentu. Argument `shmaddr` pozwala wskazać adres, pod którym system powinien dołączyć segment pamięci wspólnej. Zaleca się by miał on wartość `NULL`, oznaczającą, że system operacyjny sam dobierze odpowiedni adres. W argumencie `shmflg` przekazywane są flagi modyfikujące działanie funkcji `shmat`. Najciekawszą z nich jest flaga `SHM_RDONLY`, pozwalająca dołączyć segment pamięci w trybie tylko do odczytu. Po zakończeniu pracy z segmentem należy go odłączyć, korzystając z funkcji `shmdt`

```
int shmdt(const void *shmaddr);
```

Funkcja ta przyjmuje w argumencie adres zwrócony przez funkcję `shmat` i zwraca 0 w przypadku powodzenia oraz -1 w przypadku błędu. Na koniec warto również wspomnieć o funkcji `shmctl`

```
void *shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Pozwala ona, między innymi, usunąć segment pamięci wspólnej z systemu. W tym celu należy ją wywołać przekazując jako drugi argument polecenie `IPC_RMID` (argument `buf` jest wówczas ignorowany). Po wykonaniu tego polecenia nie będzie możliwe dołączenie segmentu do kolejnych procesów. Segment zostanie usunięty po odłączeniu przez wszystkie procesy, które uprzednio dołączyły go do swojej przestrzeni adresowej.

Pamięć wspólna w IPC POSIX.

Aby korzystać z pamięci wspólnej w IPC POSIX do programu należy dołączyć pliki nagłówkowe: `sys/mman.h`, `sys/stat.h` oraz `fcntl.h`. Ponadto program należy zlinkować z biblioteką `rt`

Do utworzenia segmentu pamięci wspólnej (lub otwarcia istniejącego już segmentu) służy funkcja `shm_open`

```
int shm_open(const char *name, int oflag, mode_t mode);
```

Funkcja ta zwraca deskryptor plików reprezentujący segment pamięci wspólnej. W przypadku błędu zwracana jest wartość -1. Argument `name` określa nazwę segmentu, zaś argument `oflag` określa tryb otwarcia (patrz materiały do ćwiczenia 6). Po utworzeniu segmentu należy określić jego rozmiar. W tym celu należy skorzystać z funkcji `ftruncate`

```
int ftruncate(int fd, off_t length);
```

W pierwszym argumencie przekazywany jest deskryptor zwrócony przez funkcję `shm_open`. W drugim argumencie przekazywany jest pożądany rozmiar segmentu (w bajtach). Funkcja ta zwraca 0 w przypadku powodzenia oraz -1 w przypadku wystąpienia błędu. Otwarty segment pamięci wspólnej należy dołączyć do przestrzeni adresowej procesu. Operację tą można zrealizować za pomocą funkcji `mmap`

```
void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

Funkcja ta zwraca adres dołączonego segmentu lub, w przypadku błędu, wartość `(void *) -1`. W argumentach przekazywane są:

- `addr` określa adres pod którym powinien zostać dołączony segment pamięci wspólnej; zaleca się przekazanie wartości `NULL`, wskazującej, że system sam powinien dobrać odpowiedni adres,
- `len` to liczba bajtów segmentu mapowanych do przestrzeni adresowej procesu,
- `prot` określa prawa dostępu do mapowanej pamięci; prawa te są określone flagami `PROT_READ` (odczyt), `PROT_WRITE` (zapis), `PROT_EXEC` (prawo wykonania), `PROT_NONE` (brak uprawnień),
- `flags` – specyfikacja użycia segmentu (np. `MAP_SHARED`, `MAP_PRIVATE`, `MAP_FIXED`)
- `fd` jest deskryptorem plików zwróconym przez funkcję `shm_open`,
- `offset` określa przesunięcie mapowanego obszaru względem początku segmentu pamięci wspólnej; z reguły przyjmuje wartość 0.

Po zakończeniu pracy z segmentem pamięci wspólnej należy go odłączyć od przestrzeni adresowej procesu. Służy do tego funkcja `munmap`

```
int munmap(void *addr, size_t len);
```

W argumencie `addr` należy przekazać adres, pod którym segment został dołączony a w argumencie `len` rozmiar segmentu. Odłączony segment można następnie oznać do usunięcia. Służy do tego funkcja `shm_unlink`

```
int shm_unlink(const char *name);
```

W argumencie `name` przekazywana jest nazwa segmentu do usunięcia. Po wykonaniu funkcji `shm_unlink` nie będzie już możliwe otwarcie tego segmentu funkcją `shm_open`. Co więcej, po odłączeniu go przez wszystkie procesy, które uprzednio dołączyły go do swojej przestrzeni adresowej, zostanie on usunięty z zasobów systemu. Funkcje `munmap` i `shm_unlink` zwracają 0 w przypadku powodzenia oraz -1 w przypadku błędu.

Ostatnia modyfikacja: wtorek, 25 kwietnia 2023, 12:40



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

Pobierz aplikację mobilną



Wybierz język



Oznacz jako wykonane

Informacje wstępne

W obrębie jednego procesu może istnieć wiele wątków, które działają we wspólnej przestrzeni adresowej i są wykonywane współbieżnie. Wątki są identyfikowane za pomocą ID typu `pthread_t`. Każdy wątek posiada własny odrębny stos.

Proces kończy swoje działanie, gdy zakończą swoje działanie wszystkie wątki, ale również zakończenie procesu, poprzez np. zwrocenie wartości z funkcji `main`, kończy działanie wszystkich wątków.

Wątki współdzielą:

- przestrzeń adresową (w szczególności zmienne globalne)
- identyfikatory związane z procesem (PID, UID, PPID, ...)
- deskryptory plików
- sposób obsługi sygnałów (signal disposition - ignorowanie/obsługa domyślna/handler)
- limity i liczniki zużycia zasobów
- inne: rygle na pliki, umask, katalog główny/bieżący, wartość nice

Każdy wątek posiada własne:

- thread ID (TID)
- **maska** sygnałów
- wartość errno
- dane własne (thread local)
- stos (w szczególności zmienne lokalne)
- inne: polityki szeregowania, CPU affinity, security capabilities, alternate signal stack

Do obsługi wątków w standardzie POSIX służy biblioteka `pthreads`. Aby jej używać, należy dołączyć plik nagłówkowy `pthreads.h` oraz doliczować bibliotekę `pthread` (-lpthread)

Tworzenie wątków

Do tworzenia nowych wątków służy funkcja `pthread_create`

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg)
```

gdzie

- `thread` - wskaźnik na miejsce gdzie zapisany zostanie identyfikator utworzonego wątku
- `attr` - dodatkowe ustawienia (opcjonalne, `NULL` = ustawienia domyślne)
- `start_routine` - adres funkcji, która ma zostać wykonana w utworzonym wątku
- `arg` - argument, z którym ma zostać wywołana

Funkcja na którą wskazuje `start_routine` przyjmuje i zwraca wskaźnik na dowolne dane (`void*`). Nie ma gwarancji, że nowo utworzony wątek zacznie swoje działanie natychmiast - nie należy zatem przekazywać jako argumentów zmiennych lokalnych które mogą ulec zniszczeniu zanim nowy wątek zacznie działać.

Typ identyfikatora wątku `pthread_t` jest zależny od implementacji - może nie być to typ całkowitoliczbowy. Do porównywania równości dwóch wartości typu `pthread_t` służy funkcja `pthread_equal`:

```
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

zwracająca 0 jeśli `tid1` nie jest równy `tid2`. Swój własny identyfikator wątek może pobrać przy użyciu `pthread_self`:

```
pthread_t pthread_self(void);
```

Argument `attr` pozwala kontrolować różne aspekty działania tworzonego wątku. Przed przekazaniem adresu struktury `pthread_attr_t` należy ją zainicjalizować przy użyciu funkcji `pthread_attr_init`:

```
int pthread_attr_init(pthread_attr_t *attr);
```

Po wywołaniu pthread_create należy zwolnić potencjalnie zaalokowane przez to wywołanie zasoby używając funkcji pthread_attr_destroy:

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

Cykl życia wątku

Działanie wątku może zostać zakończone w wyniku kilku zdarzeń:

- jeśli wątek główny zakończy swoje działanie poprzez zwrócenie wartości z funkcji main, cały proces (wszystkie jego wątki) kończy swoje działanie
- jeśli którykolwiek wątek procesu wywoła exit, _Exit lub _exit, cały proces kończy swoje działanie
- jeśli którykolwiek wątek otrzyma sygnał, którego domyślnym sposobem obsługi jest zakończenie procesu, cały proces kończy swoje działanie
- jeśli wątek zwróci wartość ze swojej funkcji, kończy on swoje działanie
- jeśli wątek wywoła pthread_exit, kończy on swoje działanie
- jeśli wątek zostanie anulowany przez inny wątek, kończy on swoje działanie

Funkcja

```
void pthread_exit(void *rval_ptr);
```

przyjmuje wartość która zostanie użyta jako wartość zwrócona przez wątek. Zwrócenie wartości val z funkcji wątku jest równoważne z wywołaniem pthread_exit(val) (poza wątkiem głównym - w nim zwrócenie val jest równoważne z exit(val)). Wywołanie pthread_exit z wątku głównego powoduje zakończenie wątku głównego, ale pozostałe wątki nadal mogą działać - cały proces zakończy się wówczas dopiero gdy zakończą swoje działanie wszystkie utworzone wątki, a kodem wyjścia procesu będzie 0.

Przerwać działanie wątku z innego wątku można przy pomocy funkcji pthread_cancel:

```
int pthread_cancel(pthread_t tid);
```

Reakcja wątku na bycie anulowanym zależy od jego ustawień. Wątek może dopuszczać (zachowanie domyślne) lub odrzucać żądania anulowania, sterować tym zachowaniem można przy użyciu funkcji pthread_setcancelstate

```
int pthread_setcancelstate(int state, int *oldstate);
```

podając jako pierwszy argument PTHREAD_CANCEL_ENABLE lub PTHREAD_CANCEL_DISABLE. Anulowanie wątku z zablokowanym anulowaniem powoduje, że wątek zostanie anulowany w chwili gdy z powrotem odblokuje możliwość anulowania. Jeśli wątek dopuszcza anulowanie, może to robić w dwóch trybach:

- PTHREAD_CANCEL_DEFERRED (domyślnie) - wątek kontynuuje swoje działanie do momentu napotkania tzw. *cancellation point*
- PTHREAD_CANCEL_ASYNCHRONOUS - wątek kończy swoje działanie natychmiast

Zmienić tryb anulowania można przy pomocy funkcji pthread_setcanceltype:

```
int pthread_setcanceltype(int type, int *oldtype);
```

Cancellation point to wywołanie jednej z wymienionej jako takowy przez standard POSIX funkcji (np. read, write, pause). W szczególności cancellation point stanowi wywołanie funkcji pthread_testcancel.

Domyślnie po zakończeniu działania wątku wartość którą zwrócił można pobrać przy użyciu funkcji pthread_join:

```
int pthread_join(pthread_t thread, void **rval_ptr);
```

Jeśli wątek został anulowany, pod rval_ptr zapisana zostanie wartość PTHREAD_CANCELED. Jeśli chcemy, by dane zakończonego wątku nie były przechowywane do czasu wywołania pthread_join, a były usuwane natychmiast, możemy wątek uznać wątkiem odłączonym. Można to uczynić na dwa sposoby: w momencie tworzenia wątku, lub później.

Aby utworzyć wątek od razu jako wątek odłączony, należy do pthread_create przekazać adres struktury pthread_attr_t po wywołaniu na niej pthread_attr_setdetachstate

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

jako drugi argument przekazując PTHREAD_CREATE_DETACHED.

Istniejący wątek można po utworzeniu odłączyć przy użyciu funkcji pthread_detach:

```
int pthread_detach(pthread_t tid);
```

Wątki i sygnały

Sygnały są dostarczane do jednego wątku w procesie. Sygnały związane ze zdarzeniami sprzętowymi są z reguły dostarczane do wątku, który je spowodował, pozostałe - arbitralnie.

Wątki mają odrębne maski sygnałów, ale wspólne ustawienia ich obsługi (signal disposition). Do ustawienia maski sygnału wątku służy funkcja pthread_sigmask:

```
int pthread_sigmask(int how, const sigset_t* set, sigset_t* oset);
```

o sygnaturze identycznej do sigprocmask. Działanie sigprocmask dla programu wielowątkowego jest niezdefiniowane.

Aby wysłać sygnał do konkretnego wątku, należy użyć funkcji pthread_kill lub pthread_sigqueue:

```
int pthread_kill(pthread_t thread, int signo);
int pthread_sigqueue(pthread_t thread, int sig, const union sigval value);
```

Ostatnia modyfikacja: wtorek, 25 kwietnia 2023, 16:32



AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

AGH

Platforma obsługiwana przez:

[Centrum e-Learningu i Innowacyjnej Dydaktyki AGH](#)

[Centrum Rozwiązań Informatycznych AGH](#)

Pobierz aplikację mobilną



Wybierz język



Oznacz jako wykonane

Mutex-y

Mutex (*MUTual EXclusion*, wzajemne wykluczanie) jest blokadą, którą może uzyskać **tylko jeden wątek**. Mutexy służą głównie do realizacji sekcji krytycznych, czyli bezpiecznego w sensie wielowątkowym dostępu do zasobów współdzielonych.

Schemat działania na mutexach jest następujący:

1. pozyskanie blokady
2. modyfikacja lub odczyt współdzielonego obiektu
3. zwolnienie blokady

Mutex w **pthreads** jest opisywany przez strukturę typu **pthread_mutex_t**, zaś jego atrybuty **pthread_mutexattr_t**.

Obiekt **pthread_mutex_t** (wzajemnego wykluczania) musi być przed użyciem zainicjalizowany, co odbywa się za pomocą funkcji [pthread_mutex_init](#).

int pthread_mutex_init (pthread_mutex_t *mutex, pthread_mutexattr_t *attr);

- mutex – wcześniej stworzony przez nas mutex
- attr – atrybuty tworzonego mutexu (domyślne ustawienia: NULL)

Zablokowanie obiektu przez wątek może zostać wykonane poprzez jedną z następujących funkcji:

- **int pthread_mutex_lock(pthread_mutex_t *mutex)** – która jeśli obiektu mutex-a jest zablokowany przez inny wątek usypia obecny wątek, aż mutex zostanie odblokowany. Z kolei jeśli obiekt mutex-a jest już zablokowany przez obecny wątek to albo:
 - usypia wywołujący ją wątek (jeśli jest to mutex typu "fast")
 - zwraca natychmiast kod błędu EDEADLK (jeśli jest to mutex typu "error checking")
 - normalnie kontynuuje pracę, zwiększając licznik blokad mutex-a przez dany wątek (jeśli mutex jest typu "recursive"); odpowiednia liczba razy odblokowań musi nastąpić aby mutex powrócił do stanu "unlocked"
- **int pthread_mutex_trylock(pthread_mutex_t *mutex)** – która zachowuje się podobnie jak powyższa, z tym że obecny wątek nie jest blokowany jeśli mutex jest już zablokowany przez inny wątek, a jedynie ustawia flagę EBUSY.
- **pthread_mutex_timedlock** – jest rozwinięciem funkcji **pthread_mutex_lock** – podawany jest maksymalny czas czekania wątku na odblokowanie (zablokowanego przez inny wątek) mutex-a.

Odblokowanie mutex-a wykonywane jest za pomocą funkcji [pthread_mutex_unlock](#).

Ostrzeżenie: Należy zwrócić uwagę, aby nie używać mutex-ów w funkcjach obsługujących sygnały (signal handlers), gdyż może to doprowadzić do zablokowania się programu.

Jednym z najprostszych zastosowań mutex-ów jest ochrona zmiennej przed równoczesnym dostępem z wielu wątków. Pokazuje to następujący przykład:

Chronienie dostępu do zmiennej

```
int x;
pthread_mutex_t x_mutex = PTHREAD_MUTEX_INITIALIZER;

void my_thread_safe_function(...) {
    /* Każdy dostęp do zmiennej x powinien się odbywać w następujący sposób: */
    pthread_mutex_lock(&x_mutex);
    /* operacje na x... */
    pthread_mutex_unlock(&x_mutex);
}

...
```

Zakleszczenia

Zakleszczenia są najczęściej spowodowane nieprawidłowym używaniem mechanizmu mutexów przez programistę. Może to wystąpić np. w sytuacji gdy próbujemy zamknąć mutex w wątku, w którym już wcześniej to zrobiliśmy. Zachowanie programu jest wtedy uzależnione od typu używanego mutexu.

Wyróżniamy trzy rodzaje mutexów:

- **Szybki mutex** (fast mutex) – domyślny typ, próba zamknięcia takiego mutexu z wątku, w którym wcześniej już to zrobiliśmy spowoduje zakleszczenie.
- **Rekursywny mutex** – nie powoduje zakleszczenia. Zapamiętuje ile razy wątek zablokował danego mutexa i oczekuje na taką samą ilość odblokowań.
- **Mutex sprawdzający błędy** (error-checking mutex) – taka próba spowoduje błąd **EDEADLK** podczas wywoływania `pthread_mutex_lock`

By stworzyć niestandardowe mutexy używamy następujących typów i funkcji: `int pthread_mutexattr_init(*attr)` – inicjalizuje zmienną z atrybutami

`int pthread_mutexattr_destroy(pthread_mutexattr_t *attr)` – zwalnianie

`int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type)` – ustawia typ mutexu

`int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr, int *type)` – pobiera typ mutexu

- `pthread_mutexattr_t attr` – zmienna przechowująca atrybuty mutexa
- `type` – typ mutexu – może przyjmować następujące wartości:

`PTHREAD_MUTEX_NORMAL`

`PTHREAD_MUTEX_ERRORCHECK`

`PTHREAD_MUTEX_RECURSIVE`

Współdzielenie mutexu z innymi procesami

Funkcje

- `int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared)`
- `int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr, int *pshared)`

Zmiana priorytetu wątku posiadającego blokadę

Dostępne, gdy istnieje rozszerzenie [TPP](#) (oraz [TPI](#)).

Wartość atrybutu decyduje o strategii wykonywania programu, gdy wiele wątków o różnych priorytetach stara się o uzyskanie blokady. Atrybut może mieć wartości:

1. `PTHREAD_PRIO_NONE`,
2. `PTHREAD_PRIO_PROTECT`,
3. `PTHREAD_PRIO_INHERIT` (opcja [TPI](#)).

W przypadku `PTHREAD_PRIO_NONE` priorytet wątku, który pozyskuje blokadę nie zmienia się.

W dwóch pozostałych przypadkach z mutexem powiązany zostaje pewien priorytet i gdy wątek uzyska blokadę, wówczas jego priorytet jest podbijany do wartości z mutexu (o ile oczywiście był wcześniej niższy). Innymi słowy w obrębie sekcji krytycznej wątek może działać z wyższym priorytetem.

Sposób ustalania priorytetu mutexu zależy od atrybutu:

- `PTHREAD_PRIO_INHERIT` – wybierany jest maksymalny priorytet spośród wątków oczekujących na uzyskanie danej blokady;
- `PTHREAD_PRIO_PROTECT` – priorytet jest ustalany przez programistę funkcją `pthread_mutexattr_setprioceiling` lub `pthread_mutex_setprioceiling` (opisane w następnej sekcji).

Dodatkowo jeśli wybrano wartość `PTHREAD_PRIO_PROTECT`, wówczas wszelkie próby założenia blokady funkcjami `pthread_mutex_XXXlock` z poziomu wątków o priorytecie niższym niż ustawiony dla mutexa nie powiodą się – zostanie zwrócona wartość błędu `EINVAL`.

Funkcje

- `int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol)` ([doc](#))
- `int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr, int *protocol)` ([doc](#))

Semafora nienazwane

- Nowy semafor jest tworzony przez funkcję [sem_init](#)
- Operacja Sygnalizuj jest wykonywana za pomocą [sem_post](#), która zwiększa licznik semafora. Funkcja ta nigdy nie powoduje blokady wywołującego ją wątku.
- Operacja Czekaj jest wykonywana za pomocą jednej z następujących funkcji:

- [sem_wait](#) – proces jest usypiany aż dany semafor ma nie-zerową wartość, po czym następuje atomowa operacja zmniejszenia licznika semafora
- [sem_trywait](#) – jest nie blokującym wariantem [sem_wait](#) – jeśli semafor ma zerowy licznik, funkcja nie usypia wątku lecz zwraca -1 ustawia numer błędu na EAGAIN
- Semafor jest usuwany za pomocą [sem_destroy](#).

Warunki Sprawdzające (Condition Variables)

Czasami konieczne jest monitorowanie przez wątek pewnych warunków. Implementacja ich sprawdzania z wykorzystaniem konwencjonalnych mechanizmów (ciągłego sprawdzania czy warunek jest spełniony) byłaby mocno nieefektywna, gdyż powodowałaby że wątek byłby ciągle zajęty.

Rozwiązaniem tego problemu jest mechanizm warunków sprawdzających ([Condition Variables](#)). Pozwala on na uśpienie wątku aż do momentu gdy pewne warunki na dzielonych danych zostaną spełnione.

Za pomocą zmiennych warunkowych możemy wstrzymywać wykonywanie wątku, aż do momentu gdy zajdzie określony przez nas warunek. Zmiennej warunkowej towarzyszy mutex, który zapewnia wyłączność w trakcie odczytu/zmiany wartości flagi.

Gdy wątek dochodzi do sekcji zależnej od pewnego warunku (np. flagi), wykonywana jest sekwencja:

- Wątek zajmuje mutexa następnie sprawdza warunek.
- Jeżeli warunek jest spełniony – wtedy wątek wykonuje kolejne instrukcje.
- Jeżeli warunek nie jest spełniony – wtedy wątek jednocześnie odblokowuje mutex i wstrzymuje działanie aż do spełnienia warunku (poinformowania o zmianie warunku przez inny wątek).

Przy zmianie warunku muszą być podjęte następujące kroki:

- Zajęcie mutexa towarzyszącego zmiennej warunkowej
- Podjęcie akcji, która może zmienić warunek
- Zasygnalizowanie oczekującym wątkom zmiany warunku.
- Odblokowanie mutexa.

Zmienne warunkowe przechowywane są w typie `pthread_cond_t`.

Zmienne warunkowe obsługują następujące funkcje:

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr) - inicjalizacja zmiennej
int pthread_cond_destroy(pthread_cond_t *cond) - usunięcie zmiennej
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex) - ustawia wątek w tryb oczekiwania w czasie, którego Mutex jest
odblokowany
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *timeout) - usypia wątek na określoną
ilość czasu
int pthread_cond_broadcast(pthread_cond_t *cond) - powiadamia wszystkie oczekujące wątki
int pthread_cond_signal(pthread_cond_t *cond) - powiadamia tylko jeden wątek
```

Zmienna warunkowa może być również inicjalizowana makrem **PTHREAD_COND_INITIALIZER**, które zainicjalizuje ją standardowymi atrybutami.

Do obsługi atrybutów służy funkcje: `int pthread_condattr_init(pthread_condattr_t *attr)` – inicjalizacja

`int pthread_condattr_destroy(pthread_condattr_t *attr)` – zwolnianie

`int pthread_condattr_getpshared(const pthread_condattr_t *attr, int *pshared)` – pobiera atrybut process-shared

`int pthread_condattr_setpshared(pthread_condattr_t *attr, int pshared)` – ustawia atrybut process-shared

Poniżej znajduje się "urywek" kodu wykorzystujący warunki sprawdzające. Dwie zmienne dzielone (x i y) są sprawdzana pod kątem większości x od y. Każda zmiana dowolnej ze zmiennych powoduje obudzenie "zainteresowanych" wątków.

Przykład użycia warunków sprawdzających

```

int x,y;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

...
// (Watek 1)
// Czekanie aż x jest większe od y jest
// przeprowadzane następująco:
pthread_mutex_lock(&mutex);
while (x <= y) {
    pthread_cond_wait(&cond, &mutex);
}
...
pthread_mutex_unlock(&mutex);

...
// (Watek 2)
// Kazda modyfikacja x lub y może
// powodować zmianę warunków. Należy
// obudzić pozostałe wątki, które korzystają
// z tego warunku sprawdzającego.

pthread_mutex_lock(&mutex);
/* zmiana x oraz y */
if (x > y)
    pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mutex);

```

Ostatnia modyfikacja: poniedziałek, 8 maja 2023, 10:02



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

Pobierz aplikację mobilną



Wybierz język



Oznacz jako wykonane

Gniazda

Uniwersalny mechanizm dwukierunkowej komunikacji procesów lokalnych i zdalnych

Wprowadzenie

Gniazda (ang. sockets) są kolejnym z dostępnych mechanizmów komunikacji międzyprocesowej. W odróżnieniu od poznanych do tej pory (potoki, sygnały, komunikaty) obok komunikacji lokalnej umożliwiają również komunikację między procesami działającymi na różnych maszynach. Dostarczają nam warstwy abstrakcji nad fizycznym (światłowód, przewód miedziany, sieć bezprzewodowa) i logicznym (działanie protokołów sieciowych) sposobem połączenia maszyn.

Podstawowe cechy

Dziedzina

Dziedzina określa jaka rodzina protokołów będzie wykorzystywana do komunikacji i w jakiej przestrzeni będą umieszczane nazwy identyfikujące gniazda.

- **AF_UNIX/AF_LOCAL** — komunikacja lokalna w obrębie jednej maszyny
- **AF_INET** — komunikacja internetowa w oparciu o protokół sieciowy IPv4
- **AF_INET6** — komunikacja internetowa w oparciu o protokół sieciowy IPv6

Inne dostępne to **AF_PACKET** (obsługa "surowych" ramek warstwy łącznika danych), **AF_NETLINK** (komunikacja z interfejsami jądra), a także **AF_IPX**, **AF_X25**, **AF_AX25**, **AF_ATMPVC**, **AF_APPLETALK**...

W starszych źródłach można natrafić na używane w tym kontekście stałe które rozpoczynają się od **PF_***. Ich znacznie jest w praktyce dokładnie takie samo:

The manifest constants used under 4.x BSD for protocol families are **PF_UNIX**, **PF_INET**, and so on, while **AF_UNIX**, **AF_INET**, and so on are used for address families. However, already the BSD man page promises: "The protocol family generally is the same as the address family", and subsequent standards use **AF_*** everywhere.

manual funkcji socket(2)

Tryb komunikacji

Komunikacja z wykorzystaniem gniazd może odbywać się w różnych trybach które różnią się między sobą swoimi cechami. Nie ma jednego uniwersalnego trybu — każda zaleta pociąga za sobą jakąś wadę.

- **SOCK_STREAM** — niezawodna, uporządkowana, dwukierunkowa komunikacja strumieniowa oparta o połączenia; podobna w swojej naturze do rozmowy telefonicznej
 - **niezawodna** — wszystkie wysłane dane zostaną dostarczone (chyba że będzie to niemożliwe, co zostanie wykryte); protokół komunikacyjny zawiera mechanizmy wykrywania i obsługi utraty pakietów danych
 - **uporządkowana** — dane są dostarczane do adresata w kolejności wysłania ich przez nadawcę; protokół komunikacyjny zawiera mechanizmy które porządkują pakiety danych których kolejność została zamieniona w transporcie
 - **strumieniowa** — dane przekazywane są do odbiorcy jako sekwencja kolejnych bajtów, niezależnie od tego jakimi „porcjami” wysyłał je klient; protokół komunikacyjny nie dostarcza „obiektu” wiadomości/komunikatu, konieczna jest własna ich implementacja (np. przez „znaczniki” końca kolejnych wiadomości umieszczone w strumieniu)
 - **oparta o połączenia ("połączeniowa")** — przekazywanie danych wymaga zestawienia połączenia, które zostanie wykorzystane do komunikacji; protokół komunikacyjny dostarcza metod nawiązania połączenia i jego utrzymania
- **SOCK_DGRAM** — zawodna, nieuporządkowana, bezpołączeniowa komunikacja datagramowa; podobna w swojej naturze do przesyłania listów
 - **zawodna** — dane mogą zaginąć w trakcie transportu a nadawca nie otrzyma o tym żadnej informacji; protokół komunikacyjny sam w sobie nie wykrywa i nie obsługuje utraty pakietów danych
 - **nieuporządkowana** — te same dane (w przypadku kiedy udało im się dotrzeć) mogą zostać dostarczone wielokrotnie lub kolejność kolejnych komunikatów może zostać wymieszana

- **bezołączeniowa** — przed wysłaniem danych nie jest zestawiane połączenie, są one po prostu wysyłane na adres odbiorcy oczekującego datagramów
- **datagramowa** — odbiorca otrzymuje dane w postaci komunikatów, zostaje zachowany podział na kolejne wiadomości — nie są łączone w jeden strumień

Inne dostępne to **SOCK_SEQPACKET** (łączy cechy **SOCK_STREAM** i **SOCK_DGRAM**), **SOCK_RAW** (pozwala na samodzielne konstruowanie struktur — ramek, pakietów — wysyłanych przez system operacyjny do sieci), **SOCK_RDM...**

Protokół

Protokół określa sposób transportowania danych przesyłanych przez gniazdo. W większości przypadków dla danego trybu komunikacji w danej dziedzinie istnieje jeden konkretny "słuszny protokół". Dla **AF_INET/AF_INET6...**

- ... w trybie **SOCK_STREAM** zostanie użyty protokół TCP
- ... w trybie **SOCK_DGRAM** zostanie użyty protokół UDP

Adresy gniazd

Z każdym gniazdem, niezależnie od tego jaką rodzinę protokołów wykorzystuje, powiązana jest struktura opisująca jego adres. Jest ona wykorzystywana zarówno do określenia gdzie gniazdo ma nasłuchiwać, jak również do przechowywania informacji o podłączonym do serwera kliencie. Standardowym typem opisującym taką strukturę jest **struct sockaddr** zdefiniowany następująco:

```
struct sockaddr {
    sa_family_t sa_family;
    char        sa_data[14];
};
```

W praktyce wykorzystywane są jednak dedykowane dla poszczególnych dziedzin struktury adresowe, które **na początku zawierają odpowiednik sa_family ustawiony na odpowiednią rodzinę adresów** i są rzutowane na **struct sockaddr** (lub dokonywane jest rzutowanie ze **struct sockaddr** do nich).

AF_INET

Wykorzystywana jest struktura **sockaddr_in** zdefiniowana w pliku nagłówkowym **netinet/in.h**:

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;  /* internet address */
};

/* Internet address. */
struct in_addr {
    uint32_t       s_addr;    /* address in network byte order */
};
```

Jeśli chcemy by struktura opisywała dowolny adres IP posiadany przez maszynę, jako **sin_addr.s_addr** w strukturze ustawiamy stały **INADDR_ANY**. Aby zezwalać wyłącznie na połączenia lokalne należy użyć **INADDR_LOOPBACK**.

Numer portu (**sin_port**) jest 16-bitową liczbą (od 0 do 65535). Porty o numerach mniejszych niż 1024 uznawane są za uprzywilejowane i nasłuchiwanie na nich wymaga posiadania odpowiednich uprawnień (najczęściej superużytkownika). Podanie jako numeru portu 0 (zero) oznacza, że system operacyjny ma wybrać dowolny dostępny port (z zakresu tzw. portów efemerycznych — na Linuksie zwykle są to numery od 32768 do 60999).

Aby przekształcić zwykły numer portu do "network byte order" (w którym pierwszy bajt jest najbardziej znaczący), należy skorzystać z funkcji [htobe16\(...\)](#) lub [hton\(...\)](#).

Z danym portem i adresem sieciowym można powiązać na maszynie tylko jedno gniazdo — ponowne powiązanie wymaga wcześniejszego zamknięcia poprzedniego gniazda.

AF_INET6

Wykorzystywana jest struktura **sockaddr_in6** zdefiniowana w pliku nagłówkowym **netinet/in.h**:

```

struct sockaddr_in6 {
    sa_family_t      sin6_family;    /* AF_INET6 */
    in_port_t        sin6_port;     /* port number */
    uint32_t         sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr;     /* IPv6 address */
    uint32_t         sin6_scope_id; /* Scope ID (new in 2.4) */
};

struct in6_addr {
    unsigned char    s6_addr[16];   /* IPv6 address */
};

```

Jeśli chcemy by struktura opisywała dowolny adres IPv6 posiadany przez maszynę, jako `sin6_addr` w strukturze ustawiamy stałą `in6addr_any`. Aby zezwalać wyłącznie na połączenia lokalne należy użyć `in6addr_loopback`. Należy mieć na uwadze, że w zależności od konfiguracji systemu gniazda w dziedzinie `AF_INET6` nasłuchujące na wszystkich adresach IPv6 mogą domyślnie przyjmować również połączenia na wszystkich adresach IPv4 — wówczas adres przechowywany w strukturze zostanie w specyficzny sposób przekształcony.

W odniesieniu do `sin6_port` obowiązują takie same zasady jak w przypadku `sin_port` w przypadku rodziny adresów `AF_INET`.

AF_UNIX

Wykorzystywana jest struktura `sockaddr_un` zdefiniowana w pliku nagłówkowym `sys/un.h`:

```

#define UNIX_PATH_MAX    108

struct sockaddr_un {
    sa_family_t sun_family;          /* AF_UNIX */
    char       sun_path[UNIX_PATH_MAX]; /* pathname */
}

```

`sun_path` jest zazwyczaj ścieżką do pliku który reprezentuje gniazdo (sama komunikacja nie odbywa się jednak za pomocą systemu plików i nie zadziała też przez dysk sieciowy – np. Network File System), a uprawnienia do niego kontrolują możliwość podłączenia się do socketu. W przypadku gdy chcemy powiązać gniazdo ze ścieżką i taki plik już istnieje, wystąpi błąd (nawet jeśli jest to plik gniazda pozostały po poprzednim wykonaniu naszego programu). Trzeba pamiętać o jego usunięciu np. funkcją `int unlink(const char *pathname)` z nagłówka `unistd.h`.

W systemach z rodziny Linux istnieje możliwość utworzenia tzw. gniazda abstrakcyjnego (np. kiedy nasz system plików nie pozwala na utworzenie i-węzła reprezentującego gniazdo). Realizuje się to przez ustawienie pierwszego bajtu ścieżki (zerowego indeksu tablicy) na bajt zerowy (`'\0'`) — pozostałe bajty tablicy tworzą identyfikator takiego gniazda.

Długość ścieżki nie może przekroczyć `UNIX_PATH_MAX` (wliczając w to bajt zerowy umieszczony na jej końcu, także w przypadku gniazd abstrakcyjnych). W przypadku Linuksa stała ta ma wartość 108, w przypadku macOS jest to 104. Istnieją systemy gdzie ścieżka do gniazda UNIX ma maksymalnie 92 bajty.

Funkcje związane z adresami

Korzystanie z czterech poniższych funkcji wymaga dołączenia nagłówków `sys/socket.h`, `netinet/in.h` oraz `arpa/inet.h`, a ponadto wcześniejszego zdefiniowania `_BSD_SOURCE` lub `_SVID_SOURCE`:

int inet_aton(const char *cp, struct in_addr *inp)

W przypadku sukcesu (podany string zawierał poprawny adres) `inet_aton(...)` zwraca **w odróżnieniu od wielu funkcji niezerową wartość, a w przypadku błędu 0**.

- `cp` — wskaźnik na bufor znakowy zawierający adres IP zapisany jako tekst
- `inp` — wskaźnik na zaallokowany obszar pamięci przeznaczonej na przechowywanie struktury z adresem

char *inet_ntoa(struct in_addr in)

Na podstawie podanej struktury tworzy tekstową reprezentację adresu IPv4 i zwraca wskaźnik do bufora zawierającego ten adres. **Bufor ten jest ponownie wykorzystywany przez kolejne wywołania tej funkcji, więc powstający napis należy we własnym zakresie skopiować w inne miejsce pamięci!**

- `in` — struktura opisująca adres IP

int inet_pton(int af, const char *src, void *dst)

Funkcja ta przekształca podany jako string adres z danej rodziny adresów w odpowiedniego rodzaju strukturę opisującą adres.

Dla poprawnego adresu zwraca `1`, dla błędного zwraca `0`, a dla nieobsługiwanej rodziny adresów zwraca `-1` i ustawia `errno`.

- `af` — rodzina adresów; `AF_INET` dla IPv4, `AF_INET6` dla IPv6 — **inne wartości nie są obsługiwane**
- `src` — wskaźnik na bufor znakowy zawierający odpowiedni dla rodziny adresów zapisany jako tekst

- **dst** — wskaźnik na zaalokowany obszar pamięci przeznaczonej na przechowywanie właściwej dla rodziny adresów struktury (**struct in_addr** dla **AF_INET** bądź **struct in6_addr** dla **AF_INET6**)

const char *inet_ntop(int af, const void *src, char *dst, socklen_t size)

Funkcja zamienia adres zawarty w strukturze właściwej dla danej rodziny adresów na jego tekstowy zapis.

Zwraca wskaźnik na napis jeśli wywołanie zakończyło się sukcesem lub **NULL** i ustawia **errno** w przypadku błędu.

- **af** — rodzina adresów; **AF_INET** dla IPv4, **AF_INET6** dla IPv6 — **inne wartości nie są obsługiwane**
- **src** — wskaźnik na właściwą dla rodzinny adresów strukturę przechowującą adres
- **dst** — wskaźnik na zaalokowany dla bufora znakowego obszar pamięci w którym powinna zostać umieszczona tekstowa reprezentacja adresu
- **size** — ilość bajtów dostępnych w buforze docelowym (minimum **INET_ADDRSTRLEN** dla **AF_INET** i **INET6_ADDRSTRLEN** dla **AF_INET6**)

Poniższa funkcja została zdefiniowana w nagłówku **netdb.h**:

struct hostent *gethostbyname(const char *name)

Funkcja zwraca adres/adresy powiązane z podaną nazwą domenową. W przypadku podania jako argument adresu IP w postaci tekowej, zwraca ten adres jako odpowiednią strukturę.

gethostbyname(...) zostało uznane za przestarzałe i zaleca się korzystanie z **getaddrinfo(...)**; funkcja nie radzi sobie z adresami IPv6.

Funkcja zwraca wskaźnik na **struct hostent** jeśli wywołanie zakończyło się sukcesem lub **NULL** i ustawia **h_errno** (odpowiednik **errno** ze specjalizowanymi kodami błędów) w przypadku błędu. Kolejne wywołania mogą nadpisać struktury otrzymane w wyniku poprzednich wywołań — aby tego uniknąć konieczne jest wykonania kopi struktury wartość po wartości.

```
struct hostent {
    char *h_name;           /* official name of host */
    char **h_aliases;       /* alias list */
    int h_addrtype;         /* host address type */
    int h_length;           /* length of address */
    char **h_addr_list;     /* list of addresses */
}
#define h_addr h_addr_list[0] /* for backward compatibility */
```

Chociaż **h_addr_list** jest zadeklarowana jako tablica stringów, to w rzeczywistości przechowuje struktury o rozmiarze **h_length**, odpowiadające rodzinie adresów znajdującej się w **h_addrtype** (**AF_INET**); ostatni element tablicy jest **NUL**em. Należy wykonywać odpowiednie rzutowania (na **struct in_addr** bądź **struct in6_addr**). Jeśli interesuje nas dowolny z adresów powiązanych z nazwą (chociaż nierzadko otrzymamy tylko jeden), to możemy odwołać się do **h_addr**.

Poniższa funkcja dostępna jest w pliku nagłówkowym **unistd.h** i wymaga wcześniejszego zdefiniowania **_BSD_SOURCE**:

int gethostname(char *name, size_t len)

Funkcja służy do pobierania nazwy lokalnego komputera. Gwarantowane jest że jej długość nie przekroczy stałej **HOST_NAME_MAX**, którą można znaleźć w pliku nagłówkowym **limits.h** (dla Linuksa ta stała wynosi 64).

W przypadku sukcesu **gethostname(...)** zwraca **0**, a w przypadku błędu **-1** i ustawia **errno**.

- **name** — wskaźnik na obszar pamięci w którym ma zostać umieszczona nazwa hosta
- **len** — wielkość zaalokowanego obszaru przeznaczona na nazwę (nie wliczamy końcowego **'\0'**!)

Funkcje do obsługi zmiany kolejności bajtów

Różne architektury procesorów mogą różnić się między sobą sposobem przechowywania liczb w pamięci. Wyróżnia się dwie metody:

- **little-endian** — jako pierwszy przechowywany/wysyłany jest najmniej znaczący (najmłodszy) bajt liczby (tego sposobu używają procesory x86 i x86-64)
- **big-endian** — jako pierwszy przechowywany/wysyłany jest najbardziej znaczący (najstarszy) bajt liczby (ten sposób powszechnie obowiązuje w protokołach sieciowych)

Dla przykładu, liczba **0xAABBCCDD** będzie przechowywana w obu systemach następująco:

mem[0] mem[1] mem[2] mem[3]

little-endian	0xDD	0xCC	0xBB	0xAA
big-endian	0xAA	0xBB	0xCC	0xDD

W przypadku wielu architektur kolejność bajtów może być przełączana — są to np. nowsze wersje ARM, SPARC i PowerPC, a także MIPS.

W związku z tym, aby uchronić się przed niewłaściwym zinterpretowaniem liczb w sytuacji kiedy klient i serwer stosują różną kolejność bajtów, przyjmuje się że przed wysyłką dane powinny być zawsze skonwertowane do obowiązującego w sieciach systemu big-endian.

Poniższa rodzina funkcji dostępna jest na Linuksie w pliku nagłówkowym `endian.h` i wymaga wcześniejszego zdefiniowania `_DEFAULT_SOURCE` (w miejscu **NN** należy podstawić ilość bitów, jaką zajmuje liczba danego typu — 16 dla liczby 2-bajtowej, 32 dla liczby 4-bajtowej i 64 dla liczby 8-bajtowej):

uintNN_t htobeNN(uintNN_t host_NNbits)

Przekształca podaną liczbę **NN**-bitową z kodowania hosta na kodowanie big-endian (jeśli host używa big-endian, funkcja zwraca liczbę w niezmienionej postaci).

uintNN_t htoleNN(uintNN_t host_NNbits)

Przekształca podaną liczbę **NN**-bitową z kodowania hosta na kodowanie little-endian (jeśli host używa little-endian, funkcja zwraca liczbę w niezmienionej postaci).

uintNN_t beNNtoh(uintNN_t big_endian_NNbits)

Przekształca podaną liczbę **NN**-bitową z kodowania big-endian na kodowanie hosta (jeśli host używa big-endian, funkcja zwraca liczbę w niezmienionej postaci).

uintNN_t leNNtoh(uintNN_t little_endian_NNbits)

Przekształca podaną liczbę **NN**-bitową z kodowania little-endian na kodowanie hosta (jeśli host używa little-endian, funkcja zwraca liczbę w niezmienionej postaci).

Poniższe funkcje zdefiniowane są w pliku nagłówkowym `arpa/inet.h` (zauważ, że brak tutaj funkcji pozwalającej na konwersję liczb 64-bitowych):

uint32_t htonl(uint32_t hostlong)

Przekształca podaną liczbę 32-bitową (l w nazwie jak long) z kodowania hosta na kodowanie big-endian (jeśli host używa big-endian, funkcja zwraca liczbę w niezmienionej postaci).

uint16_t htons(uint16_t hostshort)

Przekształca podaną liczbę 16-bitową (s w nazwie jak short) z kodowania hosta na kodowanie big-endian (jeśli host używa big-endian, funkcja zwraca liczbę w niezmienionej postaci).

uint32_t ntohl(uint32_t netlong)

Przekształca podaną liczbę 32-bitową (l w nazwie jak long) z kodowania big-endian na kodowanie hosta (jeśli host używa big-endian, funkcja zwraca liczbę w niezmienionej postaci).

uint16_t ntohs(uint16_t netshort)

Przekształca podaną liczbę 16-bitową (s w nazwie jak short) z kodowania big-endian na kodowanie hosta (jeśli host używa big-endian, funkcja zwraca liczbę w niezmienionej postaci).

Funkcje do obsługi gniazd

Wszystkie poniższe funkcje zostały zdefiniowane w pliku nagłówkowym `sys/socket.h`. Dla zapewnienia przenośności kodu rozsądnie jest również dołączyć plik nagłówkowy `sys/types.h`.

int socket(int domain, int type, int protocol)

Funkcja ta tworzy gniazdo i zwraca numer powiązanego z nim deskryptora albo `-1` w przypadku błędu i ustawia `errno`.

- **domain** — jedna z [opisanych wcześniej stałych](#) określających rodzinę protokołów wykorzystywanych do komunikacji
- **type** — jeden z [opisanych wcześniej trybów komunikacji](#), ewentualnie zORowany z następującymi flagami:
 - `SOCK_NONBLOCK` — gniazdo w trybie nieblokującym (ustawienie `O_NONBLOCK` na deskryptorze gniazda)
 - `SOCK_CLOEXEC` — zamknij deskryptor gniazda w chwili wywołania `execve` (ustawienie `FD_CLOEXEC` na deskryptorze gniazda)
- **protocol** — protokół wykorzystywany do komunikacji; podanie wartości innej niż `0` (automatyczny wybór na podstawie pozostałych ustawień) jest konieczne tylko jeśli istnieje wiele protokołów związanych z daną kombinacją dziedziny i trybu; w przypadku gniazd `AF_PACKET` określa rodzaj ramek warstwy łącza które chcemy otrzymywać

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)

Po utworzeniu gniazda przy użyciu `socket(...)` nie ma ono przypisanej swojej nazwy, w związku z czym nie ma możliwości wysyłania do niego danych (wyjątkiem są gniazda "połączeniowe" po stronie klienta — nie muszą być zaadresowane). `bind(...)` służy właśnie do związania gniazda z jego nazwą (adresem).

W przypadku sukcesu `bind(...)` zwraca `0`, a w przypadku błędu `-1` i ustawia `errno`.

- **sockfd** — numer deskryptora gniazda

- **addr** — wskaźnik do [opisanej wcześniej struktury](#) definiującej adres z którym ma zostać związane gniazdo
- **addrlen** — wielkość struktury przekazanej w drugim argumencie (np. wynik `sizeof(...)`)

W domenie internetu (`AF_INET/AF_INET6`) skorzystanie z funkcji `sendto(...)` (w przypadku komunikacji datagramowej) lub dowolnej funkcji do wysyłki danych (w przypadku komunikacji strumieniowej) powoduje automatyczne związywanie gniazda z portem efemerycznym — tak, jak gdyby wywołano `bind(...)` przekazując strukturę z numerem portu ustawionym na 0 (zero).

W domenie komunikacji lokalnej (`AF_UNIX`) nie ma potrzeby korzystania po stronie klienta z funkcji `bind(...)` jeśli korzystamy z komunikacji strumieniowej lub nie potrzebujemy odbierać odpowiedzi na datagramy. W przeciwnym razie należy wywołać funkcję `bind(...)` z argumentem `addrlen` ustawionym na `sizeof(sa_family_t)` lub ustawić na gnieździe flagę `SO_PASSCRED`; są to dwie metody, które powodują związywanie gniazda z wygenerowaną losowo abstrakcyjną nazwą (autobind).

int listen(int sockfd, int backlog)

Funkcja ta dotyczy gniazd connection-oriented (`SOCK_STREAM`) — odpowiada za rozpoczęcie akceptowania połączeń od klientów.

W przypadku sukcesu `listen(...)` zwraca `0`, a w przypadku błędu `-1` i ustawia `errno`.

- **sockfd** — numer deskryptora gniazda
- **backlog** — maksymalna ilość połączeń które mogą oczekiwać na zaakceptowanie (kolejne połączenia będą od razu odrzucane lub będą całkowicie ignorowane przez system)

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen)

Działanie tej funkcji zależy od tego na jakiego rodzaju gnieździe zostanie wywołania.

Dla gniazd strumieniowych: służy do połączenia się klienta z serwerem, z powodzeniem może zostać wywołana tylko raz.

Dla gniazd datagramowych: ustawia domyślny adres na który adres będą wysyłane datagramy oraz jedyny adres z którego będą one odbierane (na danym gnieździe) — w odróżnieniu od `connect(...)` na gniazdach strumieniowych, można ją wywołać wielokrotnie.

W przypadku sukcesu `connect(...)` zwraca `0`, a w przypadku błędu `-1` i ustawia `errno`.

- **sockfd** — numer deskryptora gniazda zwrócony przez `socket(...)`
- **addr** — wskaźnik do [opisanej wcześniej struktury](#) definiującej adres z którym ma zostać związane gniazdo
- **addrlen** — wielkość struktury przekazanej w drugim argumencie (np. wynik `sizeof(...)`)

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)

int accept4(int sockfd, struct sockaddr *addr, socklen_t *addrlen, int flags)

Funkcje te służą do akceptowania oczekujących połączeń na gniazdach połączeniowych. Mogą blokować się do czasu pojawienia się połączenia lub zawsze natychmiast powracać, w zależności od tego czy na deskryptorze gniazda ustawione jest `O_NONBLOCK`.

Po wywołaniu zostaje zaakceptowane pierwsze z oczekujących na gnieździe połączeń i zostaje zwrócony deskryptor służący do komunikacji z klientem który się połączył lub `-1` i ustawia `errno` gdy wystąpił błąd.

- **sockfd** — numer deskryptora gniazda
- **addr** — wskaźnik do miejsca w pamięci przygotowanego do przyjęcia struktury z adresem klienta który się połączył lub `NULL` jeśli adres nie ma być zapisywany
- **addrlen** — wskaźnik do zmiennej określającej zaalokowaną wielkość struktury `addr` (jeśli struktura opisująca adres połączonego klienta jest większa niż `addrlen`, zostanie przyjęta do tego rozmiaru); po wywołaniu `accept(...)` w tej zmiennej znajdzie się faktyczna wielkość zapisanej struktury, jeśli była ona mniejsza niż `addrlen`; jeśli `addr` było równe `NULL`, `addrlen` również musi być `NULL`em
- **flags** — flagi `SOCK_NONBLOCK` i `SOCK_CLOEXEC` o znaczeniu identycznym jak w przypadku `socket(...)`, ale w odniesieniu do deskryptora gniazda służącego do komunikacji z klientem; gdy ustawione na `0`, to `accept4(...)` działa dokładnie tak samo jak `accept(...)`

ssize_t write(int sockfd, const void *buf, size_t len)

ssize_t send(int sockfd, const void *buf, size_t len, int flags)

ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addrlen)

ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags)

Służą do wysyłania danych z użyciem gniazda. W przypadku gdy dla gniazda datagramowego nie ustawiono domyślnego odbiorcy, nie jest możliwe korzystanie z pierwszych dwóch funkcji i konieczne jest wykorzystanie `sendto(...)`.

Funkcja `sendmsg(...)` jest zaawansowana i nie będzie opisywana. Może zostać użyta m. in. do przesyłania między dwoma procesami na tej samej maszynie deskryptora pliku — wymaga to jednak ręcznego utworzenia wiadomości o odpowiedniej strukturze.

Zwracają ilość wysłanych bajtów lub `-1` i ustawiają `errno` w razie niepowodzenia.

- **sockfd** — numer deskryptora gniazda

- **buf** — wskaźnik do danych które chcemy wysłać
- **len** — długość danych w **buf** które chcemy wysłać
- **flags** — flagi określające sposób wysyłki danych lub ich rodzaj:
 - **MSG_DONTWAIT** — wysyła dane w sposób nieblokujący (tak jak po ustawieniu **SOCK_NONBLOCK** dla gniazda)
 - **MSG_MORE** — dla socketów strumieniowych po TCP wpływa na sposób pakietyzacji danych, dla socketów datagramowych po UDP opóźnia wysyłkę komunikatu do momentu aż nastąpi wywołanie **send(...)** bez tej flagi i łączy dane ze wszystkich wywołań w jeden komunikat
 - **MSG_NOSIGNAL** — nie wysyłaj do procesu **SIGPIPE** jeśli druga strona zerwała połączenie
 - ...
- Gdy **flags** jest ustawione na **0**, to **send(...)** działa dokładnie tak samo jak **write(...)**.
- **dest_addr** — wskaźnik do struktury opisującej adres odbiorcy danych; dla gniazd połączeniowych powinien być **NULL**em
- **addrlen** — długość struktury opisywanej przez **dest_addr**; dla gniazd połączeniowych powinna być ustawiona na **0**

ssize_t read(int sockfd, const void *buf, size_t len)

ssize_t recv(int sockfd, const void *buf, size_t len, int flags)

ssize_t recvfrom(int sockfd, const void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen)

ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags)

Służą do odbierania danych z użyciem gniazda.

Funkcja **recvmsg(...)** jest zaawansowana i podobnie jak **sendmsg(...)** nie będzie opisywana.

Zwracają ilość odebranych bajtów lub **-1** i ustawiają **errno** w razie niepowodzenia. Zwrócenie **0** oznacza że druga strona połączenia zamknęła kanał komunikacji.

- **sockfd** — numer deskryptora gniazda
- **buf** — wskaźnik do miejsca w pamięci w którym chcemy zapisać dane
- **len** — maksymalna ilość bajtów które zostaną odczytane z gniazda i zapisane w **buf** (w przypadku komunikacji datagramowej: jeśli **len** jest mniejsze niż długość odebranego datagramu, to nadmiarowe bajty zostaną utracone!)
- **flags** — flagi określające sposób odbioru danych:
 - **MSG_DONTWAIT** — odbiera dane w sposób nieblokujący (tak jak po ustawieniu **SOCK_NONBLOCK** dla gniazda)
 - **MSG_WAITALL** — blokuje wywołanie do momentu aż zostanie odebranych dokładnie **len** bajtów (chyba że zostanie odebrany sygnał lub połączenie zostanie przerwane)
 - **MSG_PEEK** — odczytuje oczekujące na gnieździe dane i nie usuwa ich z bufora gniazda (kolejne wywołanie ponownie je zwróci)
 - **MSG_NOSIGNAL** — nie wysyłaj do procesu **SIGPIPE** jeśli druga strona zerwała połączenie
 - ...
- Gdy **flags** jest ustawione na **0**, to **recv(...)** działa dokładnie tak samo jak **read(...)**.
- **src_addr** — wskaźnik do miejsca w pamięci przygotowanego do przyjęcia struktury z adresem klienta który wysłał dane lub **NULL** jeśli adres nie ma być zapisywany
- **addrlen** — wskaźnik do zmiennej określającej zaalokowaną wielkość struktury **src_addr** (jeśli struktura opisująca adres połączonego klienta jest większa niż **addrlen**, zostanie przycięta do tego rozmiaru); po wywołaniu **recvfrom(...)** w tej zmiennej znajdzie się faktyczna wielkość zapisanej struktury, jeśli była ona mniejsza niż **addrlen**; jeśli **src_addr** było równe **NULL**, **addrlen** również musi być **NULL**em

int shutdown(int sockfd, int how)

Funkcja ta służy do kończenia komunikacji z użyciem gniazda, wykonując przy okazji czynności przewidziane protokołem dla poprawnego jej zakończenia (np. w przypadku protokołu TCP powoduje wysłanie pakietów z flagą **FIN**). Pozwala określić którą "stronę" gniazda zamykamy.

W przypadku sukcesu **shutdown(...)** zwraca **0**, a w przypadku błędu **-1** i ustawia **errno**.

- **sockfd** — numer deskryptora gniazda
- **how** — określa sposób zamknięcia gniazda:
 - **SHUT_RD** — zamyka kanał odczytu z gniazda
 - **SHUT_WR** — zamyka kanał zapisu do gniazda
 - **SHUT_RDWR** — zamyka oba kanały komunikacji

int close(int sockfd)

Funkcja ta zamyka deskryptor gniazda. Od tego momentu wszelkie operacje na tym deskryptorze są niedozwolone.

W przypadku sukcesu **close(...)** zwraca **0**, a w przypadku błędu **-1** i ustawia **errno**.

- **sockfd** — numer deskryptora gniazda

int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen)

Funkcja ta pozwala uzyskać aktualny adres gniazda. Może być użyteczna, jeśli zażądaliśmy związania gniazda z portem efemerycznym i chcemy się dowiedzieć jaki numer portu został wybrany przez system operacyjny.

W przypadku sukcesu `getsockname(...)` zwraca `0`, a w przypadku błędu `-1` i ustawia `errno`.

- `sockfd` — numer deskryptora gniazda
- `addr` — wskaźnik do miejsca w pamięci przygotowanego do przyjęcia struktury z adresem gniazda
- `addrlen` — wskaźnik do zmiennej określającej zaalokowaną wielkość struktury `addr` (jeśli struktura opisująca adres gniazda jest większa niż `addrlen`, zostanie przycięta do tego rozmiaru); po wywołaniu `getsockname(...)` w tej zmiennej znajdzie się faktyczna wielkość zapisanej struktury, jeśli była ona mniejsza niż `addrlen`

`int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen)`

Funkcja ta pozwala na ustawianie opcji związanych z gniazdem — charakterystycznych dla samego gniazda lub dla protokołu który gniazdo wykorzystuje.

W przypadku sukcesu zwraca `0`, a w przypadku błędu `-1` i ustawia `errno`.

- `sockfd` — numer deskryptora gniazda
- `level` — poziom ustawień (rodzaj ustawianych opcji); typowo `SOL_SOCKET` dla ustawień tyczących się samego gniazda, może pojawić się tu numer protokołu sieciowego
- `optname` — "nazwa" opcji, czyli stała powiązana z konkretnym parametrem
- `optval` — wskaźnik do wartości na którą chcemy ustawić daną opcję
- `optlen` — wielkość wartości z poprzedniego argumentu

Spośród licznych dostępnych opcji najbardziej warte uwagi są:

- `SO_REUSEADDR` — decyduje o możliwości związania gniazda "bardziej szczegółowego" (związanego z konkretnym adresem) na adres i port który jest już zajęty w wyniku zbindowania innego gniazda do `INADDR_ANY` (wszystkie interfejsy hosta); pozwala też na ponowne zbindowanie procesu-serwera TCP na ten sam adres i port natychmiast po jego zrestartowaniu (jeśli to serwer zamyka połączenie jako pierwszy, to standardowo konieczne jest oczekiwanie przez kilkanaście sekund aż minie timeout dla "zabłąkanych pakietów" — w tym czasie próby bindowania bez `SO_REUSEADDR` zwracają błąd `EADDRINUSE` — "Address already in use"); przyjmuje zmienną typu `int` o wartości `1` (włącz) lub `0` (wyłącz)
- `SO_KEEPALIVE` — decyduje o wysyłaniu specjalnych pustych pakietów "keepalive" które zapobiegają zerwaniu połączenia przez urządzenia sieciowe "po drodze" w przypadku gdy przez dłuższy czas nie pojawiają się dane; przyjmuje zmienną typu `int` o wartości `1` (włącz) lub `0` (wyłącz)
- `SO_PASSCRED` — decyduje o odbieraniu komunikatu sterującego `SCM_CREDENTIALS`, zawierającego identyfikatory użytkownika i procesu który podłączył się do gniazda lokalnego; przyjmuje zmienną typu `int` o wartości `1` (włącz) lub `0` (wyłącz)

`int getsockopt(int sockfd, int level, int optname, void *optval, socklen_t *optlen)`

Funkcja ta pozwala na odczytanie opcji związanych z gniazdem — charakterystycznych dla samego gniazda lub dla protokołu który gniazdo wykorzystuje.

W przypadku sukcesu zwraca `0`, a w przypadku błędu `-1` i ustawia `errno`.

- `sockfd` — numer deskryptora gniazda
- `level` — poziom ustawień (rodzaj odczytywanych opcji); typowo `SOL_SOCKET` dla ustawień tyczących się samego gniazda, może pojawić się tu numer protokołu sieciowego
- `optname` — "nazwa" opcji, czyli stała powiązana z konkretnym parametrem
- `optval` — wskaźnik do zaalokowanego obszaru pamięci do którego ma trafić odczytana wartość opcji
- `optlen` — wskaźnik do zmiennej określającej zaalokowaną wielkość `optval`; po wywołaniu `getsockopt(...)` w tej zmiennej znajdzie się faktyczna wielkość odczytanej wartości opcji

Możemy odczytywać opcje ustawiane przez `setsockopt(...)`, a oprócz tego także między innymi:

- `SO_DOMAIN` — dziedzinę gniazda; `optval` typu `int`
- `SO_TYPE` — tryb komunikacji gniazda; `optval` typu `int`
- `SO_PROTOCOL` — protokół wykorzystywany przez gniazdo; `optval` typu `int`
- `SO_ACCEPTCONN` — czy gniazdo znajduje się w stanie nasłuchiwanego (`listen(...)`); `optval` typu `int` reprezentujące wartość logiczną
- `SO_PEERCRED` — w przypadku wcześniejszego ustawienia `SO_PASSCRED` na gnieździe lokalnym opcja zawiera strukturę z identyfikatorami użytkownika i procesu który podłączył się do gniazda (UID, GID oraz PID); typu `struct ucred` dostępnego w nagłówku `sys/socket.h` po zdefiniowaniu `_GNU_SOURCE`:

```
struct ucred {  
    pid_t pid; /* process ID of the sending process */  
    uid_t uid; /* user ID of the sending process */  
    gid_t gid; /* group ID of the sending process */  
};
```

`int socketpair(int domain, int type, int protocol, int sv[2])`

Funkcja służy do utworzenia pary połączonych ze sobą nienazwanych gniazd. Można je wykorzystać na przykład do komunikacji między procesem macierzystym a potomnym.

W przypadku sukcesu `socketpair(...)` zwraca `0`, a w przypadku błędu `-1` i ustawia `errno`.

- `domain` — jedną dopuszczalną na Linuksie opcją jest podanie `AF_UNIX`
- `type` — jeden z [opisanych wcześniej trybów komunikacji](#), ewentualnie zORowany z następującymi flagami:
 - `SOCK_NONBLOCK` — gniazdo w trybie nieblokującym (ustawienie `O_NONBLOCK` na deskryptorze gniazda)
 - `SOCK_CLOEXEC` — zamknij deskryptor gniazda w chwili wywołania `execve` (ustawienie `FD_CLOEXEC` na deskryptorze gniazda)
- `protocol` — podajemy `0`, czyli automatyczny wybór
- `sv` — tablica w której zostaną umieszczone deskryptory obu gniazd; są nieroróżnialne

Monitorowanie wielu deskryptorów

Czasami zachodzi potrzeba jednoczesnego oczekiwania na zdarzenia (możliwość zapisu, możliwość odczytu) na wielu deskryptorach. Istnieją dwa podstawowe mechanizmy pozwalające na wydajne (czytaj: nie ma potrzeby iterowania się cały czas po nieblokujących deskryptorach) oczekiwanie na zdarzenia na zbiorze deskryptorów.

Z wykorzystaniem mechanizmu epoll

epoll jest najmłodszym z mechanizmów pozwalających na monitorowanie wielu deskryptorów plików. Nie posiada ograniczeń jak chodzi o ilość monitorowanych deskryptorów (w odróżnieniu od funkcji `select(...)`) i wszystkie monitorowane deskryptory obsługuje w czasie stałym (`poll(...)` stosuje liniowe przeszukiwanie). Jest wzorowany na mechanizmie kqueue obecnym we FreeBSD.

Wszystkie poniższe funkcje i typy zdefiniowane są w pliku `sys/epoll.h`.

Aby skorzystać z mechanizmu epoll, należy najpierw utworzyć instancję mechanizmu monitorowania.

`int epoll_create1(int flags)`

Funkcja zwraca numer deskryptora pliku, który umożliwia skonfigurowanie monitorowania innych deskryptorów i uruchomienie właściwego oczekiwania na zdarzenia. Po skończonej pracy instancję mechanizmu epoll można usunąć wywołując na tym deskryptorze operację `close(...)`.

- `flags` — opcjonalne flagi:
 - `EPOLL_CLOEXEC` — zamknij deskryptor w chwili wywołania `execve` (ustawienie `FD_CLOEXEC` na deskryptorze gniazda)
- Gdy `flags` jest ustawione na `0`, to `epoll_create1(...)` działa dokładnie tak samo jak `epoll_create(...)` (z dokładnością do pominięcia nieistotnego argumentu `size`).

`int epoll_create(int size)`

Wariant przestarzały. Argument `size` służył do podpowiadania jądro systemu na ile monitorowanych deskryptorów powinno być przygotowane; obecnie jest ignorowany (jądro powiększa odpowiednio struktury dynamicznie), ale ze względów kompatybilności wstępnej należy podawać jako `size` wartości większe od zera.

Zwracana wartość ma identyczne znaczenie jak w przypadku `epoll_create1(...)`.

Po utworzeniu instancji mechanizmu należy zarejestrować w nim deskryptory które chcemy obserwować.

`int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)`

Funkcja służy do zarejestrowania/wyrejestrowania/zmiany sposobu obsługi danego deskryptora pliku przez daną instancję epoll.

- `epfd` — numer deskryptora instancji mechanizmu epoll (zwrócony z `epoll_create(...)`)
- `op` — rodzaj operacji do wykonania:
 - `EPOLL_CTL_ADD` — zarejestrowanie deskryptora pliku do monitorowania
 - `EPOLL_CTL_MOD` — zmiana sposobu monitorowania deskryptora pliku
 - `EPOLL_CTL_DEL` — wyrejestrowanie deskryptora pliku z monitorowania
- `fd` — numer deskryptora pliku w odniesieniu do którego chcemy wykonać operację
- `event` — wskaźnik na strukturę `struct epoll_event` opisującą sposób monitorowania deskryptora i dane dodatkowe które mają być zwracane w przypadku wystąpienia zdarzeń na deskryptorze

W przypadku sukcesu `epoll_ctl(...)` zwraca `0`, a w przypadku błędu `-1` i ustawia `errno`.

`struct epoll_event`

```
struct epoll_event {
    uint32_t events;      /* Epoll events */
    epoll_data_t data;    /* User data variable */
};

typedef union epoll_data {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;
```

Pole **events** struktury to maska bitowa, w której ustawiane są szczegóły związane ze sposobem monitorowania deskryptora:

- **EPOLLIN** — możliwy odczyt
- **EPOLLOUT** — możliwy zapis
- **EPOLLPRI** — pojawiły się dane priorytetowe
- **EPOLLRDHUP** — drugi koniec gniazda został odłączony (klient rozłączył się lub zamknął swój kanał przeznaczony do zapisu)
- **EPOLLET** (Edge Triggered) — domyślnie epoll działa w trybie level triggered, czyli jeśli np. zarejestrowaliśmy się na zdarzenie "możliwy odczyt" i po otrzymaniu powiadomienia nie odczytaliśmy wszystkich oczekujących danych, to powiadomienie zostanie powtózone; w wariancie edge triggered powiadomienie emitowane jest jednokrotnie, w momencie wystąpienia zdarzenia
- **EPOLLONESHOT** — powiadamia tylko o najbliższym zdarzeniu na danym deskryptorze (aby otrzymać kolejne, należy ponownie go zarejestrować)
- **EPOLLEXCLUSIVE** — w sytuacji kiedy kilka instancji epoll monitoruje ten sam deskryptor, domyślnie wszystkie otrzymują powiadomienia o zdarzeniach na nim; ta flaga pozwala na to by zdarzenia trafiły tylko do tych instancji, które użyły flagi **EPOLLEXCLUSIVE**

Pole **data** zawiera unię, w której możemy przekazać informację jaką ma nam zwrócić epoll w przypadku wystąpienia zdarzenia na deskryptorze. Ta informacja powinna umożliwiać nam ustalenie na którym deskryptorze miało miejsce zdarzenie — epoll nie przekazuje nam tej informacji "sam z siebie"! (Typowo ustawiamy w tej unii pole **fd** na identyczną wartość, jak argument **fd** w wywołaniu **epoll_ctl(...)** w którym odwołujemy się do struktury z tą unią.)

int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout)

Funkcja oczekuje na zdarzenia na które zarejestrowano się w danej instancji epoll i zapisuje we wskazanym obszarze powiadomienia o nich. Oczekiwanie może zostać przerwane przez sygnał.

Funkcja zwraca ilość zdarzeń które faktycznie zostały zapisane w podanym obszarze pamięci lub zwraca **-1** i ustawia **errno** w przypadku wystąpienia błędu.

- **epfd** — numer deskryptora instancji mechanizmu epoll (zwrócony z **epoll_create(...)**)
- **events** — wskaźnik na zaalokowaną tablicę elementów **struct epoll_event**, w której funkcja ma umieścić powiadomienia o zdarzeniach
- **maxevents** — ilość elementów zaalokowanej przez nas tablicy
- **timeout** — maksymalny czas oczekiwania na zdarzenia (wyrażony w milisekundach); wartość **0** oznacza że funkcja ma natychmiast powrócić, jeśli nie ma oczekujących zdarzeń, zaś wartość **-1** oznacza oczekивание w nieskończoność

int epoll_pwait(int epfd, struct epoll_event *events, int maxevents, int timeout, const sigset_t *sigmask)

Działa jak **epoll_pwait(...)**, ale dodatkowo przyjmuje maskę sygnałów do ustawienia w wątku na czas oczekiwania na zdarzenia (można to porównać do atomowego wykonania ustawienia podanej maski, wykonania **epoll_wait(...)**, a następnie przywrócenia poprzedniej).

Z wykorzystaniem funkcji **poll(...)**

poll(...) działa niemalże identycznie jak **select(...)**, jest jednak w stanie monitorować dowolną ilość deskryptorów (dla **select(...)** istnieje ograniczenie ustalone na etapie komplikacji biblioteki standardowej C), stąd stosowanie **poll(...)** jest preferowane względem **select(...)**.

Wszystkie poniższe funkcje i typy zdefiniowane są w pliku **poll.h**. Dodatkowo funkcja **ppoll(...)** wymaga zdefiniowania **_GNU_SOURCE** przed dołączeniem nagłówka.

struct pollfd

W przypadku **poll(...)** interesujące nas zdarzenia definiujemy dla każdego deskryptora z osobna, nie mamy dzięki temu ograniczenia na numer deskryptora który chcemy monitorować. Po wykonaniu funkcji również dla każdego deskryptora z osobna otrzymujemy informacje jakie zdarzenia na nim wystąpiły.

```
struct pollfd {  
    int fd;          /* file descriptor */  
    short events;    /* requested events */  
    short revents;   /* returned events */  
};
```

- **fd** — numer deskryptora (jeśli będzie ujemny, struktura zostanie pominięta — można to wykorzystać do szybkiego jednorazowego "wyłączenia" monitorowania konkretnego deskryptora przed **poll(...)** przez pomnożenie jego numeru przez **-1**)
- **events** — maska bitowa wartości określających monitorowane zdarzenia:
 - **POLLIN** — możliwy odczyt
 - **POLLOUT** — możliwy zapis
 - **POLLURG** — pojawiły się do odczytania dane priorytetowe
 - **POLLRDHUP** (pod warunkiem zdefiniowania **_GNU_SOURCE**) — drugi koniec gniazda został odłączony (klient rozłączył się lub zamknął swój kanał przeznaczony do zapisu)
- **revents** — maska bitowa wartości określających jakie zdarzenia zaszły (uzupełniana po wywołaniu funkcji), może przyjmować wartości takie jak **events** a dodatkowo:
 - **POLLERR** — wystąpił błąd
 - **POLLHUP** — nastąpiło rozłączenie
 - **POLLNVAL** — nieprawidłowy deskryptor (nie jest otwarty)

int poll(struct pollfd *fds, nfds_t nfds, int timeout)

Funkcja oczekuje na zdarzenia, a po ich wystąpieniu aktualizuje struktury które opisywały zdarzenia na które oczekujemy. Oczekiwanie może zostać przerwane przez sygnał.

Funkcja zwraca ilość deskryptorów na których wystąpiły obserwowane zmiany (zostanie zwrócone **0** jeśli upłynął określony przez nas limit czasu i nie wystąpiło żadne zdarzenie) lub **-1** i ustawia **errno** w przypadku wystąpienia błędu.

- **fds** — tablica struktur zawierających informacje jakimi zdarzeniami na jakim deskryptorze jesteśmy zainteresowani oraz miejsce na zapisanie jakie zdarzenia wystąpiły
- **nfds** — ilość struktur w tablicy (długość tablicy)
- **timeout** — maksymalny czas oczekiwania na zdarzenia podany w milisekundach (wartość ujemna oznacza oczekiwanie w nieskończoność)

int ppoll(struct pollfd *fds, nfds_t nfds, const struct timespec *timeout_ts, const sigset_t *sigmask)

Działa jak **poll(...)**, ale dodatkowo przyjmuje maskę sygnałów do ustawienia na czas oczekiwania na zdarzenia (można to porównać do atomowego wykonania ustawienia podanej maski, wykonania **poll(...)**, a następnie przywrócenia poprzedniej). Dodatkowo timeout jest strukturą zamiast liczbą.

- **timeout_ts** — zdefiniowana w nagłówku **sys/time.h** struktura opisująca maksymalny czas oczekiwania na zdarzenia

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

- **sigmask** — wskaźnik na zbiór sygnałów do zamaskowania (**NULL** oznacza zignorowanie tego argumentu)

Z wykorzystaniem funkcji select(...)

Wszystkie poniższe funkcje i typy zdefiniowane są w pliku nagłówkowym **sys/select.h**.

fd_set

Typ o stałym rozmiarze pozwalający na przechowywanie zbioru deskryptorów plików z których numer żadnego nie przekracza wartości stałej **FD_SETSIZE**.

void FD_CLR(int fd, fd_set *set)

Funkcja służy do usunięcia wskazanego deskryptora **fd** ze zbioru **set**.

int FD_ISSET(int fd, fd_set *set)

Funkcja służy do sprawdzenia czy wskazany deskryptor **fd** znajduje się w zbiorze **set**.

void FD_SET(int fd, fd_set *set)

Funkcja służy do dodania wskazanego deskryptora **fd** do zbioru **set**.

void FD_ZERO(fd_set *set)

Funkcja służy do wyczyszczenia zbioru **set**.

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)

Funkcja monitoruje trzy wskazane zbiorы deskryptorów plików pod kątem możliwości natychmiastowego (nieblokującego) wykonania na nich odpowiedniej operacji (dla **readfds** — odczytu; może być to również odczyt końca pliku **EOF**, dla **writefds** — zapisu, a w przypadku **exceptfds** monitorowane jest wystąpienie sytuacji wyjątkowych). Jednocześnie, możemy określić maksymalny czas oczekiwania na zdarzenia na monitorowanych deskryptorach.

Funkcja zwraca sumaryczną ilość deskryptorów na których wystąpiły obserwowane zmiany (zostanie zwrócone **0** jeśli upłynął określony przez nas limit czasu i nie wystąpiło żadne zdarzenie) lub **-1** i ustawia **errno** w przypadku wystąpienia błędu. Dodatkowo w zbiorach przekazanych jako argumenty zostaną pozostawione jedynie te deskryptory na których zaszło odpowiednie zdarzenie, wiedząc zatem jakie deskryptory dodawaliśmy do zbiorów możemy się po nich przeiterować wywołując dla każdego **FD_ISSET(...)** w celu sprawdzenia czy to właśnie on spowodował powrót z **select(...)**.

- **nfds** — największy numer deskryptora w naszych zbiorach powiększony o **1**
- **readfds** — wskaźnik na zbiór deskryptorów monitorowanych pod kątem możliwości odczytu lub **NULL** jeśli tego nie monitorujemy
- **writefds** — wskaźnik na zbiór deskryptorów monitorowanych pod kątem możliwości zapisu lub **NULL** jeśli tego nie monitorujemy
- **exceptfds** — wskaźnik na zbiór deskryptorów monitorowanych pod kątem zdarzeń wyjątkowych lub **NULL** jeśli tego nie monitorujemy
- **timeout** — wskaźnik na strukturę (z nagłówka **sys/time.h**) opisującą maksymalny czas oczekiwania na zdarzenia lub **NULL** jeśli ustalamy limitu; wypełnienie jej zerami sprawia że **select(...)** nie czeka na zdarzenia, a jedynie zwraca nam (przez modyfikację zbiorów) informacje na temat tego które deskryptory gotowe są do poszczególnych operacji; po powrocie z wywołania funkcji struktura będzie zawierała informacje na temat czasu przez jaki oczekiwano by jeszcze na zdarzenia gdyby się takie nie pojawiły

```

struct timeval {
    time_t      tv_sec;      /* seconds */
    suseconds_t tv_usec;     /* microseconds */
};

```

int pselect(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, const struct timespec *timeout, const sigset(SIGSETSIGMASK))

Działa jak `select(...)`, ale dodatkowo przyjmuje maskę sygnałów do ustawienia na czas oczekiwania na zdarzenia (można to porównać do atomowego wykonania ustawienia podanej maski, wykonania `select(...)`, a następnie przywrócenia poprzedniej). Dodatkowo przyjmuje nieco inną strukturę opisującą timeout.

- `timeout` — jak dla `timeout` w `select(...)`, tylko format minimalnie inny (nanosekundy zamiast mikrosekund); również pochodzi z nagłówka `sys/time.h`

```

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

```

- `sigmask` — wskaźnik na zbiór sygnałów do zamaskowania (`NULL` oznacza zignorowanie tego argumentu)

Typowy "cykl życia" gniazda

Gniazdo połączeniowe

Serwer	Klient
<code>socket</code>	<code>socket</code>
<code>bind</code>	
<code>listen</code>	<code>connect</code>
<code>accept</code>	
powtarzające się <code>send/recv</code>	powtarzające się <code>send/recv</code>
<code>shutdown</code>	<code>shutdown</code>
<code>close</code>	<code>close</code>

Gniazdo bezpołączeniowe

Serwer	Klient
<code>socket</code>	<code>socket</code>
<code>bind</code>	<code>bind</code> (gdy uniksowe gniazdo datagramowe)
	<code>connect</code>
powtarzające się <code>sendto/recvfrom</code>	powtarzające się <code>send/recv</code>
<code>close</code>	<code>close</code>

Przydatne narzędzia

W trakcie korzystania z gniazd wielokrotnie zachodzi potrzeba sprawdzenia poprawności działania naszych programów lub ustalenia miejsca występowania problemu. Możemy w tym celu skorzystać z wielu programów — poniżej wymieniono pewne podstawowe.

ss (część pakietu narzędzi iproute)

Wywołanie: `ss [opcje] [filtr]`

Pozwala na wyświetlenie listy obecnie nawiązanych połączeń oraz listy aktualnie nasłuchujących gniazd wraz z ich parametrami. Komenda domyślnie wyświetla listę aktualnie nawiązanych połączeń. Nazwy gniazd abstrakcyjnych UNIX poprzedzone są @.

- `-l/--listening` — wyświetl listę nasłuchujących gniazd
- `-p/--processes` — wyświetl informacje na temat programu korzystającego z gniazda (PID i nazwa)
- `-n/--numeric` — nawet jeśli do numeru portu przypisana jest nazwana usługa, pokaż go jako numer
- `-t/--tcp` — wyświetl tylko gniazda korzystające z protokołu TCP
- `-u/--udp` — wyświetl tylko gniazda korzystające z protokołu UDP
- `-x/--unix` — wyświetl tylko gniazda w domenie lokalnych gniazd UNIX
- `-4/--ipv4` — wyświetl tylko gniazda w domenie IPv4
- `-6/--ipv6` — wyświetl tylko gniazda w domenie IPv6
- `-f typ/--family=typ` — wyświetl tylko gniazda podanego rodzaju (unix, inet, inet6, ...); argument można podać wielokrotnie; `-x`, `-4` i `-6` to aliasy

netstat (część pakietu narzędzi net-tools)

Uwaga: zgodnie z manuałem — *This program is mostly obsolete. Replacement for netstat is ss.*

Wywołanie: `netstat [opcje]`

Pozwala na wyświetlenie listy obecnie nawiązanych połączeń oraz listy aktualnie nasłuchujących gniazd wraz z ich parametrami. Komenda domyślnie wyświetla listę aktualnie nawiązanych połączeń. Nazwy gniazd abstrakcyjnych UNIX poprzedzone są @.

- **-l/—listening** — wyświetl listę nasłuchujących gniazd
- **-p/—programs** — wyświetl informacje na temat programu korzystającego z gniazda (PID i nazwa)
- **--numeric-ports** — nawet jeśli do numeru portu przypisana jest nazwana usługa, pokaż go jako numer
- **-t/—tcp** — wyświetl tylko gniazda korzystające z protokołu TCP
- **-u/—udp** — wyświetl tylko gniazda korzystające z protokołu UDP
- **-x/—unix** — wyświetl tylko gniazda w domenie lokalnych gniazd UNIX
- **-4/—inet** — wyświetl tylko gniazda w domenie IPv4
- **-6/—inet6** — wyświetl tylko gniazda w domenie IPv6
- **-A typ1,typ2,.../—protocol=typ1,typ2,...** — wyświetl tylko gniazda podanego rodzaju (unix, inet, inet6, ...); argument można podać wielokrotnie; **-x**, **-4** i **-6** to aliasy

telnet

Wywołanie (wariant BSD): `telnet host [port]`

Pozwala na nawiązanie połączenia z gniazdami korzystającymi z protokołu TCP. Domyślnie połączenie nawiązywanie na porcie 23, który jest standardowym portem wykorzystywanym przez usługę Telnet (zdalny terminal). Możliwość nawiązywania połączeń z innymi usługami sieciowymi jest de facto "skutkiem ubocznym" działania tego protokołu. Nie nadaje się do ręcznego korzystania z protokołów przesyłających dane inaczej niż w formie czytelnego dla człowieka tekstu (ale można np. pomóc sobie bash-em i użyć go do zamiany zapisu szesnastkowego na konkretne wartości: `echo -e "\x12\x13" | telnet 127.0.0.1 31415`).

netcat

Wywołanie (wariant Ncat z projektu Nmap): `nc [opcje] [host] [port]`

Pozwala na nawiązywanie połączeń z gniazdami korzystającymi z TCP, UDP i lokalnej komunikacji UNIX. Umożliwia również przyjmowanie połączeń (może pracować jako serwer). Domyślnie netcat pracuje jako klient korzystający z TCP.

- **-l** — pracuj jako serwer
- **-u/—udp** — używaj UDP zamiast TCP
- **-U/—unixsock** — używaj lokalnej komunikacji UNIX zamiast TCP
 - nie podajemy portu, a jako host podajemy ścieżkę do gniazda
 - domyślnie gniazdo pracuje strumieniowo, użycie **-u/—udp** przełącza je w tryb datagramowy

Ostatnia modyfikacja: wtorek, 9 maja 2023, 07:13



AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

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

Pobierz aplikację mobilną



Wybierz język