

# 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 warstwę abstrakcji nad fizycznym (światłowód, przewód miedziany, sieć bezprzewodowa) i logicznym (działanie protokołów sieciowych) sposobem połączenia maszyn.

### Podstawowe cechy

#### Dziedzina

Dziedzina określa jaką rodzinę 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
AF_INET6 – komunikacja internetowa w oparciu o protokół sieciowy
```

Inne dostępne to AF\_PACKET (obsługa "surowych" ramek warstwy łącza 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 umieszczane 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 zagiąć 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
- bezpołączeniowa — 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 n

```
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 stałą **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 `htons(...)`.

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

```
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 `unistd.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 jak  
inp — wskaźnik na zaalokowany obszar pamięci przeznaczonej na pr  
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 powstały 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 błędnego zwraca 0, a dla nieobsługiwanej rodziny adresów zwraca -1 i ustawia errno.

```
af – rodzina adresów; AF_INET dla IPv4, AF_INET6 dla IPv6 – inne
src – wskaźnik na bufor znakowy zawierający odpowiedni dla rodzi
dst – wskaźnik na zaalokowany obszar pamięci przeznaczony na pr
const char *inet_ntop(int af, const void *src, char *dst, socklen_t
```

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 NULLa i ustawia errno w przypadku błędu.

```
af – rodzina adresów; AF_INET dla IPv4, AF_INET6 dla IPv6 – inne
src – wskaźnik na właściwą dla rodziny adresów strukturę przech
dst – wskaźnik na zaalokowany dla bufora znakowego obszar pamięc
size – ilość bajtów dostępnych w buforze docelowym (minimum INET
```

---

Poniższa funkcja została zdefiniowana w nagłówku netdb.h:

```
struct hostent *gethostbyname(const char *name)
```

Funkcja zwraca adres/adresy powiązane z podaną nazwą domenową. W

gethostbyname(...) zostało uznane za przestarzałe i zaleca się kor

Funkcja zwraca wskaźnik na struct hostent jeśli wywołanie zakońc

```
struct hostent {
    char *h_name;           /* official name of host */
    char **h_aliases;       /* alias list */
    int h_addrtype;         /* host address type */
    char *h_addr;           /* pointer to array of addresses
```



```
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 `NULlem`.

Należy wykonywać odpowiednie rzutowania (na `struct in_addr` bądź `struct in6_addr`).

Jeśli interesuje nas dowolny z adresów powiązanych z nazwą (choć nierzadko otrzymamy tylko jeden), to możemy odwołać się do `h_addr`.

Poniższa funkcja dostępna jest w pliku nagłówkowym `unistd.h` i wy

```
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

`len` – wielkość zaalokowanego obszaru przeznaczona na nazwę (nie  
Funkcje do obsługi zmiany kolejności bajtów

Różne architektury procesorów mogą różnić się między sobą sposob

Wyróżnia się dwie metody:

`little-endian` – jako pierwszy przechowywany/wysyłany jest najmni  
`big-endian` – jako pierwszy przechowywany/wysyłany jest najbardzi

Dla przykładu, liczba 0xAABBCCDD będzie przechowywana w obu syst

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 rodziny funkcji dostępna jest na Linuksie w pliku nagłówkowym endian.h

i wymaga wcześniejszego zdefiniowania

\_DEFAULT\_SOURCE (w miejsce 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 kodowa

uintNN_t htoleNN(uintNN_t host_NNbits)
Przekształca podaną liczbę NN-bitową z kodowania hosta na kodowa

uintNN_t beNNtoh(uintNN_t big_endian_NNbits)
Przekształca podaną liczbę NN-bitową z kodowania big-endian na k

uintNN_t leNNtoh(uintNN_t little_endian_NNbits)
Przekształca podaną liczbę NN-bitową z kodowania little-endian n
```

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 kod

uint16_t htons(uint16_t hostshort)
Przekształca podaną liczbę 16-bitową (s w nazwie jak short) z ko

uint32_t ntohl(uint32_t netlong)
```

Przekształca podaną liczbę 32-bitową (l w nazwie jak long) z kod

```
uint16_t ntohs(uint16_t netshort)
```

Przekształca podaną liczbę 16-bitową (s w nazwie jak short) z ko

## 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 desk

`domain` – jedna z opisanych wcześniej stałych określających rodzi

`type` – jeden z opisanych wcześniej trybów komunikacji, ewentualn

`SOCK_NONBLOCK` – gniazdo w trybie nieblokującym (ustawienie `O_NON`

`SOCK_CLOEXEC` – zamknij deskryptor gniazda w chwili wywołania exe

`protocol` – protokół wykorzystywany do komunikacji; podanie warto

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addr
```

Po utworzeniu gniazda przy użyciu `socket(...)` nie ma ono przypisan

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 adr

`addrlen` – wielkość struktury przekazanej w drugim argumente (np

W domenie internetu (`AF_INET/AF_INET6`) skorzystanie z funkcji se

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 związanie gniazda z wygenerowaną losowo abstrakcyjną nazwą (`autobind`).

```
int listen(int sockfd, int backlog)
```

Funkcja ta dotyczy gniazd `connection-oriented` (`SOCK_STREAM`) – od

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  
int connect(int sockfd, const struct sockaddr *addr, socklen_t a
```

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  
addrlen – wielkość struktury przekazanej w drugim argumencie (np  
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen  
int accept4(int sockfd, struct sockaddr *addr, socklen_t *addrlen
```

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 deskrypcorze 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
addrlen – wskaźnik do zmiennej określającej zaalokowaną wielkość
flags – flagi SOCK_NONBLOCK i SOCK_CLOEXEC o znaczeniu identycznym
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 flag
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 przesłania 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 ust
MSG_MORE – dla socketów strumieniowych po TCP wpływa na sposób p
MSG_NOSIGNAL – nie wysyłaj do procesu SIGPIPE jeśli druga strona
...
```

Gdy `flags` jest ustawione na 0, to `send(...)` działa dokładnie tak samo jak `write(...)`.

```
dest_addr – wskaźnik do struktury opisującej adres odbiorcy dany
addrlen – długość struktury opisywanej przez dest_addr; dla gnia
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 fl
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
flags – flagi określające sposób odbioru danych:
MSG_DONTWAIT – odbiera dane w sposób nieblokujący (tak jak po us
MSG_WAITALL – blokuje wywołanie do momentu aż zostanie odebranych
MSG_PEEK – odczytuje oczekujące na gnieździe dane i nie usuwa ich
MSG_NOSIGNAL – nie wysyła do procesu SIGPIPE jeśli druga strona
...
```

Gdy `flags` jest ustawione na 0, to `recv(...)` działa dokładnie tak samo jak `read(...)`.

```
src_addr – wskaźnik do miejsca w pamięci przygotowanego do przyjęcia
addrlen – wskaźnik do zmiennej określającej zaalokowaną wielkość
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 *ad
```

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  
addrlen – wskaźnik do zmiennej określającej zaalokowaną wielkość  
int setsockopt(int sockfd, int level, int optname, const void *o
```

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_S  
optname – "nazwa" opcji, czyli stała powiązana z konkretnym para  
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  
SO_KEEPALIVE – decyduje o wysyłaniu specjalnych pustych pakietów  
SO_PASSCRED – decyduje o odbieraniu komunikatu sterującego SCM_C
```

```
int getsockopt(int sockfd, int level, int optname, void *optval,
```

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  
optname – "nazwa" opcji, czyli stała powiązana z konkretnym para  
optval – wskaźnik do zaalokowanego obszaru pamięci do którego ma  
optlen – wskaźnik do zmiennej określającej zaalokowaną wielkość
```

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  
SO_ACCEPTCONN – czy gniazdo znajduje się w stanie nasłuchiwania  
SO_PEERCRED – w przypadku wcześniejszego ustawienia SO_PASSCRED  
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 – jedyną dopuszczalną na Linuksie opcją jest podanie AF_U  
type – jeden z opisanych wcześniej trybów komunikacji, ewentualn  
SOCK_NONBLOCK – gniazdo w trybie nieblokującym (ustawienie O_NON  
SOCK_CLOEXEC – zamknij deskryptor gniazda w chwili wywołania exe
```



```
protocol – podajemy 0, czyli automatyczny wybór  
sv – tablica w której zostaną umieszczone deskryptory obu gniazd
```

## 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 (us
```

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 podpowiedzenia jądra systemu na ile monitorowanych deskryptorów powinno być przygotowane; obecnie jest ignorowany (jądro powiększa odpowiednie struktury dynamicznie), ale ze względów kompatybilności wstecznej 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 podanego deskryptora pliku przez daną instancję `epoll`.

```
epfd – numer deskryptora instancji mechanizmu epoll (zwrócony z  
op – rodzaj operacji do wykonania:  
EPOLL_CTL_ADD – zarejestrowanie deskryptora pliku do monitorowan  
EPOLL_CTL_MOD – zmiana sposobu monitorowania deskryptora pliku  
EPOLL_CTL_DEL – wyrejestrowanie deskryptora pliku z monitorowani  
fd – numer deskryptora pliku w odniesieniu do którego chcemy wyk  
event – wskaźnik na strukturę struct epoll_event opisującą sposób
```

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 szc

EPOLLIN – możliwy odczyt

EPOLLOUT – możliwy zapis

EPOLLPRI – pojawiły się dane priorytetowe

EPOLLRDHUP – drugi koniec gniazda został odłączony (klient rozłą

EPOLLET (Edge Triggered) – domyślnie epoll działa w trybie level

EPOLLONESHOT – powiadamia tylko o najbliższym zdarzeniu na danym

EPOLLEXCLUSIVE – w sytuacji kiedy kilka instancji epoll monitoru

Pole data zawiera unie, w której możemy przekazać informację jak

```
int epoll_wait(int epfd, struct epoll_event *events, int maxeven
```

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  
events – wskaźnik na zaalokowaną tablicę elementów struct epoll\_  
maxevents – ilość elementów zaalokowanej przez nas tablicy  
timeout – maksymalny czas oczekiwania na zdarzenia (wyrażony w m

```
int epoll_pwait(int epfd, struct epoll_event *events, int maxeve
```

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 ustalane na etapie kompilacji 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 k

struct pollfd {
    int    fd;           /* file descriptor */
    short  events;       /* requested events */
    short  revents;      /* returned events */
};
fd – numer deskryptora (jeśli będzie ujemny, struktura zostanie
events – maska bitowa wartości określających monitorowane zdarze
POLLIN – możliwy odczyt
POLLOUT – możliwy zapis
POLLURG – pojawiły się do odczytania dane priorytetowe
POLLRDHUP (pod warunkiem zdefiniowania _GNU_SOURCE) – drugi koni
revents – maska bitowa wartości określających jakie zdarzenia za
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 zdarzenia
nfds – ilość struktur w tablicy (długość tablicy)
timeout – maksymalny czas oczekiwania na zdarzenia podany w mili
int ppoll(struct pollfd *fds, nfds_t nfds, const struct timespec
Działa jak poll(...), ale dodatkowo przyjmuje maskę sygnałów do us

timeout_ts – zdefiniowana w nagłówku sys/time.h struktura opisuj
struct timespec {
    long    tv_sec;           /* seconds */
    long    tv_nsec;         /* nanoseconds */

```

```
};  
sigmask – wskaźnik na zbiór sygnałów do zamaskowania (NULL oznacza
```

## Z wykorzystaniem funkcji select(...)

Wszystkie poniższe funkcje i typy zdefiniowane są w pliku nagłów

`fd_set`

Typ o stałym rozmiarze pozwalający na przechowywanie zbioru desk

`void FD_CLR(int fd, fd_set *set)`

Funkcja służy do usunięcia wskazanego deskryptora `fd` ze zbioru `s`

`int FD_ISSET(int fd, fd_set *set)`

Funkcja służy do sprawdzenia czy wskazany deskryptor `fd` znajduje

`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 *`

Funkcja monitoruje trzy wskazane zbiory 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 powiększo
readfds – wskaźnik na zbiór deskryptorów monitorowanych pod ką
writefds – wskaźnik na zbiór deskryptorów monitorowanych pod ką
exceptfds – wskaźnik na zbiór deskryptorów monitorowanych pod ką
timeout – wskaźnik na strukturę (z nagłówka sys/time.h) opisując
struct timeval {
    time_t      tv_sec;      /* seconds */
    suseconds_t tv_usec;     /* microseconds */
};
int pselect(int nfds, fd_set *readfds, fd_set *writefds, fd_set
Działa jak select(...), ale dodatkowo przyjmuje maskę sygnałów do

timeout – jak dla timeout w select(...), tylko format minimalnie i
struct timespec {
    long    tv_sec;          /* seconds */
    long    tv_nsec;         /* nanoseconds */
};

sigmask – wskaźnik na zbiór sygnałów do zamaskowania (NULL oznac
```

## Typowy "cykl życia" gniazda

Gniazdo połączeniowe

Serwer Klient

socket socket

bind

listen connect

accept

powtarzające się send/recv powtarzające się send/recv

shutdown shutdown

close close

Gniazdo bezpołączeniowe

Serwer Klient

socket socket

bind bind (gdy uniksowe gniazdo datagramowe)

connect

powtarzające się sendto/recvfrom powtarzające się send/recv

close close

## 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

- l/--listening – wyświetl listę nasłuchujących gniazd
- p/--processes – wyświetl informacje na temat programu korzystaj
- n/--numeric – nawet jeśli do numeru portu przypisana jest nazwa
- 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 UN
- 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 (u

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

- l/--listening – wyświetl listę nasłuchujących gniazd
- p/--programs – wyświetl informacje na temat programu korzystają
- numeric-ports – nawet jeśli do numeru portu przypisana jest na
- 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 UN
-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 p
```

telnet

Wywołanie (wariant BSD): telnet host [port]

Pozwala na nawiązanie połączenia z gniazdami korzystającymi z pr

netcat

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

Pozwala na nawiązywanie połączeń z gniazdami korzystającymi z TC

-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 przełącz