6. Rozbudowany interfejs gniazd

6.1. Opcje gniazd

- Domyślne działanie gniazda można zmieniać za pomocą opcji. Funkcje, które pozwalają pobierać i ustawiać opcje gniazd to: `getsockopt` i `setsockopt`

```
int getsockopt(int socket,            // deskryptor otwartego gniazda
               int level,             // kto ma przetwarzać opcję
               int optName,           // nazwa opcji
               void *optVal,          // zmienna dla wartości opcji
               unsigned int *optLen); // rozmiar zmiennej z opcją

int setsockopt(int socket,            // deskryptor otwartego gniazda
               int level,             // kto ma przetwarzać opcję
               int optName,           // nazwa opcji
               const void *optVal,    // zmienna z wartością opcji
               unsigned int optLen);  // rozmiar zmiennej z opcją
```

  W przypadku poprawnego wykonania funkcje zwracają 0, w przypadku błędu -1 i kod błędu w zmiennej `errno`.

- Opcje mogą dotyczyć różnych poziomów oprogramowania sieciowego (parametr `level`):
  `SOL_SOCKET` - oprogramowanie poziomu gniazd - dotyczy wszystkich gniazd
  `IPPROTO_IP` - oprogramowanie IPv4
  `IPPROTO_IPV6` - oprogramowanie IPv6
  `IPPROTO_TCP` - oprogramowanie TCP

- Opis opcji:
```
man 7 socket
man 7 tcp
man 7 ip
```

- Dwa typy opcji:
  - opcje, które włączają lub wyłączają pewną właściwość
  - opcje, które pobierają lub przekazują specjalne wartości

Przykłady opcji

Nazwa		Typ	Wartość
Poziom **SOL_SOCKET**			
SO_BROADCAST	zezwolenie na wysyłanie w trybie rozgłaszania	int	0, 1
SO_KEEPALIVE	testowanie okresowe, czy połączenie żyje	int	0, 1
SO_LINGER	zwlekanie z zamykaniem, jeśli w buforze są dane do wysłania	struct linger	czas
SO_RVCBUF	rozmiar bufora odbiorczego	int	bajty
SO_SNDBUF	rozmiar bufora wysyłkowego	int	bajty
SO_RCVLOWAT	znacznik dolnego ograniczenia bufora odbiorczego	int	bajty
SO_SNDLOWAT	znacznik dolnego ograniczenia bufora wysyłkowego	int	bajty
SO_RCVTIMEO	czas oczekiwania na pobranie	struct timeval	czas
SO_SNDTIMEO	czas oczekiwania na wysłanie	struct timeval	czas
SO_REUSEADDR	zezwolenie współdzielenie przez dwa gniazda pary adres lokalny port	int	0, 1
SO_TYPE	pobranie typu gniazda (tylko getsockname))	int	liczba
SO_OOBLINE	wykorzystywane podczas przetwarzania danych poza pasmowych	int	0,1

- Gniazda połączone TCP dziedziczą niektóre opcje po gnieździe nasłuchującym. Należą do nich SO_KEEPALIVE, SO_LINGER, SO_RVCBUF, SO_SNDBUF.

Opcja SO_BROADCAST

```
# Nadawca

#include <stdio.h>      /* printf(), fprintf() */
#include <sys/socket.h> /* socket(), bind() */
#include <arpa/inet.h>  /* sockaddr_in */
#include <stdlib.h>     /* atoi() */
#include <string.h>     /* memset() */
#include <unistd.h>     /* close() */


int main(int argc, char *argv[])
{
  int gniazdo;
  struct sockaddr_in rozglAdr;
  char *rozglIP;
  unsigned short rozglPort;
  char *tekst;
  int rozglaszanie;
  unsigned int tekstDl;

  if (argc < 4)
  {
     fprintf(stderr,"Uzycie:  %s <Adres IP> <Port> <Tekst>\n", argv[0]);
     exit(1);
  }

  rozglIP = argv[1];          /* Pierwszy arg:  adres rozgloszeniowy */
  rozglPort = atoi(argv[2]);  /* Drugi arg:  port rozgloszeniowy */
  tekst = argv[3];            /* Trzeci arg:  tekst rozglaszany */

  if ((gniazdo= socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    { perror("socket()"); exit(1); }

  rozglaszanie = 1;
  if (setsockopt(gniazdo, SOL_SOCKET, SO_BROADCAST, &rozglaszanie,
    sizeof(rozglaszanie)) < 0)
    { perror("setsockopt()"); exit(1); }

  memset(&rozglAdr, 0, sizeof(rozglAdr));
  rozglAdr.sin_family = AF_INET;
  rozglAdr.sin_addr.s_addr = inet_addr(rozglIP);
  rozglAdr.sin_port = htons(rozglPort);

  tekstDl= strlen(tekst);
  for (;;)
  {
   /* Rozglaszaj co 3 sekundy */
    if (sendto(gniazdo,tekst,tekstDl,0,(struct sockaddr *)&rozglAdr,
       sizeof(rozglAdr)) != tekstDl)
    { perror("sendto() wyslal inna liczbe bajtow niz powinien"); exit(1); }
      sleep(3);
  }
}
```

```
# Odbiorca
#
#include <stdio.h>      /* printf(), fprintf() */
#include <sys/socket.h> /* socket(), connect(), sendto(), recvfrom() */
#include <arpa/inet.h>  /* sockaddr_in, inet_addr() */
#include <stdlib.h>     /* atoi() */
#include <string.h>     /* memset() */
#include <unistd.h>     /* close() */

#define MAXTEKST 255 /* najdluszy odbierany tekst */

int main(int argc, char *argv[])
{
  int gniazdo;
  struct sockaddr_in rozglAdr;
  unsigned int rozglPort;
  char tekst[MAXTEKST+1];
  int tekstDl;

  if (argc != 2)
  {
     fprintf(stderr,"Uzycie: %s <Port rozgloszeniowy>\n", argv[0]);
     exit(1);
  }

  rozglPort = atoi(argv[1]);   /* Pierwszy arg: port rozgloszeniowy */
  if ((gniazdo= socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
  { perror("socket()"); exit(1); }

  memset(&rozglAdr, 0, sizeof(rozglAdr));
  rozglAdr.sin_family = AF_INET;
  rozglAdr.sin_addr.s_addr = htonl(INADDR_ANY);
  rozglAdr.sin_port = htons(rozglPort);

  if (bind(gniazdo,(struct sockaddr *)&rozglAdr,sizeof(rozglAdr)) < 0)
  { perror("bind()"); exit(1); }

  if ((tekstDl = recvfrom(gniazdo, tekst, MAXTEKST, 0, NULL, 0)) < 0)
  { perror("recvfrom()"); exit(1); }

  tekst[tekstDl] = '\0';
  printf("Otrzymano : %s\n", tekst);

  close(gniazdo);
  exit(0);
}
```

Opcja SO_REUSEADDR

Opcja ta jest ustawiana wtedy, kiedy chcemy, aby:

- Serwer nasłuchujący mógł rozpocząć pracę i wywołać funkcję bind nawet wówczas, kiedy istnieją ustanowione wcześniej połączenia, które używają tego portu jako swojego portu lokalnego. Przykład: serwer nasłuchujący skończył pracę, ale jego potomek obsługuje jeszcze klienta; chcemy wznowić pracę serwera bez konieczności czekania na zakończenie procesu potomnego.
- Wiele egzemplarzy tego samego serwera mogło rozpoczynać pracę przez ten sam port, warunek - mają inne adresy lokalne IP.

```
  int serwGniazdo, wynik;
  struct sockaddr_in serwAdr;
  unsigned short serwPort;
  int opcja;

  serwGniazdo = socket(PF_INET, SOCK_STREAM, 0);

  opcja=1;
  wynik = setsockopt(sock,SOL_SOCKET,SO_REUSERADDR,
                   (void *)&opcja, sizeof(opcja));

  memset(serwAdr, 0, sizeof(serwAdr));
  echoSerwAdr.sin_family = AF_INET;
  echoSerwAdr.sin_addr.s_addr = htonl(INADDR_ANY);
  echoSerwAdr.sin_port = htons(serwPort);

/* Przypisz gniazdu lokalny adres */
  wynik =bind(serwGniazdo,(struct sockaddr *) &serwAdr,sizeof(serwAdr));
```

Opcja SO_LINGER

- Opcja określa sposób działania funkcji `close` dla protokołu połączeniowego. Domyślnie funkcja `close` od razu powraca do programu, który ją wywołał, jeśli pozostały jeszcze jakieś dane do wysłania w buforze wysyłkowym gniazda, to system spróbuje je dostarczyć partnerowi. Program jednak nie wie, czy dane te zostały dostarczone pratnerowi. Opcja `SO_LINGER` pozwala zmienić to działanie domyślne. Aplikacja będzie zablokowana w funkcji `close()`, dopóki wszystkie dane nie zostaną dostarczone odbiorcy.

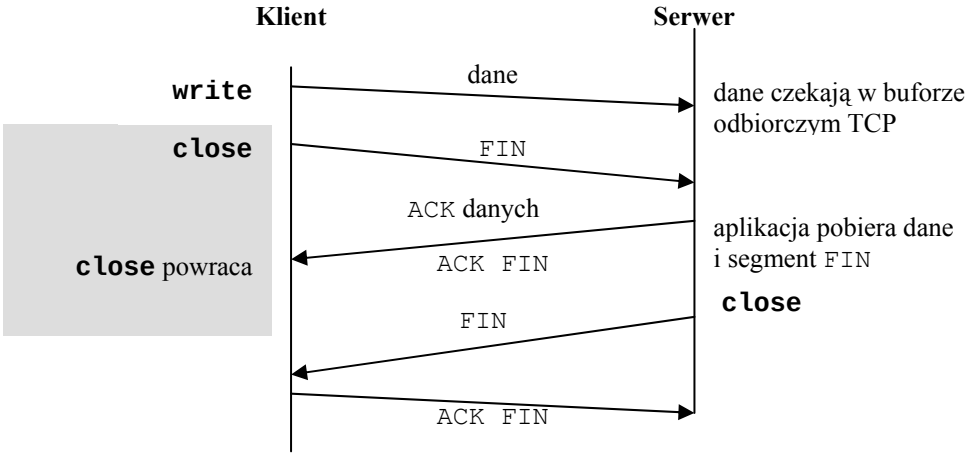
  Działanie opcji `SO_LINGER` definiowane jest za pomocą struktury:

```
struct linger {
  int l_onoff;    /* 0 - wyłączone, niezero - włączone */
  int l_linger;   /* czas zwlekania */
};
```

  Jeśli:

  - `l_onoff` jest równe 0 - ignorowana jest druga składowa i działanie funkcji `close` pozostaje niezmienione
  - `l_onoff` jest różne od 0, `l_linger` jest równe 0 - połączenie zostanie natychmiast zerwane przez warstwę TCP (usunięte zostaną wszystkie dane z bufora wysyłkowego gniazda, wysłany zostanie segment RST zamiast sekwencji kończącej)
  - `l_onoff` jest różne od 0, `l_linger` jest różne od 0 - proces będzie uśpiony dopóty, dopóki albo wszystkie dane będą wysłane i nadejdzie potwierdzenie od partnera, albo upłynie czas zwlekania (ang. *linger*). Jeśli wróci się z funkcji w wyniku upłynięcia czasu zwlekania, to zwrócony będzie kod błędu `EWOULDBLOCK` i wszystkie dane pozostawione w buforze wysyłkowym zostaną zniszczone.

- Włączone zwlekanie



Otrzymaliśmy potwierdzenie danych i przesłanego do partnera segmentu FIN. Nadal nie wiemy, czy aplikacja partnera przeczytała dane. Jak uzyskać tę informację?

Opcje SO_RVCBUF i SO_SNDBUF

- Każde gniazdo ma bufor wysyłkowy i odbiorczy.
- Bufor odbiorczy wykorzystywany jest przez oprogramowanie warstwy TCP i UDP do przechowywania danych zanim przeczyta je aplikacja.
  - Wielkość bufora odbiorczego TCP jest równa rozmiarowi okna oferowanego partnerowi. Bufor ten nie może się przepełnić; jeśli partner zignoruje rozmiar okna, warstwa TCP odrzuci nadmiarowe dane. Nadawca będzie musiał je powtórzyć.
  - W przypadku UDP jeśli datagram nie mieści się w buforze odbiorczym gniazda, zostanie odrzucony.
- Każde gniazdo TCP ma bufor wysyłkowy> Do niego kopiowane są dane z bufora użytkowego aplikacji. Jeśli gniazdo jest gniazdem blokującym (ustawienie domyślne), powrót z funkcji `write` będzie oznaczał, że wszystkie dane z bufora aplikacji zostały umieszczone w tym buforze. Dane są usuwane z tego bufora dopiero po otrzymaniu potwierdzenia ACK.
- Gniazdo UDP nie ma bufora wysyłkowego. Posługuje się tylko jego rozmiarem do określenia maksymalnego rozmiaru datagramu, który można wysłać poprzez to gniazdo.

- Przykład: chcemy zwiększyć rozmiar bufora odbiorczego gniazda

```
int rvcBufferSize;
int sockOptSize;
sockOptSize=sizeof(rvcBufferSize);
if (getsockopt(sock,SOL_SOCKET,SO_RCVBUF,&rvcBufferSize, &sockOptSize) < 0)
  error("getsockopt");
printf("Poczatkowa wielkosc bufora: %d\n", rvcBufferSize);

/* Podwajamy wielkość bufora */
rvcBufferSize *=2;
if (setsockopt(sock,SOL_SOCKET,SO_RCVBUF,
               &rvcBufferSize,sizeof(rcvBufferSize)) < 0)
  error("setsockopt");
```

Opcje SO_RCVLOWAT i SO_SNDLOWAT

- Funkcja `select` do stwierdzenia gotowości gniazda do czytania lub pisania wykorzystuje znaczniki dolnego ograniczenia bufora wysyłkowego i odbiorczego (ang. *low-water mark*).
- Znacznik dolnego ograniczenia bufora odbiorczego jest to niezbędna liczba bajtów w buforze odbiorczym gniazda potrzebna do tego aby `select` przekazała informację, że gniazdo nadaje się do pobierania danych.
- Znacznik dolnego ograniczenia bufora wysyłkowego jest to niezbędna wielkość dostępnej przestrzeni w buforze wysyłkowym gniazda potrzebna do tego aby `select` przekazała informację, że gniazdo nadaje się do wysyłania danych. W przypadku UDP znacznik ten oznacza górną granicę maksymalnego rozmiaru datagramów UDP, które można odsyłać do tego gniazda. Gniazdo to nie ma bufora wysyłkowego, ma tylko rozmiar bufora wysyłkowego.

- Przykład: chcemy otrzymać 48 bajtów, zanim nastąpi powrót z operacji czytania

```
int lowat;
int lowatSize;
sockOptSize=sizeof(rvcBufferSize);
int wynik;

lowat=48;
wynik=setsockopt(sock,SOL_SOCKET,SO_RCVLOWAT,&lowat,sizeof(rcvBufferSize));
```

Opcja SO_KEEPALIVE

- Opcja włącza i wyłącza sondowanie połączenie TCP (ang. keepalive probe). Sondowanie polega na wysyłaniu segmentu ACK, na który partner musi odpowiedzieć.

6.2. Funkcje wejścia-wyjścia

```
#include <unistd.h>

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

#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int s, const void *msg, size_t len, int flags);
```
Wysyła komunikat do połączonego hosta. Podobna do **`write`**, ale daje możliwość określenia opcji dla połączenia.

```
ssize_t sendto(int  s,  const  void *msg, size_t len, int flags,
               const struct sockaddr *to, socklen_t tolen);
```
Wysyła komunikat do określonego hosta. Wykorzystywana w połączeniach UDP.

```
ssize_t sendmsg(int s, const struct msghdr *msg, int flags);
```
Najbardziej ogólna funkcja. Wysyła komunikat zbudowany z wielu bloków. Daje możliwość określenia opcji dla połączenia.

```
ssize_t recv(int s, void *buf, size_t len, int flags);
```
Odbiera komunikat do połączonego hosta. Podobna do **`read`**, ale daje możliwość określenia opcji dla połączenia.

```
ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);
```
Odbiera komunikat od hosta. Adres hosta jest zwracany w argumencie `from`. Wykorzystywana w połączeniach UDP.

```
ssize_t recvmsg(int s, struct msghdr *msg, int flags);
```
Najbardziej ogólna funkcja Odbiera komunikat zbudowany z wielu bloków. aje możliwość określenia opcji dla połączenia.

- Przykłady opcji w funkcjach wysyłających (argument flags):

MSG_OOB	Wyślij dane poza pasmowe (ang. *urgent*, *out-of-band*)
MSG_DONTWAIT	Wyślij bez blokowania. Zwraca w `errno` wartość `EWOULDBLOCK`, jeśli funkcja nie może być od razu wykonana. Dotyczy tylko jednego wywołania. Nie trzeba wtedy ustawiać gniazda w trybie nieblokującym.

Przykłady opcji w funkcjach odbierających (argument flags):

MSG_OOB	Odbierz dane poza pasmowe, jeśli nie są one umieszczone w normalnym strumieniu danych.
MSG_DONTWAIT	Odbierz bez blokowania. Zwraca w `errno` wartość `EWOULDBLOCK`, jeśli funkcja nie może być od razu wykonana. Dotyczy tylko jednego wywołania. Nie trzeba wtedy ustawiać gniazda w trybie nieblokującym.
MSG_WAITALL	Czekaj, aż odebrane zostaną wszystkie dane. Uwaga: funkcja może przekazać mniejszą od żądanej liczbę bajtów, gdy przechwycono sygnał lub połączenie zostało zakończone.
MSG_PEEK	Podgląd komunikatu – odbierz dane bez usuwania ich z bufora wejściowego.

6.3. Gniazda nieblokujące

- Modele wejścia-wyjścia:
  - wejście-wyjście blokujące - domyślnie każde gniazdo jest blokujące, program czeka aż pojawią się dane lub będzie można je wysłać
  - wejście-wyjście nieblokujące - powróć z funkcji natychmiast, jeśli nie można zakończyć operacji wejścia-wyjścia zwróć błąd (`EWOULDBLOCK`); program musi odpytywać (ang. *polling*) taki deskryptor, czy operacja jest już gotowa do wykonania
  - wejście-wyjście zwielokrotnione - specjalna funkcja systemowa (`select`, `poll`), w której proces się blokuje, zamiast w funkcji wejścia-wyjścia; zaletą jest możliwość oczekiwania na wiele deskryptorów
  - wejście-wyjście sterowane sygnałami - jądro systemu może generować sygnał `SIGIO`, który poinformuje proces o gotowości deskryptora do rozpoczęcia operacji wejścia-wyjścia; proces nie jest blokowany
  - wejście-wyjście asynchroniczne (Posix) - proces zleca jądru rozpoczęcie operacji wejścia-wyjścia i późniejsze zawiadomienie o jej zakończeniu

- **Przykład:** Wejście-wyjście sterowane sygnałami (tradycyjna nazwa - wejście-wyjście asynchroniczne)

  - Czynności związane z korzystaniem z gniazda w trybie wejścia-wyjścia sterowanego sygnałami
    1. Ustanowienie procedury obsługi sygnału `SIGIO`
    2. Ustalenie właściciela gniazda
    3. Włączenie obsługiwania gniazda w trybie wejścia-wyjścia sterowanego sygnałami

  ad 1.

  Protokół UDP - sygnał `SIGIO` jest generowany m.in. wtedy, kiedy
  - nadejdzie datagram przeznaczony do gniazda

  Protokół TCP - sygnał `SIGIO` jest generowany m.in. wtedy, kiedy
  - zakończono obsługiwanie żądania połączenia z gniazdem nasłuchującym
  - zainicjowano obsługiwanie żądania rozłączenia
  - zakończono obsługiwanie żądania rozłączenia
  - jedna ze stron połączenia została zamknięta
  - do gniazda dostarczono dane
  - z gniazda wysłano dane (zwolniono miejsce w buforze wysyłkowym)

  Ze względu na częste występowanie sygnału SIGIO w przypadku połączenia TCP, jest on rzadko wykorzystywany do sterowania we-wy.

  ad 2.

  Do wykonywania operacji na deskryptorze pliku służy funkcja `fcntl()`:

  ```
  #include <fcntl>
  int fcntl(int fd, int cmd, ... /* int arg */);
  ```

  Ustanowienie właściciela gniazda wymaga polecenia `F_SETOWN`, pozwala przypisać gniazdo procesowi o ustalonym identyfikatorze:

  ```
  fcntl(gniazdo,F_SETOWN,getpid())
  ```

  ad. 3.

  Włączenie obsługiwania gniazda w trybie wejścia-wyjścia sterowanego sygnałami można również zrealizować za pomocą funkcji `fcntl()`:

  ```
  int flagi;
  flagi=fcntl(gniazdo,F_GETFL,0);
  flagi |= FASYNC; // lub flagi |= O_ASYNC
  fcntl(gniazdo,F_SETFL,flagi)
  ```

```
/* Serwer echa */

#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>    /* fcntl() */
#include <sys/file.h> /* O_NONBLOCK i FASYNC */
#include <signal.h>
#include <errno.h>

#define ECHOMAX 255

void ObslugaBledu(char *komunikat);
void CzasDoWykorzystania();
void ObslugaSIGIO(int typSygnalu);

int sock;

int main(int argc, char *argv[]) {
  struct sockaddr_in SerwAdr;
  unsigned short SerwPort;
  struct sigaction obsluga;

  if (argc != 2)   {
    fprintf(stderr, "Wywołanie:  %s <SERWER PORT>\n", argv[0]);
    exit(1);
  }

  SerwPort = atoi(argv[1]);
  if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    ObslugaBledu("socket()");

  memset(&SerwAdr,0,sizeof(SerwAdr));
  SerwAdr.sin_family = AF_INET;
  SerwAdr.sin_addr.s_addr=htonl(INADDR_ANY);
  SerwAdr.sin_port = htons(SerwPort);

  if (bind(sock,(struct sockaddr *)&SerwAdr,sizeof(SerwAdr)) < 0)
    ObslugaBledu("bind()");

/* Ustanowienie procedury obsługi sygnału SIGIO */
  obsluga.sa_handler = ObslugaSIGIO;
  if (sigfillset(&obsluga.sa_mask) < 0)
    ObslugaBledu("sigfillset()");
  obsluga.sa_flags = 0;

  if (sigaction(SIGIO, &obsluga, 0) < 0)
    ObslugaBledu("sigaction() SIGIO");

/* Ustalenie właściciela gniazda */
if (fcntl(sock, F_SETOWN, getpid()) < 0)
   ObslugaBledu("Nie mozna ustawic wlasciciela procesu ");

/* Włączenie obsługiwania gniazda w trybie wejścia-wyjścia
   sterowanego sygnałami */
  if (fcntl(sock,F_SETFL,O_NONBLOCK|FASYNC)< 0)
    ObslugaBledu("Nie mozna ustawic gniazda klienta w trybie
                  O_NONBLOCK|FASYNC");
```

```
for (;;)
  CzasDoWykorzystania();
}

void CzasDoWykorzystania() {
  printf(".\n");
  sleep(3);
}

void ObslugaSIGIO(int typSygnalu){
  struct sockaddr_in KlientAdr;
  unsigned int KlientDl;
  int rozmiar;
  char bufor[ECHOMAX];

  do {
  KlientDl = sizeof(KlientAdr);
  if ((rozmiar = recvfrom(sock, bufor, ECHOMAX, 0,
         (struct sockaddr *)&KlientAdr, &KlientDl)) < 0) {
    if (errno != EWOULDBLOCK)
      ObslugaBledu("recvfrom()");
  }
  else {
    printf("Przetwarzam klienta %s\n", inet_ntoa(KlientAdr.sin_addr));
    if (sendto(sock, bufor, rozmiar, 0,
       (struct sockaddr *)&KlientAdr, sizeof(KlientAdr)) != rozmiar)
      ObslugaBledu("sendto()");
    }
  }  while (rozmiar >= 0);
}
```

Pytania:

- Dlaczego w funkcji obsługi sygnału występuje pętla?

6.4. Sprawdzanie zamknięcia gniazda partnera

- Wykrywanie zamkniętego gniazda podczas czytania

```
  int gniazdo, wynik;
  char bufor[BUFWE];

  gniazdo = socket(PF_INET, SOCK_STREAM, 0));

/ * Klient łączy się */

  wynik = recv(gniazdo, bufoe, BUFWE, 0);
  if (wynik > 0) {
  /* Otrzymano dane, przetwarzaj je */
  }
  else if (wynik < 0 ) {
  /* wystąpił błąd, sprawdż jaki */
  }
  else if (wynik == 0) {
  /* partner zamknął połączenie */
  close(gniazdo);
}
```

- Wykrywanie zamkniętego gniazda podczas zapisu

```
  int serwGniazdo, klientGniazdo;
  int rozmiar;
  char bufor[BUFWE];
  int wynik;

  serwGniazdo = socket(PF_INET, SOCK_STREAM, 0));

/* ustawienie parametrów serwera */

  klientGniazdo =  accept(serwGniazdo, NULL, NULL);

...

  wynik=send(klientGniazdo, bufor, rozmiar, 0);
  if (wynik == 0) {
  /* dane wysłano */
  }
  else if (wynik < 0 ) {
  /* wystąpił błąd, sprawdż jaki */
    if (errno == EPIPE) {
  /* Partner zamknął gniazdo  */
  close(klientGniazdo);
  }
}
```

6.5. Dane pozapasmowe

- Dane pozapasmowe (ang. *out-of-band date*) to dane, które są przesyłane z wyższym priorytetem. Każdy rodzaj warstwy transportowej obsługuje te dane w inny sposób.
- Do obsługi danych pozapasmowych w protokole TCP wykorzystywany jest tryb pilny (ang *urgent mode*). Można w ten sposób przesłać jeden znak jako dane pilne.

- Przesyłanie danych pozapasmowych:

```
send(socket,"?",1,MSG_OOB);
```

- Odczytywanie danych pzapasmowych:
  a) dane odbierane są w specjalnym jednobajtowym buforze danych pozapasmowych (domyślne działanie gniazda); do odczytu można użyć wtedy `recv` z ustawioną flagą `MSG_OOB :`

```
recv(socket,buf,1,MSG_OOB);
```

  b) dane odbierane są przemieszane z danymi zwykłymi (gniazdo ma ustawioną opcję `SO_OOBINLINE`); należy wtedy odszukać dane pilne w zwykłych danych

- Powiadomienie o nadejściu danych pozapasmowych:
  - proces odbierający jest powiadamiany o nadejściu danych pozapasmowych za pomocą sygnału `SIGURG;` proces musi być właścicielem gniazda
  - jeśli proces korzysta z funkcji `select`, to nadejście danych pozapasmowych jest sygnalizowane pojawieniem się sytuacji wyjątkowej (trzeci zestaw badanych deskryptorów)

Przykład: wykorzystanie danych pozapasmowych do śledzenia aktywności połączenia

- Klient echa

```
#define CZEKAJ 5

int sock, odp=1;

void obsluga_syg(int sygnal)
{
  if ( sygnal == SIGURG )
  {
    char c;
    recv(sock, &c, sizeof(c), MSG_OOB);
    odp = ( c == 'T' );              /* żyję */
    fprintf(stderr,"[jest]");
  }
  else if ( sygnal == SIGALRM )
  {
    if ( odp )
    {
      send(sock, "?", 1, MSG_OOB);   /* żyjesz?? */
      alarm(CZEKAJ);
      odp = 0;
    }
    else
      fprintf(stderr, "Brak polaczenia!");
  }
}

...

  struct sigaction act;
  memset(&act, 0, sizeof(act));
  act.sa_handler = obsluga_syg;
  act.sa_flags = SA_RESTART;
  sigaction(SIGURG, &act, 0);
  sigaction(SIGALRM, &act, 0);
...
  sock = socket(PF_INET, SOCK_STREAM, 0);
// ustal właściciela gniazda
  if ( fcntl(sock, F_SETOWN, getpid()) != 0 )
    { perror("F_SETOWN"); exit(1); }
...
  if ( connect(sock, (struct sockaddr*) &adr, sizeof(adr)) == 0 )
  {
// ustaw okres oczekiwania
    alarm(CZEKAJ);
    do
    {
      // pobierz dane z klawiatury i prześlij do serwera
...   // czekaj na odpowiedź
    }
    while ( ... );
  }
...
```

- Serwer echa

```
int sock;
struct sigaction act;

void obsluga_syg(int sygnal)
{
  if ( sygnal == SIGURG )
  {
    char c;
    recv(sock, &c, sizeof(c), MSG_OOB);
    if ( c == '?' )                              /* Czy żyjesz? */
    send(sock, "T", 1, MSG_OOB);                /* TAK! */
  }
  else {if ( sygnal == SIGCHLD )
    wait(0);
  }
}

...

  bzero(&act, sizeof(act));
  act.sa_handler = sig_handler;
  act.sa_flags = SA_RESTART;
  sigaction(SIGURG, &act, 0);
  if ( fcntl(sock, F_SETOWN, getpid()) != 0 )
    perror("F_SETOWN");
  do
  {
    bytes = recv(sock, buffer, sizeof(buffer), 0);
    if ( bytes > 0 )
      send(sock, buffer, bytes, 0);
  }
  while ( bytes > 0 );
  close(sock);
  exit(0);
}
```