Wykład #3
Wprowadzenie do API gniazd. Obsługa podstawowych sygnałów w programach sieciowych.

Cel: zapoznanie się z podstawowymi strukturami i funkcjami operującymi na gniazdach i obsługa sygnałów w programach sieciowych

Struktury adresowe

sockaddr

sockaddr_in

sockaddr_in6

sockaddr_storage

Ogólna gniazdowa struktura adresowa

- Funkcje gniazd działają niezależnie od rodziny protokołów
- Struktura **sockaddr**: <sys/socket.h>
- Wymagane rzutowanie struktur dla funkcji gniazd

```
struct    sockaddr {
     sa_family_t    sa_family; /*address family: AF_xxx value */
     char   sa_data[14]; /* protocol-specific address */
};
```

Gniazdowe struktury adresowe IPv4: `struct sockaddr_in`

- Plik nagłówkowy <netinet/in.h>
- Może występować zmienna określająca długość
- Sieciowa kolejność bajtów
- sin_zero – uzupełnienie długości do sizeof(sockaddr)
- AF_INET – sin_family

Gniazdowe struktury adresowe – IPv4

```
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 */
};
```

Gniazdowe struktury adresowe – typy i ich rozmiary

Typ danych	Opis	Plik nagłówkowy
int8_t uint8_t int16_t uint16_t int32_t uint32_t	8-bitowa liczba całkowita ze znakiem 8-bitowa liczba całkowita bez znaku 16-bitowa liczba całkowita ze znakiem 16-bitowa liczba całkowita bez znaku 32-bitowa liczba całkowita ze znakiem 32-bitowa liczba całkowita bez znaku	<sys/types.h>
sa_family_t socklen_t	Rodzina adresów w strukturze adresowej gniazd Rozmiar gniazdowej struktury adresowej, zwykle uint32_t	<sys/socket.h>
in_addr_t in_port_t	Adres IPv4, zwykle uint32_t Port TCP lub UDP, uint16_t	<netinet/in.h>

Gniazdowe struktury adresowe IPv6: `struct sockaddr_in6`

- Plik nagłówkowy <netinet/in.h>
- Może występować zmienna określająca długość
- Sieciowa kolejność bajtów
- Flow info (20b etykieta przepływu, 12b nieużywane)
- AF_INET6 – sin6_family

Gniazdowe struktury adresowe – IPv6

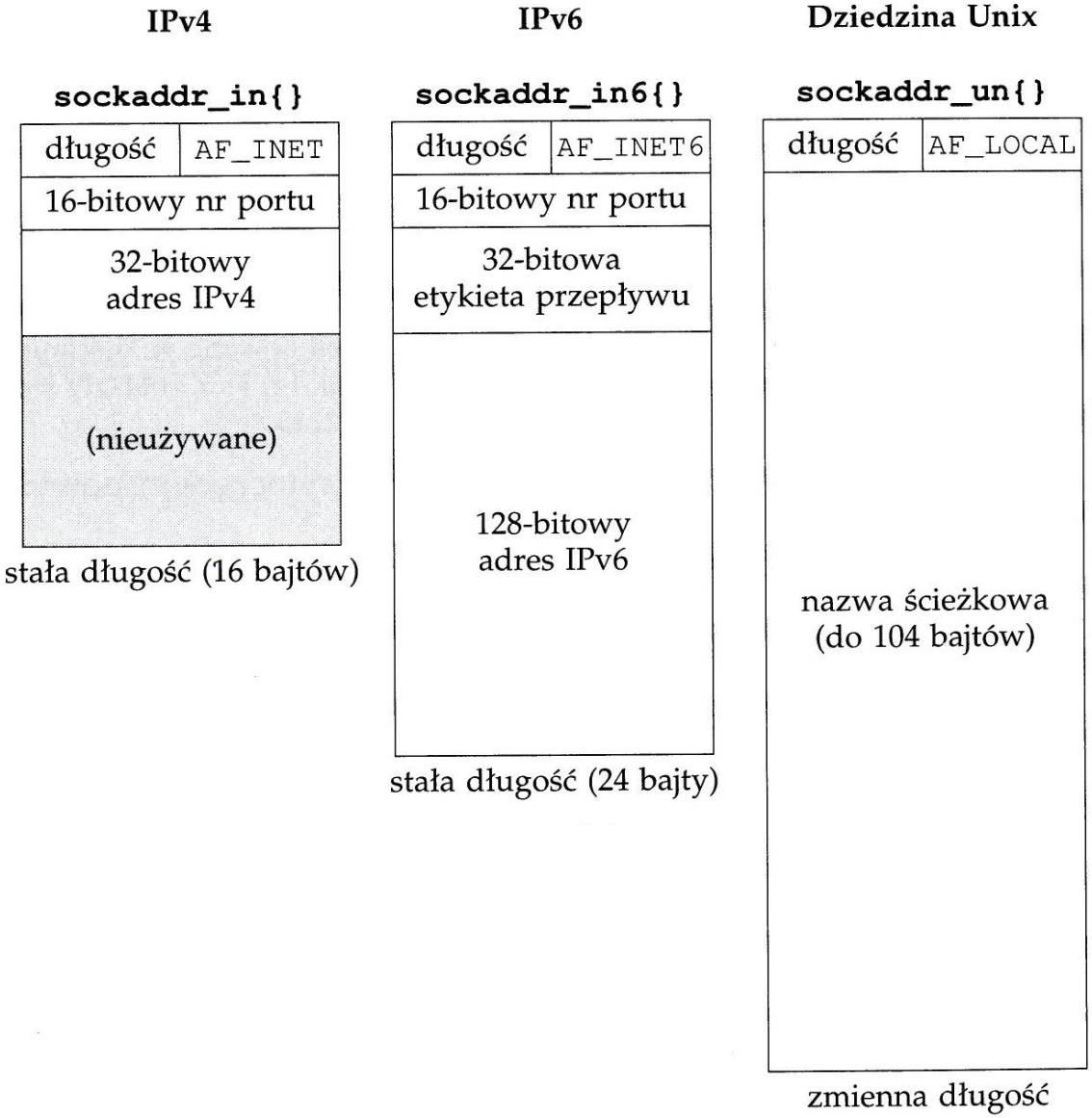
```
 struct sockaddr_in6 {
       sa_family_t     sin6_family;   /*AF_INET6 */
       in_port_t       sin6_port;     /*port number */
       uint32_t        sin6_flowinfo; /*IPv6 flow information */
       struct in6_addr sin6_addr;     /*IPv6 address */
       uint32_t        sin6_scope_id; /*Scope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

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

Uogólniona struktura adresowa **`sockaddr_storage`**

- Można w niej przechowywać każdy typ adresu
- Wspiera wyrównanie bajtów dla każdego z adresów

```
struct sockaddr_storage {
       sa_family_t ss_family; /* Address family */
       __ss_aligntype __ss_align; /* Force desired alignment.*/
       char __ss_padding[_SS_PADSIZE];
};
```

Porównanie struktur adresowych

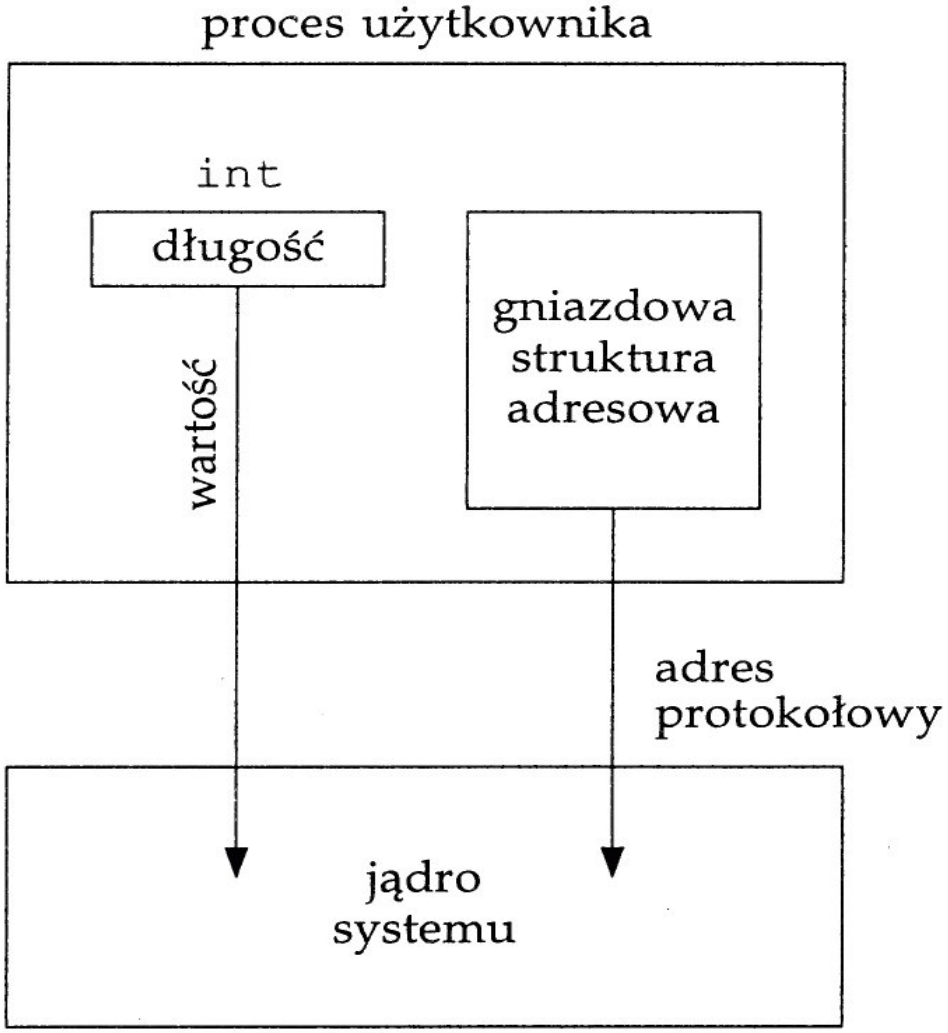


Gniazdowe struktury adresowe – przekazywanie struktur(1/2)

- Gniazdowe struktury adresowe przekazywane są przez odniesienie (wskaźnik)
- bind(), connect(), sendto() – przekazywanie danych od użytkownika do jądra np.:

```
1. struct sockaddr_in  serv;
2. ……(wypełnienie adresu)……
3. connect (sockfd, (sockaddr *) &serv,
           sizeof(serv));
```

Przekazywanie struktur od procesu użytkownika do jądra

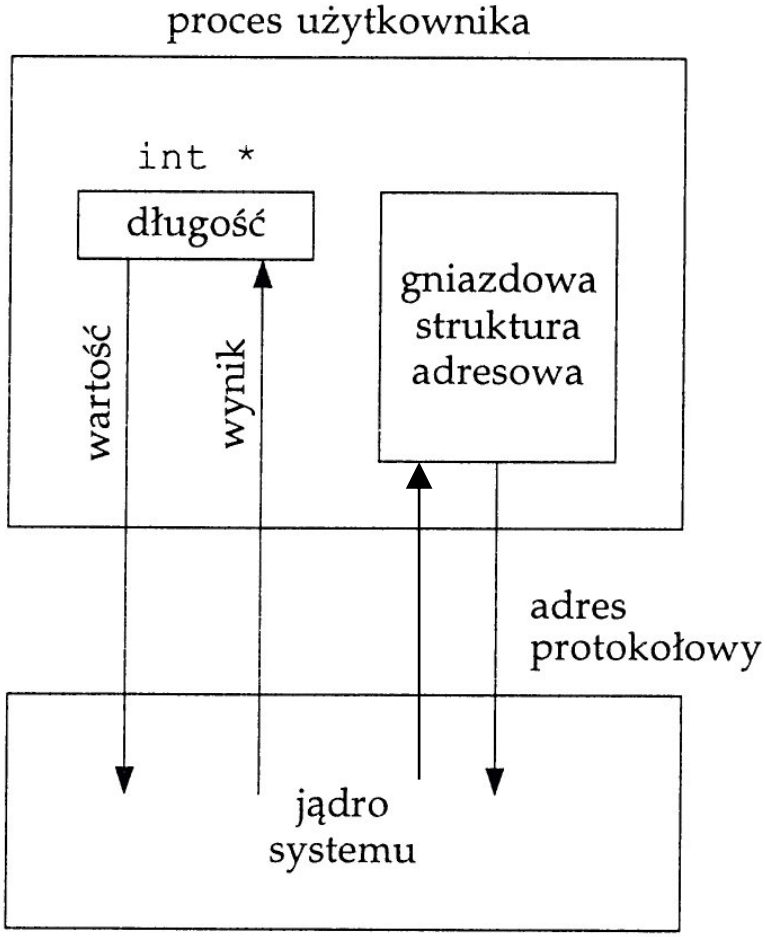


Gniazdowe struktury adresowe – przekazywanie struktur (2/2)

- accept(), recfrom(), getsockname(), getpeername() – przekazywanie danych od jądra do użytkownika, np.:

```
1. struct sockaddr_in     cli;
2. socklen_t    len;
3. len = sizeof(cli); /* len is a value */
4. getpeername(sockfd, (sockaddr *) &cli, &len);
```

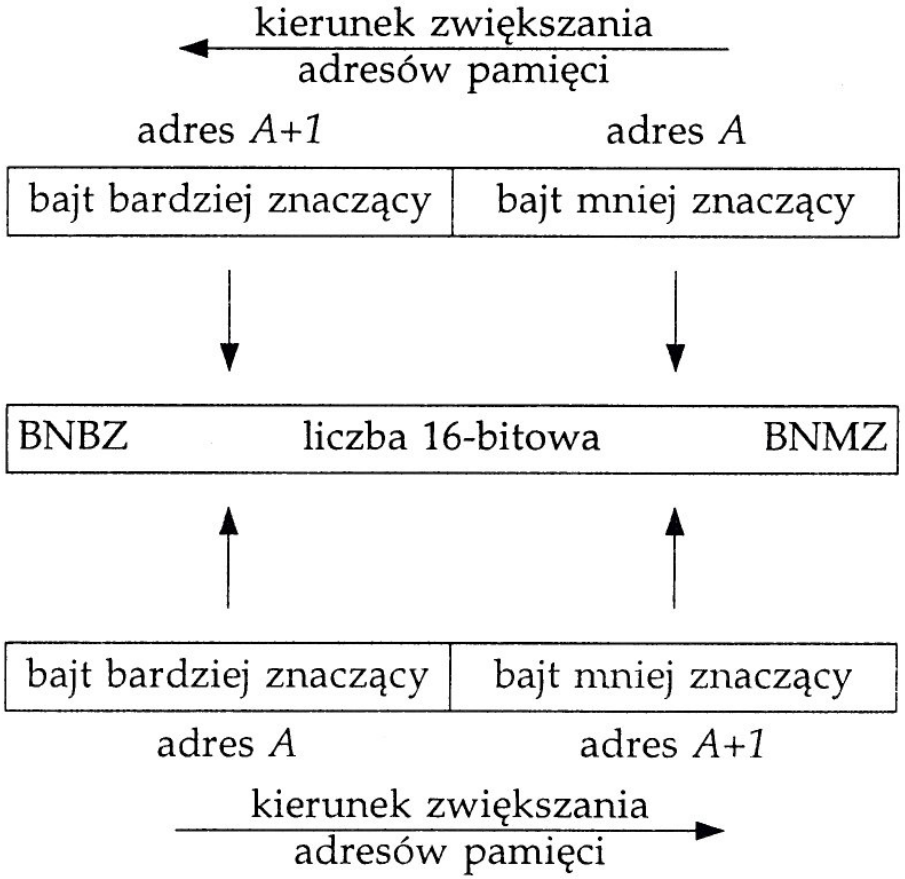
Przekazywanie struktur od jądra do procesu użytkownika



Kolejność bajtów

Kolejność bajtów

kolejność *little-endian*, z pierwszeństwem bajtu mniej znaczącego:



kolejność *big-endian*, z pierwszeństwem bajtu bardziej znaczącego:

Kolejność sieciowa

Program sprawdzający kolejność bajtów

```
1.  int main(int argc, char **argv) {
2.      union {
3.           short s;
4.           char c[sizeof(short)];
5.      } un;
6.      un.s = 0x0102; //258/513
7.
8.      if (sizeof(short) == 2) {
9.          if (un.c[0] == 1 && un.c[1] == 2)
10.                    printf("big-endian\n");
11.         else if (un.c[0] == 2 && un.c[1] == 1)
12.             printf("little-endian\n");
13.         else
14.             printf("unknown\n");
15.     } else
16.               printf("sizeof(short) = %d\n", sizeof(short));
17.     exit(0);
18.  }
```

Funkcje ustalające kolejność bajtów

- #include <netinet/in.h>
- Funkcje zwracające wartość w kolejności sieciowej
  - uint16_t **htons**(uint16_t *host16bitvalue) ;*
  - uint32_t **htonl**(uint32_t *host32bitvalue) ;*
- Funkcje zwracające wartość w kolejności systemu operacyjnego
  - uint16_t **ntohs**(uint16_t *net16bitvalue) ;*
  - uint32_t **ntohl**(uint32_t *net32bitvalue) ;*

Funkcje operujące na bajtach

#include <strings.h>
void bzero(void **dest, size_t nbytes);*
void bcopy(const void **src, void *dest, size_t nbytes);*
int bcmp(const void **ptr1, const void *ptr2, size_t nbytes);*

#include <string.h>
void *memset(void **dest, int c, size_t len);*
void *memcpy(void **dest, const void *src, size_t nbytes);*
int memcmp(const void **ptr1, const void *ptr2, size_t nbytes);*

Funkcje przekształcające adresy IPv4 i IPv6
ciąg znaków ↔ postać binarna

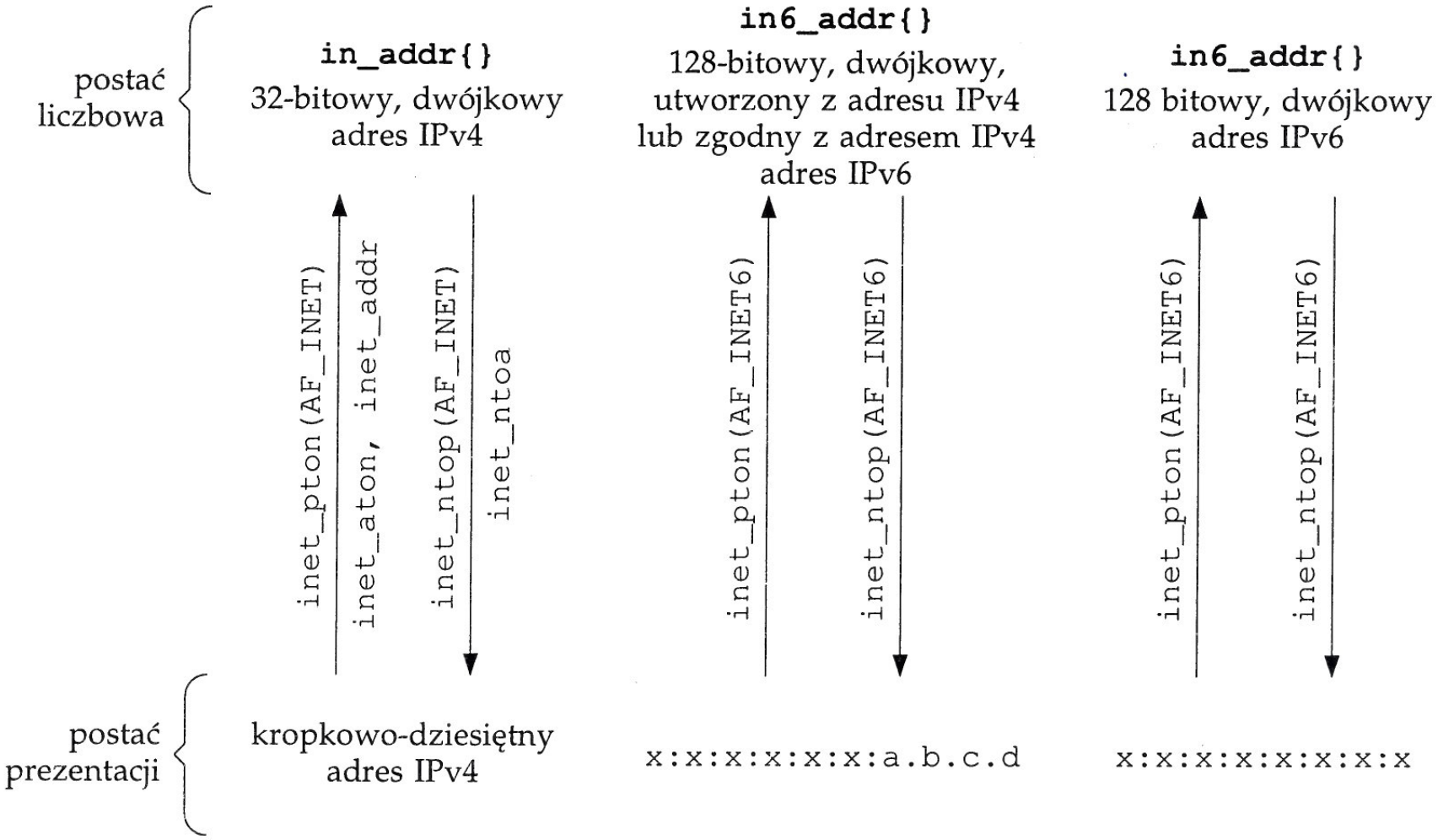
Funkcje przekształcające adresy- tylko dla IPv4 :

- int **inet_aton**(const char **strptr, struct in_addr *addrptr);*
  (zwraca 1 jeśli adres jest poprawny !!!, zero w przeciwnym wypadku)
- in_addr_t **inet_addr**(const char **strptr);*
  (zwraca stałą INADDR_NONE w razie błędu, lub 32-bitową liczbę dwójkową w sieciowej kolejności bajtów
- char ***inet_ntoa**(struct in_addr *inaddr);*

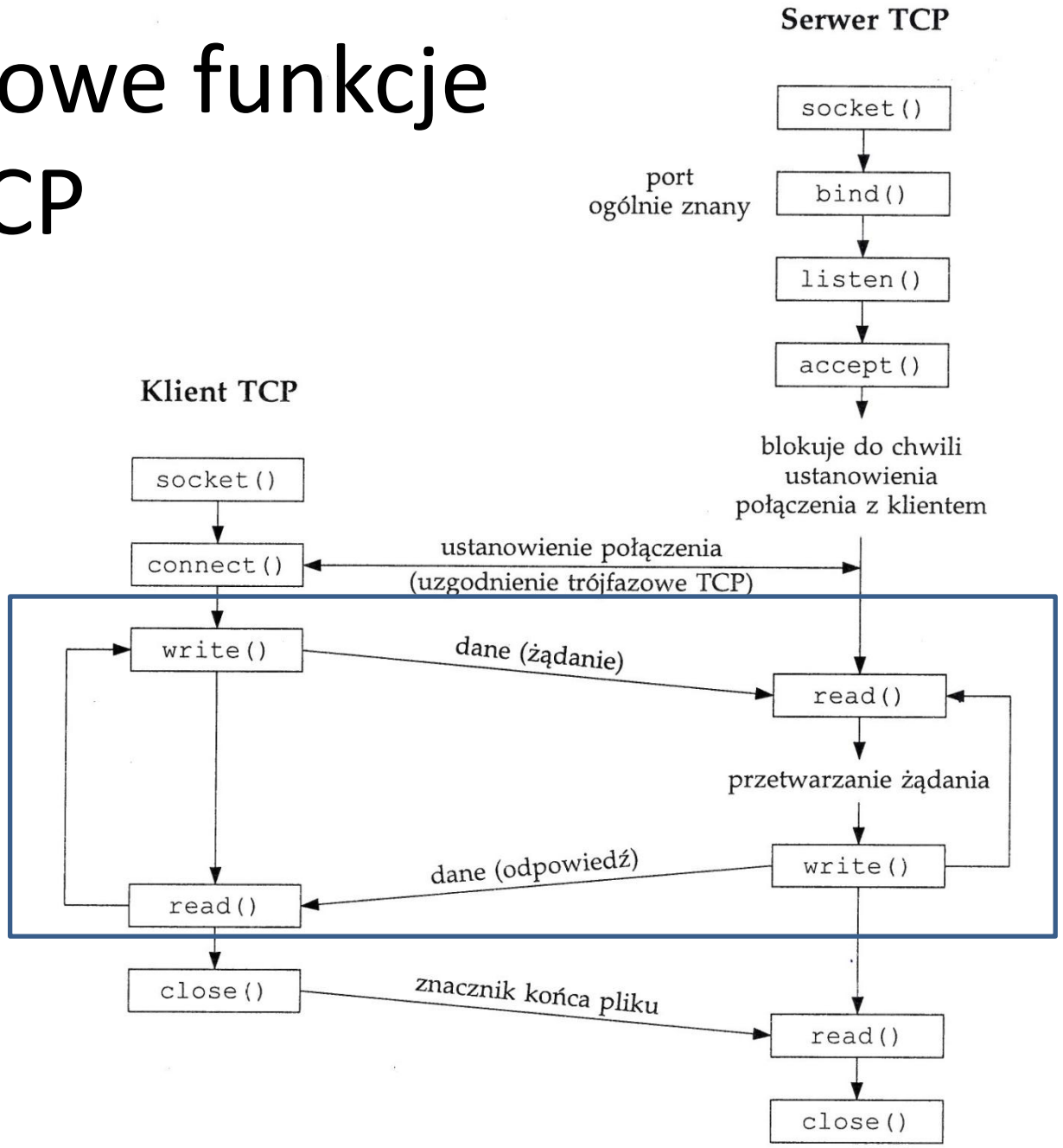
Funkcje przekształcające adresy- IPv4 i IPv6

- int **inet_pton**(int *family, const char *strptr, void *addrptr);*
  - Zwraca: 1 jeśli OK, 0 jeśli parametr wejściowy nie zawiera poprawnego adresu, -1 w wypadku innego błędu

- const char ***inet_ntop**(int *family, const void *addrptr, char *strptr, size_t len);*
  - Zwraca wskaźnik do wyniku, lub NULL w przypadku błędu. *strptr musi wskazywać na zaalokowaną pamięć.


postać liczbowa
in_addr{}
32-bitowy, dwójkowy adres IPv4
in6_addr{}
128-bitowy, dwójkowy, utworzony z adresu IPv4 lub zgodny z adresem IPv4 adres IPv6
in6_addr{}
128 bitowy, dwójkowy adres IPv6
inet_pton(AF_INET)
inet_aton, inet_addr
inet_ntop(AF_INET)
inet_ntoa
inet_pton(AF_INET6)
inet_ntop(AF_INET6)
inet_pton(AF_INET6)
inet_ntop(AF_INET6)
postać prezentacji
kropkowo-dziesiętny adres IPv4
x:x:x:x:x:x:a.b.c.d
x:x:x:x:x:x:x:x

Podstawowe funkcje gniazd TCP



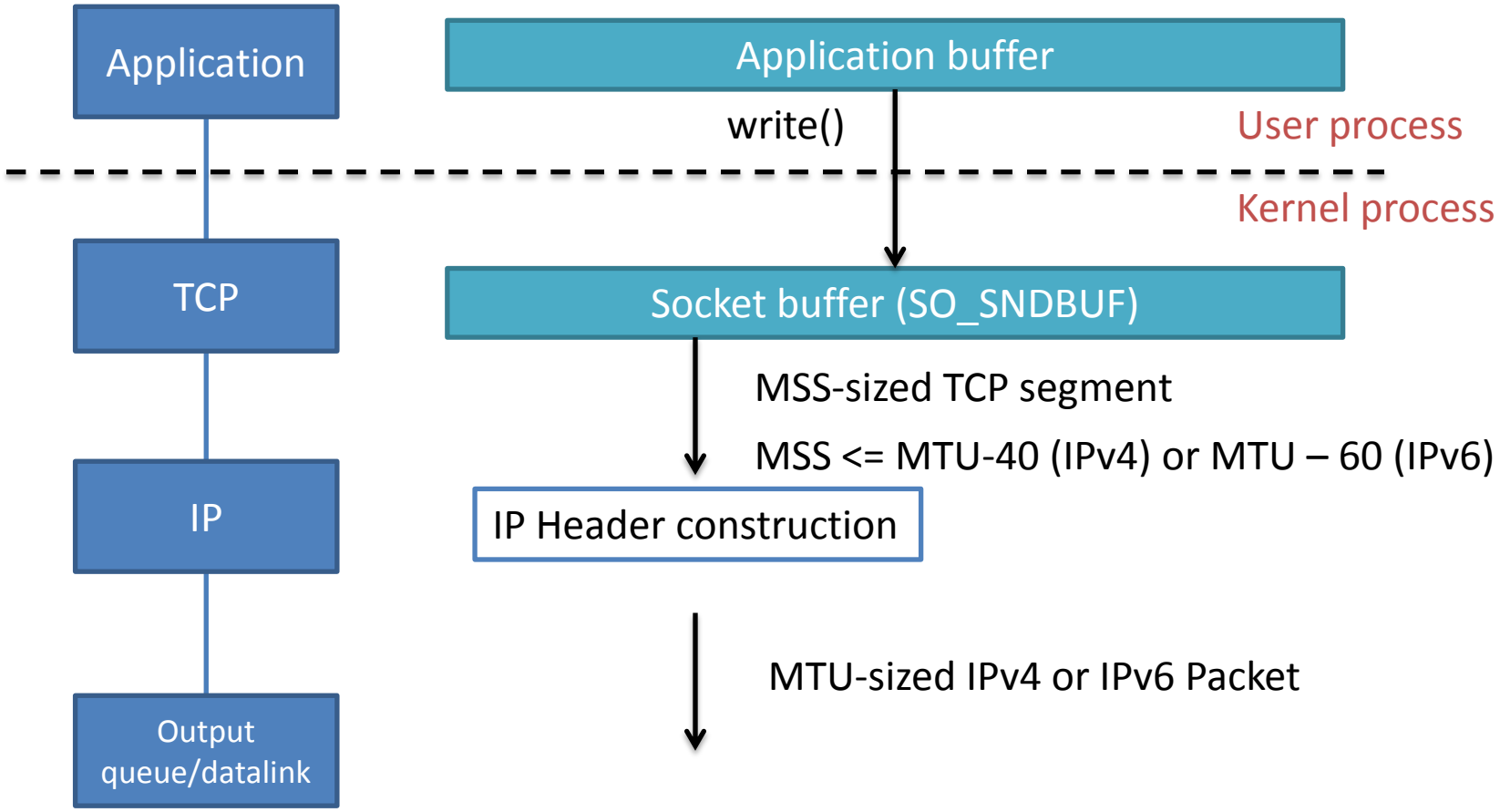
Funkcje read() i write() – operacje na gniazdach

- Funkcje **read**() i **write**() znajdują głównie zastosowanie dla gniazd połączonych
- Funkcja **read**():
  - ssize_t read(int fd, void *buf, size_t count);
  - Może zwrócić mniej oktetów niż wartość zmiennej count (to nie jest błąd)
  - Jeśli bufor odbiorczy jest pusty to normalnie funkcja read() jest blokowana
- Funkcja **write**()
  - ssize_t write(int fd, const void *buf, size_t count);
  - może zapisać mniej oktetów niż wartość zmiennej count (to nie jest błąd)
  - Należy sprawdzać wynik funkcji write() i w pętli wywoływać tyle razy, ile jest wymagane do zapisania danych

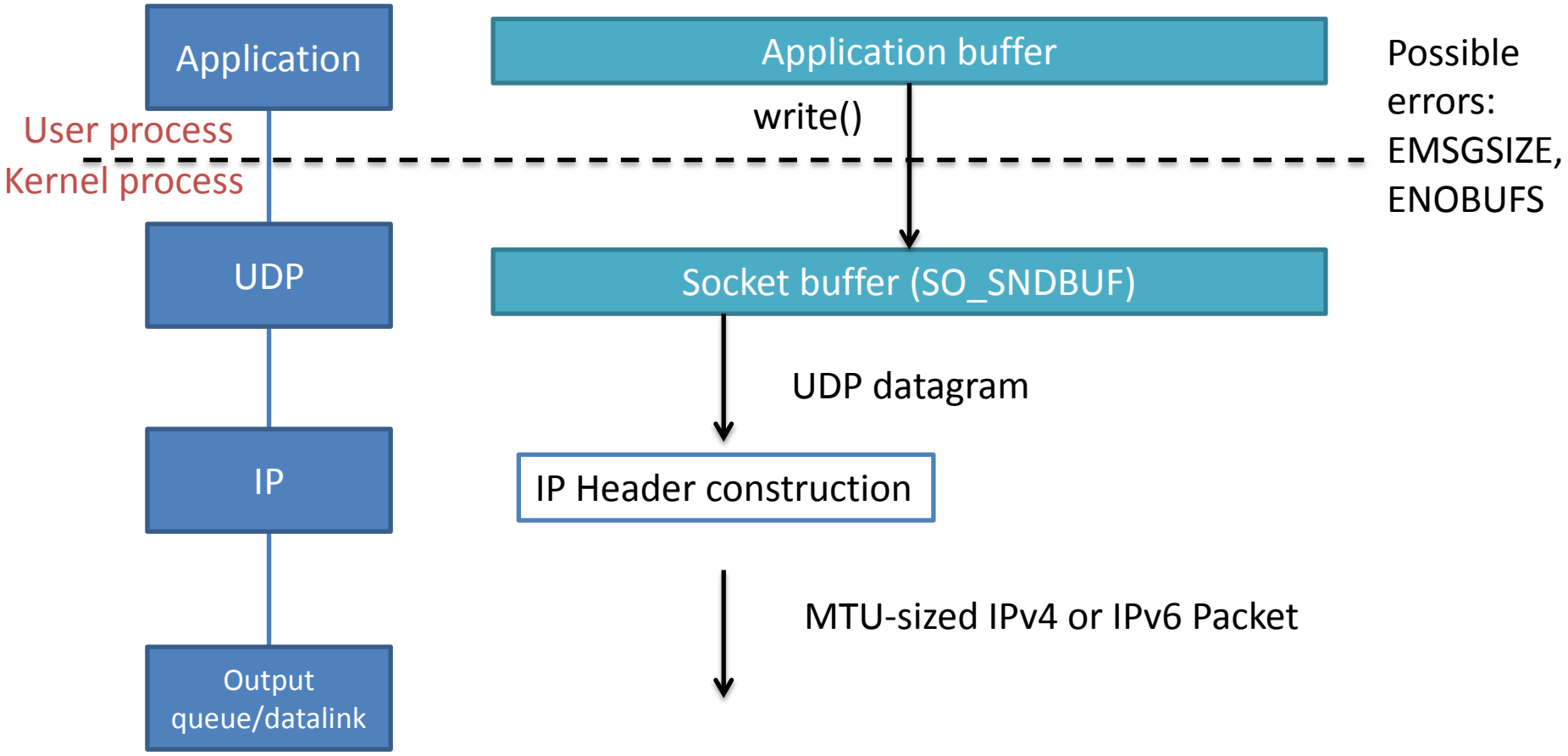
Opóźnienia w funkcjach we/wy dla gniazd **strumieniowych**:

- Funkcje odbierające zwracają sterowanie jeśli:
  - **Nadejdą dane z ustawionym bitem PUSH.**
  - **Bufor odbiorczy gniazda jest pełny.**
  - **Upłynęło 0.5 sekundy od odebrania ostatnich danych.**

TCP – transmisja segmentu



UDP – transmisja segmentu



writen()
Write "n" bytes to a descriptor

```
1. ssize_t
2. writen(int fd, const void *vptr, size_t n)
3. {
4.      size_t          nleft;
5.      ssize_t         nwritten;
6.      const char      *ptr;
7.      ptr = vptr;
8.      nleft = n;
9.      while (nleft > 0) {
10.         if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
11.             if (nwritten < 0 && errno == EINTR)
12.                     nwritten = 0;    /* and call write() again */
13.             else
14.                     return(-1);      /* error */
15.          }
16.             nleft -= nwritten;
17.             ptr   += nwritten;
18.     }
19.     return(n);
20.}
```

readn()

```
1.  ssize_t                 /* Read "n" bytes from a descriptor. */
2.  readn(int fd, void *vptr, size_t n){
3.          size_t          nleft;
4.          ssize_t         nread;
5.          char            *ptr;
6.          ptr = vptr;     nleft = n;
7.          while (nleft > 0) {
8.                          if ( (nread = read(fd, ptr, nleft)) < 0) {
9.                                  if (errno == EINTR)
10.                                         nread = 0;                  /* and call read() again */
11.                                 else
12.                                         return(-1);
13.                         } else if (nread == 0)
14.                                 break;                              /* EOF */
15.                         nleft -= nread;
16.                         ptr   += nread;
17.             }
18.     return(n - nleft);                          /* return >= 0 */
19. }
```

Funkcje recv() i send() – operacje na gniazdach

- Funkcje **recv**() i **send**() znajdują zastosowanie głównie dla gniazd połączonych
- Funkcja **recv**():
  - `ssize_t recv(int fd, void *buf, size_t count, int flags);`
  - Może zwrócić mniej oktetów niż wartość zmiennej count (to nie jest błąd)
  - Domyślnie jeśli bufor odbiorczy jest pusty, to funkcja jest blokowana
- Funkcja **send**():
  - `ssize_t send(int fd, const void *buf, size_t count, int flags);`
  - może zapisać mniej oktetów niż wartość zmiennej count (to nie jest błąd)
  - Należy sprawdzać wynik funkcji send() i w pętli wywoływać tyle razy, ile jest wymagane do zapisania danych

Flagi funkcji send() i recv()

- MSG_DONTROUTE – pakiet przeznaczony dla sieci lokalnej
- MSG_OOB – dane pozapasmowe
- MSG_DONTWAIT – operacja nieblokowalna (bez potrzeby ustawienia opcji gniazda)
- MSG_PEEK – podgląd nadchodzącego komunikatu
- MSG_WAITALL – czytanie wszystkich danych, zgodnie z parametrem count

Funkcja isfdtype()

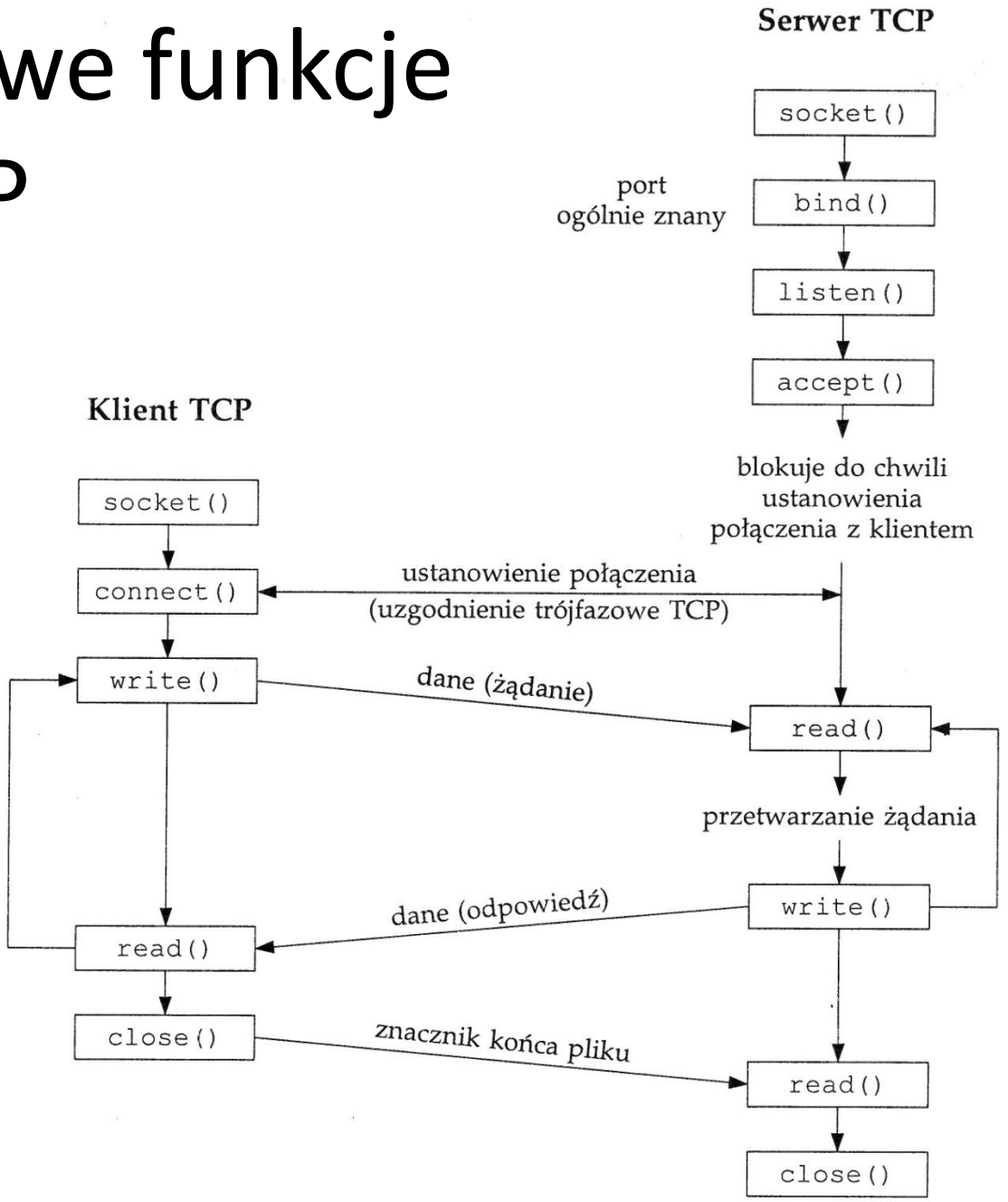
#include <sys/stat.h>

#include <sys/socket.h>

int isfdtype(int *fd*, **int** *fdtype*);

fdtype == S_IFSOCK dla gniazd

Podstawowe funkcje gniazd TCP



SOCKET() - podstawowa funkcja tworząca gniazdo

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
int socket (int family, int type, int
  protocol);
```

- family – rodzina protokołów
- type – typ gniazda (strumieniowy, datagramowy, skwencyjny)
- protocol – typ protokołu (IPPROTO_TCP/UDP/ SCTP)

SOCKET() - Family

Name	Purpose	Man page
AF_UNIX, AF_LOCAL	Local communication	unix(7)
AF_INET	IPv4 Internet protocols	ip(7)
AF_INET6	IPv6 Internet protocols	ipv6(7)
AF_NETLINK	Kernel user interface device	netlink(7)
AF_IPX	IPX - Novell protocols	
AF_X25	ITU-T X.25 / ISO-8208 protocol	x25(7)
AF_ATMPVC	Access to raw ATM PVCs	
AF_APPLETALK	Appletalk	ddp(7)
AF_PACKET	Low level packet interface	packet(7)
AF_AX25	Amateur radio AX.25 protocol	

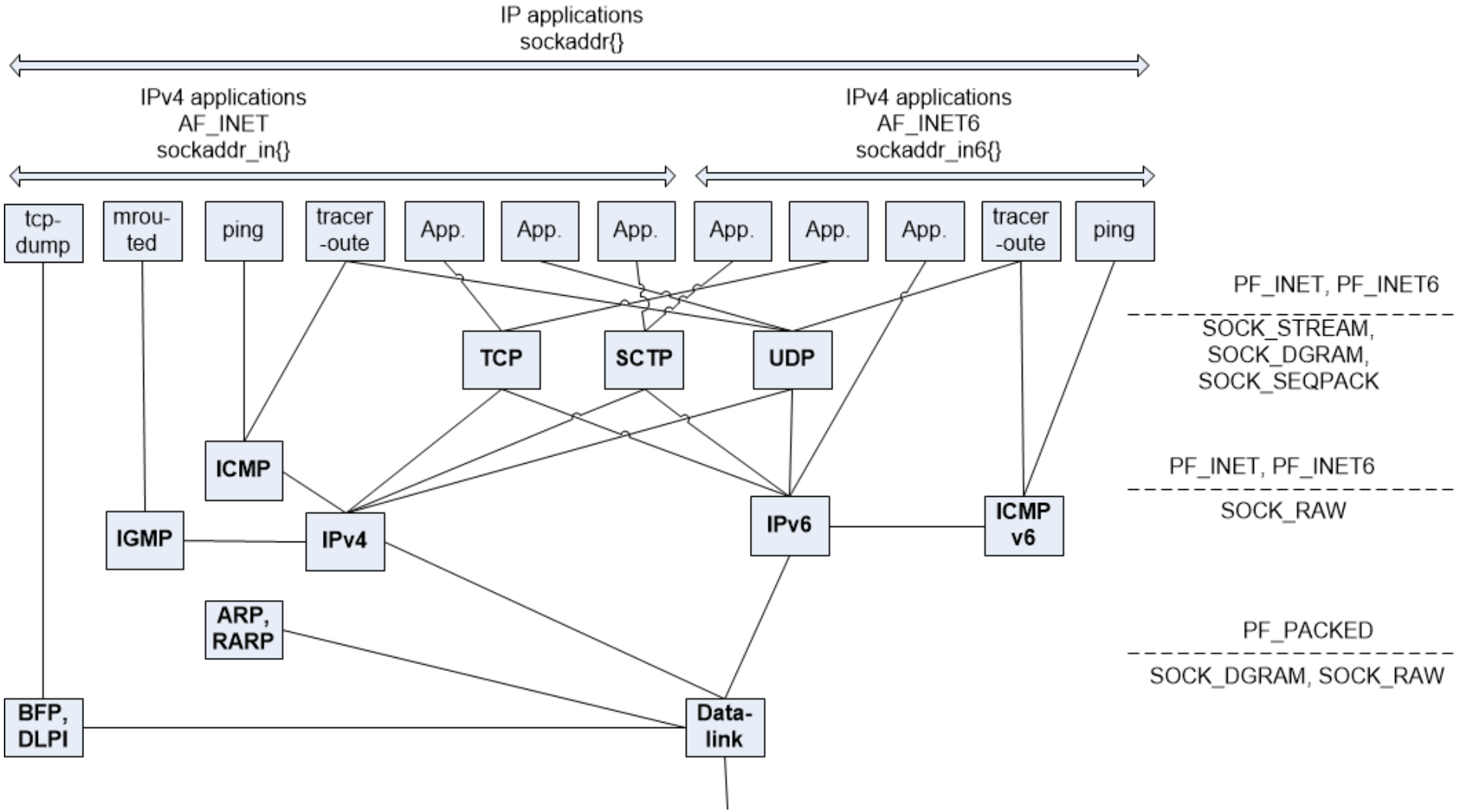
SOCKET() - Type

Typ gniazda	Opis
SOCK_STREAM	Przesyłanie danych traktowanego jak coś ciągłego, jako strumień, przychodzące dane nie muszą mieć zawsze tej samej wielkości; używany protokół to; zapewniona integralność danych
SOCK_DGRAM	Usługa bezpołączeniowa, musi czytać całą wiadomość za jednym razem; ma ona z góry określony rozmiar.
SOCK_ SEQPACKET	Usługa taka jak dla **SOCK_DGRAM** ale dodatkowo z zapewnieniem braku strat jak dla **SOCK_STREAM**
SOCK_RAW	Dostęp do warstwy sieciowej

Dodatkowe opcje/flagi gniazda dla systemu Linux >= 2.6.27:

SOCK_NONBLOCK - ustawia gniazdo w tryb nieblokujący

SOCK_CLOEXEC - powoduje zamknięcie deskryptora gniazda w przypadku wywołania funkcji z rodziny exec()


IP applications
sockaddr{}
IPv4 applications
AF_INET
sockaddr_in{}
IPv4 applications
AF_INET6
sockaddr_in6{}
tcp-dump
mrou-ted
ping
tracer -oute
App.
App.
App.
App.
App.
App.
tracer -oute
ping
TCP
SCTP
UDP
ICMP
IGMP
IPv4
IPv6
ICMP v6
ARP, RARP
BFP, DLPI
Data-link
PF_INET, PF_INET6
SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACK
PF_INET, PF_INET6
SOCK_RAW
PF_PACKED
SOCK_DGRAM, SOCK_RAW

CONNECT()

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

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

- Ustanawia połączenie ze stacją zdalną
  - Dla TCP inicjuje uzgadnianie trójfazowe
  - Dla UDP określa, że pakiety będą wysyłane i dobierane tylko dla danego adresu
- Przekazuje: 0, jeśli wszystko w porządku; -1 jeśli wystąpił błąd i ustawia zmienną **errno** na odpowiedni kod błędu
- Najistotniejsze kody błędów dla TCP:
  - ECONNREFUSED - No-one listening on the remote address
  - ENETUNREACH - Network is unreachable.
  - EHOSTUNREACH – Host is unreachable.
  - ETIMEDOUT - Timeout while attempting connection. The server may be too busy to accept new connections.
- **Jeśli wystąpi błąd** (nie udało się osiągnąć stanu ESTABLISHED), to aby ponownie z sukcesem wywołać funkcję connect() należy **zamknąć gniazdo**

BIND()

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

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

- Przypisanie adresu i portu do gniazda – proces czeka na połączenie na adresie i porcie, do którego się dowiąże

Process specifies		Result
IP address	port	
Wildcard	0	Kernel chooses IP address and port
Wildcard	nonzero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP, kernel chooses port
Local IP address	nonzero	Process specifies IP address and port

```
struct sockaddr_in servaddr;
servaddr.sin_addr.s_addr = htonl (INADDR_ANY); /* wildcard */

struct sockaddr_in6 servaddr;
servaddr.sin6_addr.s_addr = in6_addr_any; /* wildcard */
```

LISTEN()

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

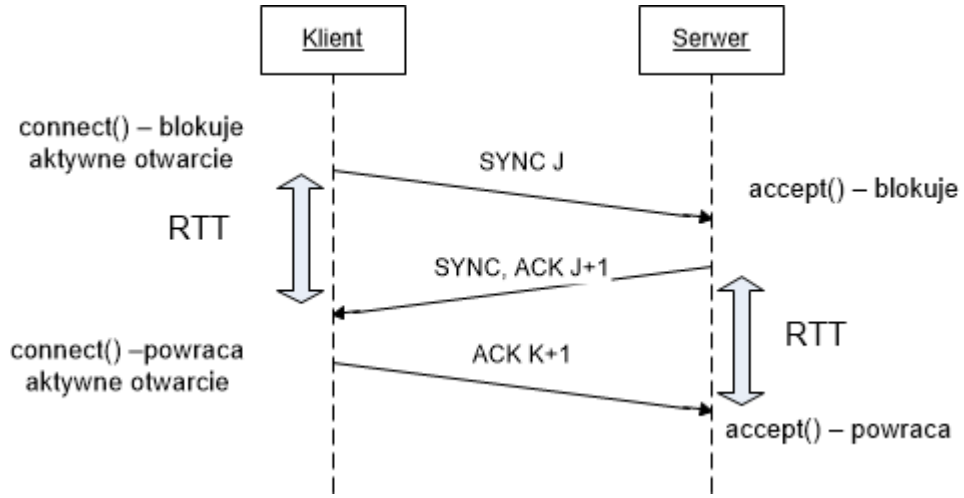
- Funkcja listen() zmienia stan gniazda z gniazda niepołączonego na gniazdo pasywne, które oczekuje na połączenia (jądro powinno akceptować przychodzące żądania nawiązania połączenia dla tego gniazda). Dla TCP gniazdo przechodzi ze stanu CLOSED do LISTEN.
- Drugi argument określa ile żądań połączeń może zostać przez jądro systemu ustawionych w kolejce do danego gniazda.

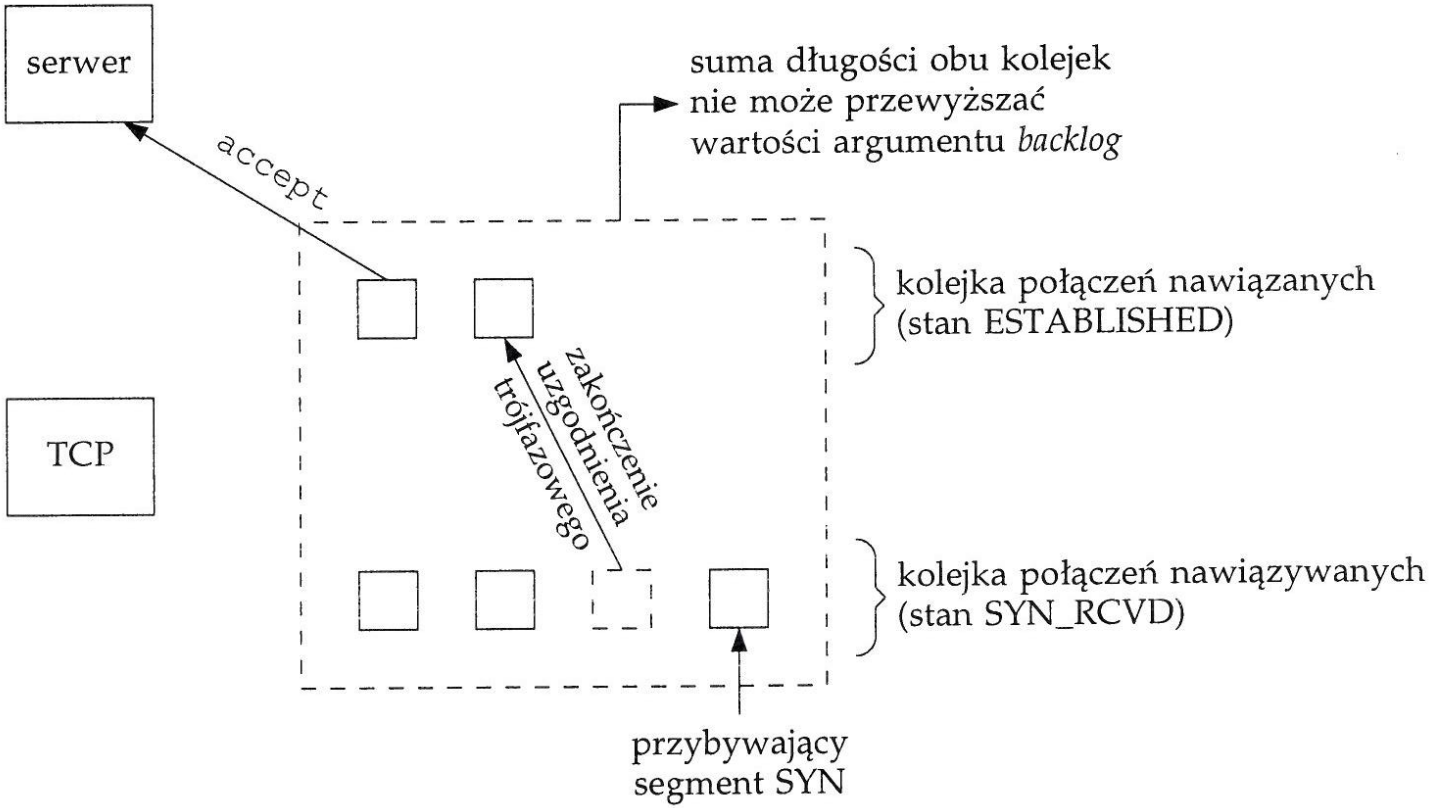
LISTEN()

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

- W systemie LINUX **backlog:**
  - Długość kolejki dla ustanowionych połączeń (stan CONNECTED)
  - Długość kolejki dla połączeń nawiązywanych może być ustawiona za pomocą zmiennej: /proc/sys/net/ipv4/tcp_max_syn_backlog (dom. 256)
  - Backlog nie może być większy od zmiennej: /proc/sys/net/core/somaxconn
  - Jeśli mechanizm **Syncookies** (do blokowania ataków typu **Syn flood**) jest uaktywniony to ta zmienna nie ma znaczenia.

LISTEN()


Klient
Serwer
connect() – blokuje
aktywne otwarcie
SYNC J
accept() – blokuje
RTT
SYNC, ACK J+1
RTT
connect() –powraca
aktywne otwarcie
ACK K+1
accept() – powraca


serwer
accept
suma długości obu kolejek
nie może przewyższać
wartości argumentu *backlog*
kolejka połączeń nawiązanych
(stan ESTABLISHED)
TCP
zakończenie
uzgodnienia
trójfazowego
kolejka połączeń nawiązywanych
(stan SYN_RCVD)
przybywający
segment SYN

ACCEPT()

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr,
           socklen_t *addrlen);
int accept4(int sockfd, struct sockaddr *addr,
           socklen_t *addrlen, int flags);
```

- Zwraca:
  - gniazdo połączone dla zaakceptowanego połączenia
  - adres drugiej strony połączenia
- Flagi:
  - **SOCK_NONBLOCK**
  - **SOCK_CLOEXEC**

Funkcja close()

```
#include <unistd.h>
int close(int fd);
```

- Zamyka deskryptor – nie można go już użyć w programie
- System próbuje wysłać pozostałe dane z kolejek
- **Jeśli licznik odniesień do gniazda = 0, to system wywołuje procedurę zakończenia połączenia (wysyła segment FIN)**
- W celu natychmiastowego zerwania połączenia wywołuje się funkcję (wysyła segment FIN)

```
int shutdown(int sockfd, int how);
```

Funkcja shutdown()

```
#include <sys/socket.h>
int shutdown(int sockfd, int how);
```

- Zamyka częściowo lub całkowicie połączenie full-duplex dla gniazda SOCK_STREAM (SOCK_SEQPACK) niezależnie od wartości licznika odniesień dla gniazda
- W zależności od parametru **how**:
  - **SHUT_RD** – gniazdo zamykane do czytania
  - **SHUT_WR** – gniazdo zamykane do pisania (wysyłany segment FIN)
  - **SHUT_RDWR** – gniazdo zamykane do pisania i czytania (wysyłany segment FIN)
- W przypadku poprawnego zakończenia zwraca 0, w przypadku błędu -1 i ustawia zmienną errno

shutdown() summary

Function	Description
shutdown(), SHUT_RD	No more receives can be issued on socket; process can still send on socket; socket receive buffer discarded; any further data received is discarded by TCP; no effect on socket send buffer.
shutdown(), SHUT_WR	No more sends can be issued on socket; process can still receive on socket; contents of socket send buffer send to other end, followed by normal connection termination (FIN); no effect on socket receive buffer;
shutdown(), SHUT_RDWR	No more receives or sends can be issued on socket; contents of socket send buffer send to other end. Normal connection termination (FIN) sent following data in send buffer and socket receive buffer discarded;

Funkcje getsockname() i getpeername()

- Funkcje przekazują adresy gniazd połączonych:
  - int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
    - rodzina, adres i port **lokalny** gniazda
  - int getpeername(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
    - rodzina, adres i port **zdalny** gniazda
- Różne przypadki użycia: gdy proces nie precyzuje adresu i portu w funkcji bind(), po wywołaniu funkcji exec(), …

Funkcje systemu UNIX FORK() i EXEC_()

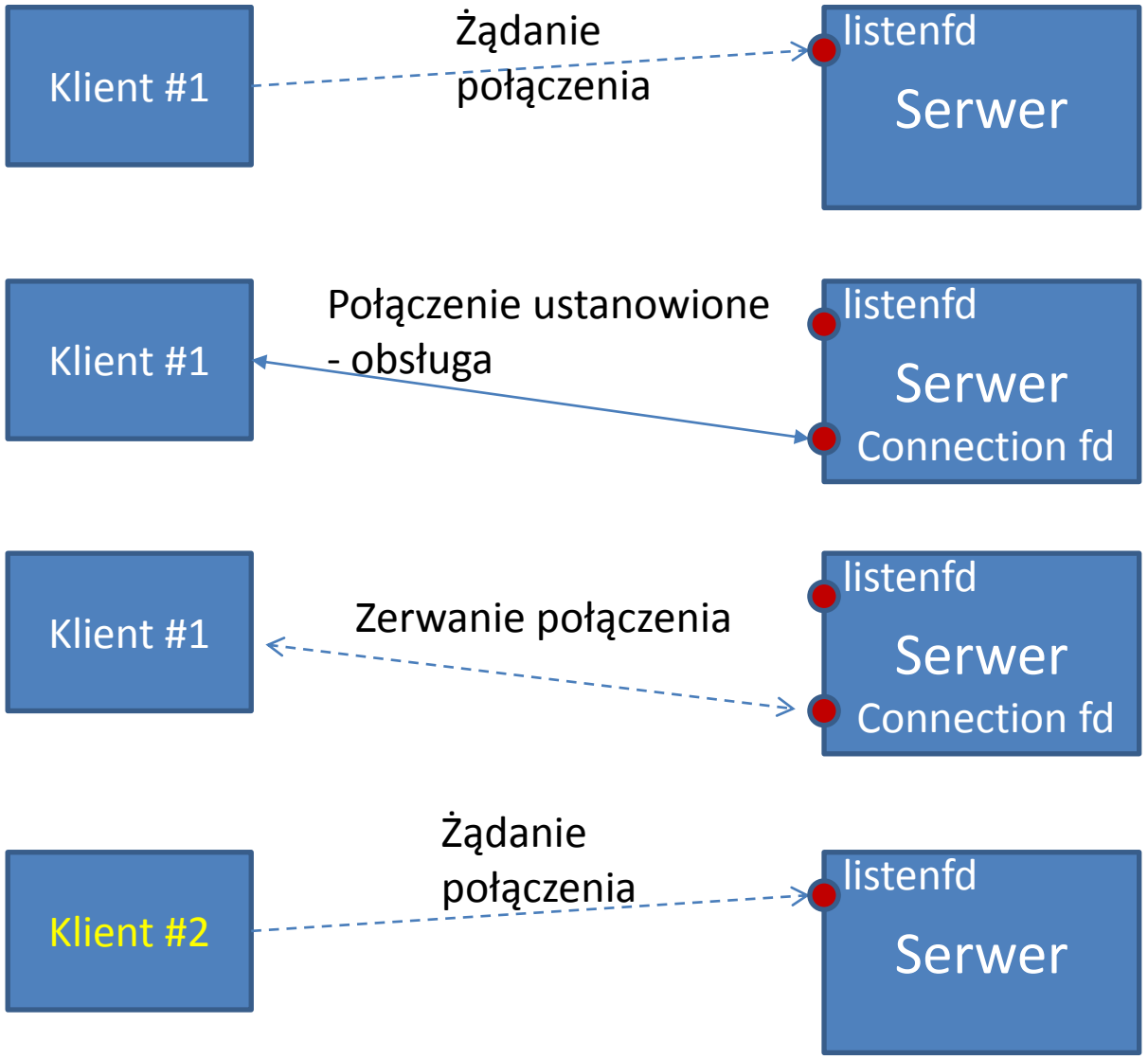
- Wymagane dla tworzenia serwerów współbieżnych
- **fork()** – rozwidlenie procesu
- exec() – podmiana kodu procesu
- pipe() – komunikacja przez strumienie pomiędzy spokrewnionymi procesami

Typy serwerów: iteracyjne i współbieżne

Typy serwerów

- Dwa typy serwerów:
  - Iteracyjne
  - Współbieżne
- Serwer iteracyjny – przykład podany wcześniej – klienci obsługiwani w kolejności zgłoszeń
- Serwer współbieżny – klienci obsługiwani w tym samym czasie przez zbiór procesów, z użyciem wątków lub multipleksacji gniazd

Serwer Iteracyjny – fazy obsługi



Prosty program typu klient-serwer

- Klient
  - Przygotowuje struktury adresowe i inicjuje gniazdo
  - Łączy się z serwerem
  - Odbiera informację
  - Wyświetla informację na ekranie
  - Koniec
- Serwer
  - Przygotowuje struktury adresowe i inicjuje gniazdo
  - Nasłuchuje
  - Akceptuje połączenie od klienta
  - Wysyła informacje
  - Kończy połączenie
  - Nasłuchuje ….

Serwer Iteracyjny - przykład

```
1.  int
2.  main(int argc, char **argv)
3.  {
4.   ……..
5.      struct sockaddr_in6     servaddr, cliaddr;

6.      if ( (listenfd = socket (AF_INET6, SOCK_STREAM, 0)) < 0){
7.          fprintf(stderr,"socket error : %s\n", strerror(errno));
8.          return 1;
9.      }

10.  ……..

11.  if ( bind( listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0){
12.         fprintf(stderr,"bind error : %s\n", strerror(errno));
13.         return 1;
14.  }

21.  if ( listen (listenfd, LISTENQ) < 0){
22.         fprintf(stderr,"listen error : %s\n", strerror(errno));
23.         return 1;
24.  }
```

```
21.
22.  for ( ; ; ) {
23.         len = sizeof(cliaddr);
24.         if ( (connfd = accept(listenfd, (struct sockaddr *) &cliaddr, &len)) < 0){
25.             fprintf(stderr,"accept error : %s\n", strerror(errno));
26.             continue;
27.         }

28.         bzero(str, sizeof(str));
29.         inet_ntop(AF_INET6, (struct sockaddr  *) &cliaddr.sin6_addr,  str, sizeof(str));
30.         printf("Connection from %s\n", str);

31.         ticks = time(NULL);
32.         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
33.         if( write(connfd, buff, strlen(buff))< 0 )
34.             fprintf(stderr,"write error : %s\n", strerror(errno));
35.         close(connfd);
36.     }
37.  }
```

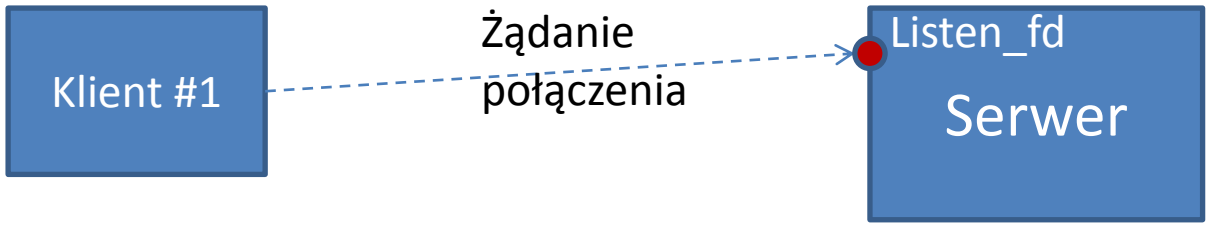
daytimetcpcliv6.c

Klient

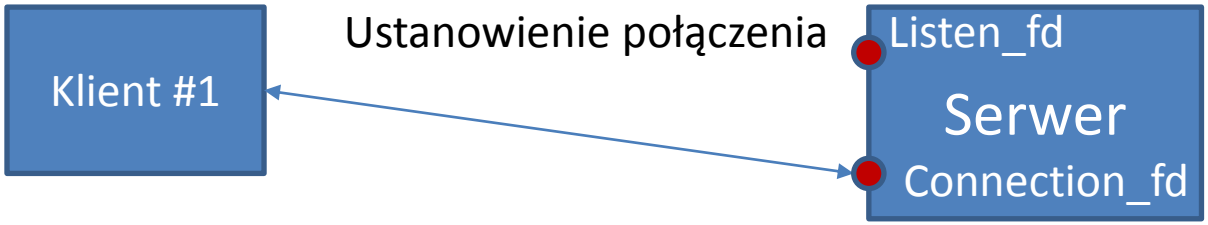
```
1.  int
2.  main(int argc, char **argv)
3.  {
4.      int                     sockfd, n;
5.      struct sockaddr_in6     servaddr;
6.      char                    recvline[MAXLINE + 1];

7.      if (argc != 2){
8.          fprintf(stderr, "usage: a.out <IPaddress> : %s\n",
        strerror(errno));
9.          return 1;
10.     }
11.     if ( (sockfd = socket(AF_INET6, SOCK_STREAM, 0)) <
        0){
12.         fprintf(stderr,"socket error : %s\n",
        strerror(errno));
13.         return 1;
14.     }

15.     bzero(&servaddr, sizeof(servaddr));
16.     servaddr.sin6_family = AF_INET6;
17.     servaddr.sin6_port   = htons(13);      /* daytime server
        */
18.     if (inet_pton(AF_INET6, argv[1], &servaddr.sin6_addr)
        <= 0){
19.         fprintf(stderr,"inet_pton error for %s : %s \n",
        argv[1], strerror(errno));
20.         return 1;
21.     }

22.     if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) <
        0){
23.         fprintf(stderr,"connect error : %s \n",
        strerror(errno));
24.         return 1;
25.     }

26.     while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
27.         recvline[n] = 0;       /* null terminate */
28.         if (fputs(recvline, stdout) == EOF){
29.             fprintf(stderr,"fputs error : %s\n",
        strerror(errno));
30.             return 1;
31.         }
32.     }
33.     if (n < 0)
34.         fprintf(stderr,"read error : %s\n",
        strerror(errno));

35.     fprintf(stderr,"OK\n");
36. //   flush(stderr);

37.     exit(0);
38. }
```

Serwer współbieżny z użyciem funkcji fork() – zestawienie połączenia (1/3)

- Krok #1

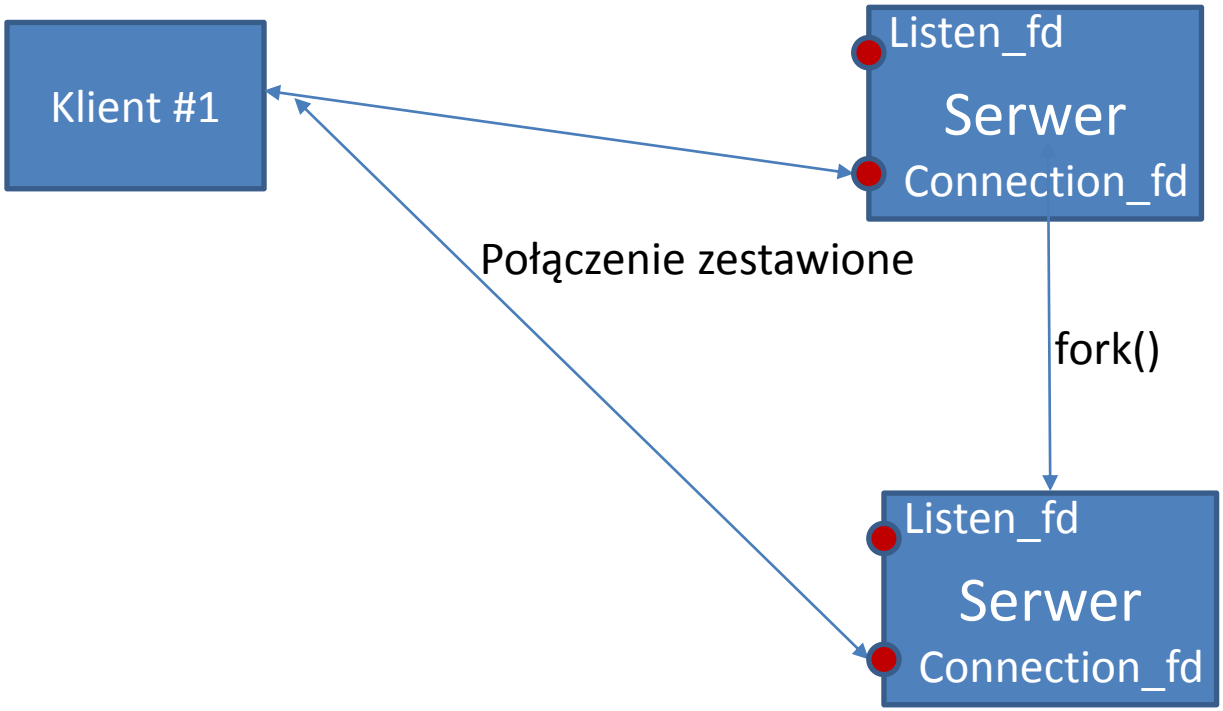


- Krok #2



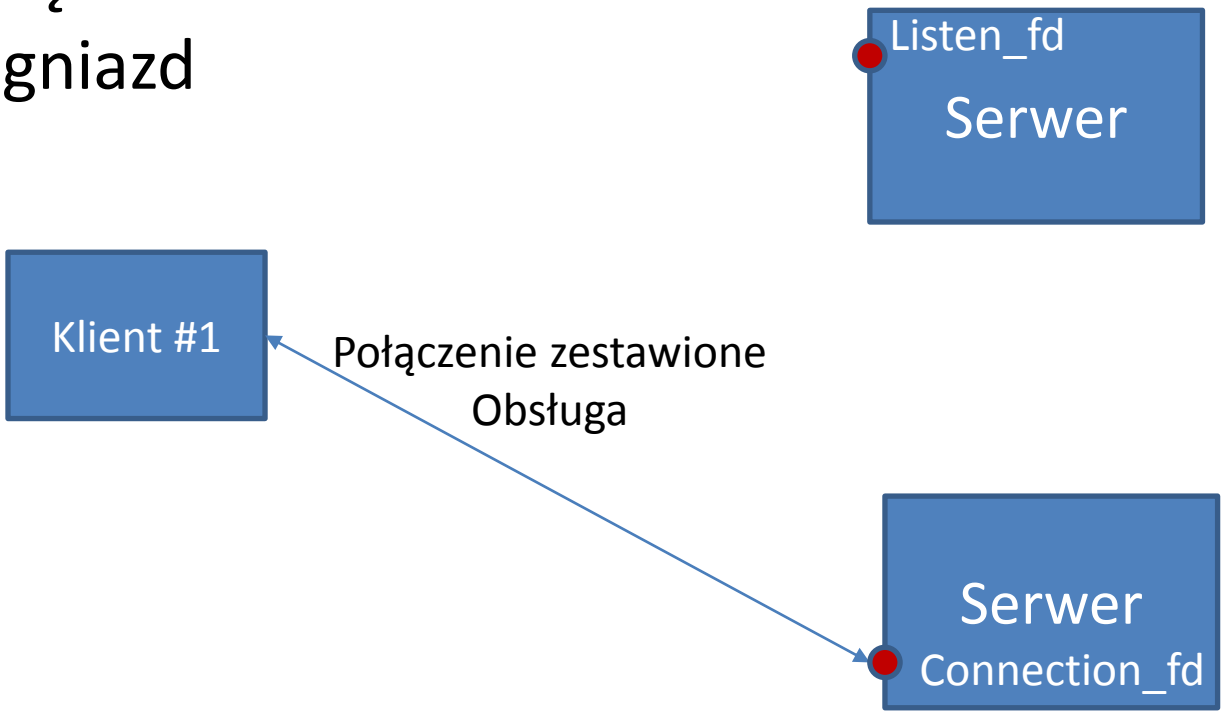
Serwer współbieżny z użyciem funkcji fork() – zestawienie połączenia (2/3)

- Krok #3



Serwer współbieżny z użyciem funkcji fork() – zestawienie połączenia (3/3)

- Krok #4 – zamknięcie niepotrzebnych gniazd



Serwer współbieżny z użyciem funkcji fork() - przykład

```
21.  socket(…); bind(…); listen(…);
22.  signal(SIGCHLD, sig_chld);
23.  for ( ; ; ) {
24.       if ( (connfd = accept(listenfd, (struct sockaddr *) &cliaddr, &len)) < 0){
25.            Obsługa przerwania();
26.       }
27.       if( (pid=fork()) == 0 ){   /*dziecko*/
28.           close(listenfd);
29.            OBSŁUGA();
30.            close(connfd);
31.            exit(0);
32.       else { /*rodzic*/
33.            close(connfd);
34.       }
35.  }
```

echo_tcpservv6_cmd.c

Serwer współbieżny z użyciem funkcji fork() – porównanie z serwerem iteracyjnym

Serwer iteracyjny

```
21.  for ( ; ; ) {
22.          len = sizeof(cliaddr);
23.          if ( (connfd = accept(listenfd, (struct sockaddr *)
&cliaddr, &len)) < 0){
24.                  fprintf(stderr,"accept error : %s\n",
strerror(errno));
25.                  continue;
26.          }

27.          bzero(str, sizeof(str));
28.          inet_ntop(AF_INET6, (struct sockaddr  *)
&cliaddr.sin6_addr,  str, sizeof(str));
29.          printf("Connection from %s\n", str);

30.          ticks = time(NULL);
31.          snprintf(buff, sizeof(buff), "%.24s\r\n",
ctime(&ticks));
32.          if( write(connfd, buff, strlen(buff))< 0 )
33.                  fprintf(stderr,"write error : %s\n",
strerror(errno));
34.              close(connfd);
35.        } //for
```

Serwer współbieżny

```
21.  signal(SIGCHLD, sig_chld);
22.  for ( ; ; ) {
23.          len = sizeof(cliaddr);
24.          if ( (connfd = accept(listenfd, (struct sockaddr *) &cliaddr,
&len)) < 0){
25.          fprintf(stderr,"accept error : %s\n", strerror(errno));
26.                  continue;
27.           }
28.   if( (pid=fork()) == 0 ){ //potomek
29.              close(listenfd);
30.              bzero(str, sizeof(str));
31.              inet_ntop(AF_INET6, (struct sockaddr  *)
&cliaddr.sin6_addr,  str, sizeof(str));
32.              printf("Connection from %s\n", str);

33.              ticks = time(NULL);
34.              snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
35.              if( write(connfd, buff, strlen(buff))< 0 )
36.                fprintf(stderr,"write error : %s\n",  strerror(errno));
37.              close(connfd);
38.              exit(0);
39.        }else{ close(connfd); } //rodzic
40.         } //for
```

Trzy zasady obowiązujące przy pisaniu serwera sieciowego z użyciem funkcji fork():

- Jeśli tworzymy procesy potomne to w procesie macierzystym musimy obsłużyć sygnał **SIGCHLD**
- Jeśli przechwytujemy sygnały, to musimy obsługiwać zdarzenia spowodowane przerwanym działaniem funkcji systemowych
- Aby nie pozostawiać po sobie procesów w stanie „zombie”, musimy poprawnie zdefiniować procedurę obsługi sygnału **SIGCHILD**, używając w niej funkcji waitpid() lub jawnie zignorować ten sygnał

Obsługa sygnałów

Domyślna reakcja na otrzymanie sygnału przez proces w zależności od otrzymanego sygnału:

- **Term** – zakończenie procesu.
- **Ign** – zignorowanie sygnału.
- **Core** – zakończenie procesu i zrzut pamięci (core dump).
- **Stop** – zatrzymanie procesu.
- **Cont** – wznowienie procesu, jeśli jest zatrzymany.

Proces może zmienić domyślną reakcję na otrzymanie sygnału za pomocą funkcji sigaction(2) or signal(2) z wyjątkiem sygnałów SIGKILL i SIGSTOP.

Funkcja signal() – przechwytywanie sygnałów

- **signal()** – przestarzała, może zachowywać się różnie na różnych systemach UNIX (przechwytywane sygnały przerywają działanie funkcji systemowych i mogą lub nie do nich powracać)

```
#include <signal.h>
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum,
   sighadler_t handler);
```

Funkcja sigaction() – przechwytywanie sygnałów

- **sigaction()** – funkcja zgodna ze standardem POSIX, możliwość ustawienia czy po obsłużeniu przerwania funkcje systemowe mają być kontynuowane czy przerywane (flaga **SA_RESTART**)

```
1. #include <signal.h>
2. int sigaction(int signum, const
   struct sigaction *act, struct
   sigaction *oldact);
```

Struct sigaction

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void
    *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restore)(void);
};
```

- flaga **SA_SIGINFO** determinuje, która funkcja obsługi sygnału zostanie wywołana: **`sa_handler`** czy **`sa_sigaction`**
- **`sa_mask`** określa, jakie sygnały powinny być blokowane przy wywołaniu funkcji obsługującej sygnał

Sigaction - flagi

- SA_NOCLDSTOP – dla SIGCHLD blokuj informację o zatrzymaniu procesu potomnego
- **SA_NOCLDWAIT** – nie transformuj potomków w procesy „zombie”
- SA_SIGINFO – określa ile parametrów ma funkcja obsługi sygnałów
- **SA_RESTART** – restart funkcji przerwanych przez sygnał
- SA_RESETHAND – przywróć domyślną obsługę sygnału po wywołaniu funkcji obsługi sygnału
- SA_NODEFER – nie blokuj otrzymywania sygnału z funkcji obsługi sygnału

Obsługa sygnałów dla funkcji sieciowych

- Funkcje blokujące po ustawieniu flagi **SA_RESTART** są restartowane (kontynuowane) – w przeciwnym wypadku są przerywane i zwracają błąd EINTR:
  - read(), readv(), write(), writev(),
  - wait(), wait3(), wait4(), waitid(), and waitpid().
  - funkcje gniazd: accept(), connect(), recv(), recvfrom(), recvmsg(), send(), sendto(), and sendmsg(), jeśli nie ustawiono na gnieździe czasu oczekiwania (patrz następny slajd).

Obsługa sygnałów dla funkcji sieciowych

- Dla systemu LINUX następujące funkcje gniazd nigdy nie są restartowane po przerwaniu działania przez sygnał, nawet w przypadku ustawienia flagi **SA_RESTART** jeśli ustanowiono **czas oczekiwania (timeout)** na gnieździe za pomocą funkcji **setsockopt**(); poniższe funkcje zawsze zwracają błąd EINTR, kiedy ich **działanie jest przerywane** przez obsługę sygnału:
  - **accept**(), **recv**(), **recvfrom**(), and **recvmsg**(), jeśli ustawiono czas oczekiwania na pobranie danych (opcja **SO_RCVTIMEO**);
  - **connect**(), **send(**), **sendto**(), and **sendmsg**(), jeśli ustawiono czas oczekiwania na wysłanie danych (opcja **SO_SNDTIMEO**).

Funkcje wait() i waitpid()

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- Umożliwiają uzyskanie informacji o zmianie stanu procesu potomka:
  - Zamknięcie procesu
  - Zatrzymanie procesu
  - Wznowienie procesu
- Są wymagane do zwolnienia zasobów przez zakończone procesy potomne
- Jeśli proces macierzysty nie użyje tych funkcji, wtedy zakończone procesy potomne pozostają jako „zombie” i zajmują zasoby systemowe

Funkcja waitpid()

pid_t waitpid(pid_t **pid**, int *status, int options);

- Parametr **pid**:
  - **< -1** – oczekiwanie na dowolny proces, którego grupa jest równa wartości bezwzględnej z **pid**
  - **-1** – oczekiwanie na dowolny proces
  - **0** – oczekiwanie na dowolny proces, którego grupa jest równa grupie procesu oczekującego
  - **> 0** – oczekiwanie na proces o podanym **pid**

Funkcja waitpid()

pid_t waitpid(pid_t pid, int ***status**, int **options**);

- Parametr **options**:
  - **WNOHANG** – jeśli nie ma potomków, które zakończyły działanie funkcja nie jest blokowana
  - **WUNTRACED** – sygnalizuj zatrzymanie potomka
  - **WCONTINUED** – sygnalizuj wznowienie działania potomka w wyniku otrzymania sygnału **SIGCONT** (dla jądra Linux >= 2.6.10)
- Parametr **status** – zwraca informację o stanie procesu

Parametr **status** (1/2)

- **Do odczytania statusu zakończenia procesu używa się następujące makra:**
  - **WIFEXITED(***status***)** zwraca prawdę jeśli proces zakończył się „normalnie” – za pomocą funkcji exit(), _exit() lub powrót z funkcji main(),
  - **WEXITSTATUS(***status***)** - zwraca status powrotu procesu, który jest zakodowany na 8 najmniej znaczących bitach zmiennej status. Jest to wartość przekazana w funkcjach exit(), _exit() lub parametr słowa kluczowego return. Działa prawidłowo jeśli makro **WIFEXITED** zwróciło prawdę.
  - **WIFSIGNALED(***status***)** – zwraca prawdę jeśli proces został zakończony przez sygnał.
  - **WTERMSIG(***status***)** – zwraca numer sygnału, który spowodował zakończenie potomka. returns the number of the signal that caused the child process to terminate. Działa prawidłowo jeśli makro **WIFSIGNALED** zwróciło prawdę.

Parametr **status** (2/2)

- **Do odczytania statusu zakończenia procesu używa się następujące makra (ciąg dalszy):**
  - **WCOREDUMP(***status***)** - zwraca prawdę jeśli proces wykonał zrzut pamięci (core dump). Działa prawidłowo jeśli makro **WIFSIGNALED** zwróciło prawdę. Może nie być zaimplementowane w niektórych systemach Unix. Używać z: #ifdef WCOREDUMP ... #endif.
  - **WIFSTOPPED(***status***)** - zwraca prawdę jeśli potomek został zatrzymany przez sygnał. Można używać jedynie z flagą **WUNTRACED** lub gdy proces jest debagowany.
  - **WSTOPSIG(***status***)** - zwraca numer sygnału, który spowodował zatrzymanie procesu. Działa prawidłowo jeśli makro **WIFSTOPPED** zwróciło prawdę.
  - **WIFCONTINUED(***status***)** (od Linux 2.6.10) – zwraca prawdę, jeśli proces został ponownie przywrócony do działania przez sygnał **SIGCONT**.

Funkcja wait()

pid_t wait(int *status);

- Funkcja **wait**() blokuje działanie procesu wywołującego dopóki jeden z procesów potomnych się nie zakończy . Równoważna wywołaniu funkcji waitpid(-1, status, 0);
- Tylko jeden parametr
- Zawsze blokująca
- Nie można otrzymać informacji o zatrzymaniu potomka (sygnał **SIGSTOP**)

Dlaczego waitpid() a nie wait() w programach sieciowych?

- waitpid() posiada dodatkowe flagi: można ją uruchomić jako funkcję nieblokującą
- Jeśli proces jest w trakcie obsługi sygnału, to nie otrzymuje informacji o przychodzących sygnałach – wyklucza to zastosowanie funkcji wait() (jako blokującej) w obsłudze sygnału gdy nie wiemy ile potomków się zakończyło

Serwer współbieżny z użyciem funkcji fork() - przykład

```
21.  socket(…); bind(…); listen(…);
22.  signal(SIGCHLD, sig_chld);
23.  for ( ; ; ) {
24.       if ( (connfd = accept(listenfd, (struct sockaddr *) &cliaddr, &len)) < 0){
25.            …;  //obsługa ewentualnego przerwania
26.       }
27.       if( (pid=fork()) == 0 ){ //potomek
28.            close(listenfd);
29.            OBSŁUGA();
30.            close(connfd);
31.            exit(0);
32.       else {
33.            close(connfd);
34.       }
35.  }
```

Przykład oprogramowania klient-serwer – obsługa sygnału SIGCHILD

```
void
sig_chld(int signo)
{
    pid_t       pid;
    int         stat;

    while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
        printf("child %d terminated\n", pid);
    return;
}
```

Obsługa sygnału **SIGCHLD - podsumowanie**

- Sygnał SIGCHLD jest wysyłany do procesu macierzystego po zakończeniu procesu potomka.
- Domyślną akcją obsługi sygnału SIGCHLD jest zignorowanie tego sygnału. Jeśli w programie nie zostanie ten sygnał obsłużony, to zostanie zignorowany i powstanie proces „zombie”. Powinno się ten sygnał obsługiwać, aby zwalniać zasoby systemowe.
- Jeśli ustanowi się procedurę obsługi sygnału SIGCHLD, proces jest w stanie „zombie” tylko od czasu zakończenia procesu do wywołania funkcji np. waitpid() w procesie macierzystym
- Dla systemu Linux > 2.6.9, **jeśli w programie macierzystym jawnie zdeklaruje się, że sygnał SIGCHLD ma być ignorowany (SIG_IGN),** to wtedy system natychmiast po zakończeniu procesu usuwa proces z tablicy procesów (**nie jest tworzony proces „zombie**”).