Gniazda

dr inż. Sławomir Samolej
Katedra Informatyki i Automatyki
Politechnika Rzeszowska

Program przedmiotu oparto w części na materiałach opublikowanych na:
http://wazniak.mimuw.edu.pl/

oraz
na materiałach opracowanych przez
dr inż. Jędrzeja Ułasiewicza:
jedrzej.ulasiewicz.staff.iiar.pwr.wroc.pl

Interfejs gniazd

- Jest pochodną potoków i został po raz pierwszy zaproponowany w wersji Uniksa pochodzącej z Berkeley
- Gniazda używa się jak potoki, ale uogólniono je w celu nawiązywania komunikacji w sieci komputerowej
- Proces działający w jednym komputerze może wykorzystać gniazda, aby porozumieć się z procesem działającym w innym, co pozwala na implementację rozproszonych w sieci systemów klient-serwer
- Interfejs gniazd został także udostępniony dla systemów Windows w postaci specyfikacji Windows Sockets, znanej jako WinSock. Dzięki temu można pisać programy przeznaczone dla Microsoft Windows, które będą współdziałać w sieci z komputerami opartymi na Uniksie w ramach systemu klient-serwer.

Czym jest gniazdo?

- Gniazdo jest mechanizmem komunikacyjnym, który pozwala na implementację systemów klient-serwer albo na lokalnym komputerze, albo w sieciach.
- Usługi Uniksa, takie jak drukowanie, oraz narzędzia sieciowe, takie jak rlogin i ftp, zwykle komunikują się za pomocą gniazd
- Gniazda są tworzone i używane w odmienny sposób niż potoki, ponieważ istnieje w nich wyraźne rozróżnienie pomiędzy serwerem i klientem
- Mechanizm gniazd w naturalny sposób wspiera implementacje z pojedynczym serwerem i wieloma klientami.

Połączenie poprzez gniazda - serwer

- Na początku aplikacja serwera tworzy gniazdo, które jest zasobem systemu operacyjnego przypisanym danemu procesowi serwera
- Po stworzeniu gniazda poprzez funkcję systemową **socket**, żaden inny proces na razie nie ma do niego dostępu
- Następnie serwer nadaje gniazdu nazwę (wywołanie funkcji **bind**) (gniazda lokalne – nazwa pliku, gniazda sieciowe – numer portu/punkt dostępu)
- Proces serwera oczekuje następnie na połączenie klienta z nazwanym gniazdem.
- Jeśli z gniazdem chce połączyć się jednocześnie wielu klientów, wówczas wywołanie systemowe **listen** tworzy kolejkę, przeznaczoną dla nadchodzących połączeń.
- Serwer może zaakceptować połączenie z klientem przy użyciu systemowej funkcji **accept**
- Kiedy serwer wywołuje **accept**, tworzone jest nowe gniazdo, niezależne od nazwanego. Gniazdo to jest używane wyłącznie do komunikacji z danym klientem.
- Nazwane gniazdo oczekuje na dalsze połączenie z innymi klientami; jeśli serwer jest odpowiednio napisany, wówczas może obsługiwać wiele połączeń jednocześnie. W przypadku prostego serwera klienci czekają w kolejce **listen**, aż serwer będzie ponownie gotowy.

Połączenie poprzez gniazda - klient

- Klient tworzy nie nazwane gniazdo za pomocą funkcji **socket**
- Wywołuje **connect**, aby nawiązać połączenie z serwerem używając nazwanego gniazda serwera jako adresu
- Po nawiązaniu połączenia, gniazda są używane w podobny sposób, jak deskryptory plików, zapewniając dwukierunkową komunikację.

Prosty klient – serwer

```
// Pliki nagłówkowe

int main() //klient
{   int sockfd;
    int len;
    struct sockaddr_un address;
    int result;
    char ch = 'A';
    sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
    address.sun_family = AF_UNIX;
    strcpy(address.sun_path, "server_socket");
    len = sizeof(address);
    result = connect(sockfd, (struct sockaddr *)&address,
len);
    if(result == -1) {       perror("oops: client1");
                             exit(1);}
    write(sockfd, &ch, 1);
    read(sockfd, &ch, 1);
    printf("char from server = %c\n", ch);
    close(sockfd);
    exit(0);
}
```

```
// Pliki nagłówkowe

int main() //serwer
{  int server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_un server_address;
    struct sockaddr_un client_address;
    unlink("server_socket");
    server_sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
    server_address.sun_family = AF_UNIX;
    strcpy(server_address.sun_path, "server_socket");
    server_len = sizeof(server_address);
    bind(server_sockfd, (struct sockaddr *)&server_address,
                  server_len);
    listen(server_sockfd, 5);
    while(1) {  char ch;
       printf("server waiting\n");
      client_len = sizeof(client_address);
       client_sockfd = accept(server_sockfd,
          (struct sockaddr *)&client_address, &client_len);
       read(client_sockfd, &ch, 1);
       ch++;
       write(client_sockfd, &ch, 1);
       close(client_sockfd);}}
```

Atrybuty gniazd

- Gniazda charakteryzują się trzema atrybutami:
  - Domeną
  - Typem
  - Protokołem
- Posiadają również **adres**, który jest używany jako ich nazwa.
  - Adresy mają różne formaty, w zależności od **domeny** (domain) znanej jako **rodziny protokołów** (protocol family).
  - Każda **rodzina protokołów** może korzystać z jednej lub większej liczby **rodzin adresów** (address family), aby zdefiniować format adresu.

Domeny gniazd

- Domeny gniazd określają środowisko sieciowe, z którego będą korzystać gniazda. Najpopularniejsze z nich to:
  - AF_INET – sieci internetowe – należy podać adres IP oraz PORT gniazda
  - AF_UNIX – gniazda ulokowane na pojedynczym komputerze odwzorowane w systemie plików
  - AF_ISO – sieci oparte na protokołach standardu ISO
  - AF_NS – Xerox Network System

Typy gniazd

- Domena gniazd może obejmować kilka metod komunikacji, a każdą może cechować odmienna charakterystyka
- W domenie AF_UNIX typowo ustala się typ gniazd na SOCK_STREAM (przesyłanie strumieniowe)
- W domenie AF_INET:
  - SOCK_STREAM dla protokołu TCP/IP
  - SOCK_DGRAM dla protokołu UDP/IP

Protokoły gniazd

- Będą używane domyślne.

Funkcje obsługujące gniazda (1)

- Utworzenie gniazda:

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

- domain:
  - AF_UNIX - wewnętrzy system plików
  - AF_INET – protokoły internetowe
  - AF_ISO – protokoły standardu ISO
  - AF_NS – protokoły standardu Xerox Network Systems
  - AF_IPX – protokoły Novell IPX
  - AF_APPLETALK – protokoły Appletalk DDS
- type:
  - SOCK_STREAM
  - SOCK_DGRAM
- protocol:
  - 0 – protokół domyślny
- Wartość zwracana:
  - Deskryptor gniazda podobny do deskryptora pliku. Pozwala na zapis lub odczyt danych z zastosowaniem funkcji **read** i **write**. Funkcja **close** kończy połączenie przez gniazdo.

Adresy gniazd

- Adres gniazda AF_UNIX:

```
struct sockaddr_un {
sa_family_t sun_family;    /* AF_UNIX */
char sun_path[];           /* ścieżka do pliku */
                                        };
```

- Adres gniazda AF_INET:

```
struct sockaddr_in {
short int sin_family;                   /* AF_INET */
unsigned short int sin_port;            /* Numer portu */
struct in_addr sin_addr;                /* Adres internetowy */
};
```

- Struktura opisująca adres internetowy:

```
struct in_addr {
unsigned long int s_addr;
};
```

Uwaga: Cztery bajty adresu IP są łączone w pojedynczą, 32-bitową liczbę.

Gniazdo AF_INET jest w pełni określone przez swoją domenę, adres IP oraz numer portu. Z punktu widzenia aplikacji wszystkie gniazda zachowują się jak deskryptory plików i są adresowane za pomocą unikatowej liczby całkowitej.

Nadawanie nazwy gniazdu

- AF_UNIX -> związanie ze ścieżką do pliku
- AF_INET -> związanie z numerem portu IP

int bind(int socket,
const struct sockaddr *address, size_t address_len);

- Następuje przypisanie adresu określonego parametrem **address** do nie nazwanego gniazda, które jest związane z deskryptorem pliku **socket**. Długość struktury adresowej jest przekazana w argumencie **address_len**.
- Wartości zwracane:
  - 0 – sukces, -1 – błąd, zmienna errno ustawiana na wartości:
    - EBADF – deskryptor pliku jest błędy
    - ENOTSOCK – deskryptor pliku nie odnosi się do gniazda
    - EINVAL – deskryptor odnosi się do już nazwanego gniazda
    - EADDRNOTAVAIL – adres jest niedostępny
    - EADDRINUSE – adres został już przydzielony jakiemuś gniazdu

Tworzenie kolejki na gnieździe

int listen(int socket, int backlog);

- socket – deskryptor gniazda
- backlog – maksymalna liczba elementów w kolejce (jeśli będzie więcej zapytań, to zostaną odrzucone na poziomie systemu operacyjnego)
- Wartości zwracane:

  0 – sukces, -1 – błąd:

  - Zmienna errno może zawierać: EBADF, EINVAL, ENOTSOCK

Akceptowanie połączeń

```
int accept(     int socket,
                struct sockaddr *address,
                size_t *address_len);
```

- Funkcja **accept** powróci, kiedy klient spróbuje połączyć się z gniazdem, zdefiniowanym parametrem **socket**.
- Klientem będzie pierwsze oczekujące połączenie z kolejki danego gniazda.
- Funkcja **accept** tworzy nowe gniazdo do celów komunikacyjnych z danym klientem i zwraca jego deskryptor.
- Gniazdo będzie miało ten sam typ, co gniazdo zdefiniowane w wywołaniu funkcji **listen**
- **Uwaga:** Wcześniej gniazdo musi otrzymać nazwę w wywołaniu funkcji **bind** i posiadać kolejkę połączeń przydzieloną przez **listen**. Adres wywołującego klienta zostanie umieszczony w strukturze **sockaddr**, na którą wskazuje **address**. Jeśli adres klienta nie ma znaczenia, można użyć tu wskaźnika pustego.
- Jeśli w kolejce gniazda nie ma żadnych oczekujących połączeń, funkcja **accept** zablokuje się (to znaczy program zawiesi działanie) aż do momentu nawiązania połączenia przez klienta.

Żądanie nawiązania połączenia

int connect(int socket,
const struct sockaddr *address,
size_t address_len);

- Gniazdo określone przez parametr **socket** jest łączone z gniazdem serwera, określonym przez **address**, który ma długość **address_len**.
- Gniazdo musi być poprawnym deskryptorem pliku, zwróconym przez funkcję **socket**
- W przypadku pomyślnego wykonania connect zwraca 0, a w razie błędu: -1. Zmienna errno może przyjąć wtedy wartości:
  - EBADF - w parametrze socket określono niepoprawny deskryptor pliku
  - EALREADY – gniazdo nawiązało już połączenie
  - ETIMEDOUT – minął czas na nawiązanie połączenia
  - ECONNREFUSED – serwer odrzucił próbę połączenia
- Uwagi: Jeśli połączenie nie może zostać natychmiast nawiązane, **connect** zablokuje się na nieokreślony czas. Po upłynięciu limitu czasu połączenie zostanie zaniechane i funkcja **connect** zwróci błąd.

Zamykanie gniazda

- Do zamykania gniazda służy funkcja **close()**.
- Poprawnie powinno się zamknąć gniazdo po obu stronach.

Prosty klient – serwer sieciowy

```
// Pliki nagłówkowe
int main() //klient
{   int sockfd;
    int len;
    struct sockaddr_in address;
    int result;
    char ch = 'A';
     sockfd = socket(AF_INET, SOCK_STREAM, 0);
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = inet_addr("127.0.0.1");
    address.sin_port = 9734;
    len = sizeof(address);
   result = connect(sockfd,
                          (struct sockaddr *)&address, len);
    if(result == -1) {   perror("oops: client2"); exit(1); }
    write(sockfd, &ch, 1);
    read(sockfd, &ch, 1);
    printf("char from server = %c\n", ch);
    close(sockfd);
    exit(0);
}
```

```
// Pliki nagłówkowe
int main() //serwer
{    int server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_in server_address;
    struct sockaddr_in client_address;
    server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = inet_addr("127.0.0.1");
    server_address.sin_port = 9734;
    server_len = sizeof(server_address);
    bind(server_sockfd,
            (struct sockaddr *)&server_address, server_len);
    listen(server_sockfd, 5);
    while(1) {       char ch;       printf("server waiting\n");
       client_len = sizeof(client_address);
       client_sockfd = accept(server_sockfd,
          (struct sockaddr *)&client_address, &client_len);
       read(client_sockfd, &ch, 1);
       ch++;
       write(client_sockfd, &ch, 1);
       close(client_sockfd);}}
```

Problemy…

- Raport z wykonania polecenia netstat:

```
$ server2 &
[4] 1225
$ server waiting
client2
server waiting
char from server = B
$ netstat
Active Internet connections
Proto Recv-Q Send-Q Local Address Foreign Address (State) User
tcp    1       0 localhost:1574 localhost:1174 TIME_WAIT root
```

- Port lokalny ma wartość 1574 zamiast 9735!
- Wynika to z różnic reprezentacji liczb (kolejności bajtów w liczbach wielobajtowych) na komputerze i w „porządku sieciowym”.

Funkcje konwertujące porządek bajtów z hosta do sieci i odwrotnie

```
#include <netinet/in.h>
unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

- htonl = host to network, long
- htons = host to network, short
- ntohl = network to host, long
- ntohs = network to host, short

Klient – serwer sieciowy z poprawnymi portami

```
int main() //klient
{
   int sockfd;
   int len;
   struct sockaddr_in address;
   int result;
   char ch = 'A';
   sockfd = socket(AF_INET, SOCK_STREAM, 0);
   address.sin_family = AF_INET;
   address.sin_addr.s_addr = inet_addr("127.0.0.1");
   address.sin_port = htons(9734);
   len = sizeof(address);
   result = connect(sockfd,
                    (struct sockaddr *)&address, len);
   if(result == -1) { perror("oops: client3"); exit(1); }
   write(sockfd, &ch, 1);
   read(sockfd, &ch, 1);
   printf("char from server = %c\n", ch);
   close(sockfd);
   exit(0);
}
```

```
int main() //serwer
{
   int server_sockfd, client_sockfd;
   int server_len, client_len;
   struct sockaddr_in server_address;
   struct sockaddr_in client_address;
   server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
   server_address.sin_family = AF_INET;
   server_address.sin_addr.s_addr = htonl(INADDR_ANY);
   server_address.sin_port = htons(9734);
   server_len = sizeof(server_address);
   bind(server_sockfd,
         (struct sockaddr *)&server_address, server_len);
   listen(server_sockfd, 5);
   while(1) {
      char ch;
      printf("server waiting\n");
      client_len = sizeof(client_address);
      client_sockfd = accept(server_sockfd,
         (struct sockaddr *)&client_address, &client_len);
      read(client_sockfd, &ch, 1);
      ch++;
      write(client_sockfd, &ch, 1);
      close(client_sockfd);
   }}
```

Uwagi

- Serwer jest przygotowany do odbierania danych od komputera o dowolnym adresie IP
- Numer portu niezależnie od platformy sprzętowej jest tłumaczony poprawnie
- Wynik działania funkcji netstat:

```
$ netstat
Active Internet connections
Proto Recv-Q Send-Q Local Address Foreign Address (State) User
Tcp    1      0 localhost:9734 localhost:1175 TIME_WAIT root
```

Obsługa wielu klientów I

- Po zaakceptowaniu nowego połączenia przez serwer tworzone jest kolejne gniazdo, a pierwotnie nasłuchujące gniazdo jest nadal dostępne do połączeń
- Jeśli serwer nie jest w stanie natychmiast przyjąć tych połączeń, będą one oczekiwać w kolejce
- Ponieważ pierwotnie gniazdo jest wciąż dostępne, a gniazda zachowują się jak deskryptory plików, to przy rozgałęzieniu procesy z zastosowaniem instrukcji **fork** odpowiednie wartości zmiennych zostaną przekopiowane i proces obsługi wielu klientów można zrównoleglić.
- Obsługą danego połączenia będzie się zajmował proces potomny, a proces pierwotny będzie oczekiwał na kolejne połączenia.

Serwer sieciowy ze współbieżną obsługą wielu klientów

```
int main()
{
   int server_sockfd, client_sockfd;
   int server_len, client_len;
   struct sockaddr_in server_address;
   struct sockaddr_in client_address;
   server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
   server_address.sin_family = AF_INET;
   server_address.sin_addr.s_addr =
htonl(INADDR_ANY);
   server_address.sin_port = htons(9734);
   server_len = sizeof(server_address);
   bind(server_sockfd,
         (struct sockaddr *)&server_address,
server_len);
   listen(server_sockfd, 5);
```

```
signal(SIGCHLD, SIG_IGN);
   while(1) {
      char ch;
      printf("server waiting\n");
     client_len = sizeof(client_address);
      client_sockfd = accept(server_sockfd,
         (struct sockaddr *)&client_address, &client_len);
      if(fork() == 0) {
         read(client_sockfd, &ch, 1);
         sleep(5);
         ch++;
         write(client_sockfd, &ch, 1);
         close(client_sockfd);
         exit(0);
      }
      else {
         close(client_sockfd);
      }
   }
}
```

Wynik przykładowej sesji z 3 klientami:

```
$ ./server4&
[7] 1571
$server waiting
./client3 & ./client3 & ./client3 & ps -ax
[8] 1572
[9] 1573
[10] 1574
server waiting
server waiting
server waiting
PID TTY STAT TIME COMMAND
1557 pp0 S 0:00 ./server4
1572 pp0 S 0:00 ./client3
1573 pp0 S 0:00 ./client3
1574 pp0 S 0:00 ./client3
1575 pp0 R 0:00 ps -ax
1576 pp0 S 0:00 ./server4
1577 pp0 S 0:00 ./server4
1578 pp0 S 0:00 ./server4
```

```
$ char from server = B
char from server = B
char from server = B
ps -ax
PID TTY STAT TIME COMMAND
1557 pp0 S 0:00 ./server4
1580 pp0 R 0:00 ps -ax
[8] Done ./client3
[9]- Done ./client3
[10]+ Done ./client3
$
```

Funkcja systemowa **select** (1)

- Pozwala na programowi zaczekać, aż nadejdą dane wejściowe (bądź też zakończy się zapisywanie danych wyjściowych) od kilku niskopoziomowych deskryptorów pliku jednocześnie
- Serwer może wtedy odpowiadać na żądanie pojawiające się na jednym z wielu otwartych gniazd
- Funkcja **select** operuje na strukturach danych **fd_set**, które są zbiorami otwartych deskryptorów plików.
- Makra do obsługi zbiorów:

```
#include <sys/types.h>
#include <sys/time.h>
void FD_ZERO(fd_set *fdset);          // inicjuje zbiór jako pusty
void FD_CLR(int fd, fd_set *fdset);   // zeruj el. zbioru odpowiadający fd
void FD_SET(int fd, fd_set *fdset);   // ustaw el. zbioru odpowiadający fd
int FD_ISSET(int fd, fd_set *fdset);  // zwraca wartość niezerową, jeśli deskry-
                                      // ptor pliku fd jest elementem fd_set
```

Funkcja systemowa **select** (2)

- Funkcja **select** przyjmuje także limit czasu swojego działania, aby zapobiec zablokowaniu się przez nieokreślony czas
- Parametrem funkcji jest struktura typu time_t:

```
struct timeval {
time_t tv_sec;        /* sekundy */
long tv_usec;         /* mikrosekundy */};
```

- Prototyp funkcji:

```
int select(                    // zwraca 0 – upłynął czas
                               // zwraca -1 – błąd
                               // zwraca liczbę zmienionych deskrypt.
int nfds,                      // liczba deskryptorów, którą należy sprawdzić
fd_set *readfds,               // wskazanie na zbiór deskryptorów do odczytu
fd_set *writefds,              // wskazanie na zbiór deskryptorów do zapisu
fd_set *errorfds,              // wskazanie na zbiór deskryptorów w stanie błędu
struct timeval *timeout);      // czas, po którym funkcja powróci, jeśli w deskryptorach
                               // nie wykryto zmian
```

- Funkcja powróci jeśli dowolny deskryptor **readfds** jest gotowy do odczytu, dowolny deskryptor w zbiorze **writefds** jest gotowy do zapisu, albo dowolny deskryptor w zbiorze **errorfds** jest w stanie błędu
- Po powrocie z **select** zbiory deskryptorów zostaną zmodyfikowane, aby wskazać, które spośród nich są gotowe do odczytu, zapisu, bądź też są w stanie błędu

Serwer sieciowy z zastosowaniem funkcji select

```
int main()
{
   int server_sockfd, client_sockfd;
   int server_len, client_len;
   struct sockaddr_in server_address;
   struct sockaddr_in client_address;
   int result;
   fd_set readfds, testfds;
   server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
   server_address.sin_family = AF_INET;
   server_address.sin_addr.s_addr =
                                htonl(INADDR_ANY);
   server_address.sin_port = htons(9734);
   server_len = sizeof(server_address);
   bind(server_sockfd,
   (struct sockaddr *)& server_address, server_len);
   listen(server_sockfd, 5);
   FD_ZERO(&readfds);
   FD_SET(server_sockfd, &readfds);

   while(1) {
      char ch;       int fd;       int nread;
      testfds = readfds;
      printf("server waiting\n");
      result = select(FD_SETSIZE, &testfds, (fd_set *)0,
         (fd_set *)0, (struct timeval *) 0);
      if(result < 1) { perror("server5"); exit(1);  }
```

```
      for(fd = 0; fd < FD_SETSIZE; fd++) {
         if(FD_ISSET(fd,&testfds)) {
          if(fd == server_sockfd) {
               client_len = sizeof(client_address);
               client_sockfd = accept(server_sockfd,
                (struct sockaddr *)& client_address,
                                        &client_len);
               FD_SET(client_sockfd, &readfds);
               printf("adding client on fd %d\n", client_sockfd);
             }
             else {
               ioctl(fd, FIONREAD, &nread);
               if(nread == 0) {
                  close(fd);
                  FD_CLR(fd, &readfds);
                  printf("removing client on fd %d\n", fd);
               }
               else {
                  read(fd, &ch, 1);
                  sleep(5);
                  printf("serving client on fd %d\n", fd);
                  ch++;
                  write(fd, &ch, 1);
               } }  }  }  }  }
```

Uwagi

- Tworzone i nazywane jest gniazdo serwera
- Tworzona jest kolejka połączeń oraz inicjowany zbiór **readfds**, aby obsłużyć wejście z gniazda **server_sockfd**
- Następuje czekanie na żądania od klientów. Limit czasu ustawiony jest na pusty wskaźnik, więc nie zajdzie przekroczenie limitu czasu
- Program zakończy pracę i zgłosi błąd, jeśli **select** zwróci wartość mniejszą od 1
- Kiedy zostanie wykryta aktywność, to sprawdzane są po kolei wszystkie deskryptory (**FD_ISSET**), aby odnaleźć ten aktywny
- Jeśli aktywny jest deskryptor **server_sockfd**, oznacza to, że pojawiło się nowe żądanie połączenia, więc dodajemy odpowiedni deskryptor **client_sockfd** do zbioru deskryptorów
- Jeśli aktywność nie jest związana z serwerem, to musi być związana z klientem. Jeśli odebrano 0 bajtów, to oznacza, że klient zakończył pracę i można usunąć jego deskryptor ze zbioru.
- W przeciwnym wypadku następuje „obsługa” klienta.

Wynik przykładowej sesji z 3 klientami:

```
$ ./server5 &
[7] 1670
$ server waiting
./client3 & ./client3 & ./client3 & ps -ax
[8] 1671
[9] 1672
[10] 1673
adding client on fd 4
server waiting
adding client on fd 5
server waiting
adding client on fd 6
server waiting
PID TTY STAT TIME COMMAND
1670 pp0 S 0:00 ./server5
1671 pp0 S 0:00 ./client3
1672 pp0 S 0:00 ./client3
1673 pp0 S 0:00 ./client3
1674 pp0 R 0:00 ps -ax
```

```
$ serving client on fd 4
server waiting
char from server = B
serving client on fd 5
char from server = B
serving client on fd 6
server waiting
removing client on fd 4
removing client on fd 5
server waiting
char from server = B
removing client on fd 6
server waiting
[8] Done ./client3
[9]- Done ./client3
[10]+ Done ./client3
$
```