4. Algorytmy serwera

4.1. Typy serwerów

- Serwer iteracyjny (ang. *iterative server*) obsługuje zgłoszenia klientów sekwencyjnie, jedno po drugim.
- Serwer współbieżny (ang. *concurrent server*) obsługuje wiele zgłoszeń jednocześnie.

- Serwer połączeniowy (ang. *connection-oriented server*) używa protokołu TCP.
- Serwer bezpołączeniowy (ang. *connectionless server*) używa protokołu UDP.

- Serwer wielostanowy (ang. *stateful server*) pamięta stany interakcji z klientami.
- Serwer bezstanowy (ang. *stateless server*) nie przechowuje informacji o stanie interakcji z klientem.

TCP a UDP

- Protokół TCP:
  - zapewnia połączenie typu punkt-punkt
  - gwarantuje niezawodność połączenia - klient albo ustanawia połączenie z serwerem albo gdy żądanie połączenia nie może być zrealizowane otrzymuje zwrotny komunikat
  - gwarantuje niezawodność przesyłania danych - po nawiązaniu połączenia dane są dostarczane w takiej samej kolejności w jakiej zostały wysłane, nie są gubione ani powielane; jeśli połączenie zostanie zerwane, nadawca jest o tym informowany.
  - steruje szybkością przepływu danych - nadawca nie może szybciej wysyłać danych niż odbiorca jest w stanie je odebrać
  - działa w trybie full-duplex - pojedynczy kanał może służyć do równoczesnego przesyłania danych w dwóch kierunkach, klient może przesyłać dane do serwera zaś serwer do klienta
  - jest protokołem strumieniowym - do odbiorcy przesyłany jest strumień bajtów, nie musi być on tak samo zgrupowany jak był wysłany, możliwe jest za to przesyłanie dużych bloków danych

- Protokół UDP:
  - nie wymaga utworzenia połączenia, nie potrzebuje narzutu czasu związanego z tworzeniem i utrzymywaniem połączenia
  - umożliwia połączenia typu wiele-wiele
  - jest protokołem zawodnym - komunikat może być zagubiony, powielony, dostarczony w innej kolejności
  - nie ma mechanizmu kontroli przesyłania danych - jeśli datagramy napływają szybciej niż mogą być przetworzone, są odrzucane bez powiadamiania o tym użytkownika
  - jest protokołem datagramowym - nadawca określa liczbę bajtów do wysłania i taką samą liczbę bajtów otrzymuje odbiorca w jednym komunikacie; rozmiar komunikatu jest ograniczony.

Serwer iteracyjny a serwer współbieżny

- Czas przetwarzania zgłoszenia przez serwer (T_P, ang. *request processing time*) to całkowity czas trwania obsługi jednego, wyizolowanego zgłoszenia.
- Obserwowany czas odpowiedzi dla klienta (T_R, ang. *observed processing time*) to całkowity czas upływający od wysłania głoszenia przez klienta do chwili uzyskania odpowiedzi serwera.
- T_R nigdy nie jest krótszy od T_P , a może być dużo dłuższy, gdy serwer ma do obsłużenia kolejkę zgłoszeń.

- Przykładowe kryteria wyboru:

  - Średni obserwowany czas odpowiedzi serwera iteracyjnego:

    $$T_R = (N/2 + 1)\ T_P$$

    gdzie N - średnia długość kolejki zgłoszeń. Można założyć, że jeśli mała kolejka w serwerze iteracyjnym nie wystarczy, to należy użyć serwera współbieżnego.

  - Czas przetwarzania zgłoszenia T_P dla serwera iteracyjnego powinien być mniejszy niż:

    $$T_{P\,MAX} = 1/KR$$

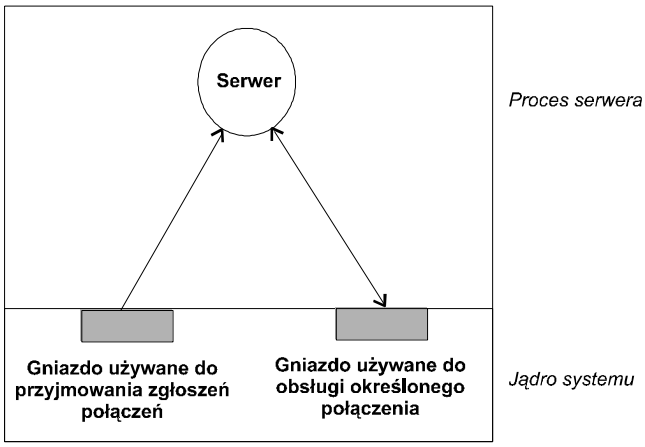
    gdzie: K - liczba klientów, R - liczba zgłoszeń nadsyłanych przez pojedynczego klienta w ciągu sekundy. Jeśli warunek ten nie zostanie spełniony, należy rozważyć serwer współbieżny.

  - Serwer współbieżny poprawi czas odpowiedzi, jeśli:
    - przygotowanie odpowiedzi wymaga wielu operacji we-wy
    - czas przetwarzania jest zróżnicowany dla różnych żądań
    - serwer jest uruchomiony na maszynie wieloprocesorowej

4.2. Algorytm działania iteracyjnego serwera połączeniowego (TCP)

Zasada: *jeden proces kolejno obsługuje połączenia z poszczególnymi klientami*.

1. Utwórz gniazdo i zwiąż je z powszechnie znanym adresem odpowiadającym usłudze udostępnianej przez serwer (funkcje `socket`, `bind`)
2. Ustaw bierny tryb pracy gniazda (funkcja `listen`).
3. Przyjmij kolejne zgłoszenie połączenia nadesłane na adres tego gniazda i uzyskaj przydział nowego gniazda do obsługi tego połączenia (funkcja `accept`).
4. Odbieraj kolejne zapytania od klienta, konstruuj odpowiedzi i wysyłaj je do klienta zgodnie z protokołem zdefiniowanym w warstwie aplikacji.
5. Po zakończeniu obsługi danego klienta zamknij połączenie i wróć do kroku 3, aby przyjąć następne połączenie.



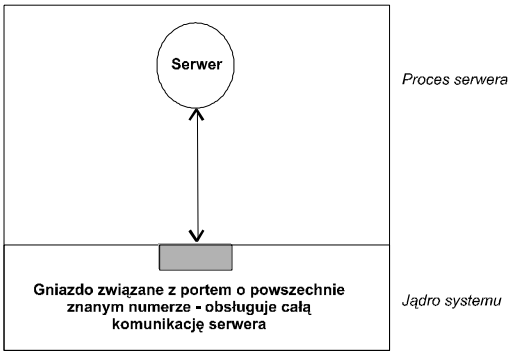
- Serwer połączeniowy:
  - wykorzystuje wszystkie zalety protokołu TCP

  ale
  - potrzebuje czasu na nawiązanie połączenia TCP i jego zakończenie
  - dla każdego połączenia tworzy oddzielne gniazdo
  - nie przesyła pakietów przez połączenie, które jest bezczynne

4.3. Algorytm działania iteracyjnego serwera bezpołączeniowego (UDP)

Zasada: *jeden proces kolejno obsługuje zapytania od poszczególnych klientów*.

1. Utwórz gniazdo i zwiąż je z powszechnie znanym adresem odpowiadającym usłudze udostępnianej przez serwer (funkcje `socket`, `bind`).
2. Odbieraj kolejne zapytania od klientów, konstruuj odpowiedzi i wysyłaj je zgodnie z protokołem warstwy aplikacji.



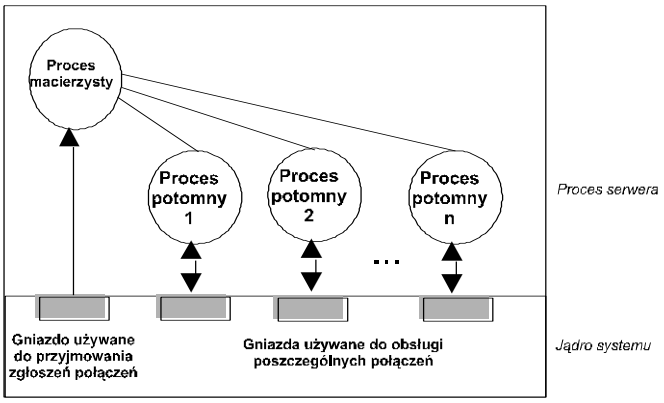
- Serwer bezpołączeniowy
  - nie jest narażony na wyczerpanie zasobów
  - pozwala na pracę w trybie rozgłaszania

  ale
  - trzeba wbudować mechanizmy gwarantujące niezawodność (część może być po stronie klienta)

4.4. Algorytm działania współbieżnego serwera połączeniowego (TCP)

Zasada: proces główny serwera przyjmuje zgłoszenia połączeń i tworzy nowe procesy potomne lub wątki do obsługi każdego połączenia; proces potomny po wykonaniu usługi zamyka połączenie.

Proces główny, krok 1.	Utwórz gniazdo i zwiąż je z powszechnie znanym adresem odpowiadającym usłudze realizowanej przez serwer..
Proces główny, krok 2.	Ustaw bierny tryb pracy gniazda, tak aby mogło być używane przez serwer.
Proces główny, krok 3.	Przyjmuj kolejne zgłoszenia połączeń od klientów posługując się funkcją `accept`; dla każdego połączenia utwórz nowy proces potomny lub wątek, który przygotuje odpowiedź.
Proces potomny/wątek, krok 1.	Rozpoczynając działanie, przejmij od procesu głównego nawiązane połączenie (tzn. gniazdo przeznaczone dla tego połączenia).
Proces potomny/wątek, krok 2.	Korzystając z tego połączenia, prowadź interakcję z klientem; odbieraj zapytania i wysyłaj odpowiedzi.
Proces potomny/wątek, krok 3.	Zamknij połączenie i zakończ się. Proces potomny/wątek kończy działanie po obsłużeniu wszystkich zapytań od jednego klienta.



- Implementacja współbieżności za pomocą procesów potomnych:
  - Kod procesu macierzystego i potomnego może być:
    - zawarty w jednym programie
    - kod procesu potomnego może być zawarty w odrębnym programie uruchamianym za pomocą funkcji z rodziny `exec`.
  - Należy zwrócić uwagę na czyszczenie po procesach potomnych i nie pozostawianie zombi.
  - Zalety: łatwy w implementacji, żaden z klientów nie może zmonopolizować serwera, załamanie jednego procesu potomnego nie wpływa na inne.
  - Wady: utworzenie nowego procesu jest czasochłonne, trudne porozumiewanie się procesów między sobą (brak wspólnej pamięci), duży program zużywa znaczące zasoby.

- Implementacja współbieżności za pomocą wątków
  - Zalety: mniejszy czas potrzebny na utworzenie nowego wątku, współdzielenie pamięci, różne metody synchronizacji,
  - Wady: mniejsza stabilność w porównaniu do procesów potomnych – błąd w jednym wątku może mieć wpływ na cały serwer, ograniczenie liczby wątków dla jednego programu

4.5. Algorytm działania wieloprocesowego współbieżnego serwera bezpołączeniowego (UDP)

Zasada: proces macierzysty (główny) serwera przyjmuje zapytania od klientów i tworzy nowe procesy potomne/wątki do obsługi każdego zapytania.

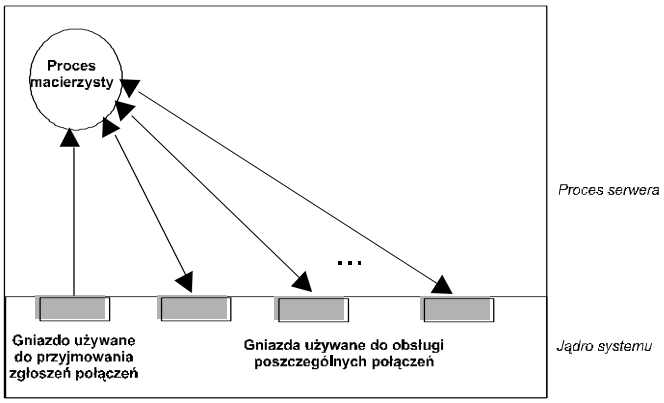
Proces macierzysty, krok 1.	Utwórz gniazdo i zwiąż je z powszechnie znanym adresem odpowiadającym usłudze realizowanej przez serwer.
Proces macierzysty, krok 2.	Odbieraj kolejne zapytania od klientów posługując się funkcją `recvfrom`; dla każdego zapytania utwórz nowy proces potomny, który przygotuje odpowiedź.
Proces potomny/wątek, krok 1.	Rozpoczynając działanie, przejmij określone zapytanie od klienta i przejmij dostęp do gniazda.
Proces potomny/watek, krok 2.	Skonstruuj odpowiedź zgodnie z protokołem warstwy aplikacji i wyślij ją do klienta posługując się funkcja `sendto`.
Proces potomny/wątek, krok 3.	Zakończ się (proces potomny/wątek kończy więc działanie po obsłużeniu jednego zapytania).

- Wady: czas potrzebny do utworzenia procesu potomnego/wątku

4.6. Współbieżność pozorna - algorytm działania jednoprocesowego, współbieżnego serwera połączeniowego (mulipleksacja)

Zasada: proces serwera czeka na gotowość któregoś z gniazd - nadejście nowego żądania połączenia lub nadejście zapytania od klienta przez gniazdo już istniejące.

1. Utwórz gniazdo i zwiąż je z portem o powszechnie znanym numerze odpowiadającym usłudze realizowanej przez serwer. Dodaj gniazdo do listy gniazd, na których są wykonywane operacje we/wy.
2. Wywołaj funkcję `select`, aby czekać na zdarzenie we/wy dotyczące istniejących gniazd.
3. W razie gotowości pierwotnie utworzonego gniazda, wywołaj funkcje `accept`, w celu przyjęcia kolejnego połączenia i dodaj nowe gniazdo do listy gniazd, na których są wykonywane operacje we/wy.
4. W razie gotowości innego gniazda, wywołaj funkcję `read`, aby odebrać kolejne zapytanie nadesłane przez klienta, skonstruuj odpowiedź i wywołaj funkcję `write`, aby przesłać odpowiedź klientowi.
5. Przejdź do kroku 2.



- Zalety: jeden proces, współdzielenie pamięci, szybka obsługa nowych połączeń
- Wady: może być bardzo złożony i trudny w pielęgnacji, klient może zmonopolizować serwer

- **`select()`** - asynchroniczne wykonywanie we-wy

SKŁADNIA

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

int select(int maxfdpl, fd_set *readfds, fd_set *writefds,
                        fd_set *exceptfds, struct timeval *timeout);
```

OPIS

- Funkcja `select` umożliwia procesowi asynchroniczne wykonywanie operacji we-wy. Proces czeka na zgłoszenie gotowości przez którykolwiek z deskryptorów z podanego zbioru. Deskryptory są implementowane jako wektory bitów.
- Parametry na we:
- `maxfdpl` - maksymalna liczba deskryptorów, które będą sprawdzane
- `readfds` - deskryptory sprawdzane dla danych wejściowych
- `writefds` - deskryptory sprawdzane dla danych wyjściowych
- `excepfds` - deskryptory sprawdzane dla sytuacji wyjątkowych
- `timeout` - pozwala określić maksymalny czas oczekiwania na wystąpienie zdarzenia.
- Wartość zwracana przez funkcję `select`:

> 0 – liczba deskryptorów, które zgłosiły gotowość
0 – upłynął czas oczekiwania
-1 – błąd.

Argumenty deskryptorów po powrocie z funkcji `select` zawierają tylko, te deskryptory, które były aktywne. Uwaga: w niektórych implementacjach zmianie ulega również argument `timeout`.

- Struktura `timeval` zdefiniowana jest w pliku `time.h` następująco:

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

W zależności od wartości argumentu `timeout` w działaniu funkcji `select` wyróżnić można trzy przypadki:

- Oba pola struktury `timeval` są równe 0 - funkcja kończy się natychmiast po sprawdzeniu wszystkich deskryptorów; mówimy wtedy o odpytywaniu deskryptorów (ang. *polling*)
- W strukturze `timeval` określono niezerowy czas oczekiwania – funkcja czeka nie dłużej niż `timeout` na gotowość któregoś z deskryptorów;
- Argument `timeout` jest równy `NULL` – powrót z funkcja następuje dopiero wtedy, gdy jeden z deskryptorów gotowy jest do wykonania operacji we-wy.

- Zdefiniowano następujące makrodefinicje do obsługi zestawów deskryptorów:

```
/* zeruj wszystkie bity w fdset */
FD_ZERO(fd_set fdset);

/* umieść 1 w bicie dla fd w fdset */
FD_SET(int fd, fd_set *fdset);

/* zeruj bit dla fd w fdset */
FD_CLR(int fd, fd_set *fdset);

/* sprawdź bit dla fd w fdset */
FD_ISSET(int fd, fd_set *fdset);
```

- Przykład:

Istnieje stała `FD_SETSIZE`, która określa maksymalną liczbę deskryptorów obsługiwanych przez `select`:

```
int sock;       // deskryptor gniazda
fd_set rfds;    // zbiór deskryptorów do czytania

FD_ZERO(&rfds);
FD_SET(sock, &rfds);
select(FD_SETSIZE,rfds,NULL,NULL,NULL);
if (FD_ISSET(sock, &rfds)) printf("Gniazdo gotowe do czytania\n");
```

4.7. Przykładowa biblioteka podstawowych funkcji dla programów serwera

- Biblioteka zaproponowana w Comer, Stevens "Sieci komputerowe", tom 3.
- Podstawowe funkcje: utworzenie gniazda biernego

```
socket = passiveTCP(usługa, dkol); // serwer połączeniowy
socket = passiveUDP(usługa);       // serwer bezpołączeniowy
```

- Implementacja funkcji `passiveTCP` (plik `passiveTCP.c`)

```
/*
 *------------------------------------------------------------
 * passiveTCP - utwórz gniazdo bierne dla serwera
 *              używającego protokołu TCP.
 *------------------------------------------------------------
 */
#include "passivesock.h"

int passiveTCP(char *service, int qlen)
{
  return passivesock(service, "tcp", qlen);
}
```

- Implementacja funkcji `passiveUDP` (plik `passiveUDP.c`)

```
/*
 *------------------------------------------------------------
 * passiveUDP - utworz gniazdo bierne dla serwera
 *              używajacego protokolu UDP.
 *------------------------------------------------------------
 */
#include "passivesock.h"

int passiveUDP(char *service)
{
  return passivesock(service, "udp", 0);
}
```

- Implementacja funkcji tworzącej gniazdo bierne `passivesock` (plik `passivesock.c`)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>

u_short portbase=0;/* przesuniecie bazowe portu
                      dla serwera nieuprzywilejowanego */

/*
 *------------------------------------------------------
 * passivesock - ustaw gniazdo dla serwera używającego TCP
 *               lub UDP i przypisz mu adres
 *------------------------------------------------------
 */
int passivesock(char *service, char *transport, int qlen)
{
  struct servent *pse;  /* wskaznik do struktury opisu usługi */
  struct protoent *ppe;  /* wskaznik do struktury opisu protokołu  */
  struct sockaddr_in sin;  /* internetowy adres punktu końcowego */
  int s, type;  /* deskryptor, typ gniazda  */

  memset(&sin, 0, sizeof(sin));
  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = INADDR_ANY;

  /* Odwzoruj nazwę usługi na numer portu */
  if(pse = getservbyname(service, transport))
    sin.sin_port = htons(ntohs((u_short)pse->s_port) + portbase);
  else if((sin.sin_port = htons((u_short)atoi(service))) == 0 )
    errexit("can't get \" %s \" service entry \n",service);

  /* Odwzoruj nazwę protokołu na jego numer */
  if ( (ppe = getprotobyname(transport)) == 0)
    errexit("can't get \" %s \" protocol entry \n",protocol);

 /* Wybierz typ gniazda odpowiedni dla protokołu */
  if (strcmp(transport, "udp") == 0)
    type = SOCK_DGRAM;
  else
    type = SOCK_STREAM;

  /* Utwórz gniazdo */
  s = socket(PF_INET, type, ppe->p_proto);
  if (s < 0)
    errexit("can't create socket: %s\n", sys_errlist[errno]);

  /* Przypisz adres gniazdu */
  if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
    errexit("can't bind to %s port: %s\n", service, sys_errlist[errno]);

  if (type == SOCK_STREAM && listen(s, qlen) < 0)
    errexit("can't listen on %s port: %s\n", service, sys_errlist[errno]);

  return s;
}
```

- Przykłady serwerów TCP i UDP napisanych z użyciem proponowanej biblioteki:
  Comer, Stevens "Sieci komputerowe", tom 3
  - iteracyjny serwer usługi DAYTIME, wersja TCP, str. 163-165
  - iteracyjny serwer usługi TIME, wersja UDP, str-157-158
  - współbieżny wieloprocesowy serwer usługi ECHO , wersja TCP, str.171-174
  - współbieżny jednoprocesowy serwer usługi ECHO, wersja TCP, str. 180-183

4.8. Przykład wieloprocesowego serwera współbieżnego echo

Serwer współbieżny echa - wersja 1

Plik TCPEchoSerwer.c

```
#include "TCPEchoSerwer.h"  /* prototypy funkcji */
#include <sys/wait.h>          /* waitpid() */

unsigned int potomekLicz=0;  /* liczba potomków */

void sig_child(int sig)
{
/* wait nieblokujące */
  while (waitpid(-1,NULL,WNOHANG)>0)
   potomekLicz--;
}

int main(int argc, char *argv[]) {
  int serwGniazdo;
  int klientGniazdo;
  unsigned short echoSerwPort;
  pid_t procesID; /* numer procesu */
  unsigned int potomekLicz=0;  /* liczba potomków */

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

  signal(SIGCHLD,sig_child);

  echoSerwPort = atoi(argv[1]);
  serwGniazdo = UtworzGniazdoTCP(echoSerwPort);
  for (;;) {
    klientGniazdo=AkceptujPolaczenieTCP(serwGniazdo);
    if ((procesID=fork()) < 0) /* błąd */
      Zakoncz("fork()");
    else if (procesID==0) {   /* proces potomka */
      close(serwGniazdo);  /*zamyka gniazdo serwera*/
      PrzetwarzajKlienta(klientGniazdo);
      close(klientGniazdo); /*zamyka gniazdo połączenia */
      exit(0);  /* zakończenie procesu potomka */
    }

    /* proces serwera */
    printf("Uruchomiono proces potomny %d\n",procesID);
    close(klientGniazdo);
    potomekLicz++;
  }
}
```

Plik TCPEchoSerwer.h

```
#include <stdio.h>     /* printf(), fprintf() */
#include <sys/socket.h> /*socket(),bind(),connect()*/
#include <arpa/inet.h>  /* sockaddr_in, inet_ntoa()*/
#include <stdlib.h>     /* atoi() */
#include <string.h>     /* memset() */
#include <unistd.h>     /* close() */
void Zakoncz(char *komunikat);  /* Funkcja bledu */
void PrzetwarzajKlienta(int klientGniazdo);
int UtworzGniazdoTCP(unsigned short port);
int AkceptujPolaczenieTCP(int serwGiazdo);
```

Plik UtworzGniazdoTCP.c

```
#include <sys/socket.h> /*socket(),bind(),connect()*/
#include <arpa/inet.h>  /* sockaddr_in, inet_ntoa()*/
#include <string.h>     /* memset() */

#define MAXKOLEJKA 5
void Zakoncz(char *komunikat);

int UtworzGniazdoTCP(unsigned short port)
{
  int gniazdo;
  struct sockaddr_in echoSerwAdr;
/* Utwórz gniazdo dla przychodzących połączeń */
  if ((gniazdo =
    socket(PF_INET, SOCK_STREAM, 0)) < 0)
      Zakoncz("socket()");

/* Zbuduj lokalną strukturę adresową */
  memset(&echoSerwAdr, 0, sizeof(echoSerwAdr));
  echoSerwAdr.sin_family = AF_INET;
  echoSerwAdr.sin_addr.s_addr = htonl(INADDR_ANY);
  echoSerwAdr.sin_port = htons(port);

/* Przypisz gniazdu lokalny adres */
  if (bind(gniazdo,(struct sockaddr *) &echoSerwAdr,
           sizeof(echoSerwAdr)) < 0)
    Zakoncz("bind()");

/* Ustaw gniazdo w trybie biernym - przyjmowania
   połączeń*/
  if (listen(gniazdo, MAXKOLEJKA) < 0)
    Zakoncz("listen()");
  return gniazdo;
}
```

Plik Zakoncz.c

```
#include <stdio.h>   /* perror() */
#include <stdlib.h>  /* exit() */
void Zakoncz(char *komunikat)
{
  perror(komunikat);
  exit(1);
}
```

Plik AkceptujPolaczenieTCP.c

```
#include <stdio.h>      /* printf() */
#include <sys/socket.h> /* accept() */
#include <arpa/inet.h>  /* sockaddr_in,inet_ntoa()*/

void Zakoncz(char *komunikat);

int AkceptujPolaczenieTCP(int serwGniazdo)
{
  int klientGniazdo;
  struct sockaddr_in echoKlientAdr;
  unsigned int klientDl;

  klientDl= sizeof(echoKlientAdr);
  if((klientGniazdo=accept(serwGniazdo,
   (struct sockaddr *)&echoKlientAdr, &klientDl)) < 0)
     Zakoncz("accept()");
  printf("Przetwarzam klienta %s\n",
      inet_ntoa(echoKlientAdr.sin_addr));
  return klientGniazdo;
}
```

Plik PrzetwarzajKlienta.c

```
#include <stdio.h>  /* printf(),fprintf(),perror() */
#include <unistd.h>     /* read(), write(), close() */
#include <sys/socket.h> /* sendto(), recvfrom() */

#define BUFWE 32

void Zakoncz(char *komunikat);

void PrzetwarzajKlienta(int klientGniazdo)
{
  char echoBufor[BUFWE];
  int otrzTekstDl;

/* Odbierz komunikat od klienta */
  otrzTekstDl=recv(klientGniazdo,echoBufor,BUFWE,0);
  if (otrzTekstDl < 0)
    Zakoncz("recv()");

/* Odeslij komunikat do klienta i pobierz nastepny */
  while(otrzTekstDl > 0)
  {
    if (send(klientGniazdo,echoBufor,otrzTekstDl,0)
      != otrzTekstDl)
    Zakoncz("send()");
    otrzTekstDl=recv(klientGniazdo,echoBufor,BUFWE,0);
    if (otrzTekstDl < 0)
      Zakoncz("recv()");
  }
  close(klientGniazdo);
}
```

4.9. Przykład wielowątkowego serwera współbieżnego

Wielowątkowy serwer współbieżny echo, wersja 1

- Porównaj z wieloprocesowym serwerem współbieżnym echa w 5.8. W przykładzie wykorzystane są te same funkcje

```
#include "TCPEchoSerwer.h"
#include <pthread.h>       /* Wątki POSIX */

void *Wykonaj(void *arg);  /* Główna funkcja wątku */

/* Argument przesyłany do głównej funkcji wątku */
struct WatekArg {
  int klientGniazdo; /* Deskryptor gniazda klienta */
};

int main(int argc, char *argv[])
{
  int serwGniazdo;   /* Deskryptor gniazda serwera */
  int klientGniazdo; /* Deskryptor gniazda klienta */
  unsigned short echoSerwPort;    /* Port serwera */

  pthread_t watekID;    /* Identyfikator wątku dla pthread_create() */
  struct WatekArg *watekArg;   /* Wskaźnik do argumentu
                                  przekazywanego do wątku */

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

  echoSerwPort = atoi(argv[1]);
  serwGniazdo = UtworzGniazdoTCP(echoSerwPort);
  for (;;) {
    klientGniazdo = AkceptujPolaczenieTCP(serwGniazdo);
    /* przydziel pamięć dla argumentu wątku */
    if ((watekArg =
       (struct WatekArg *) malloc(sizeof(struct WatekArg)))
                 == NULL)
       Zakoncz("malloc()");
    watekArg -> klientGniazdo = klientGniazdo;

    /* Utwórz wątek obsługujący klienta */
    if (pthread_create(&watekID, NULL, Wykonaj, (void *) watekArg) != 0)
      Zakoncz("pthread_create()");
    printf("watek %ld ", (long int) watekID);
    printf("proces %ld\n", (long int) getpid());
  }
}
```

```
/* Główna funkcja wątku */
void *Wykonaj(void *watekArg)
{
  int klientGniazdo;  /* Deskryptor gniazda klienta */

  /* Odłączenie wątku */
  pthread_detach(pthread_self());

  /* Wybranie deskryptora gniazda klienta
     z argumentu funkcji */
  klientGniazdo =
      ((struct WatekArg *) watekArg) -> klientGniazdo;
  free(watekArg);  /*Zwolnienie pamięci dla argumentu*/

  PrzetwarzajKlienta(klientGniazdo);

  return (NULL);
}
```

Przykład wielowątkowego serwera współbieżnego - echo, wersja 2

Porównaj z poprzednim przykładem.

```
/* Comer, Sieci komputerowe TCP/IP t.III - wersja Linux */

/* TCPmtechod.c - main, TCPechod, prstats */

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>

#include <sys/types.h>
#include <sys/signal.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
#include <sys/errno.h>
#include <netinet/in.h>

#define  QLEN      32  /* maximum connection queue length  */
#define  BUFSIZE    4096

#define  INTERVAL  5  /* secs */

struct {
  pthread_mutex_t  st_mutex;
  unsigned int  st_concount;
  unsigned int  st_contotal;
  unsigned long  st_contime;
  unsigned long  st_bytecount;
} stats;

void  prstats(void);
int  TCPechod(int fd);
int  errexit(const char *format, ...);
int  passiveTCP(const char *service, int qlen);

/*------------------------------------------------------------------------
 * main - Concurrent TCP server for ECHO service
 *------------------------------------------------------------------------
 */
int main(int argc, char *argv[])
{
  pthread_t  th;
  pthread_attr_t  ta;
  char  *service = "echo";  /* service name or port number  */
  struct  sockaddr_in fsin;  /* the address of a client  */
  unsigned int  alen;    /* length of client's address  */
  int  msock;      /* master server socket    */
  int  ssock;      /* slave server socket    */
```

```
  switch (argc) {
  case  1:
    break;
  case  2:
    service = argv[1];
    break;
  default:
    errexit("usage: TCPechod [port]\n");
  }

  msock = passiveTCP(service, QLEN);

  (void) pthread_attr_init(&ta);
  (void) pthread_attr_setdetachstate(&ta, PTHREAD_CREATE_DETACHED);
  (void) pthread_mutex_init(&stats.st_mutex, 0);

  if (pthread_create(&th, &ta, (void * (*)(void *))prstats, 0) < 0)
    errexit("pthread_create(prstats): %s\n", strerror(errno));

  while (1) {
    alen = sizeof(fsin);
    ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
    if (ssock < 0) {
      if (errno == EINTR)
        continue;
      errexit("accept: %s\n", strerror(errno));
    }
    if (pthread_create(&th, &ta, (void * (*)(void *))TCPechod,
        (void *)ssock) < 0)
      errexit("pthread_create: %s\n", strerror(errno));
  }
}

/*------------------------------------------------------------------------
 * TCPechod - echo data until end of file
 *------------------------------------------------------------------------
 */
int TCPechod(int fd) {
  time_t  start;
  char  buf[BUFSIZ];
  int  cc;

  start = time(0);
  (void) pthread_mutex_lock(&stats.st_mutex);
  stats.st_concount++;
  (void) pthread_mutex_unlock(&stats.st_mutex);
  while (cc = read(fd, buf, sizeof buf)) {
    if (cc < 0)
      errexit("echo read: %s\n", strerror(errno));
    if (write(fd, buf, cc) < 0)
      errexit("echo write: %s\n", strerror(errno));
    (void) pthread_mutex_lock(&stats.st_mutex);
    stats.st_bytecount += cc;
    (void) pthread_mutex_unlock(&stats.st_mutex);
  }
  (void) close(fd);
  (void) pthread_mutex_lock(&stats.st_mutex);
  stats.st_contime += time(0) - start;
  stats.st_concount--;
  stats.st_contotal++;
  (void) pthread_mutex_unlock(&stats.st_mutex);
  return 0;
}
```

```
/*-------------------------------------------------------------------------
 * prstats - print server statistical data
 *-------------------------------------------------------------------------
 */
void prstats(void)
{
  time_t  now;

  while (1) {
    (void) sleep(INTERVAL);

    (void) pthread_mutex_lock(&stats.st_mutex);
    now = time(0);
    (void) printf("--- %s", ctime(&now));
    (void) printf("%-32s: %u\n", "Current connections", stats.st_concount);
    (void) printf("%-32s: %u\n", "Completed connections",
      stats.st_contotal);
    if (stats.st_contotal) {
      (void) printf("%-32s: %.2f (secs)\n",
        "Average complete connection time",
        (float)stats.st_contime / (float)stats.st_contotal);
      (void) printf("%-32s: %.2f\n",
        "Average byte count",
        (float)stats.st_bytecount / (float)(stats.st_contotal +
        stats.st_concount));
    }
    (void) printf("%-32s: %lu\n\n", "Total byte count",
      stats.st_bytecount);
    (void) pthread_mutex_unlock(&stats.st_mutex);

  }
}
```

4.10. Przykład jednoprocesowego serwera współbieżnego

Jednoprocesowy współbieżny serwer echo – wersja 1

```
#include "TCPEchoSerwer.h"
#include <sys/time.h>       /* dla struct timeval {} */
#include <fcntl.h>          /* dla fcntl() */

int main(int argc, char *argv[])
{
  int serwGniazdo;
  int klientGniazdo;
  int maxDeskryptor;  /* Liczba sprawdzanych deskryptorów */
  fd_set gniazdoC;    /* Zbiór przeglądanych deskryptorów */
  fd_set gniazdoS;    /* Zbiór deskryptorów dla select() */
  long timeout;       /* Timeout */
  struct timeval selTimeout;   /* Timeout dla select() */
  int wykonuj = 1;  /* 1 jesli serwer ma działać, 0 w przeciwnym wypadku */
  unsigned short portNo;    /* Port serwera */

  int x;
  if (argc < 3) {
    fprintf(stderr, "Uzycie:  %s <Timeout (sek.)> <Port>\n", argv[0]);
    exit(1);
  }
  timeout = atol(argv[1]);    /* Pierwszy arg: Timeout */
  portNo = atoi(argv[2]);     /* Drugi arg: Port */
  serwGniazdo = UtworzGniazdoTCP(portNo);

/* Przygotuj początkowy zbiór deskryptorów dla select() */
  FD_ZERO(&gniazdoC);
  FD_SET(STDIN_FILENO, &gniazdoC);
  FD_SET(serwGniazdo, &gniazdoC);
  maxDeskryptor = FD_SETSIZE;      /* z dużym nadmiarem */

  printf("Uruchamiam serwer:  nacisnij Return aby zakonczyc prace\n");

  while (wykonuj)
  {
   /* Trzeba na nowo zdefiniwoać przed każdym wywołaniem select() */
    selTimeout.tv_sec = timeout;       /* timeout (sek.) */
    selTimeout.tv_usec = 0;            /* 0 mikrosek. */
    gniazdoS=gniazdoC;

    /* Blokuj, dopóki nie jest gotowy deskryptor lub upłynął timeout */
    if (select(maxDeskryptor, &gniazdoS, NULL, NULL, &selTimeout) == 0)
      printf("Nic sie nie dzieje od %ld sek... Serwer zyje!\n", timeout);
    else
    {
      if (FD_ISSET(STDIN_FILENO, &gniazdoS)) /* sprawdź klawiature */
      {
        printf("Zamykam serwer\n");
        getchar();
        wykonuj = 0;
        continue;
      }
```

```
      /* sprawdź deskryptory gniazd */
      for (x= 0; x< FD_SETSIZE; x++)
        if ((x != STDIN_FILENO) && FD_ISSET(x, &gniazdoS))
        {
          if (x==serwGniazdo) {   /* zgłosił się nowy klient */
            klientGniazdo=accept(serwGniazdo,NULL,NULL);
            FD_SET(klientGniazdo,&gniazdoC);
            printf("Nowy klient otrzymał gniazdo %d\n", klientGniazdo);
          } else {
          printf("Czytam z gniazda %d\n",x);
          if ((PrzetwarzajKlienta(x)== 0)) {
            close (x);
            FD_CLR(x,&gniazdoC);
            printf("Klient z gniazda %d odłączony\n", klientGniazdo);
          }
        }
      } /* koniec for */
    }
  } /
  close(serwGniazdo);
  exit(0);
}
```

- Jak należy zmienić funkcję `PrzetwarzajKlienta` z poprzednich przykładów, aby klienci byli obsługiwani współbieżnie?

Jednoprocesowy serwra współbieżny echo, wersja 2

```
/* Comer, Sieci komputerowe TCP/IP t.III, s.180-182 */

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>

#define QLEN 5 // maksymalna długość kolejki połączeń
#define BUFSIZE  4096 // bufor wejściowy

extern int errno;
int  errexit(const char *format, ...);
int  passiveTCP(const char *service, int qlen);
int  echo(int fd);

int main(int argc, char *argv[])
{
  char *service = "echo"; // nazwa usługi
  struct sockaddr_in fsin; // adres klienta
  int msock;   // gniazdo nasłuchujące serwera
  fd_set rfds;   // zbiór deskryptorów do czytania
  fd_set afds;   // zbiór deskryptorów aktywnych
  int alen;      // długość adresu nadawcy
  int i, nfds;  // zmienne pomocnicze

  switch (argc) {
  case 1:
    break;
  case 2:
    service = argv[1];
    break;
  default:
    errexit("użycie: TCPmechod [port]\n");
  }
// utwórz gniazdo bierne dla serwera typu TCP
// (opis funkcji: Comer t.III, str.161
// argumenty: nazwa lub numer usługi (w postaci tekstu)
//            długość kolejki zgłoszeń klientów
// wynik: deskryptor gniazda nasłuchującego
  msock = passiveTCP(service, QLEN);

// określ maksymalną liczbę przeglądanych deskryptorów
  nfds = getdtablesize();
// przypisz wszystkim deskryptorom wartość 0
  FD_ZERO(&afds);
// ustaw bit
// związany z deskryptorem gniazda nasłuchującego
  FD_SET(msock, &afds);

  while (1) {

// utwórz maskę rdfs przeglądanych deskryptorów
    memcpy(&rfds, &afds, sizeof(rfds));
```

```
// sprawdź, czy są deskryptory gotowe do przetwarzania
    if (select(nfds, &rfds, NULL, NULL, NULL) < 0)
      errexit("select: %s\n", strerror(errno));
// po wykonaniu select rfds zawiera tylko
// deskryptory gotowe do czytania

// sprawdź, czy zgłosił się nowy klient i utwórz
// dla niego gniazdo połączeniowe
    if (FD_ISSET(msock, &rfds)) {
      int ssock;
      alen = sizeof(fsin);
      ssock = accept(msock, (struct sockaddr *)&fsin,
                                                &alen);
      if (ssock < 0)
        errexit("accept: %s\n", strerror(errno));
      // dodaj gniazdo do zbioru aktywnych deskryptorów
      FD_SET(ssock, &afds);
    }

// Przeglądaj kolejne gniazda i dla każdego gotowego
// gniazda odbierz zapytanie, przetwórz je i odeślij
    for (i=0; i<nfds; ++i)
      if ( (i != msock) && FD_ISSET(i, &rfds))
        // echo() przetwarza zapytanie skierowane
        // do i-tego klienta; zwrócenie 0 oznacza,
        // że klient zakończył połączenie
        if (echo(i) == 0) {
          close(i);
          FD_CLR(i, &afds);
        }
  } // pętla while
}

/*-------------------------------------------------------
      echo - odsyła echo przesłanych danych,
      zwraca liczbę przeczytanych bajtów
*-------------------------------------------------------*/
int echo(int fd)
{
  char buf[BUFSIZ];
 int cc;

  cc = read(fd, buf, sizeof buf);
  if (cc < 0)
    errexit("echo read: %s\n", strerror(errno));
  if (cc && write(fd, buf, cc) < 0)
    errexit("echo write: %s\n", strerror(errno));
  return cc;
}
```

4.11. Przykład serwera wieloprotokołowego

- Wykorzystanie mechanizmów zwielokrotnionego wejścia-wyjścia w serwerze wieloprotokołowym - funkcja `select` (usługa DAYTIME, protokół TCP lub UDP w jednym programie): Comer, Stevens "Sieci komputerowe", tom 3, str. 187-190)

4.12. Przykład serwera wielousługowego

- Wykorzystanie mechanizmów zwielokrotnionego wejścia-wyjścia w serwerze wielousługowym (usługi echo, chargen, daytime, time w jednym programie): Comer, Stevens "Sieci komputerowe", tom 3, str. 200-206)

4.13. Inne właściwości serwera

Uruchomienie jako demon

- Demon jest to program, który jest wykonywany w tle i nie jest związany z żadnym terminalem. Ma działać bez żadnej interakcji z użytkownikiem i zazwyczaj jest uruchomiony tak długo, jak długo działa system. Aby zrealizować te funkcje program:
    - powinien zamknąć wszystkie odziedziczone deskryptory, aby zapobiec niepotrzebnemu przetrzymywaniu zasobów
    - musi odłączyć się od terminala sterującego, aby na jego działanie nie miały wpływu sygnały generowane przez terminal użytkownika
    - powinien zmienić katalog bieżący na taki, w którym będzie mógł działać nieograniczenie długo nie utrudniając zarządzania systemem
    - powinien zmienić wartość maski umask zgodnie ze swoimi wymaganiami
    - musi odłączyć się od grupy, aby nie otrzymywać sygnałów przeznaczonych dla tej grupy

- Przykład prostej funkcji demon:

```
#include <sys/types.h>
#include <unistd.h>
#include <syslog.h>
#include <sys/stat.h>
#include <stdlib.h>

int demon() {
  pid_t pid;
  long ldes;
  int i;

/* Krok 1 – wywołaj funkcję fork() i zakończ proces macierzysty */
  if ((pid = fork()) != 0) { exit(0); }

/* Krok 2 – utwórz nową grupę procesów i nową sesji. Uczyń proces jedynym
   członkiem tych grup i jednocześnie przywódcą tych grup. W ten sposób
   pozbędziesz się terminala sterującego */

  setsid();

/* Krok 3 – ponownie wywołaj funkcję fork() i zakończ proces macierzysty.
   Nie ma już przywódcy grupy, nie można zostać przyłączonym
   do terminala sterującego */

  if ((pid = fork()) != 0) { exit(0);  }

/* Krok 4 – wybierz jako katalog bieżący "bezpieczny" katalog */

  chdir("/");

/* Krok 5 – ustal wartość maski */

  umask(0);

/* Krok 6  - zamknij odziedziczone po procesie macierzystym deskryptory
   plików */
  ldes = sysconf(_SC_OPEN_MAX);
  for (i = 0; i < ldes; i++) {
    close(i);
  }

  return 1;
}
```

Zapisy do logów

- Serwer może posiadać własne pliki logów lub korzystać z logów systemowych.
- W systemach uniksowych tradycyjnym programem obsługi logów jest `syslog`. Przykład zapisu za pośrednictwem syslog'a:

```
int main(int argc, char **argv)
{
  demon();

/* otwórz połączenie */
  openlog("test_sewera", LOG_PID, LOG_USER);

/* wyślij komunikat */
  syslog(LOG_INFO, "%s", "Uruchomiono!");

/* zamknij połączenie */
  closelog();

  return 1;
}
```

Uzyskany wpis w `/var/log/messages`:

```
Mar 10 18:31:34 blade-runner test_serwera[10289]: Uruchomiono!
```

Zmniejszenie uprawnień

- Zasada: takie przywileje, jakie są niezbędne do wykonywania pracy.
- Jeśli serwer potrzebuje uprawnień root'a tylko do wykonania czynności początkowych po uruchomieniu, można po ich wykonaniu odebrać mu te uprawnienia.
- Do jednoczesnej zmiany wszystkich identyfikatorów użytkownika (rzeczywistego, efektywnego) służy funkcja `setuid()`. Musi być wywołana, gdy właścicielem procesu jest jeszcze root.

```
int main(int argc, char **argv)
{
  struct passwd *pws;
  const char *user = "nopriv";

  pws = getpwnam(user);
  if (pws == NULL) {
    printf("Nieznany użytkownik: %s\n", user);
    return 0;
  }

  demon();

/* Zmień ID uzytkownika na nopriv */
  setuid(pws->pw_uid);

  while (1) {
    sleep(1);
  }

  return 1;
}
```

- Można również ograniczyć dostęp do systemu plików. Za pomocą funkcji `chroot()` można określić katalog, który staje się katalogiem głównym dla danego procesu. Proces będzie miał dostęp tylko do plików znajdujących się poniżej nowego głównego katalogu. Funkcja wymaga uprawnień roota. Uwaga: funkcja `chroot()` nie zmienia bieżącego katalogu.

Wzajemne wykluczanie egzemplarzy serwera

- Zasada: nie powinno się inicjować działania więcej niż jednej kopii serwera w danym czasie; ewentualna współbieżność powinna być realizowana przez sam serwer.

- Przykład:

```
#define LOCKF /var/spool/serwer.lock
lf=open(LOCKF, O_RWDR|O_CREAT, 0640);
if (lf < 0)  /* błąd podczas otwierania pliku */
  exit(1);
if (flock(lf, LOCK_EX|LOCK_NB))
  exit(0);   /* nie udało się zablokować pliku */
```

Zarejestrowanie identyfikatora procesu serwera

- Serwer często zapamiętuje identyfikator swojego procesu w pliku o ustalonej nazwie. Dzięki temu można szybko odnaleźć ten identyfikator, bez potrzeby przeglądania listy wszystkich procesów działających w systemie.
- Takim plikiem może być na przykład plik blokady serwera.
- Przykład:

```
char pbuf[10]; /* pid w postaci napisu */
/* zamień liczbę binarną na dziesiętną */
sprintf(pbuf,"%6d\n",getpid());
/* plik blokady jest już otwarty */
write(lf,pbuf,strlen(pbuf));
```

Należy przeczytać:

Douglas E. Comer, David L. Stevens: *Sieci komputerowe TCP/IP, tom 3*: str. 130-206

W. Richard Stevens: *Unix,* programowanie *usług sieciowych, tom 1: API gniazda i XTI*: str. 129-135, 182-189, 671-707