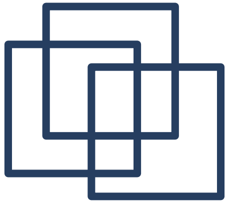
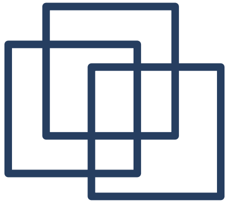


Serwery współbieżne

1. Serwery współbieżne
- serwery połączeniowe, usuwanie zakończonych procesów,
- serwery bezpołączeniowe,

2. Jednoprocesowe serwery współbieżne.
- koncepcja i implementacja.

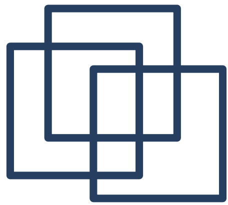


Serwer współbieżny

Zasadniczym powodem stosowania mechanizmu współbieżności w serwerach jest potrzeba zapewnienia krótkiego czasu odpowiedzi w warunkach obsługi wielu klientów. Współbieżność skraca obserwowany czas odpowiedzi gdy:

- przygotowanie odpowiedzi wymaga czasochłonnych operacji wejścia – wyjścia,
- występują znaczne różnice czasu przetwarzania dla różnych zgłoszeń,
- serwer działa na komputerze wieloprocesorowym.

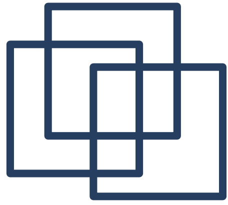
Wyższa wydajność jest uzyskiwana zwykle przez zrównoleglenie przetwarzania danych z operacjami wejścia – wyjścia.



Serwer współbieżny połączeniowy

<u>Proces główny:</u>

1. Utwórz gniazdo i zwiąż je z powszechnie znanym adresem odpowiadającym usłudze udostępnionej przez serwer – **`bind()`**.
2. Ustaw bierny tryb pracy gniazda – **`listen()`**.
3. Przyjmij zgłoszenie połączenia nadesłane na adres gniazda - **`accept()`**. Utwórz nowy proces podporządkowany – **`fork()`** odpowiedzialny za obsługę tego połączenia.
4. Wróć do kroku 3.

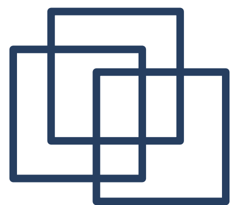


Serwer współbieżny połączeniowy

Proces podporządkowany:

1. Przejmij od procesu głównego nawiązane połączenie.
2. Korzystając z otrzymanego gniazda prowadź interakcję z klientem zgodnie z protokołem warstwy aplikacji – **`read()`**, **`write()`**.
3. Zwolnij gniazdo – **`close()`** i zakończ działanie – **`exit()`**.

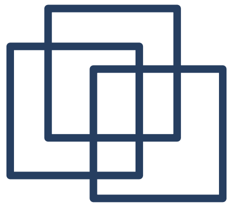
Współbieżność w działaniu serwerów typu połączeniowego polega na zrównolegleniu obsługi wielu połączeń, a nie poszczególnych zapytań.



Serwer współbieżny połączeniowy

Schemat struktury współbieżnego serwera połączeniowego.

Proces główny przyjmuje zgłoszenia połączeń. Do obsługi każdego połączenia tworzony jest proces podporządkowany.



Usuwanie procesów po ich zakończeniu

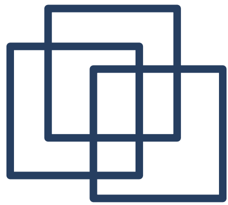
Procesy potomne po obsłużeniu klienta kończą pracę. W systemach UNIX powinny one dodatkowo zostać zakończone przez proces macierzysty. W przeciwnym razie będą nadal egzystować w systemie.

W chwili zakończenia procesu potomnego proces macierzysty otrzymuje sygnał **SIGCHILD**. Dzięki temu może on zakończyć proces potomny używając instrukcji

```
signal(SIGCHILD, gc);
```

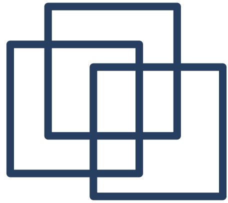
gdzie **gc** jest wskaźnikiem do przykładowej funkcji:

```
void gc(int i){
    while(waitpid(-1, NULL, WNOHANG)<=0)
        ;
}
```



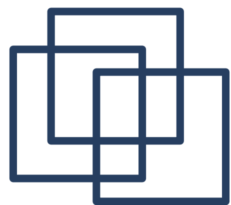
Serwer współbieżny połączeniowy

```
svr_con_echo_tcp(){
    int s1, s2, alen;
    struct sockaddr_in sin;

    if((s1=passivesock(TESTPORT, "tcp", 10))<0){
        fprintf(stderr, "passivesock: %s\n", strerror(errno));
        return -1;
    }
    signal(SIGCHLD, gc);
    while(1){
        alen = sizeof(sin);
        if((s2=accept(s1, (struct sockaddr *)&sin, &alen))<0){
            if(errno==EINTR){
                continue;
            }
            fprintf(stderr, "accept: %s\n", strerror(errno));
            return -1;
        }
```



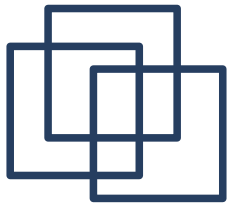
Serwer współbieżny połączeniowy

```
        switch(fork()){
            case 0: // proces potomny
                close(s1);
                echod(s2);
                if (close(s2)<0){
                    fprintf(stderr, "close: %s\n",
                            strerror(errno));
                    return -1;
                }
                exit(1);
            default: // proces macierzysty
                close(s2);
                break;
            case -1:
                fprintf(stderr, "fork: %s\n", strerror(errno));
                return -1;
        }// switch
    }
}
```



Serwer współbieżny połączeniowy

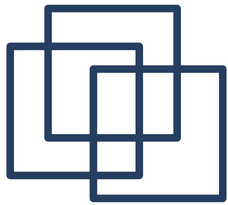
Współbieżne serwery połączeniowe jednocześnie komunikują się z wieloma klientami. W zaprezentowanym przykładzie proces główny tworzy nowy proces podporządkowany dla każdego zgłoszonego połączenia. Proces potomny obsługuje klienta po czym zamyka połączenie i kończy działanie. Proces główny bezpośrednio nie kontaktuje się z klientami.



Serwer współbieżny bezpołączeniowy

Proces główny

1. Utwórz gniazdo i zwiąż je z powszechnie znanym adresem odpowiadającym usłudze udostępnionej przez serwer - **`bind()`**.
2. Ustaw bierny tryb pracy gniazda - **`listen()`**.
3. Odbierz kolejne zapytanie od klientów - **`recvfrom()`**. Utwórz nowy proces podporządkowany – **`fork()`**, który przygotuje odpowiedź.
4. Przejdź do punktu 3.

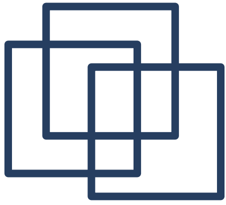


Serwer współbieżny bezpołączeniowy

Proces podporządkowany:

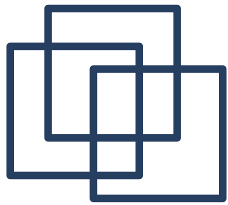
1. Przejmij od procesu głównego dostęp do gniazda.
2. Skonstruuj odpowiedź zgodnie z używanym protokołem i wyślij ją do klienta – **`sendto()`**.
3. Zakończ działanie – **`exit()`**.

Z powodu znacznego kosztu operacji tworzenia nowego procesu istnieje niewiele współbieżnych realizacji serwerów bezpołączeniowych.



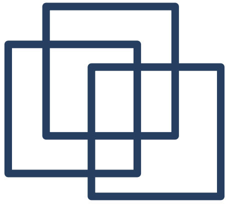
Serwer współbieżny bezpołączeniowy

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;


public class EchoConcurentUDPServer {

   private static final int LINELEN = 100;

   public static void main(String[] args) {
       if (args.length<1){
           System.out.println(
                       "wywolanie java EchoConcurentUDPServer port");
           return;
       }
       DatagramPacket p;
       DatagramSocket s=null;
```



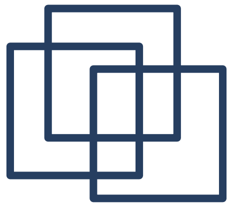
Serwer współbieżny bezpołączeniowy

```
        try {
          s = new DatagramSocket(Integer.parseInt(args[0]));
          while(true){
              p = new DatagramPacket(new byte[LINELEN], LINELEN);
              s.receive(p);
              Thread t = new Thread(new Worker(s,p));
              t.start();
              System.out.println("otrzymano :" +
                      new String(p.getData(), 0, p.getLength()));
          }
      } catch (Exception e) {
              e.printStackTrace();
      } finally {
          if (s!=null){
              s.close();
          }
      }
  }
```



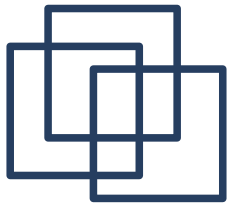
Serwer współbieżny bezpołączeniowy

```
private static class Worker implements Runnable{
  private DatagramPacket request;
  private DatagramSocket socket;

  /**
   * Konstruktor
   * @param ds gniazdo uzywane do transmisji
   * @param dp pakiet zawierający żadanie obsługi
   */
  public Worker(DatagramSocket ds, DatagramPacket dp){
      this.socket = ds;
      this.request = dp;
  }
```



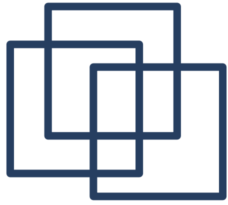
Serwer współbieżny bezpołączeniowy

```
        public void run() {
          byte[] ba = this.request.getData();
          int length = this.request.getLength();
          InetAddress ia = this.request.getAddress();
          int port = this.request.getPort();
          DatagramPacket response = new
                    DatagramPacket(ba, 0, length, ia, port);
          try {
              this.socket.send(response);
              System.out.println("wyslano :" +
                    new String(response.getData(),
                                0, response.getLength()));
          } catch (IOException e) {
              e.printStackTrace();
          }
      }
  }
}
```



Pozorna współbieżność w jednym procesie

1. Utwórz gniazdo i zwiąż je z powszechnie znanym adresem odpowiadającym usłudze udostępnionej przez serwer – **`bind()`** oraz **`listen()`**.
2. Czekaj na zdarzenia dotyczące istniejących gniazd.
3. W razie gotowości pierwotnie utworzonego gniazda przyjmij zgłoszenie połączenia nadesłane na adres gniazda - **`accept()`**. Dodaj nowe gniazdo do listy obsługiwanych gniazd.
4. W razie gotowości innego gniazda używaj funkcji **`read()`** oraz **`write()`** w celu komunikacji z wcześniej połączonym klientem.
5. Wróć do punktu 2.

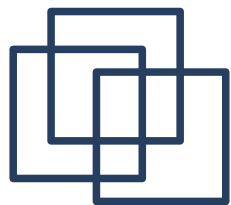


Pozorna współbieżność w jednym procesie

Pozorna współbieżność może być stosowana jeśli:

- korzyści z rzeczywistej współbieżności są mniejsze niż koszt tworzenia nowego procesu,
- kilka połączeń jest obsługiwane z wykorzystaniem wspólnego zbioru danych,
- dane są przekazywane pomiędzy niezależnymi połączeniami.

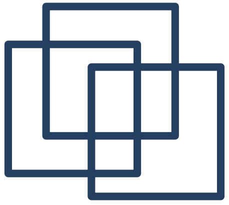
Przykład: X-Windows.



Jednoprocesowe serwery współbieżne

Schemat struktury współbieżnego serwera połączeniowego.

Proces główny przyjmuje zgłoszenia połączeń; do obsługi każdego połączenia tworzony wykorzystywane jest osobne gniazdo.



Sprawdzenie stanu gniazd

W celu wybrania gniazda, do którego przyszedł komunikat można użyć funkcji **`select()`** zadeklarowanej w pliku **`unistd.h`**:

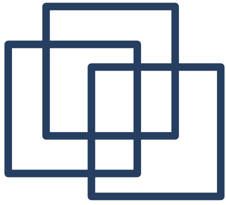
```
int select(int n, fd_set *readfds, fd_set *writefds, fd_set
  *exceptfds, struct timeval *timeout);
```

`readfds, writefds, exceptfds` – zbiory obserwowanych deskryptorów ze względu na gotowość do odczytu, zapisu oraz wystąpienia wyjątków. Po wykonaniu funkcji wskaźniki zostaną wypełnione zbiorami deskryptorów, dla których zaszło odpowiednie zdarzenie.

`n` – największy deskryptor ze wszystkich trzech zbiorów plus **`1`**.

`timeout` – maksymalny czas oczekiwania na powrót z funkcji. **`0`** – powrót natychmiastowy, **`NULL`** – potencjalnie nieskończony czas oczekiwania.

Zwraca: liczbę znalezionych deskryptorów lub -1 w przypadku błędu.



Sprawdzanie stanu gniazd

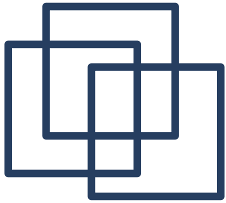
W pliku **`sys/types.h`** zdefiniowano cztery makra przeznaczone do operowania na zbiorach deskryptorów:

`FD_ZERO(fd_set *set)` – usuwa wszystkie deskryptory ze zbioru **`*set`**,

`FD_SET(int fd, fd_set *set)` – dodaje deskryptor **`fd`** do zbioru **`*set`**,

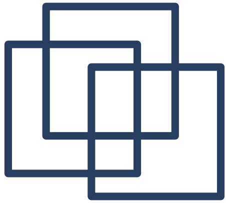
`FD_CRL(int fd, fd_set *set)` – usuwa deskryptor **`fd`** ze zbioru **`*set`**,

`FD_ISSET(int fd, fd_set *set)` – sprawdza, czy deskryptor **`fd`** znajduje się w zbiorze **`*set`**. Zwykle używane po wykonaniu funkcji **`select()`**.



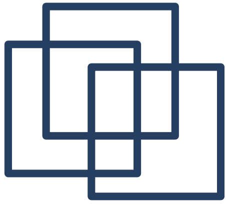
Przykład: C

```
svr_pseudocon_echo_tcp(){
     int s0, s, maxs, alen;
     struct sockaddr_in sin;
     fd_set   afds,        //zbiór aktywnych deskryptorów
              rdfs;        //zbiór znalezionych deskryptorów w
                              funkcji select()

     if((s0=passivesock(TESTPORT, "tcp", 10))<0){
        fprintf(stderr, "passivesock: %s\n", strerror(errno));
        return -1;
     }
     maxs = s0;
     FD_ZERO(&afds);
     FD_SET(s0, &afds);
```



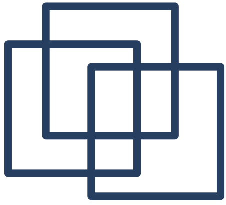
Przykład: C

```
while(1){
   bcopy((char *)&afds, (char *)&rdfs, sizeof(afds));
   if (select(maxs+1, &rdfs, NULL, NULL, 0)<0){
      fprintf(stderr, "select: %s\n", strerror(errno));
      return -1;
   }
   if (FD_ISSET(s0, &rdfs)){ //nowe połączenie
      alen = sizeof(sin);
      if((s=accept(s0, (struct sockaddr *)&sin, &alen))<0){
         fprintf(stderr, "accept: %s\n",
                  strerror(errno));
         return -1;
      }
      FD_SET(s, &afds);
      if (s>maxs)
         maxs = s;
   } // if
```



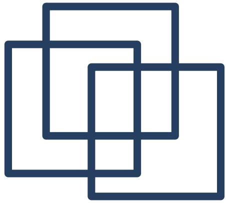
Przykład: C

```
        for (s=0; s<=maxs; s++){ // nawiązane połączenia
            if(s!=s0 && FD_ISSET(s, &rdfs)){
                echod(s);
                if (close(s)<0){
                    fprintf(stderr, "close: %s\n",
                                    strerror(errno));
                    return -1;
                }
                FD_CLR(s, &afds);
            }
        }//for
    }//while
}
```



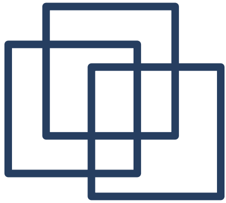
Przykład: Java

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketTimeoutException;
import java.util.Enumeration;
import java.util.Vector;

public class EchoPseudoConcurentTCPServer {
    private static final int LINELEN = 100;
    public static void main(String args[]){
        Socket s;
        OutputStream os;
        InputStream is;
        int i;

        if (args.length<1){
            System.out.println("wywolanie java
                                EchoPseudoConcurentTCPServer port");
            return;
        }
```



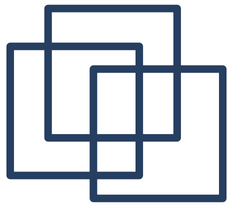
Przykład: Java

```
byte[] buffer = new byte[LINELEN];
Vector v = new Vector();
Enumeration e;
try {
    ServerSocket ss = new ServerSocket(
                                Integer.parseInt(args[0]));
    ss.setSoTimeout(1);
    v.clear();
    while(true){
        try{
            s = ss.accept();
        }catch (SocketTimeoutException ex){
            s = null;
        }
        if (s!=null){
            s.setSoTimeout(1);
            v.add(s);
        }
```



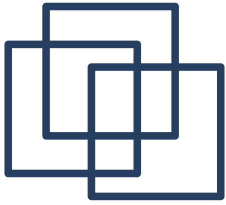
Przykład: Java

```
                    for(e=v.elements(); e.hasMoreElements(); ){
                      s = (Socket)e.nextElement();
                      is = s.getInputStream();
                      os = s.getOutputStream();
                      try{
                          i = is.read(buffer);
                      } catch(SocketTimeoutException ex){ continue; }
                      if (i>0){
                          os.write(buffer, 0, i);
                          System.out.println("wyslano :" +
                                                    new String(buffer, 0, i));
                      }
                  }
              }
          } catch (IOException ex) {
              ex.printStackTrace();
          }
      }
  }
```



Jednoprocesowe serwery współbieżne

Serwery jednoprocesowy wykonuje wszystkie zadania zarówno procesu głównego jak i procesów podporządkowanych serwera wieloprocesowego. Po zgłoszeniu gotowości przez gniazdo główne, serwer nawiązuje nowe połączenie. Gdy jest gotowe do obsługi którekolwiek z pozostałych gniazd, serwer czyta zapytanie nadesłane przez klienta i odsyła odpowiedź.



Porównanie serwerów

Serwer iteracyjny czy współbieżny.

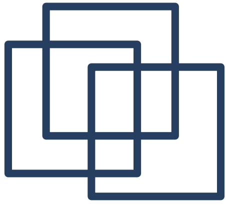
- iteracyjny - prostszy do zaprogramowania i konserwacji,
- współbieżny – krótszy czas oczekiwania na odpowiedź.

Współbieżność rzeczywista czy pozorna.

- rzeczywista – połączenia obsługiwane niezależnie,
- pozorna – niezależne połączenia korzystają lub wymieniają wspólne dane.

Serwer bezpołączeniowy czy połączeniowy.

- bezpołączeniowy – sieć lokalna, małe prawdopodobieństwo błędów transmisji,
- połączeniowy – wszystkie pozostałe zastosowania.



Podsumowanie

W ramach wykładu zostały zaprezentowane przykładowe implementacje podstawowych, omawianych wcześniej typów serwerów, z wykorzystaniem mechanizmów dostępnych w systemach UNIX.

Współbieżność w serwerach może być realizowana poprzez wielowątkowość / wieloprocesowość lub też poprzez utrzymywanie i obsługę wielu połączeń w pojedynczym wątku.