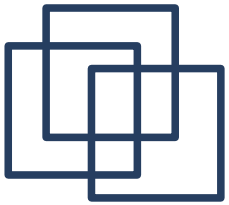
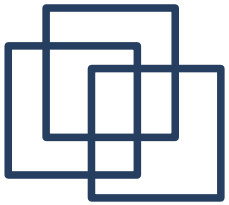


Serwery współbieżne c.d.

1. Serwery wielousługowe i wieloprotokołowe.
2. Sterowanie współbieżnością w serwerze
   - współbieżność sterowana zapotrzebowaniem,
   - alokacja wstępna procesów podporządkowanych.
3. Współbieżność w programach klienckich.

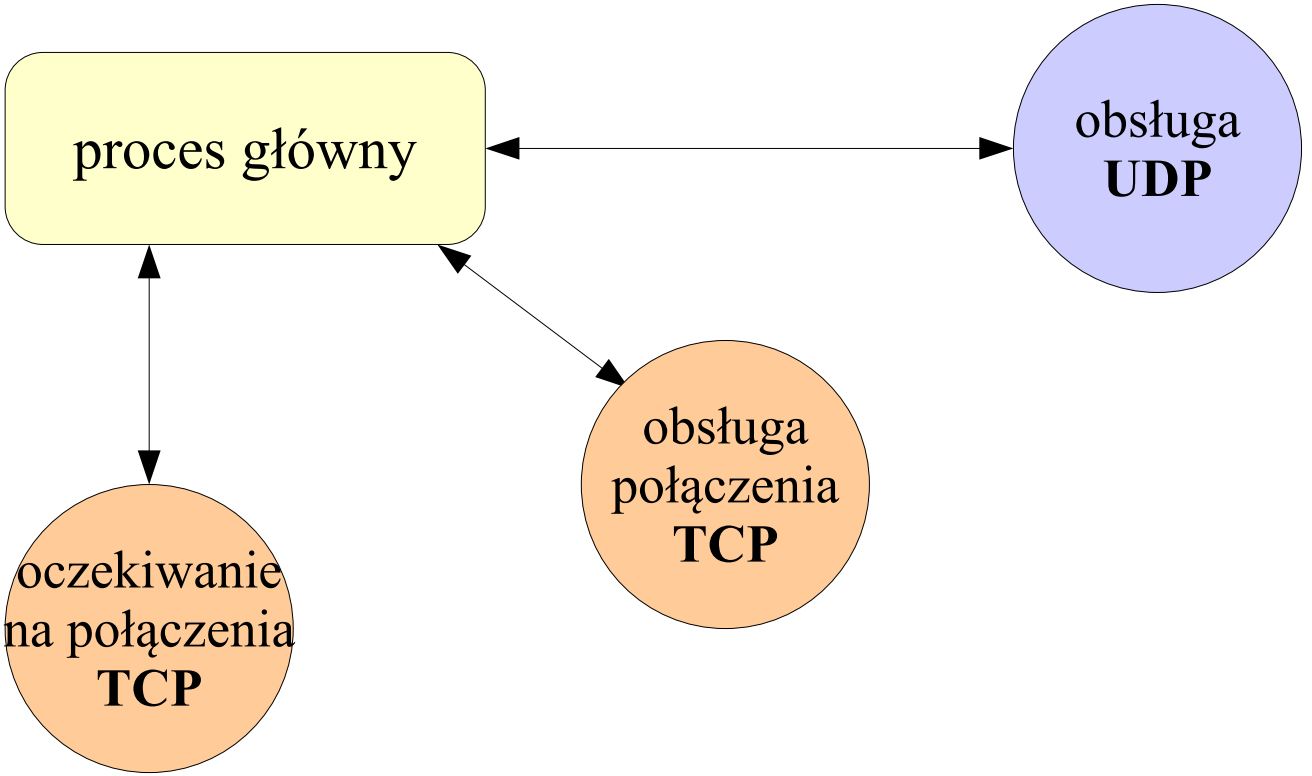


Serwery wieloprotokołowe (TCP i UDP)

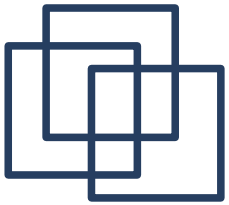
Serwery wieloprotokołowe udostępniają jedną usługę za pośrednictwem kilku protokołów warstwy transportowej. Podstawową zaletą tego rozwiązania jest łatwiejsze zarządzanie kodem realizującym usługę, np. przy zmianie wersji oprogramowania czy systemu operacyjnego, niezależnie od obsługiwanego protokołu (TCP lub UDP). Dodatkowo, serwer wieloprotokołowy zwykle potrzebuje mniej zasobów systemowych niż kilka serwerów jednoprotokołowych.

Serwery wieloprotokołowe

Schemat struktury serwera wieloprotokołowego (TCP i UDP).



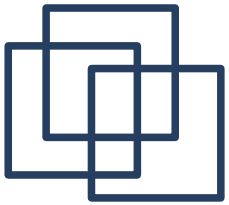
Proces główny przyjmuje zgłoszenia połączeń TCP; do obsługi połączenia wykorzystywane jest osobne gniazdo. Niezależnie obsługiwane są zgłoszenia UDP.



Serwery wieloprotokołowe

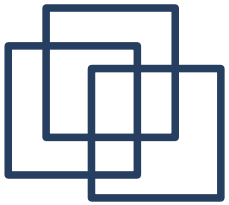
Poniższy przykład nie zawiera obsługi błędów zwracanych przez funkcje wykorzystywane do transmisji TCP/IP.

```
svr_echo_tcpudp(){
     int sTCP, sUDP, maxs, s;
     struct sockaddr_in sin;
     char buf[LINELEN];
     fd_set   afds,        //zbiór aktywnych deskryptorów

     sTCP = passivesock(TESTPORT, "tcp", 10);
     sUDP = passivesock(TESTPORT, "udp", 0);

     maxs = MAX(sTCP, sUDP);
     FD_ZERO(&afds);
     while(1){
        FD_SET(sTCP, &afds);
        FD_SET(sUDP, &afds);
```



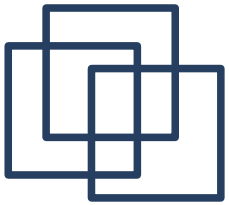
Serwery wieloprotokołowe

```
        select(maxs+1, &afds, NULL, NULL, 0);
        alen = sizeof(sin);
        if(FD_ISSET(sTCP, &afds)){ // obsługa TCP
           s = accept(sTCP, struct sockaddr*)&sin, &alen);
           daytimed(buf);
           write(s, buf, strlen(buf));
           close(s);
        }
        if(FD_ISSET(sUDP, &afds)){ // obsługa UDP
           recvfrom(sUDP, buf, sizeof(buf), 0,
              (struct sockaddr*)&sin, &alen);
           daytimed(buf);
           sendto(sUDP, buf, strlen(buf), 0,
              (struct sockaddr*)&sin, sizeof(sin));
        }
    }//while
}
```



Serwery wieloprotokołowe (TCP i UDP)

Serwer wieloprotokołowy stanowi rozwiązanie pozwalające skupić cały kod, realizujący określoną usługę, w jednym programie. Dzięki temu nie ma problemów z koordynacją zmian w przeciwieństwie do sytuacji gdy kod ten jest powielony w różnych programach. Serwer tego typu jest realizowany przez jeden proces, który tworzy główne gniazdo dla każdego z protokołów (TCP, UDP) po czym wywołuje funkcje **`select()`** w oczekiwaniu na gotowość jednego lub obydwu gniazd. Gdy gotowe jest gniazdo TCP, serwer nawiązuje połączenie z klientem i za jego pośrednictwem odpowiada na nadsyłane zapytania. Gdy gotowość zgłosi gniazdo UDP, serwer odczytuje zapytanie i wysyła odpowiedź



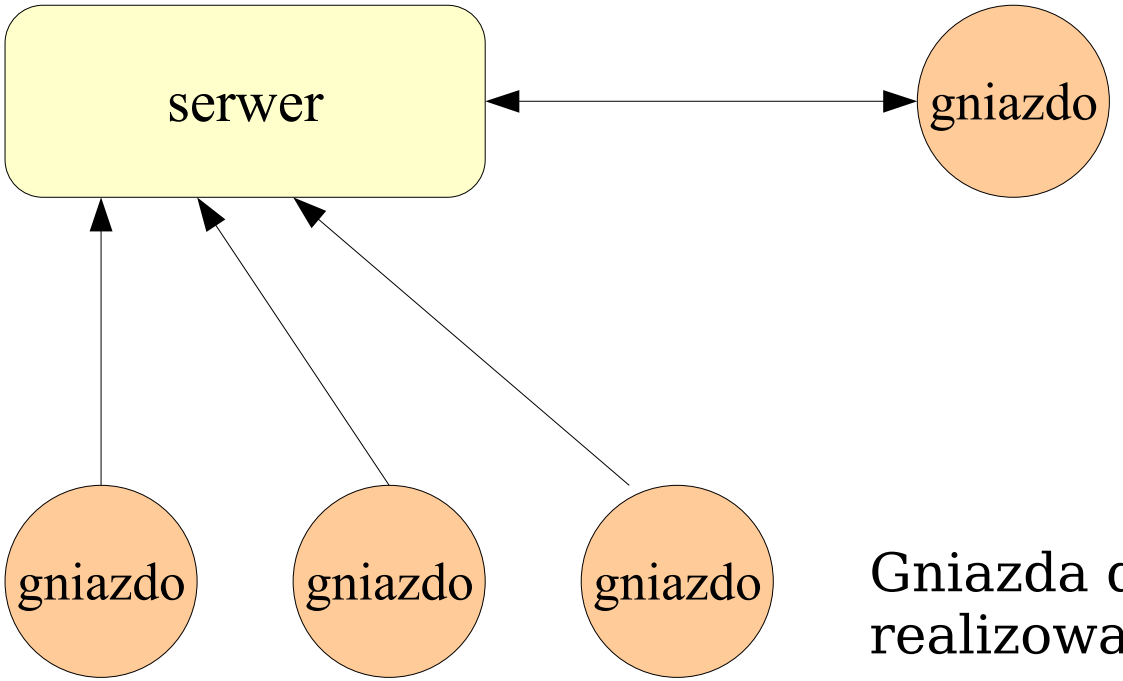
Serwery wielousługowe

Dla rodziny protokołów TCP/IP zdefiniowano duży zbiór tzw. prostych usług, pomocnych w administrowaniu, testowaniu i wykrywaniu błędów (np. DAYTIME, ECHO, TIME). W systemie, w którym dla każdej standartowej usługi istnieje oddzielny serwer, będą działać dziesiątki procesów, mimo że większość z nich zapewne nigdy nie otrzyma zgłoszenia. Dlatego połączenie wielu serwerów dla wielu różnych usług w jeden proces może radykalnie zmniejszyć liczbę aktywnych procesów.

UWAGA: system operacyjny może ograniczać maksymalną liczbę gniazd używanych w jednym procesie.

Budowa wielousługowego serwera połączeniowego

Schemat struktury iteracyjnego, połączeniowego serwera wielousługowego.



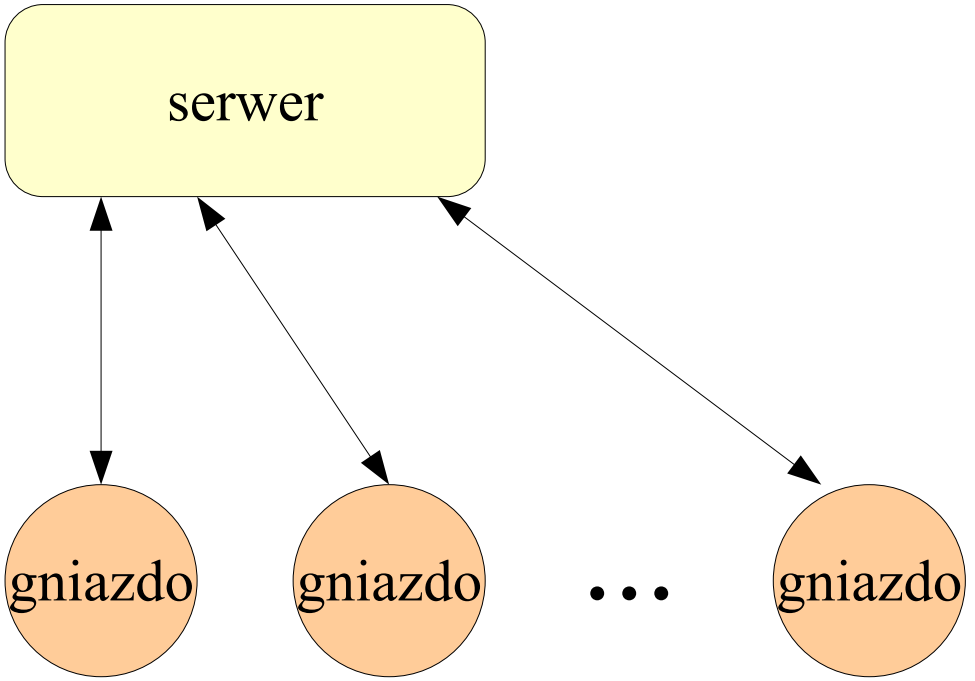
Gniazdo obsługujące określone połączenie.

Gniazda dla poszczególnych usług realizowanych przez serwer.

Serwer ma otwarte gniazdo dla każdej z usług i co najwyżej jedno do obsługi określonego połączenia.

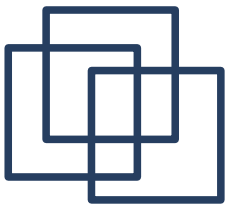
Budowa wielousługowego serwera bezpołączeniowego

Schemat struktury bezpołączeniowego serwera wielousługowego.



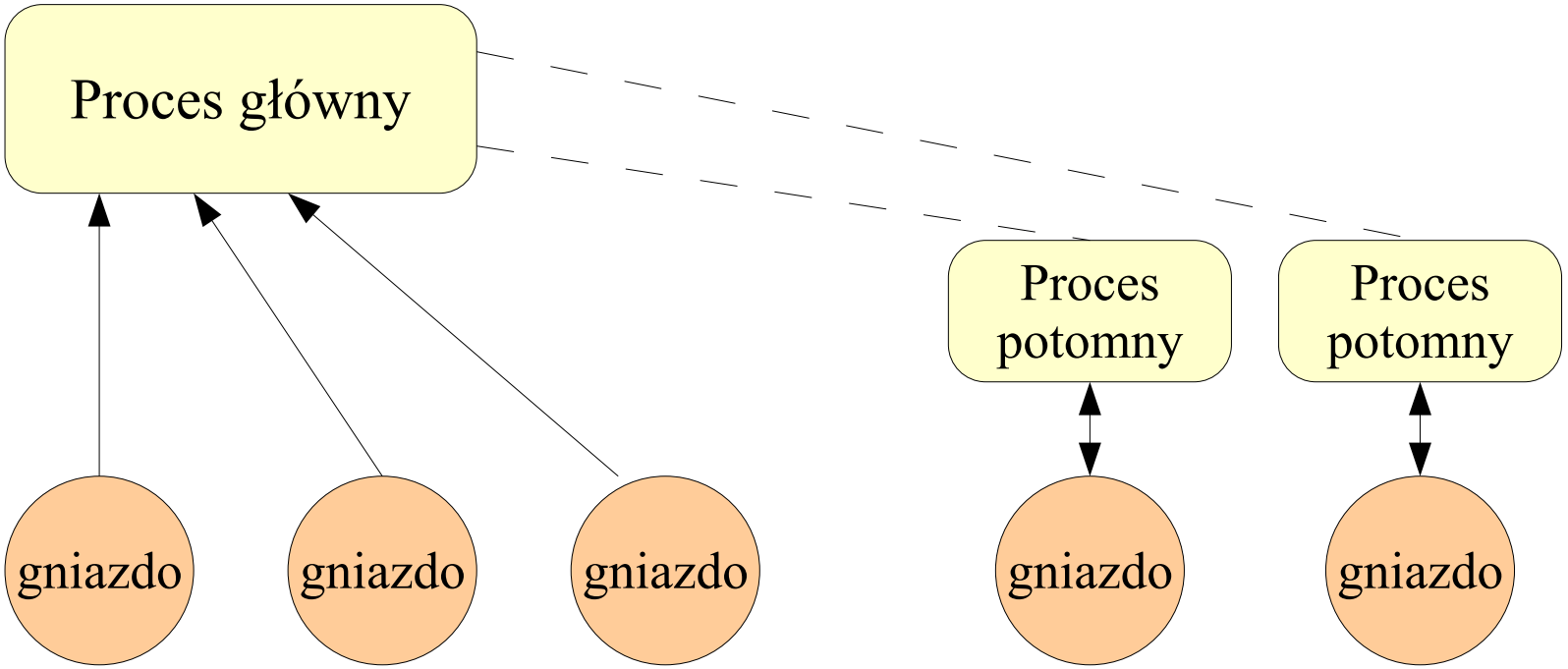
Gniazda dla poszczególnych usług realizowanych przez serwer.

Serwer czeka na nadejście datagramu do któregokolwiek z gniazd (**`select()`**). Każde gniazdo jest przypisane określonej usłudze.



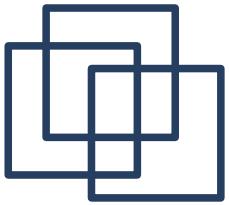
Współbieżny połączeniowy serwer wielousługowy

Schemat struktury współbieżnego, połączeniowego serwera wielousługowego.



Gniazda dla poszczególnych usług realizowanych przez serwer.

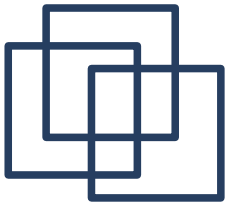
Gniazda dla poszczególnych połączeń obsługiwanych przez procesy potomne.



Serwery wielousługowe

Jedną z głównych wad dotychczas omówionych rozwiązań jest trudność wprowadzania zmian. Każda modyfikacja kodu realizującego jedną usługę wymaga ponownej kompilacji całego serwera, zatrzymania jego działania i uruchomienia nowej wersji.

W serwerze wielousługowym można oddzielić części kodu realizujące poszczególne usługi od kodu obsługującego wstępne zgłoszenia połączeń. Zabieg taki jest możliwy dzięki funkcji systemowej **`execve()`**.



Serwery wielousługowe

```
int execve(const char *filename, const char *argv[], const
  char *env[]);
```

filename – nazwa pliku z programem lub skryptem, który zostanie uruchomiony w miejsce istniejącego procesu,

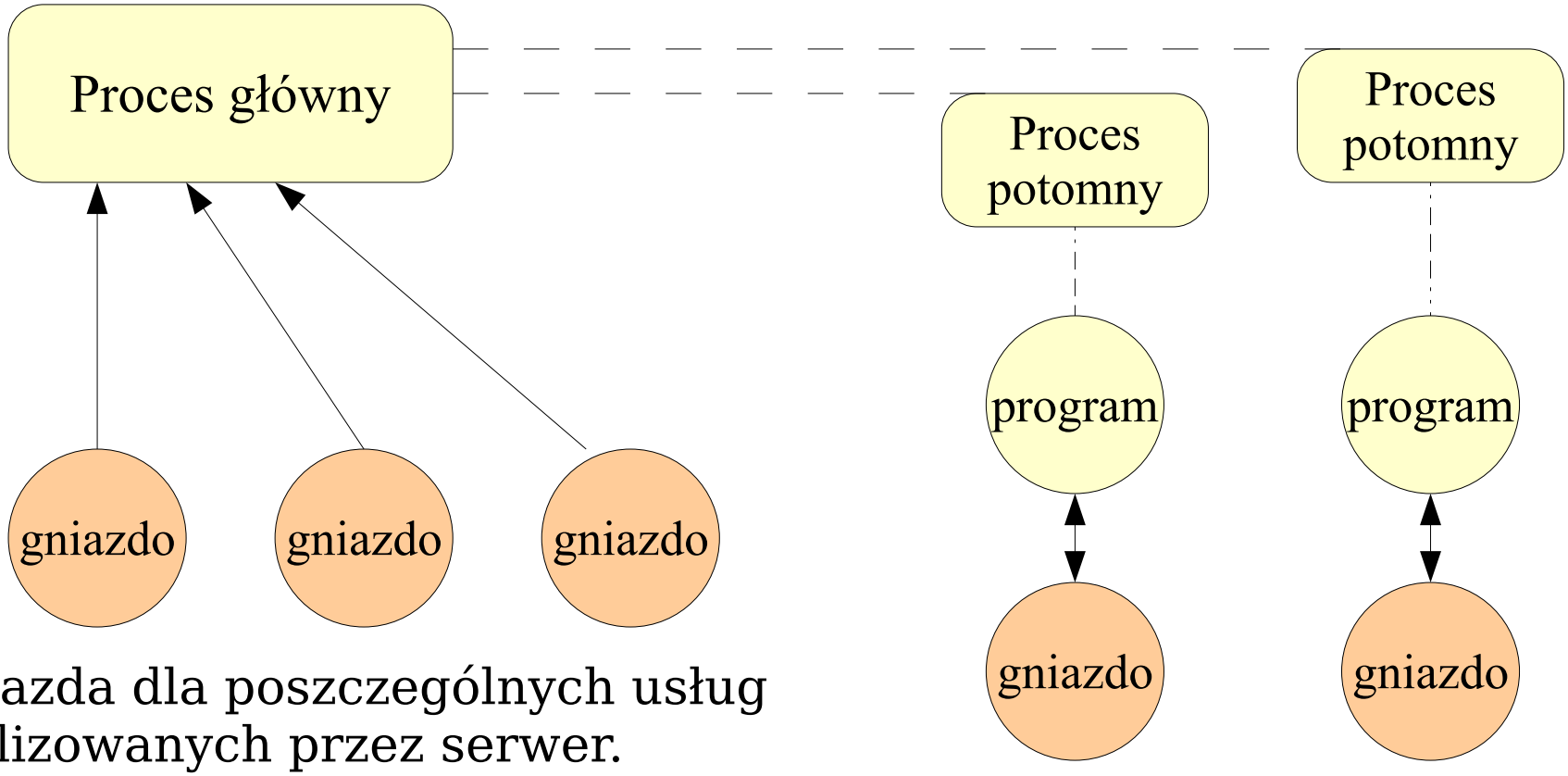
argv – tablica argumentów,

env – zmienne środowiska

Wartość zwracana: w przypadku sukcesu nie ma powrotu, w przypadku błędu -1 (**errno**).

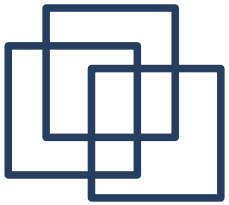
Współbieżny połączeniowy serwer wielousługowy

Schemat struktury połączeniowego, który korzysta z funkcji **`execve()`**, aby wywołać odzielny program do obsługi każdego połączenia.



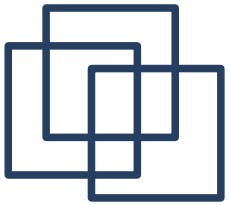
Gniazda dla poszczególnych usług realizowanych przez serwer.

Gniazda dla poszczególnych połączeń obsługiwanych przez procesy potomne.



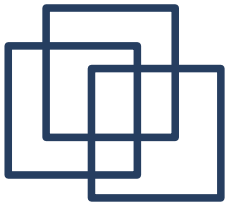
Wybór między rozwiązaniem iteracyjnym i współbieżnym

Wybór typu serwera – iteracyjnego albo współbieżnego – może być sprawą trudną, zważywszy jak szybki jest zarówno wzrost popytu użytkowników na usługi, jak i rozwój możliwości komunikacyjnych oraz wzrost szybkości przetwarzania danych. Projektanci przeważnie podejmują odpowiednie decyzje na podstawie ekstrapolacji dotychczasowych tendencji rozwojowych.



Poziom współbieżności

Poziom współbieżności działania serwera definiujemy jako liczbę procesów serwera działających w danej chwili. Współbieżny serwer połączeniowy zazwyczaj tworzy nowy proces dla każdego nawiązywanego połączenia z klientem. W praktyce liczba tych połączeń nie może być dowolnie duża. Każda implementacja protokołu TCP narzuca ograniczenie liczby jednocześnie aktywnych połączeń. Każdy system operacyjny ogranicza liczbę działających procesów. Gdy serwer wyczerpie jeden lub drugi limit, system będzie odmawiał tworzenia nowych procesów.



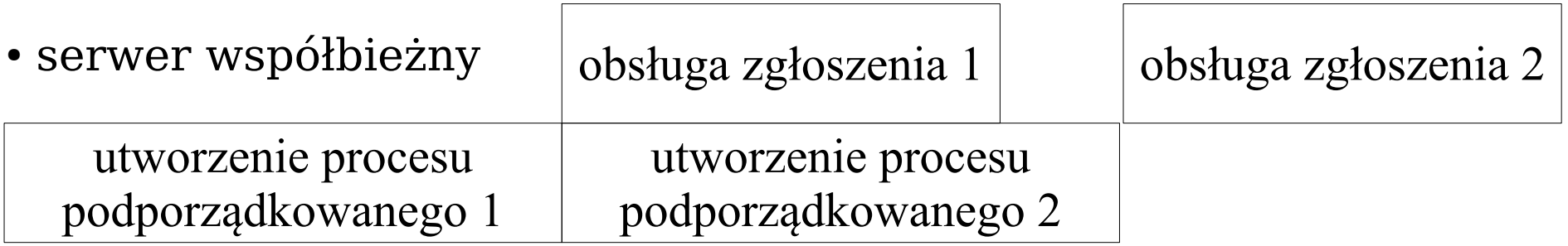
Współbieżność sterowana zapotrzebowaniem

Współbieżność sterowana zapotrzebowaniem *(demand driven concurency)* to technika polegająca na tworzeniu nowych procesów przy wzroście liczby jednocześnie obsługiwanych połączeń. Serwer zajmuje zasoby systemu tylko wtedy, gdy ich rzeczywiście używa. Jednocześnie takie serwery zapewniają krótki obserwowany czas odpowiedzi, dlatego że kolejne zgłoszenia nie muszą czekać na zakończenie obsługi zgłoszeń wcześniejszych.

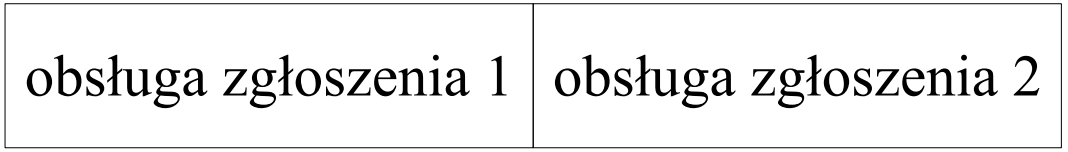
Narzut czasowy operacji systemowych

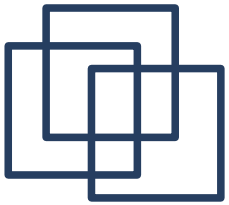
Zarówno operacja odebrania zgłoszenia z sieci, jak i utworzenie nowego procesu zajmują zauważalną ilość czasu. Powoduje to opóźnienie rozpoczęcia obsługi zgłoszenia. Jest to szczególnie odczuwalne, gdy czas obsługi zgłoszenia jest krótszy niż czas utworzenia procesu potomnego.

- serwer współbieżny



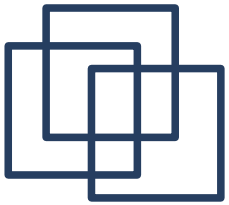
- serwer iteracyjny





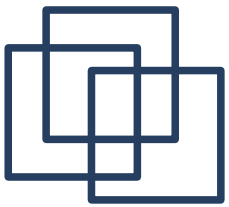
Narzut czasowy operacji systemowych

W praktyce obciążenie serwerów zgłoszeniami rzadko osiąga poziom bliski ich maksymalnej przepustowości. Ponadto niewielu projektantów decyduje się na współbieżną realizację serwera w sytuacji, gdy koszt tworzenia nowego procesu przewyższa czas przetwarzania zgłoszenia. Dlatego przypadki nadmiernych opóźnień obsługi lub odrzucania zgłoszeń nie zdarzają się często. Jednak projektując serwer, któremu stawia się wymaganie możliwie krótkiego czasu odpowiedzi przy dużym obciążeniu, trzeba rozważyć rozwiązania inne niż współbieżność sterowana zapotrzebowaniem.



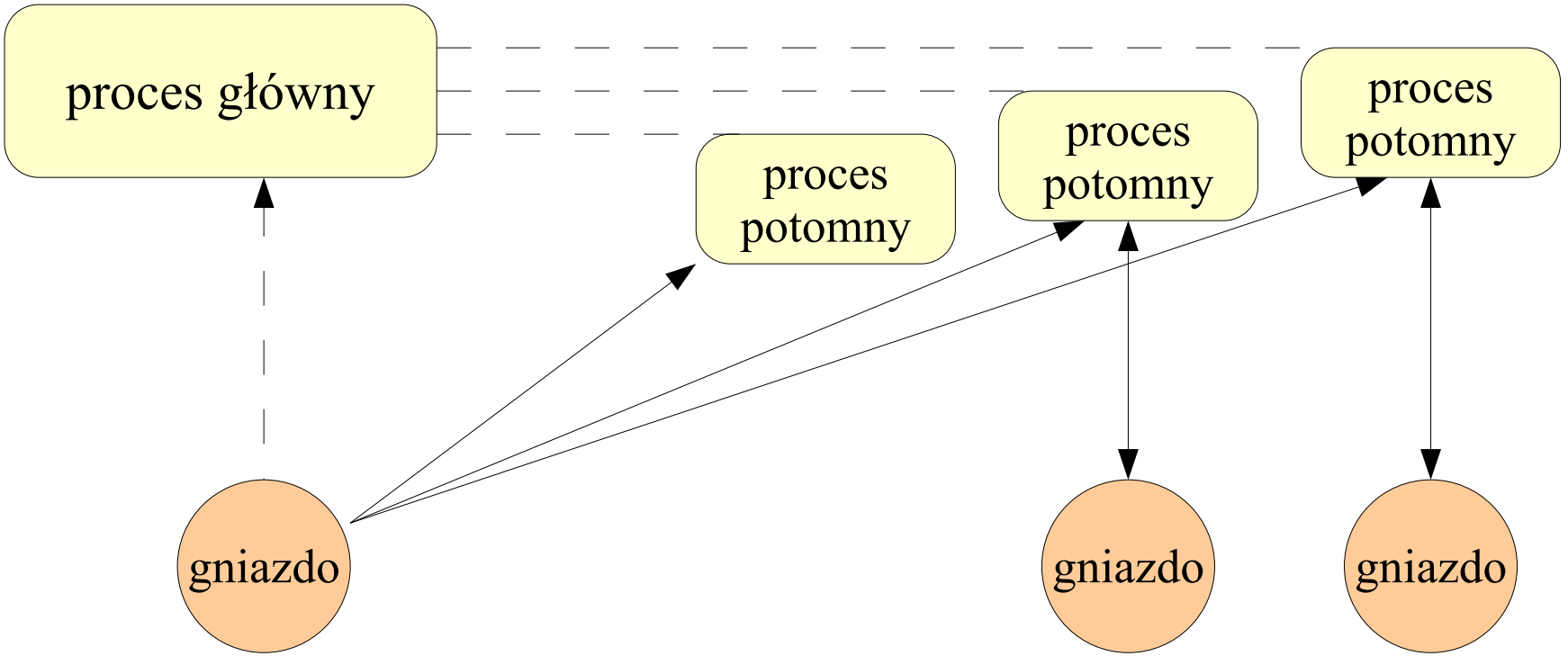
Alokacja wstępna procesów podporządkowanych

Wstępna alokacja procesów polega na tworzeniu współbieżnych procesów przy rozpoczęciu pracy przez proces główny serwera. Każdy z tych procesów czeka na nadejście zgłoszenia wywołując odpowiednią funkcję systemową. Po jego nadejściu jeden z procesów potomnych rozpoczyna obsługę klienta. Po zakończeniu obsługi proces potomny oczekuje na kolejne zgłoszenie. Dzięki temu skraca się czas obsługi zgłoszeń ponieważ serwer nie traci go na tworzenie dodatkowych procesów potomnych. Obsługa zgłoszenia może też przebiegać jednocześnie z operacją wejścia – wyjścia związaną z innym zgłoszeniem.



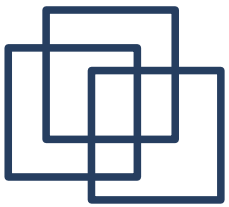
Alokacja wstępna procesów podporządkowanych

Schemat serwera połączeniowego korzystającego z wstępnej alokacji procesów.



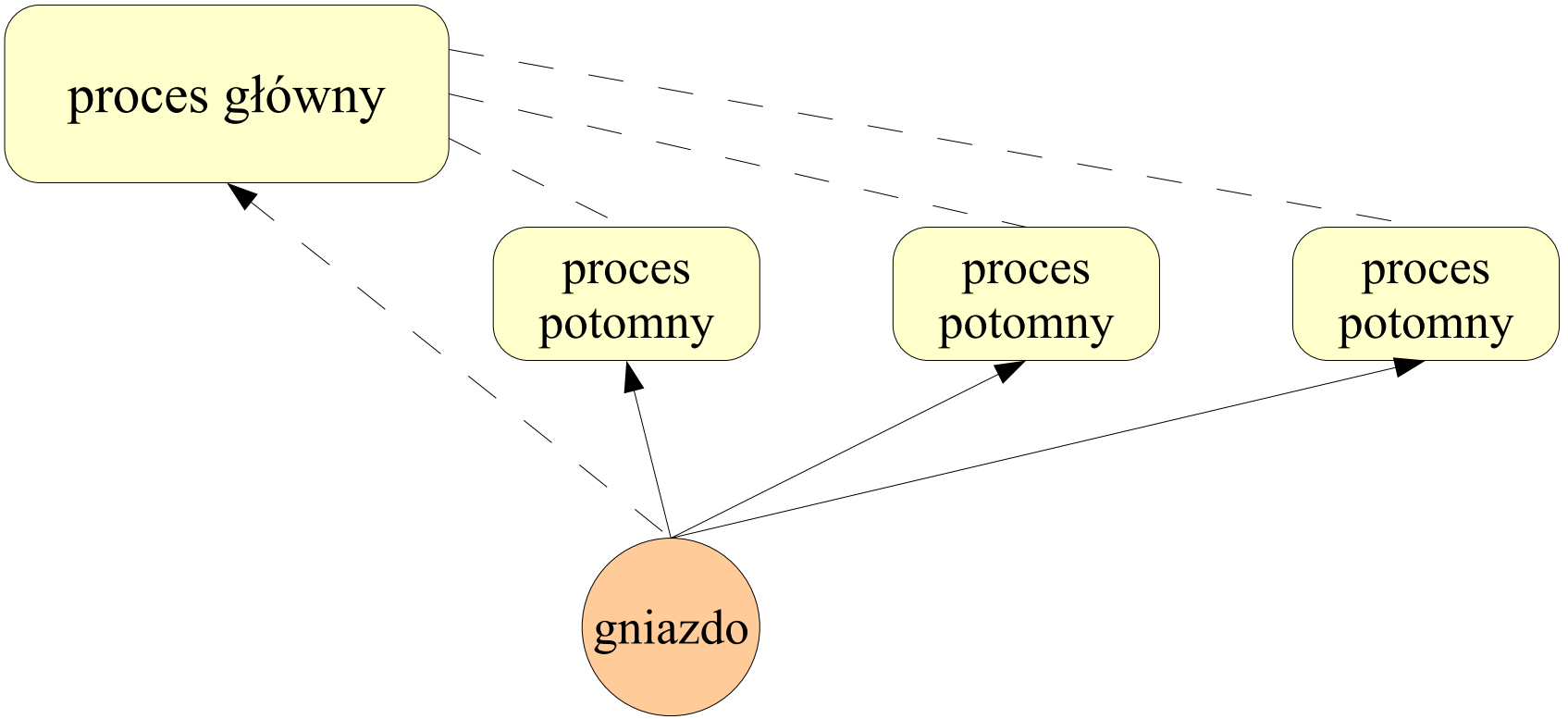
gniazdo dla zgłoszeń połączeń

gniazda używane do obsługi poszczególnych połączeń.

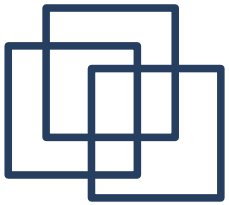


Alokacja wstępna procesów podporządkowanych

Schemat serwera bezpołączeniowego korzystającego z wstępnej alokacji procesów.

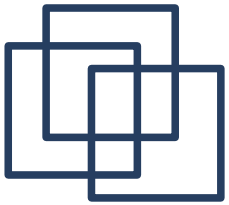


gniazdo związane z portem
o powszechnie znanym numerze



Systemy wieloprocesorowe

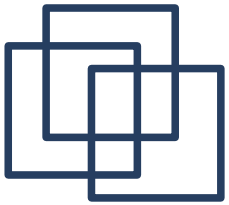
W systemach wieloprocesorowych alokacja wstępna pozwala powiązać poziom współbieżności z możliwościami serwera. W komputerze wieloprocesorowym system operacyjny przydziela każdemu procesowi oddzielny procesor. Dzięki temu, podczas nasilonego napływu zgłoszeń, każde z nich będzie obsługiwane przez inny procesor. W ten sposób osiągnięta zostanie maksymalna możliwa szybkość obsługi.



Odraczanie alokacji procesów

Koszty alokacji wstępnej nie ograniczają się jedynie do utworzenia procesu. Każdy nowy proces działający w systemie zwiększa ilość czasu procesora zużywaną przez podsystem zarządzający procesami. Oprogramowanie sieciowe także może zużywać więcej czasu w sytuacji, gdy obsługuje wiele alokowanych wstępnie procesów usiłujących odbierać zgłoszenia napływające z sieci. Te dodatkowe koszty są uzasadnione tylko wtedy, gdy dzięki alokacji wstępnej procesów można zwiększyć całkowitą przepustowość serwera lub skrócić czas oczekiwania na obsługę.

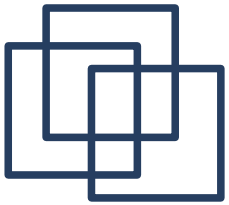
W przeciwnym razie należy rozważyć wykorzystanie techniki tzw. Odroczonej alokacji procesów *(delayed process allocation)*.



Odraczanie alokacji procesów

Serwer działający według metody odroczonej alokacji procesów rozpoczyna przetwarzanie każdego zgłoszenia w trybie iteracyjnym. Oddzielny, współbieżnie działający proces przeznaczony **tylko** do obsługi tego połączenia jest tworzony wtedy, gdy okazuje się, że jego przetwarzanie zajmie (zajmuje) zbyt dużo czasu. Proces główny może więc przykładowo sprawdzić poprawność zgłoszenia, a jeśli przetwarzane zgłoszenie jest mało czasochłonne także je obsłużyć, bez tworzenia w tym celu nowego procesu i przełączania kontekstu.

W celu realizacji tej techniki w środowisku UNIX można wykorzystać systemową funkcję **`alarm()`**.



Odraczanie alokacji procesów

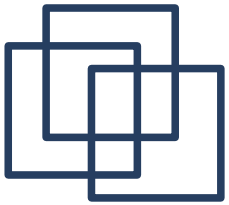
Funkcja **alarm()** powoduje wysłanie do procesu, w którym została użyta, sygnału **SIGALARM** po upływie określonego czasu. Użycie funkcji **alarm()** kasuje wszystkie wcześniej ustawione alarmy.

```
unsigned int alarm(unsigned int seconds);
```

seconds – liczba sekund, po której zostanie wysłany sygnał.

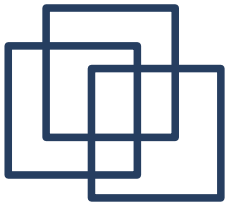
W przypadku gdy **seconds** zostanie ustawione na zero, nie zostanie wysłany sygnał.

Wartość zwracana: liczba sekund pozostała do wywołania wcześniej zdefiniowanego alarmu lub zero.



Jednoczesne stosowanie kilku technik alokacji

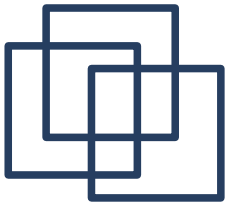
Techniki wstępnej i odroczonej alokacji są oparte na tej samej zasadzie uniezależnienia poziomu współbieżności działania serwera w danej chwili od liczby obsługiwanych zgłoszeń. Przyjęcie tej zasady umożliwia elastyczne dostosowanie poziomu współbieżności do potrzeb i w konsekwencji pozwala osiągnąć lepszą wydajność serwera.



Jednoczesne stosowanie kilku technik alokacji

Omówione techniki można stosować łącznie. Serwer na początku działa zgodnie z metodą odroczonej alokacji (nie tworzy nowych procesów). Gdy jednak już powstanie taki proces to po zakończeniu obsługi klienta może on istnieć nadal i czekać na następne zgłoszenia.

Największym problemem w tym wypadku jest stworzenie efektywnego mechanizmu ograniczenia poziomu współbieżności. Nie jest łatwo ocenić, kiedy istniejący proces potomny powinien zakończyć działanie i zwolnić zasoby. Jedna z metod polega na kończeniu procesów jeśli przez określony czas są one bezczynne – np. nie otrzymały zgłoszenia połączenia.

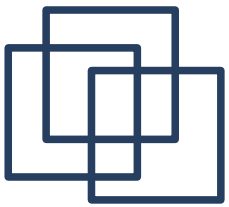


Współbieżność w programach klienckich

Współbieżna realizacja programów klienckich ma następujące zalety:

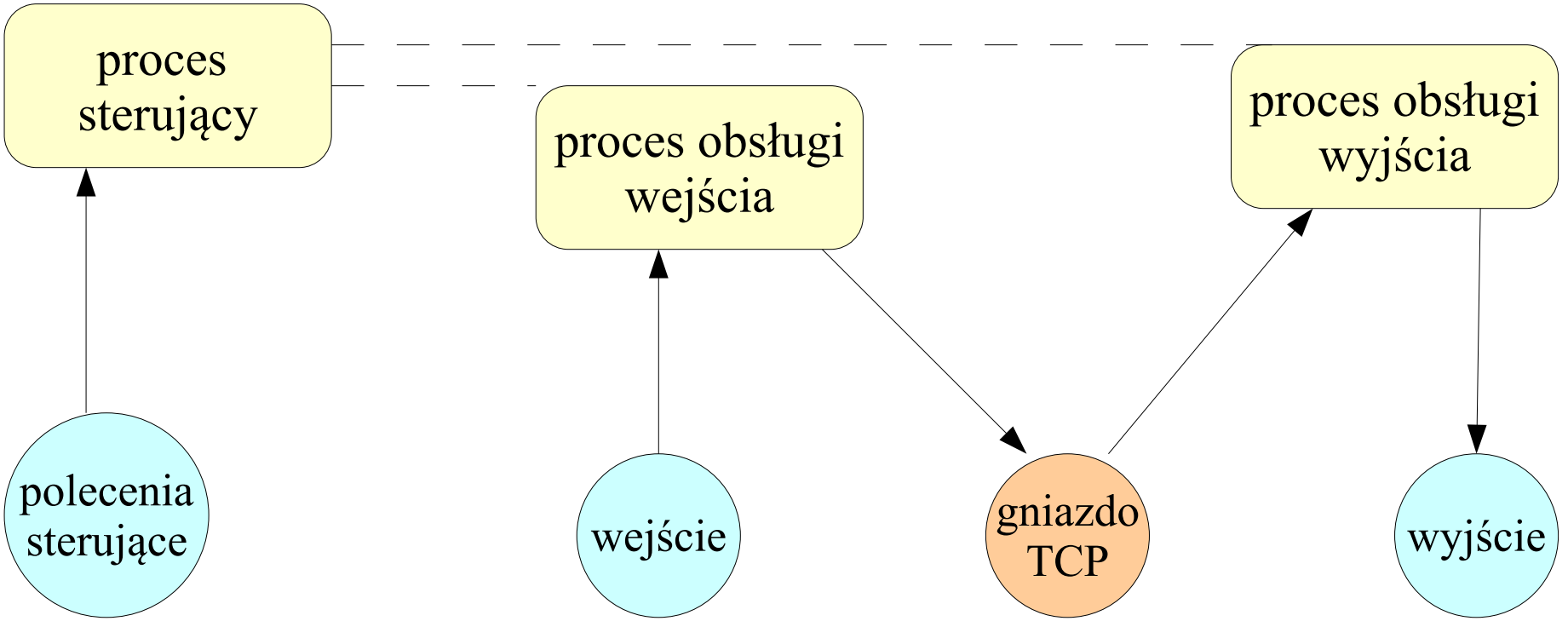
- łatwość implementacji – poszczególne funkcje mogą być realizowane przez odrębne części programu,
- ułatwiona konserwacja i rozbudowa programu – odrębne moduły mogą być rozwijane niezależnie,
- możliwość połączenia z wieloma serwerami jednocześnie – może znacznie skrócić czas oczekiwania na uzyskanie kompletnej obsługi,
- dynamiczne sterowanie przetwarzaniem danych – zmiana parametrów działania programu, uzyskanie informacji o stanie połączeń itp.

Zasadniczą zaletą współbieżnej realizacji programów klienckich jest asynchronizm. Program może jednocześnie wykonywać wiele zadań. Kolejność ich wykonywania nie jest przez program zdeterminowana.

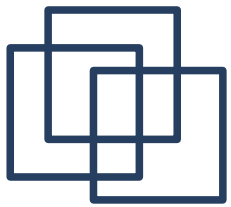


Współbieżność w programach klienckich - przykłady

Schemat jednej z możliwych struktur wieloprocesowego, połączeniowego programu klienta.

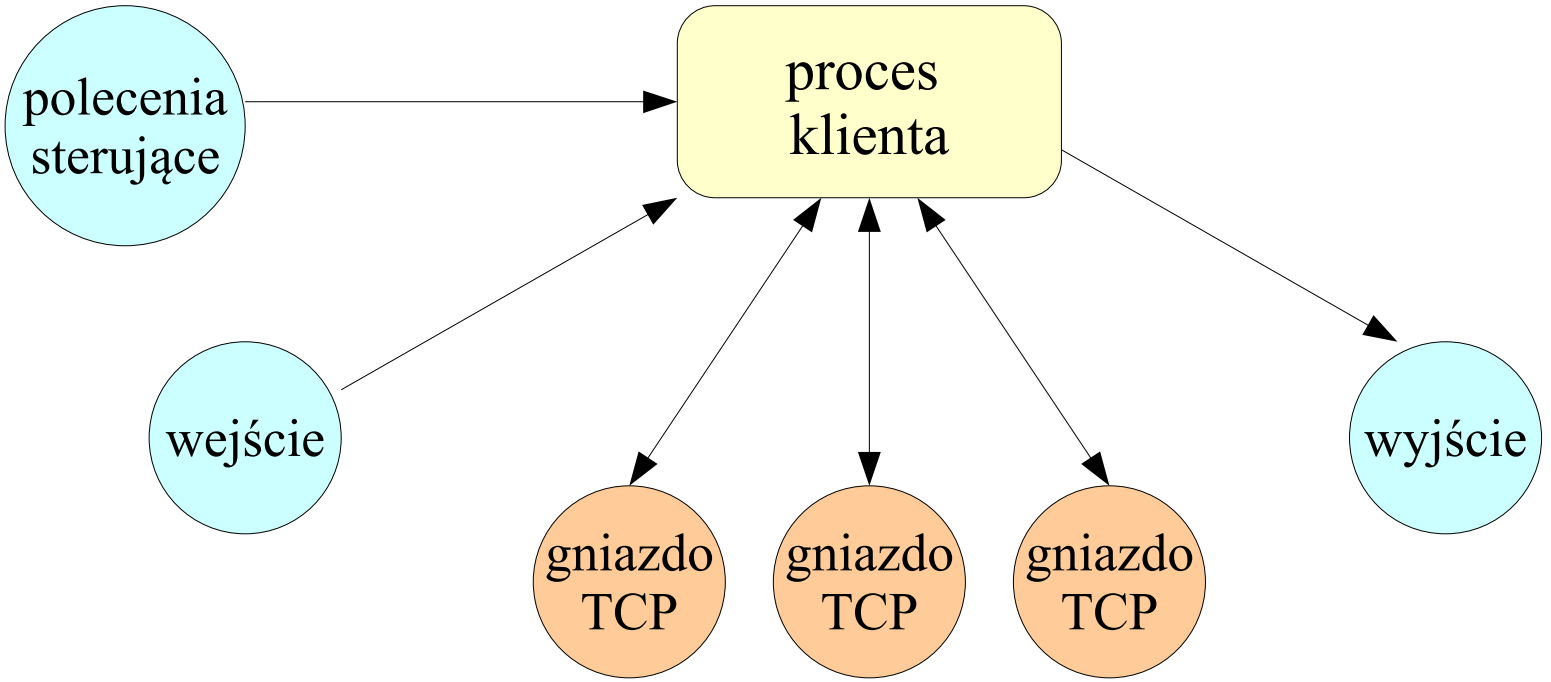


Jeden z procesów obsługuje wejście i wysyła zgłoszenia do serwera, a drugi odbiera odpowiedzi i obsługuje wyjście.

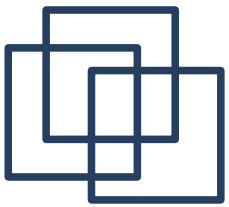


Współbieżność w programach klienckich - przykłady

Schemat jednej z możliwych struktur jednoprocesowego, połączeniowego programu klienta.

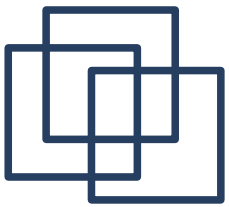


Jeden proces za pomocą funkcji **`select()`** obsługuje wiele połączeń TCP oraz wejście danych sterujących.



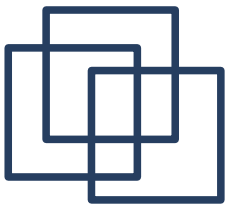
Współbieżność w programach klienckich - przykłady

```
int cli_con_echo_tcp(int argc, char **argv){
     long t;
     int ccount, scount, i, j;
     char *buf;

     scount = 0;
     ccount = 1024;
     for(i=0; i<argc; i++){
        if(strcmp(argv[i], "-c")==0){
           if (++i<argc && (ccount = atoi(argv[i]))){
              continue;
           }else{
              printf("nieprawidlowe wywolanie programu\n");
              return -1;
           }
        }else{
```



Współbieżność w programach klienckich - przykłady

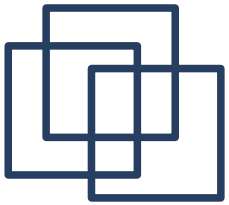
```
switch(fork()){
   case 0: // proces potomny
      if ((buf=(char *)calloc(ccount+1,
            sizeof(char)))==NULL){
         fprintf(stderr, "calloc: %s\n",
            strerror(errno));
         exit(0);
      }
      for(j=0; j<ccount; j++)
         buf[j] = 'a';
      buf[j]='\0';
      t = cli_echo_tcp(argv[i], TESTPORT, buf, 0);
      printf("%s\t%.6f\n", argv[i],
          (double)t/1000000.0);
      free(buf);
      exit(1);
```



Współbieżność w programach klienckich - przykłady

```
            default:
               scount++;
               continue;
            case -1:
               fprintf(stderr, "fork: %s\n",
                  strerror(errno));
               return -1;
         }
      }
   }
   for(i=0; i<scount; i++)
      waitpid(-1, NULL, 0);
   return 1;
}
```


33



Podsumowanie

Współbieżna organizacja programów to bardzo mocne narzędzie, użyteczne zarówno w odniesieniu do serwerów, jak i klientów. Współbieżność w programach klienckich umożliwia szybsze dostarczenie odpowiedzi użytkownikowi, pozwala uniknąć ryzyka zakleszczenia a także ułatwia oddzielenie operacji sterujących działaniem klienta od operacji przesyłania danych.