7. Sterowanie współbieżnością

- Zalety stosowania współbieżności:
  - skrócenie obserwowanego czasu odpowiedzi na zgłoszenie, w konsekwencji zwiększenie ogólnej przepustowości serwera
  - uniknięcie ryzyka zakleszczenia
- Realizacja serwerów współbieżnych:
  - poziom współbieżności – liczba procesów serwera działających w danej chwili; pytanie: czy wprowadzić maksymalny poziom współbieżności
  - współbieżność sterowana zapotrzebowaniem (ang. *demand-driven concurrency*) – poziom współbieżności wzrasta na żądanie, odpowiada liczbie zgłoszeń, które serwer przyjął, a których obsługa nie została jeszcze zakończona.
  - alokacja wstępna procesów podporządkowanych

7.1. Modyfikacje podstawowych algorytmów serwerów

- *Serwer wyprzedzająco wieloprocesowy* (ang. *preforking*): po uruchomieniu serwera przygotowywana jest z góry pewna liczba procesów potomnych. Po ustanowieniu połączenia, klientowi przypisywany jest jeden proces z puli. Można to uzyskać, na przykład poprzez umieszczenie funkcji `accept()` w procesie potomnym. Wszystkie procesy potomne wywołują funkcję `accept()` dla tego samego gniazda nasłuchującego, jądro systemu wybiera jeden proces, któremu przekazuje połączenie.
  Zalety: koszty uruchomienia procesów potomnych ponoszone tylko raz, na początku działania programu.
  Wady: Konieczność oszacowania z góry liczby uruchamianych procesów potomnych; w niektórych systemach mogą wystąpić narzuty czasu związane z budzeniem wszystkich procesów potomnych i ponownym usypianiem, po przekazaniu jednemu z nich obsługi klienta. Pewne rozwiązanie to dynamiczna pula procesów.
- *Serwer wyprzedzająco wielowątkowy (ang. prethreading)*: zasada podobna do serwera wyprzedzająco wieloprocesowego.
- *Serwer wyprzedzająco wieloprocesowy i wyprzedzająco wielowątkowy*: tworzona jest pula procesów, w ramach każdego procesu tworzona jest pula wątków. Klient jest obsługiwany przez wątek.
- *Serwer wyprzedzająco wieloprocesowy i wyprzedzająco wielowątkowy z mulipleksacją.*

Przykłady

- Przykład A. Serwer wieloprocesowy, brak ograniczenia na procesy potomne, zróżnicowana obsługa błędów

```
void sig_child(int s)
{
  while ( waitpid(-1, 0, WNOHANG) > 0 )
    ;
}

...

int listensock;
struct sockaddr_in addr;
struct sigaction act;

...
if ( sigaction(SIGCHLD, &act, 0) != 0 )
  PANIC("sigaction");                              // błąd krytyczny

...

if ( (listensock = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
  PANIC("socket");                                 // błąd krytyczny
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(mport);
addr.sin_addr.s_addr = INADDR_ANY;

if ( bind(listensock, &addr, sizeof(addr)) != 0 )
  PANIC("bind");                                   // błąd krytyczny
if ( listen(listensock, 5) != 0 )
  PANIC("listen");                                 // błąd krytyczny
for(;;)
{
  int sock_cli, adr_size = sizeof(addr);
  sock_cli = accept(listensock, &addr, &adr_size);
  if (sock_cli > 0 )
  {
    int pid;
    if ( (pid=fork()) == 0 )
      close(listensock);
      child(sock_cli);              // obsługa nowego klienta
    }
    else if (pid > 0)
      close(sock_cli);
    else
      perror("fork");  // wracamy do początku pętli
  }
}
```

- Przykład B. Ograniczenie liczby procesów potomnych

```
#define MAXCLIENTS 20   // maksymalna liczba klientów
int childCount=0;       // licznik procesów potomnych

void sig_child(int s)
{
  while ( waitpid(-1, 0, WNOHANG) > 0 )
    childCount --;
}

...

for(;;)
{
  int sock_cli, adr_size = sizeof(adr);
  while (childCount >= MAXCLIENTS)
    sleep(1);
  sock_cli = accept(listensock, &addr, &addr_size);
  if (sock_cli > 0 )
  {
    int pid;
    if ( (pid=fork()) == 0 )       // potomek
      close(listensock);
      child(sock_cli);              // obsługa nowego klienta
    }
    else if (pid > 0)              // proces macierzysty
    {
      childCount ++;
      close(sock_cli);
    }
    else                           // proces macierzysty
      perror("fork");  // wracamy do początku pętli
  }
}
```

- Przykład C. Tworzenie procesów z wyprzedzeniem

```
#define MAXCLIENTS 20   // maksymalna liczba klientów
int childCount =0; // licznik procesów potomnych

void sig_potomek(int s)
{
  while ( waitpid(-1, 0, WNOHANG) > 0 )
    childCount --;
}

...

for(;;)
{
  if (childCount < MAXCLIENTS)
  {
    if ( (pid=fork()) == 0 )       // potomek
      for (;;)
      {
        int sock_cli;
        sock_cli = accept(listensock, 0, 0);
        Child(sock_cli);            // obsłuż nowego klienta
      }
    else if (pid > 0)              // proces macierzysty
      childCount ++;
    else
      perror("fork");
  }
  else
    sleep(1);
}
```

- Przykład D. Zmienianie liczby procesów w zależności od zastosowania: tworzenie dodatkowych procesów ze zmienną częstotliwością

```
int okres=MAXOKRES;
time_t ostatni;

void sig_child(int signum)
{
    wait(0);               /* Czekaj na zakończenie potomka */
    time(&ostatni);        /* Aktualizuj czas */
    okres = MAXOKRES;      /* Ustaw okres domyślny */
}

...
  time(&ostatni);           /* Inicjalizacja znacznika czasu */
  for (;;)
  {
  if ( !fork() )
    child();               /* obsługa klienta (musi mieć exit()) */
    sleep(okres);
    /* Jeśli żaden proces potomny nie zakończył się,
       zwiększ częstotliwość*/
    if ( time(0) - ostatni >= okres )
      if ( okres > MINOKRES )   /* nie poniżej minimum */
        okres /= 2;        /* podwój częstotliwość */
  }
```

Założenie:
- liczba połączeń stabilna: tyle samo procesów kończy się, ile powstaje
- liczba połączeń wzrasta: zwiększenie częstotliwości tworzenia dodatkowych procesów

- Przykład E. Problemy z funkcją `select()`
  - `select` działa na zbiorach deskryptorów obejmujących wszystkie deskryptory do i włączając ten najwyższy będący przedmiotem zainteresowania
  - `select` nie wie, który z deskryptorów spowodował powrót z funkcji, każdy z deskryptorów musi być sprawdzony

Rozwiązanie - utrzymywanie małej tablicy deskryptorów: `select` w procesie potomnym

```
void child(int listensock) {
  fd_set set;
  int maxfd = listensock;
  int count=0;

  FD_ZERO(&set);
  FD_SET(listensock, &set);
  for (;;) {
    struct timeval timeout={2,0}; /* 2 sekundy */
    if ( select(maxfd+1, &set, 0, 0, &timeout) > 0 ) {

    /*--- Jeśli nowe połączenie, dodaj do listy --- */
      if ( FD_ISSET(listensock, &set) ) {
        if ( count < MAXCONNECTIONS ) {
          int sock_cli = accept(listensock, 0, 0);
          if ( maxfd < sock_cli) maxfd = sock_cli;
          FD_SET(sock_cli, &set);
          count++;
        }
      } /* koniec if - nowe połączenie */

      /*--- Jeśli żądanie klienta, przetwarzaj ---*/
      else {
        int i;
        for ( i = 0; i < maxfd+1; i++ ) {
          if ( FD_ISSET(i, &set) ) {
            char buffer[1024];
            int bytes;
            bytes = recv(i, buffer, sizeof(buffer), 0);
            if ( bytes < 0 ) { /* czy zamknięto połączenie */
              close(i);
              FD_CLR(i, &set);
              licznik--;
            }
            else  /* przetwarzaj żłdanie */
              send(i, buffer, bytes, 0);
          } /* koniec if - przetwarzania deskryptora */
        } /* koniec for - przegadanie gotowych deksryptorów */
      } /* koniec if - żądań klientów */

    } /* koniec if - select */
  } /* pętla główna */
  exit(0);
}
```

7.2. Klient współbieżny

Zalety:

- możliwość interakcji z użytkownikiem podczas przesyłania danych
- możliwość łączenia się z wieloma serwerami jednocześnie

Implementacja współbieżnego programu klienckiego

- funkcje klienta wykonywane są przez dwa lub więcej procesów (wątków)
- klient obsługuje zdarzenia na wielu wejściach i wyjściach w trybie asynchronicznym posługując się funkcją typu `select`

- **Przykład: program do pomiaru przepustowości sieci**

Comer, Stevens "Sieci komputerowe", tom 3, str. 228-234)

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>

int  TCPtecho(fd_set *pafds, int nfds, int ccount, int hcount);
int  reader(int fd, fd_set *pfdset);
int  writer(int fd, fd_set *pfdset);
int  errexit(const char *format, ...);
int  connectTCP(const char *host, const char *service);
long  mstime(u_long *);

#define  BUFSIZE    4096
#define  CCOUNT    64*1024
#define  USAGE  "usage: TCPtecho [ -c count ] host1 host2...\n"

char  *hname[NOFILE];
int  rc[NOFILE], wc[NOFILE];
char  buf[BUFSIZE];

int main(int argc, char *argv[]) {
  int  ccount = CCOUNT;
  int  i, hcount, maxfd, fd;
  int  one = 1;
  fd_set  afds;
  hcount = 0;
  maxfd = -1;
  for (i=1; i<argc; ++i) {
    if (strcmp(argv[i], "-c") == 0) {
      if (++i < argc &&
         (ccount = atoi(argv[i])))
        continue;
      errexit(USAGE);
    }
    /* else, a host */
    fd = connectTCP(argv[i], "echo");
/* gniazdo nieblokujące, można również użyć fcntl()*/
    if (ioctl(fd, FIONBIO, (char *)&one))
      errexit("can't mark socket nonblocking: %s\n",
        strerror(errno));
    if (fd > maxfd)
      maxfd = fd;
    hname[fd] = argv[i];
    ++hcount;
    FD_SET(fd, &afds);
  }
  TCPtecho(&afds, maxfd+1, ccount, hcount);
  exit(0);
}
```

```
int TCPtecho(fd_set *pafds, int nfds, int ccount, int hcount)
{
  fd_set  rfds, wfds;
  fd_set  rcfds, wcfds;
  int  fd, i;

  for (i=0; i<BUFSIZE; ++i)
    buf[i] = 'D';
  memcpy(&rcfds, pafds, sizeof(rcfds));
  memcpy(&wcfds, pafds, sizeof(wcfds));
  for (fd=0; fd<nfds; ++fd)
    rc[fd] = wc[fd] = ccount;

  (void) mstime((u_long *)0);

  while (hcount) {
    memcpy(&rfds, &rcfds, sizeof(rfds));
    memcpy(&wfds, &wcfds, sizeof(wfds));

    if (select(nfds, &rfds, &wfds, (fd_set *)0,(struct timeval *)0) < 0)
      errexit("select failed: %s\n", strerror(errno));
    for (fd=0; fd<nfds; ++fd) {
      if (FD_ISSET(fd, &rfds))
        if (reader(fd, &rcfds) == 0)
          hcount--;
      if (FD_ISSET(fd, &wfds))
        writer(fd, &wcfds);
    }
  }
}

int reader(int fd, fd_set *pfdset) {
  u_long  now;
  int  cc;

  cc = read(fd, buf, sizeof(buf));
  if (cc < 0)
    errexit("read: %s\n", strerror(errno));
  if (cc == 0)
    errexit("read: premature end of file\n");
  rc[fd] -= cc;
  if (rc[fd])
    return 1;
  (void) mstime(&now);
  printf("%s: %d ms\n", hname[fd], now);
  (void) close(fd);
  FD_CLR(fd, pfdset);
  return 0;
}

int writer(int fd, fd_set *pfdset) {
  int  cc;

  cc = write(fd, buf,MIN(sizeof(buf), wc[fd]));
  if (cc < 0)
    errexit("read: %s\n", strerror(errno));
  wc[fd] -= cc;
  if (wc[fd] == 0) {
    (void) shutdown(fd, 1);
    FD_CLR(fd, pfdset);
  }
}
```

```
long mstime(u_long *pms) {
  static struct timeval  epoch;
  struct timeval    now;

  if (gettimeofday(&now, (struct timezone *)0))
    errexit("gettimeofday: %s\n", strerror(errno));
  if (!pms) {
    epoch = now;
    return 0;
  }
  *pms = (now.tv_sec - epoch.tv_sec) * 1000;
  *pms +=(now.tv_usec-epoch.tv_usec+500)/1000;
  return *pms;
}
```

- Pytanie: dlaczego gniazdo jest ustawiane w trybie nieblokującym?

Należy przeczytać:

Douglas E. Comer, David L. Stevens: *Sieci komputerowe TCP/IP, tom 3*: str. 208-221, 223-235

W. Richard Stevens: *Unix, programowanie usług sieciowych, tom 1: API gniazda i XTI*: str. 806-845

Uzupełnienia

http://www.kegel.com/c10k.html
http://www.atnf.csiro.au/people/rgooch/linux/docs/io-events.html
http://bulk.fefe.de/scalable-networking.pdf