Zaawansowane operacje wejścia/wyjścia

Witold Paluszyński
witold.paluszynski@pwr.wroc.pl
http://sequoia.ict.pwr.wroc.pl/~witold/

Niniejszy dokument zawiera materiały do wykładu na temat zaawansowanych operacji wejścia/wyjścia w systemie Unix. Jest on udostępniony pod warunkiem wykorzystania wyłącznie do własnych, prywatnych potrzeb i może być kopiowany wyłącznie w całości, razem z niniejszą stroną tytułową.

Blokowanie operacji I/O

Przykłady blokowania wejścia/wyjścia:

- odczyt z plików, gdy nie ma w nich już danych (potoki, terminale, gniazdka),
- zapis do tych samych urządzeń, jeśli dane nie mogą być od razu przyjęte,
- otwarcie pliku blokuje do momentu spełnienia pewnych warunków (np. otwarcie pliku terminala do chwili nawiązania połączenia przez modem, otwarcie FIFO do zapisu gdy żaden proces nie ma go otwartego do odczytu, itp.),
- operacje na plikach z mandatory record locking,
- i inne.

Nieblokujące wejście/wyjście

- `open` z flagą `O_NONBLOCK`
- `fcntl` na deskryptorze otwartym z tą samą flagą

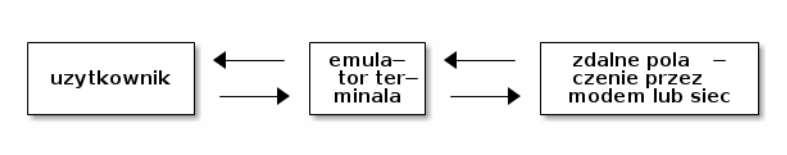
Wtedy powrót z funkcji `read`/`write` następuje z błędem wskazującym, że operacja by blokowała.

UWAGA: można zastosować starą flagę `O_NDELAY`, wtedy funkcje `read`/`write` zwracają 0, co jednak pokrywa się z sygnalizacją EOF (System V).

Równoczesne operacje I/O

Jak zorganizować operacje wejścia/wyjścia, gdy mamy np. jednocześnie kilka źródeł, z których nadchodzą dane?

Na przykład, emulator terminala, komunikujący się z jednej strony z użytkownikiem, a z drugiej ze zdalnym systemem:



Możliwe są następujące rozwiązania:

- polling: nieblokujące próby odczytu na przemian z kolejnych deskryptorów; w niektórych systemach można też użyć na gniazdkach: `ioctl(fd,FIONREAD,&nread);`
- przełączanie (multiplexing): funkcje `select` i `poll`
- asynchroniczne I/O: powiadamianie procesu przez jądro (przy pomocy sygnału) o gotowości deskryptora do operacji I/O

I/O multiplexing

Ten mechanizm nie jest objęty normą POSIX, jednak funkcja `select()` istnieje w Unixach AT&T i BSD od dawna i jest dość standardowa. Funkcja `poll()` jest nowa, i w niektórych systemach jest zaimplementowana przez `select`.

```
struct timeval {
  time_t       tv_sec;   /* seconds */
  suseconds_t  tv_usec;  /* and microseconds */
};

int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *errorfds, struct timeval *timeout);
```

Funkcja `select()` zwraca liczbę deskryptorów gotowych do operacji I/O. Po powrocie wynikającym z upłynięcia zadanego czasu funkcja zeruje wszystkie zbiory deskryptorów w systemie V, lecz pozostawia je niezmienione w systemie BSD. Te systemy również inaczej liczą deskryptory gotowe do I/O (AT&T liczy je w sensie mnogościowym, a BSD sumuje arytmetycznie liczności wszystkich trzech zbiorów deskryptorów). Ponadto, niektóre systemy w przypadku gotowości deskryptora ustawiają w strukturze `timeval` czas pozostały do wyczerpania (Linux).

Do tworzenia odpowiednich zbiorów deskryptorów, a także sprawdzania otrzymanych wyników, istnieją makra:

```
void FD_SET(int fd, fd_set *fdset);
void FD_CLR(int fd, fd_set *fdset);
int FD_ISSET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

Funkcja `poll` zapewnia podobne działanie lecz inny interface programisty:

```
struct pollfd {
  int     fd;        /* file descriptor */
  short   events;    /* requested events */
  short   revents;   /* returned events */
}

int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

Tablica struktur `pollfd` określa wszystkie deskryptory i zdarzenia, na które chcemy oczekiwać (czas określany jest tu w milisekundach). Wartość zwracana z funkcji daje liczbę gotowych deskryptorów.

Funkcja `select` — przykład

```
#include <sys/types.h>
#include <sys/time.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>

int main() {
  char buffer[128];
  int result, nread;
  fd_set inputs, testfds;
  struct timeval timeout;

  FD_ZERO(&inputs);
  FD_SET(0,&inputs);
  while(1) {
    testfds = inputs;
    timeout.tv_sec = 2;
    timeout.tv_usec = 500000;

    result = select(FD_SETSIZE, &testfds, (fd_set *)0,
                                (fd_set *)0, &timeout);
```

```
    switch(result) {
    case 0:
      printf("timeout\n");
      break;
    case -1:
      perror("select");
      exit(1);
    default:
      if (FD_ISSET(0,&testfds)) {
        ioctl(0,FIONREAD,&nread);
        if (nread == 0) {
          printf("keyboard done\n");
          exit(0);
        }
        nread = read(0,buffer,nread);
        buffer[nread] = 0;
        printf("read %d from keyboard: %s", nread, buffer);
      }
      break;
    }
  }
}
```

Funkcja `select` — inny przykład

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

int main() {
  int server_sockfd, client_sockfd;
  int server_len, client_len;
  struct sockaddr_in server_address;
  int result;
  fd_set readfds, testfds;
  server_sockfd = socket(AF_INET, SOCK_STREAM, 0);

  server_address.sin_family = AF_INET;
  server_address.sin_addr.s_addr = htonl(INADDR_ANY);
  server_address.sin_port = htons(9734);
  server_len = sizeof(server_address);

  bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
  listen(server_sockfd, 5);
```

```
  FD_ZERO(&readfds);
  FD_SET(server_sockfd, &readfds);
  while(1) {
    char ch;
    int fd, nread;

    testfds = readfds;

    printf("server waiting\n");
    result = select(FD_SETSIZE, &testfds, (fd_set *)0,
                    (fd_set *)0, (struct timeval *) 0);

    if(result < 1) {
      perror("server5");
      exit(1);
    }
    for(fd = 0; fd < FD_SETSIZE; fd++) {
      if(FD_ISSET(fd,&testfds)) {
        if(fd == server_sockfd) {
          client
          client_sockfd = accept(server_sockfd, 0, 0);
          FD_SET(client_sockfd, &readfds);
          printf("adding client on fd %d\n", client_sockfd);
        }
```

```
        else {
          ioctl(fd, FIONREAD, &nread);

          if(nread == 0) {
            close(fd);
            FD_CLR(fd, &readfds);
            printf("removing client on fd %d\n", fd);
          }

          else {
            read(fd, &ch, 1);
            sleep(5);
            printf("serving client on fd %d\n", fd);
            ch++;
            write(fd, &ch, 1);
          }
        }
      }
    }
  }
}
```

Asynchroniczne I/O

• Dla każdego deskryptora, dla którego chcemy oczekiwać na dane, należy ustawić powiadamianie przez jądro odpowiednim sygnałem. Ten mechanizm również nie jest jednak objęty normą POSIX i jest trochę inny w systemach AT&T i BSD.

AT&T Mechanizm działa dla wszystkich urządzeń opartych na systemie „streams". Należy ustawić sygnał sigpoll dla deskryptora funkcją `ioctl(fd,I_SETSIG,flags)`, gdzie argumentem flags określa się jakie wydarzenie powinno spowodować wysłanie sygnału.

BSD Tu mamy do dyspozycji dwa sygnały: sigio (dla zwykłych operacji I/O), i sigurg (dla danych *out-of-band* z połączeń sieciowych). Należy ustawić proces lub grupę procesów, która ma otrzymać sygnał dla deskryptora (funkcja `fcntl` z komendą `F_SETOWN`), oraz ustawić flagę `O_ASYNC` dla deskryptora (funkcja `fcntl` z komendą `F_SETFL`).

• W chwili otrzymania sygnału nadal nie wiemy, który deskryptor jest gotów do wykonywania operacji I/O i musimy to i tak po kolei sprawdzać.

Odwzorowanie plików do pamięci

Użycie odwzorowania pliku do pamięci pozwala wykonywać operacje I/O na pliku przez manipulacje na pamięci. Zamiast przesuwać kursor pliku funkcją `lseek`, wystarczy obliczyć inny adres w pamięci. Odwzorowanie realizuje funkcja `mmap`.

Poniższe wywołanie powoduje odwzorowanie sekcji otwartego pliku `fildes` od pozycji `off` o długości `len` do obszaru pamięci od adresu `addr`. Adres ten może być określony jako 0, wtedy funkcja sama wybiera obszar pamięci i zwraca jego adres.

```
pa = mmap(addr, len, prot, flags, fildes, off);
```

Argument `prot` określa prawa dostępu do regionu pamięci i musi być zgodny z trybem otwarcia pliku.

Argument `flags` określa różne atrybuty mapowanego regionu:

`MAP_SHARED`	operacje zapisu do pamięci powodują zapis do pliku
`MAP_PRIVATE`	operacje zapisu do pamięci nie powodują modyfikacji pliku — tworzona jest robocza kopia pliku
`MAP_FIXED`	wymusza przydział adresu zadany argumentem (normalnie jest on tylko wskazówką dla jądra)
`MAP_FILE`	flaga wymagana w systemie BSD

Odwzorowanie pamięci — przykłady

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>   /* mmap() */
#include <fcntl.h>

#ifndef MAP_FILE /* 44BSD defines this & requires it to mmap files */
#define MAP_FILE 0    /* to compile under systems other than 44BSD */
#endif

int main(int argc, char *argv[])
{
  int fdin, fdout;
  char *src, *dst;
  struct stat statbuf;

  if (argc != 3) err_quit("usage: a.out <fromfile> <tofile>");

  if ( (fdin = open(argv[1], O_RDONLY)) < 0)       /* ... */;
  if ( (fdout = open(argv[2], O_RDWR | O_CREAT | O_TRUNC,
                     FILE_MODE)) < 0)              /* ... */;

  if (fstat(fdin, &statbuf) < 0) /* need size of input file */;
  if (lseek(fdout, statbuf.st_size - 1, SEEK_SET) == -1)
    err_sys("lseek error");
  if (write(fdout, "", 1) != 1)  /* set size of output file */
    err_sys("write error");
```

Operacje I/O na plikach odwzorowanych do pamięci

Wykonywanie operacji I/O przez odwzorowaną pamięć pozwala na wykorzystanie kanałów DMA, co daje dużą szybkość operacji I/O i odciążenie głównego procesora.

Jeśli różne procesy odwzorują ten sam plik do swoich przestrzeni adresowych, to uzyskują dzięki temu obszar pamięci pozwalający na komunikację międzyprocesową.

Ze względu na efektywność, system nie wykonuje natychmiast operacji I/O wynikających z zapisów do odwzorowanej pamięci. Można wymusić wykonanie tych operacji funkcją `msync()`. Synchronizację można wywołać w dowolną stronę, tzn. zarówno zapis zawartości pamięci do pliku, jak i wczytanie zawartości pliku do pamięci.

Zwykłe zamknięcie odwzorowanego pliku nie kasuje odwzorowania `mmap`, należy w tym celu wywołać:

```
res = munmap(addr, len);
```

Skasowanie odwzorowania nie powoduje żadnych operacji I/O do pliku. Zapis do pliku w trybie `MAP_SHARED` odbywa się na bieżąco w trakcie operacji na obszarze pamięci.

```
  if ( (src = mmap(0, statbuf.st_size, PROT_READ,
                   MAP_FILE | MAP_SHARED, fdin, 0)) == (caddr_t) -1)
    err_sys("mmap error for input");
  if ( (dst = mmap(0, statbuf.st_size, PROT_READ | PROT_WRITE,
                   MAP_FILE | MAP_SHARED, fdout, 0)) == (caddr_t) -1)
    err_sys("mmap error for output");

  memcpy(dst, src, statbuf.st_size); /* does the file copy */

  exit(0);
}
```

```
typedef struct {
    int integer;
    char string[12];
} RECORD;

#define NRECORDS (100)

int main() {
   RECORD record, *mapped;
   int i, f;
   FILE *fp;

   fp = fopen("records.dat","w+");
   for(i=0; i<NRECORDS; i++) {
      record.integer = i;
      sprintf(record.string,"RECORD-%d",i);
      fwrite(&record,sizeof(record),1,fp);
   }
   fclose(fp);

   fp = fopen("records.dat","r+");
   fseek(fp,43*sizeof(record),SEEK_SET);
   fread(&record,sizeof(record),1,fp);
   record.integer = 143;
   sprintf(record.string,"RECORD-%d",record.integer);

   fseek(fp,43*sizeof(record),SEEK_SET);
   fwrite(&record,sizeof(record),1,fp);
   fclose(fp);
```

```
   f = open("records.dat",O_RDWR);
   read(f,&record,sizeof(record));
   mapped = (RECORD *)mmap(0, NRECORDS*sizeof(record),
                     PROT_READ | PROT_WRITE, MAP_SHARED, f, 0);
   mapped[43].integer = 243;
   sprintf(mapped[43].string,"RECORD-%d",mapped[43].integer);
   msync((void *)mapped, NRECORDS*sizeof(record), MS_ASYNC);
   munmap((void *)mapped, NRECORDS*sizeof(record));
   close(f);
   exit(0);
}
```

Przekazywanie otwartych deskryptorów

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

int send_fd(int clifd, int fd)
{
  char buf[2];          /* send_fd()/recv_fd() 2-byte protocol */

  buf[0] = 0;           /* null byte flag to recv_fd() */
  if (fd < 0) {
    buf[1] = -fd;       /* nonzero status means error */
    if (buf[1] == 0)
      buf[1] = 1;       /* -256, etc. would screw up protocol */
  } else {
    buf[1] = 0;         /* zero status means OK */
  }

  if (write(clifd, buf, 2) != 2)
    return(-1);

  if (fd >= 0)
    if (ioctl(clifd, I_SENDFD, fd) < 0)
      return(-1);
  return(0);
}

int send_err(int clifd, int errcode, const char *msg)
{
  int n;
```

```
  if ( (n = strlen(msg)) > 0)
    if (writen(clifd, msg, n) != n) /* send the error message */
      return(-1);

  if (errcode >= 0)
    errcode = -1;                   /* must be negative */

  if (send_fd(clifd, errcode) < 0)
    return(-1);
  return(0);
}
```

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

int recv_fd(int servfd,
            ssize_t (*userfunc)(int, const void *, size_t)) {
  int newfd, nread, flag, status = -1;
  char *ptr, buf[MAXLINE];
  struct strbuf dat;
  struct strrecvfd recvfd;

  for ( ; ; ) {
    dat.buf = buf;
    dat.maxlen = MAXLINE;
    flag = 0;
    if (getmsg(servfd, NULL, &dat, &flag) < 0)
      err_sys("getmsg error");
    nread = dat.len;
    if (nread == 0) {
      err_ret("connection closed by server");
      return(-1);
    }
    for (ptr = buf; ptr < &buf[nread]; ) {
      if (*ptr++ == 0) {
        if (ptr != &buf[nread-1])
          err_dump("message format error");
        status = *ptr & 255;
        if (status == 0) {
          if (ioctl(servfd, I_RECVFD, &recvfd) < 0)
            return(-1);
          newfd = recvfd.fd;    /* new descriptor */
```

```
        } else
          newfd = -status;
        nread -= 2;
      }
    }
    if (nread > 0)
      if ((*userfunc)(STDERR_FILENO, buf, nread) != nread)
        return(-1);

    if (status >= 0)            /* final data has arrived */
      return(newfd);            /* descriptor, or -status */
  }
}
```
