Laborator 11 - Operatii IO avansate - Linux

Materiale Ajutătoare

- lab11-slides.pdf
- lab11-refcard.pdf

Nice to read

- TLPI - Chapter 63, Alternative I/O models

Linux - multiplexarea I/O

Există situaţii în care un program trebuie să trateze operaţiile I/O de pe mai multe canale ori de câte ori acestea apar. Un astfel de exemplu este un program de tip server care foloseşte mecanisme precum pipe-uri sau socketi pentru comunicarea cu alte procese. Un program trebuie să citească practic simultan informaţii atât de la intrarea standard cât şi de la un socket (sau mai mulţi).

În aceste situaţii nu pot fi folosite operaţii obişnuite de citire sau scriere. Folosirea acestor operaţii are drept consecinţa blocarea thread-ului curent până la încheierea operaţiei. O posibilă soluţie este folosirea de operaţii non-blocante (spre exemplu folosirea flag-ul `O_NONBLOCK`) şi interogarea succesivă a descriptorilor de fişier. Totuşi, interogarea succesivă (polling) este o formă de aşteptare ocupată (busy waiting) şi este ineficientă.

Soluţia este folosirea unor mecanisme care permit unui thread să aştepte producerea unui eveniment I/O pe un set de descriptori. Thread-ul se va bloca până când unul din descriptorii din set poate fi folosit pentru citire/scriere. Un server care foloseşte un mecanism de acest tip are, de obicei, o structură de forma:

```
set = setul de descriptori urmăriţi
while (true) {
    aşteaptă producerea unui eveniment pe unul din descriptori
    pentru fiecare descriptor pe care s-a produs un eveniment I/O {
        tratează evenimentul I/O
    }
}
```

Detaliile variază de la o implementare la alta, dar secvenţa de pseudocod de mai sus reprezintă structura de bază pentru serverele care folosesc multiplexarea I/O.

select

O primă soluţie este utilizarea funcţiilor _select sau _pselect. Folosirea acestor funcţii conduce la blocarea thread-ului curent până la producerea unui eveniment I/O pe un set de descriptori de fişier, a unei erori pe set sau până la expirarea unui timer.

Funcţiile folosesc un set de descriptori de fişier pentru a preciza fişierele/socketii pe care thread-ul curent va aştepta producerea evenimentelor I/O. Tipul de date folosit pentru definirea acestui set este `fd_set`, care este, de obicei, o mască de biţi.

Funcţiile select şi pselect sunt definite conform POSIX.1-2001 în `sys/select.h`

```
#include <sys/select.h>

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

Deoarece şi apelul poll este specificat în standardul POSIX (deci portabilitate mare), dar oferă performanţe mai bune, nu vom insista asupra apelului select!

Avantaje:

- simplitate;
- portabilitate: funcţia select este disponibilă chiar şi in API-ul Win32;

Dezavantaje:

- lungimea setul de descriptori este definită cu ajutorul lui FD_SETSIZE, şi implicit are valoarea 64;
- este necesar ca seturile de descriptori să fie reconstruite la fiecare apel `select`;
- la apariţia unui eveniment pe unul din descriptori, toţi descriptorii puşi în set înainte de select trebuie testaţi pentru a vedea pe care din ei a apărut evenimentul;
- la fiecare apel, acelaşi set de descriptori este transmis în kernel.

poll

Funcţia poll consolidează argumentele funcţiei `select` şi permite notificarea pentru o gamă mai largă de evenimente. Funcţia se defineşte ca mai jos:

```
 #include <sys/poll.h>
 int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

Timeout-ul este specificat în milisecunde. În caz de valoare negativă, semnificaţia este de aşteptare pentru o perioadă nedefinita ("infinit").

Structura `struct pollfd` este definită în `sys/poll.h`:

```
#include <sys/poll.h>

struct pollfd {
     int fd;        /* file descriptor */
     short events;  /* evenimente solicitate */
     short revents; /* evenimente apărute */
};
```

Funcţia `poll` permite astfel aşteptarea evenimentelor descrise de vectorul `ufds` de dimensiune `nfds`.

În cadrul structurii `struct pollfd` avem:

- `events` este o mască de biţi în care se specifică evenimentele urmărite de `poll` pentru descriptorul `fd`.
- `revents` este, de asemenea, o mască de biţi completată de kernel cu evenimentele apărute în momentul în care apelul se întoarce (`POLLIN`, `POLLOUT`) sau valori predefinite (`POLLERR`, `POLLHUP`, `POLLNVAL`) pentru situaţii speciale.

În caz de succes funcţia returnează un număr diferit de zero reprezentând numărul de structuri pentru care `revents` nu e zero (cu alte cuvinte toţi descriptorii cu evenimente sau erori). Se returnează 0 dacă a expirat timpul (timeout milisecunde) şi nu a fost selectat nici un descriptor. În caz de eroare se returnează -1 şi se setează errno. De asemenea, funcţia `poll` poate fi întreruptă de semnale, caz în care va întoarce -1 şi errno va fi setat la `EINTR`.

Un exemplu de utilizare pentru `poll` este prezentat în continuare:

```
#define MAX_PFDS        32

[...]
struct pollfd pfds[MAX_PFDS];
int nfds;
int listenfd, sockfd;        /* listener socket; connection socket */

nfds = 0;

/* read user data from standard input */
pfds[nfds].fd = STDIN_FILENO;
pfds[nfds].events = POLLIN;
nfds++;

/* TODO ...  create server socket (listener) */

/* add listener socket */
pfds[nfds].fd = listenfd
pfds[nfds].events = POLLIN;
nfds++;

while (1) {        /* server loop */
    /* wait for readiness notification */
    poll(pfds, nfds, -1);

    if ((pfds[1].revents & POLLIN) != 0) {
        /* TODO ... handle new connection */
    }
    else if ((pfds[0].revents & POLLIN) != 0) {
        /* TODO ... read user data from standard input */
    }
    else {
```

```
        /* TODO ... handle message on connection sockets */
    }
}
[...]
```

Avantaje `poll`

- transmiterea setului de descriptori este mai simplă decât în cazul funcţiei `select`;
- setul de descriptori nu trebuie reconstruit la fiecare apel `poll`;

Dezavantaje poll

- ineficienţă - la apariţia unui eveniment, trebuie parcurs tot setul de descriptori pentru a găsi descriptorul pe care a apărut evenimentul;
- la fiecare apel, acelaşi set de descriptori (care poate fi mare) este copiat în kernel şi înapoi.

epoll

Funcţiile `select` şi `poll` nu sunt scalabile la un număr mare de conexiuni pentru că la fiecare apel al lor trebuie transmisă toată lista de descriptori. În astfel de situaţii, la fiecare pas, trebuie construită lista de descriptori şi apelat `poll` sau `select` care copiază tot setul în kernel. La apariţia unui eveniment va fi marcat corespunzător descriptorul. Utilizatorul trebuie să parcurgă tot setul de descriptori pentru a-şi da seama pe care dintre ei a apărut evenimentul. În acest fel se ajunge să se petreacă tot mai mult timp scanând după evenimente în setul de descriptori şi tot mai puţin timp făcând I/O.

Din acest motiv, diverse sisteme au implementat interfeţe scalabile dar non-portabile:

- `/dev/poll` pe Solaris;
- `kqueue` pe FreeBSD;
- `epoll` pe Linux.

Aceste interfeţe rezolvă problemele asociate cu `select` şi `poll` şi rezolvă problemele de scalabilitate.

Pentru a folosi epoll, trebuie inclus `sys/epoll.h`. Funcţiile asociate sunt `epoll_create`, `epoll_ctl` şi `epoll_wait`. Interfaţa `epoll` oferă funcţii pentru crearea unui obiect `epoll`, adăugarea sau eliminarea de descriptori de fişiere/socketi la obiectul epoll şi aşteptarea unui eveniment pe unul dintre descriptori.

Crearea unui obiect epoll

Pentru crearea unui obiect `epoll` se foloseşte funcţia `epoll_create`:

```
int epoll_create(int size);
```

Apelul `epoll_create` facilitează crearea unui descriptor de fişier ce va fi ulterior folosit pentru aşteptarea de evenimente. Descriptorul întors va trebui închis folosind `close`.

Argumentul `size` este ignorat in versiunile recente ale nucleului, acesta ajustând dinamic dimensiunea setului de descriptori asociat obiectului `epoll`.

Adăugarea/eliminarea descriptorilor la/de la obiectul epoll

Operaţiile de adăugare/eliminare de descriptori se realizează cu ajutorul funcţiei `epoll_ctl`:

```
int epoll_ctl(int epollfd, int op, int fd, struct epoll_event *event);
```

Apelul `epoll_ctl` permite specificarea evenimentelor care vor fi aşteptate. Câmpul `event` descrie evenimentul asociat descriptorului `fd` care poate fi adăugat, şters sau modificat în funcţie de valoarea argumentului op:

- `EPOLL_CTL_ADD`: pentru adaugare;
- `EPOLL_CTL_MOD`: pentru modificare;
- `EPOLL_CTL_DEL`: pentru ştergere.

Primul argument al apelului `epoll_ctl` (epollfd) este descriptorul întors de `epoll_create`.

Structura `struct epoll_event` specifică evenimentele aşteptate:

```
typedef union epoll_data {
     void *ptr;
     int fd;
```

```
     __uint32_t u32;
     __uint64_t u64;
} epoll_data_t;

struct epoll_event {
     __uint32_t events;      /* Epoll events */
     epoll_data_t data;      /* User data variable */
};
```

Aşteptarea unui eveniment I/O

Thread-ul curent aşteaptă producerea unui eveniment I/O la unul din descriptorii asociaţi obiectului `epoll` prin intermediul funcţiei `epoll_wait`:

```
int epoll_wait(int epollfd, struct epoll_event* events, int maxevents, int timeout);
```

Funcţia `epoll_wait` este echivalentul funcţiilor `select` şi `poll`. Este folosită pentru aşteptarea unui eveniment la unul din descriptorii asociaţi descriptorului `epollfd`.

La revenirea apelului utilizatorul nu va trebui să parcurgă toţi descriptorii configuraţi ci numai cei care au evenimente produse. Argumentul `events` va marca o zonă de memorie unde vor fi plasate maxim `maxevents` evenimente de nucleu. Presupunând că valoarea câmpului `timeout` este -1 (aşteptare nedefinită), apelul se va întoarce imediat dacă există evenimente asociate, sau se va bloca până la apariţia unui eveniment.

La fel ca şi în cazul `select/pselect` şi `poll/ppoll`, există apelul `epoll_pwait` care permite precizarea unei măşti de semnale.

Edge-triggered sau level-triggered

Interfaţa `epoll` are două comportamente posibile: `edge-triggered` sau `level-triggered`. Se poate folosi unul sau altul, în funcţie de prezenţa flag-ului `EPOLLET` la adăugarea unui descriptor în lista `epoll`.

Presupunem existenţa unui socket funcţionând în mod `non-blocant` pe care sosesc 100 de octeţi. În ambele moduri (edge sau level triggered) `epoll_wait` va raporta `EPOLLIN` pentru acel socket.

Vom presupune că se citesc 50 de octeţi din cei 100 primiţi. Diferenţa între cele două moduri de funcţionare apare la un nou apel epoll_wait. În modul level-triggered se va raporta imediat EPOLLIN. În modul edge-triggered nu se va mai raporta nimic, nici măcar la sosirea unor noi date pe socket. Se poate observa cum modul edge-triggered sesizează schimbarea stării descriptorului în relaţie cu evenimentul, iar level-triggered prezenţa stării. Modul edge-triggered este implementat mai eficient în kernel, chiar dacă pare mai greu de folosit.

În continuare, sunt prezentate câteva reguli care trebuie urmărite cu o metodă sau alta. Pentru ambele metode este recomandată folosirea socketţilor în modul non-blocant.

- Level-triggered

- Edge-triggered

Exemplu folosire epoll

Mai jos este prezentat un exemplu de utilizare a epoll echivalent cu exemplele pentru select şi poll (server care multiplexează mai multe conexiuni pe socketţi şi intrarea standard):

```
#define EPOLL_INIT_BACKSTORE        2

[...]
int listenfd, sockfd;        /* listener socket; connection socket */
struct epoll_event ev;

/* create epoll descriptor */
epfd = epoll_create(EPOLL_INIT_BACKSTORE);

/* read user data from standard input */
ev.data.fd = STDIN_FILENO;        /* key is file descriptor */
ev.events = EPOLLIN;
epoll_ctl(epfd, EPOLL_CTL_ADD, STDIN_FILENO, &ev);

/* TODO ...  create server socket (listener) */

/* add listener socket */
```

```
ev.data.fd = listenfd;        /* key is file descriptor */
ev.events = EPOLLIN;
epoll_ctl(epfd, EPOLL_CTL_ADD, listenfd, &ev);

while (1) {        /* server loop */
    struct epoll_event ret_ev;

    /* wait for readiness notification */
    epoll_wait(epfd, &ret_ev, 1, -1);

    if ((rev_ev.data.fd == listenfd && ((ret_ev.events & EPOLLIN) != 0)) {
        /* TODO ... handle new connection */
    }
    else if ((rev_ev.data.fd == STDIN_FILENO &&
                    ((ret_ev.events & EPOLLIN) != 0)) {
        /* TODO ... read user data from standard input */
    }
    else {
        /* TODO ... handle message on connection sockets */
    }
}
[...]
```

Linux - generalizarea multiplexarii

O problemă a funcţiilor de multiplexare de mai sus (`select`, `poll`, `epoll`) este aceea că sunt limitate la descriptori de fişier. Altfel spus, se pot aştepta doar evenimente asociate cu un fişier/socket: gata de citire, gata de scriere. De multe ori însă se doreşte să existe un punct comun de aşteptare a unui semnal, a unui semafor, a unui proces, a unei operaţii de intrare/ieşire, a unui timer. În Windows, acest lucru se poate realiza cu ajutorul funcţiei `WaitForMultipleObjects` şi datorită faptului că majoritatea mecanismelor din Windows sunt folosite cu ajutorul tipului de date `HANDLE`.

eventfd

Pentru a asigura în Linux posibilitate aşteptării de evenimente multiple s-a definit interfaţa `eventfd`. Cu ajutorul acestei interfeţe şi combinat cu interfeţele de multiplexare I/O existente, kernel-ul poate notifica o aplicaţie utilizator de orice tip de eveniment.

Interfaţa `eventfd` este prezentă în nucleul Linux începând cu versiunea 2.6.22 şi este suportată de către glibc începând cu versiunea 2.8.

Cele trei apeluri de bază pentru extinderea funcţionalităţii multiplexării I/O sunt:

- eventfd
- signalfd
- timerfd_create

Toate cele trei apeluri întorc un descriptor de fişier pe care se vor putea primi notificări (evenimente, semnale, timere). Operaţiile posibile pe descriptorul de fişier întors sunt:

- `write`: pentru transmiterea unui mesaj de notificare pe descriptor;
- `read`: pentru primirea unui mesaj care înseamnă primirea notificării;
- `select`, `poll`, `epoll`: pentru multiplexarea I/O;
- `close`: pentru închiderea descriptorului şi eliberarea resurselor asociate.

În următorul exemplu, apelul `eventfd` este folosit pentru notificarea procesului părinte de către procesul fiu. Codul este cel prezent în pagina de manual (man eventfd).

```
[...]
int efd;
uint64_t u;

/* create eventfd file descriptor */
efd = eventfd(0, 0);

switch (fork()) {
case 0:
    /* notify parent process */
    s = write(efd, &u, sizeof(uint64_t));

    printf("Child completed write loop\n");
```

```
    exit(EXIT_SUCCESS);

default:
    printf("Parent about to read\n");

    /* wait for notification */
    s = read(efd, &u, sizeof(uint64_t));
    exit(EXIT_SUCCESS);
[...]
```

signalfd

Apelul signalfd este folosit în mod similar pentru recepţionarea de semnale prin intermediul unui descriptor de fişier. Pentru a putea recepţiona un semnal cu ajutorul interfeţei signalfd, va trebui blocat în masca de semnale a procesului. La fel ca şi exemplul de mai sus, codul de mai jos este cel prezent în pagina de manual (man signalfd).

```
/* at this point Linux-specific headers are required to use struct signalfd_siginfo */
#include <linux/types.h>
#include <linux/signalfd.h>

#define SIZEOF_SIG      (_NSIG / 8)
#define SIZEOF_SIGSET   (SIZEOF_SIG > sizeof(sigset_t) ? \
                                 sizeof(sigset_t): SIZEOF_SIG)


    [...]
    sigset_t mask;
    int sfd;
    struct signalfd_siginfo fdsi;

    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);               /* CTRL-C */
    sigaddset(&mask, SIGQUIT);              /* CTRL-\ */

    /*
     * Block signals so that they arent handled
     * according to their default dispositions
     */

    sigprocmask(SIG_BLOCK, &mask, NULL);

    /* create signalfd descriptor */
    sfd = signalfd(-1, &mask);

    for (;;) {
        /* wait for signals to be delivered by user */
        s = read(sfd, &fdsi, sizeof(struct signalfd_siginfo));

        if (fdsi.ssi_signo == SIGINT) {
             <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("Got SIGINT\n");
        } else if (fdsi.ssi_signo == SIGQUIT) {
             <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("Got SIGQUIT\n");
             exit(EXIT_SUCCESS);
        } else {
              <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("Read unexpected
signal\n");
        }
    }
    [...]
```

Interfaţa eventfd permite unificarea mecanismelor de notificare ale kernel-ului într-un descriptor de fişier care va fi folosit de utilizator.

Linux - operaţii asincrone

În mod clasic, operaţiile de lucru cu datele aflate pe suporturi externe înseamnă utilizarea apelurilor sincrone de tipul `read`, `write` şi `fsync`. Aceste apeluri garantează faptul că la terminarea apelului datele sunt scrise/citite (de) pe suportul extern (sau în cache-ul asociat). Un astfel de apel poate întârzia continuarea fluxului de instrucţiuni curent până la terminarea operaţiei cerute.

Pentru fire de execuţie care nu au nevoie frecvent de operaţii de intrare-ieşire, această abordare funcţionează. În schimb, pentru aplicaţii specializate pe lucrul cu memoria externă, folosirea apelurilor sincrone (blocante) încetineşte semnificativ execuţia programului. Timpul necesar unui acces la memorie (cu atât mai mult memoria externă) depăşeşte cu mult timpul de execuţie a unei instrucţiuni strict aritmetice.

Linux AIO

Standardul `POSIX.1b` defineşte un nou set de operaţii I/O care pot reduce semnificativ timpul pe care o aplicaţie îl petrece aşteptând pentru I/O. Noile funcţii permit unui program să iniţieze una sau mai multe operaţii de I/O şi să-şi continue lucrul normal în timp ce operaţiile de I/O sunt executate în paralel.

Această funcţionalitate este disponibilă dacă se instalează biblioteca `libaio`:

```
so$ apt-cache search libaio
libaio-dev - Linux kernel AIO access library - development files
libaio1 - Linux kernel AIO access library - shared library
libaio1-dbg - Linux kernel AIO access library - debugging symbols
so$ sudo apt-get install libaio1 libaio-dev
```

Totodata programul care foloseşte acest API trebuie să includă fişierul header `libaio.h` şi să linkeze biblioteca `libaio`. Toate funcţiile şi structurile de care vom vorbi în continuare se pot găsi în acest fişier header. Dacă aţi instalat pachetul, fişierul se găseşte în `/usr/include/libaio.h`.

Structuri de bază Linux AIO

Structura `iocb` folosită pentru încapsularea unei operaţii asincrone. Structura este definită în header-ul `libaio.h`.

```
struct iocb {
        PADDEDptr(void *data, __pad1);        /* Return in the io completion event */
        PADDED(unsigned key, __pad2);        /* For use in identifying io requests */

        short               aio_lio_opcode;
        short               aio_reqprio;
        int                aio_fildes;     /* Perform async IO on this file descriptor */

        union {
                struct io_iocb_common               c;    /* common read/write operation */
                struct io_iocb_vector               v;    /* vectored read/write operations */
                struct io_iocb_poll              poll;
                struct io_iocb_sockaddr        saddr;        /* socket read/write operations */
        } u;
};
```

În principiu, nu se lucrează direct cu elementele din structura iocb. Pentru asta există funcţii de iniţializare:

- Pentru operaţii normale de citire/scriere:

```
void io_prep_pread(struct iocb *iocb, int fd, void *buf, size_t count, long long offset)
void io_prep_pwrite(struct iocb *iocb, int fd, void *buf, size_t count, long long offset)
```

- Pentru operaţii Vectored I/O de citire/scriere:

```
void io_prep_preadv(struct iocb *iocb,
                            int fd,
                            const struct iovec *iov,
                            int iovcnt,
                            long long offset)
void io_prep_pwritev(struct iocb *iocb,
                           int fd,
                           const struct iovec *iov,
                           int iovcnt,
                           long long offset)
```

Pentru folosirea acesteia o aplicaţia va include `libaio.h`. Un exemplu de iniţializare a acestei structuri este:

```
        #include <libaio.h>

        /* ... */

        struct iocb iocb;

        memset(&iocb, 0, sizeof(iocb));
        io_prep_pwrite(&iocb, fd, buffer, BUFER_SIZE, 0);
```

Context AIO

Orice operaţie sau set de operaţii Linux AIO sunt identificate printr-o valoare de tipul `io_context_t` ce reprezintă un context de operaţii asincrone.

Iniţializarea, respectiv distrugerea contextului se realizează cu ajutorul funcţiilor io_setup şi io_destroy:

```
#include <libaio.h>

io_context_t ctx;
int num_ops = 10;

/* crează un context de I/O asincron capabil să primească măcar num_ops evenimente */

if (io_setup(num_ops, &ctx) < 0) {
    /* handle error */
}

/* do work */
/* ... */

/* distruge contextul şi anulează toate operaţiile I/O asincrone necompletate */

if (io_destroy(ctx) < 0) {
    /* handle error */
}
```

Operaţii AIO

Pentru realizarea unei operaţii asincrone se foloseşte funcţia io_submit. Această funcţia declanşează pornirea operaţiilor asincrone definite în vectorul de pointeri de structuri `struct iocb` primit ca argument. Această funcţie nu blochează procesul curent.

```
#include <libaio.h>

#define NUM_AIO_OPS       10

struct iocb iocb[NUM_AIO_OPS];      /* array of asynchronous operations */
struct iocb *piocb[NUM_AIO_OPS];    /* array of pointers to asynchronous operations */
io_context_t ctx = 0;

/* init context, iocb */

/* fill piocb */
for (i = 0; i < NUM_AIO_OPS; i++)
    piocb[i] = &iocb[i];

/*
 * Submit NUM_AIO_OPS async operations in context 'ctx'
 * This does not wait for the operations to finish
 */

if (io_submit(ctx, NUM_AIO_OPS, piocb) < 0) {
    /* handle error */
}

/* Do some other stuff in paralel with the execution of async I/O operations */
```

Pentru aşteptarea încheierii unei operaţii AIO şi obţinerea de informaţii despre rezultatul acesteia se foloseşte funcţia io_getevents. Funcţia foloseşte structura `struct io_event` pentru a obţine informaţii despre încheierea unei operaţii asincrone. Un exemplu de utilizare este:

```
#include <libaio.h>
#define       NUM_AIO_OPS  10

io_context_t ctx = 0;
struct        io_event events[NUM_AIO_OPS];     /* aio result array */
/* ... */

/*
 * Wait _exactly_ NUM_AIO_OPS async operations to finish
 * min_nr - min nummber of async aio to finish for the function to return
 * max_nr - max nummber of async aio operations that can be returned
 */
rc = io_getevents(ctx, NUM_AIO_OPS, /* min_nr */
                       NUM_AIO_OPS, /* max_nr */
                       events,      /* vector to store completed events */
                       NULL)        /* no timeout */
if (rc < 0) {
    /* handle error */
}
```

Integrarea Linux AIO cu eventfd

Este utilă folosirea apelurilor de multiplexare I/O (select, poll, epoll) şi pentru aşteptarea încheierii operaţiilor asincrone. Pentru aceasta, interfaţa AIO a Linux 2.6 permite integrarea API-ului de operaţii asincrone cu mecanismul eventfd.

Pentru aceasta se configurează flag-ul `IOCB_FLAG_RESFD` iar câmpul resfd al structurii `struct iocb` va conţine un descriptor `eventfd` ce va fi notificat în momentul încheierii operaţiei asincrone. Acest lucru se poate configura din start apelând funcţia:

```
void io_set_eventfd(struct iocb *iocb, int eventfd)
```

Apelul `io_getevents` este în continuare util pentru a obţine informaţii despre încheierea operaţiilor. `eventfd` oferă doar mecanismul de aşteptare a acestora.

```
#include <libaio.h>
int efd;

// creare event cu valoare iniţială 0, fără flaguri speciale
efd = eventfd(0, 0)

/* ... */
struct iocb *iocb;

/* ... */
/* use eventfd */
io_set_eventfd(&iocb[i], efd);

/* ... */
u_int64_t efd_val;
if (read(efd, &efd_val, sizeof(efd_val)) < 0) {
    /* handle error */
}

<a  href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("%llu  operations  have
completed\n", efd_val);
```

Citirea din descriptorul eventfd reprezintă numărul de operaţii I/O încheiate. Această valoare va fi, de obicei, folosită ca al doilea şi al treilea argument al `io_getevents`.

Folosind integrarea operaţiilor asincrone cu eventfd şi mecanismele de multiplexare I/O (select, poll, epoll) se poate aştepta unificat încheierea unei operaţii asincrone sau sosirea de date pe socketi. (Hint: util pentru `Tema 5`)

Zero-copy I/O

Linux - splice

Este un apel de sistem ce permite transferul de date intre 2 descriptori de fişier, din care cel puţin unul este pipe. Avantajul este ca nu se foloseşte un buffer (byte array) în userspace. Pentru o scurtă introducere puteţi citi descrierea de apelului splice pe Wikipedia, iar pentru o mai bună aprofundare a avantajului oferit de apel, citiţi descrierea de pe LKML.

```
#define _GNU_SOURCE // trebuie definit pentru că splice este o extensie nespecificată de standardele POSIX/SYSV/BSD/etc.
#include <fcntl.h>
long splice(int fd_in, loff_t *off_in, int fd_out, loff_t *off_out, size_t len, unsigned int flags);
```

- dacă descriptorul fd_in reprezintă un pipe, atunci pointer-ul la offset off_in trebuie să fie NULL
- altfel:
  - dacă off_in este NULL, atunci datele sunt citite de la fd_in de la offset-ul curent, acesta modificându-se corespunzător
  - altfel, off_in trebuie sa fie un pointer la un întreg care reprezintă offset-ul de start de la care se va face citirea, iar offset-ul propriu descriptorului fd_in rămâne neschimbat
- comportamentul de mai sus este valabil şi pentru fd_out şi off_out, la scriere
- parametrul len specifica numărul maxim de octeţi transferaţi
- masca de biţi flags poate specifică o operaţie non-blocantă sau hint-uri pentru nucleu. Citiţi pagina de manual a funcţiei pentru detalii.

Exemplu:

```
int pipe, file1, file2;
loff_t offset = 0;
size_t count = 4096;

// ... deschideri fişiere, creare pipe
```

```
splice(file1, &offset, pipe, NULL, count, 0);

splice(pipe, NULL, file2, &offset, count, 0);
```

Vectored I/O

Vectored I/O (sau scatter/gather I/O) reprezintă o metodă prin intermediul căreia un singur apel permite scrierea de date din mai multe buffere către un flux de ieşire sau citirea de date de la un flux de ieşire în mai multe buffere. Bufferele sunt precizate ca un vector de buffere, de unde şi denumirea de vectored I/O.

Apelurile din clasa vectored I/O sunt utile în momentul în care datele sunt disparate/dezasamblate în memorie şi se doreşte "concatenarea" acestora într-un singur flux de scriere sau "desfacerea" acestora dintr-un flux de citire. Un exemplu îl reprezintă pachetele de reţea în care headerele, datele şi trailerele se găsesc, de obicei, în locaţii de memorie diferite pentru a facilita prelucrarea acestora. Folosirea Vectored I/O permite asamblarea/dezasamblarea pachetului în/din mai multe zone de memorie printr-o singură operaţie. Nu este nevoie de crearea unui buffer nou cu pachetele concatenate, drept pentru care Vectored I/O poate fi considerat o formă de zero-copy.

Apeluri:

- UNIX: readv, writev.
- Windows (fişiere) ReadFileScatter, WriteFileGather.
- Windows (socketi) WSARecv,WSASend.

readv/writev

Funcţiile readv şi writev sunt folosite în sistemele Unix ca operaţii de tipul vectored I/O. Structura de bază folosită de aceste funcţii este struct iovec:

```
#include <sys/uio.h>

struct iovec {
        void *iov_base;   /* Starting address */
        size_t iov_len;   /* bytes to transfer */
}
```

Un apel readv sau writev va permite recepţionarea/transmiterea unui număr de buffere reprezentate de structura struct iovec. Funcţiile întorc numărul total de octeţi citiţi sau scrişi.

writev scrie datele către care trimit elementele din iov în fişier, în ordinea în care acestea apar în vector

```
#include <sys/uio.h>

/* ... */
char *str0 = "Ana ";
char *str1 = "are multe ";
char *str2 = "mere, pere, etc.";
struct iovec iov[3];
ssize_t nwritten;

iov[0].iov_base = str0;
iov[0].iov_len  = strlen(str0);
iov[1].iov_base = str1;
iov[1].iov_len  = strlen(str1);
iov[2].iov_base = str2;
iov[2].iov_len  = strlen(str2);

nwritten = writev(fd, iov, 3);
if (nwritten < 0) {
     /* handle error */
}
```

Exerciţii

În rezolvarea laboratorului folosiţi arhiva de sarcini lab11-tasks.zip

1. (**1 punct**) Exemplu folosire `poll`
   - Intraţi în directorul `1-pollpipe`
   - Programul creează folosind `fork` o aplicaţie de test pentru `poll`. Aplicaţia foloseşte un server (părintele) şi

`CLIENT_COUNT` clienţi (copiii) ce comunică prin pipe-uri anonime.
- server-ul:
  - construieşte un vector de pipe-uri (în funcţia `main`);
  - creează clienţii;
  - blochează în aşteptarea datelor pe acestea şi tipăreşte datele primite;
  - termină execuţia după ce a primit date de la fiecare client;
- clienţii:
  - aşteaptă un număr aleator de secunde (mai mic decât 10);
  - scriu în pipe-ul corespunzător un şir de `MSG_SIZE` caractere de forma `:` ('a' + random() % 30);
  - scrierile şi citirile în pipe-uri de până la PIPE_BUF octeţi (4096 pe Linux) sunt atomice.
- **Urmariti** codul pentru a vedea un exemplu de folosire al functiei `poll`
- Compilati si rulati programul.
- Pentru nelamuriri puteti consulta secţiunea poll şi pipe-uri în Linux.

2. (**2 puncte**) Utilizare `epoll`
   - Modificaţi codul anterior pentru a folosi `epoll`
   - Intrati in folderul `2-epollpipe` si urmariti comentariile marcate cu *TODO*
   - **Hints**:
     - Consultaţi secţiunea epoll
3. (**2 puncte**) Utilizare `eventfd`
   - În execiţiul anterior, notificaţi serverul de terminarea unui client utilizând `eventfd`.
   - Clientul
     - va scrie un mesaj de notificare serverului înainte de ieşire care va conţine în primii 32 de biţi `MAGIC_EXIT`, iar în ultimii 32 de biţi indexul clientului
   - Serverul
     - creează eventfd-ul şi îl adaugă la descriptor-ul epoll;
     - la primirea unui eveniment prin acesta, dacă primii 32 biţi sunt MAGIC_EXIT, atunci **scoate** capătul pipe-ului corespunzător din epoll şi îl inchide;
   - **Hints**:
     - Consultaţi secţiunea eventfd
4. (**5 puncte**)
   1. (**3 puncte**) Asynchronous I/O (KAIO)
      - Intraţi în directorul `4-kaio`.
      - Parcurgeţi fişierul kaio.c.
      - Completaţi zonele lipsă pentru a programa scrierea a 4 fişiere cu numele date de variabila files.
      - Folosiţi API-ul KAIO (io_setup, io_destroy, io_submit, io_getevents).
      - Folosiţi **doar** io_getevents pentru aşteptarea încheierii operaţiilor asincrone.
      - **Hints**:
        - Parcurgeţi secţiunea Linux AIO.
        - Consultaţi exemplul lui Davide Libenzi.
        - Urmăriţi comentariile cu *TODO 1*
      - Compilaţi şi rulaţi programul. Va trebui să aveţi 4 fişiere de dimensiune 8192 octeţi create în `/tmp`.
   2. (**2 puncte**) Asynchronous I/O (KAIO) + eventfd
      - Folosiţi eventfd pentru aşteptarea operaţiilor asincrone.
        - Decomentaţi în `kaio.c` linia cu #define USE_EVENTFD
        - La iniţializarea structurilor `iocb`, folosiţi funcţia `io_set_eventfd` pentru a activarea folosirea `eventfd`
        - Completaţi funcţia `wait_aio` pentru a şstepta terminarea operaţiilor asincrone folosind `eventfd`
      - **Hints**:
        - Parcurgeţi secţiunea Linux AIO.
        - Urmăriţi comentariile cu *TODO 2*
        - Consultaţi exemplul lui Davide Libenzi.
      - Compilaţi şi rulaţi programul. Va trebui să aveţi 4 fişiere de dimensiune 8192 octeţi create în `/tmp`.

BONUS

1. **signalpipe (1 so karma)** Utilizarea `signalfd`
   - Modificati codul de la exercitiu 2 pentru a permite notificarea de terminare a clientilor bazata pe semnale.
   - server-ul:
     - creează un descriptor via signalfd pentru semnalul SIGCHLD si-l adauga la epoll
     - la primirea unui semnal, prin read(2) pe descriptorul creat, determina PID-ul copilului defunct, afiseaza un mesaj si scote pipe-ul din epoll.
   - **Hints**:
     - Consultaţi secţiunea signalfd
     - man signalfd

Soluţii