Laborator 11 - Operații IO avansate - Linux

Materiale Ajutătoare

- lab11-slides.pdf [http://elf.cs.pub.ro/so/res/laboratoare/lab11-slides.pdf]
- lab11-refcard.pdf [http://elf.cs.pub.ro/so/res/laboratoare/lab11-refcard.pdf]

Nice to read

- TLPI - Chapter 63, Alternative I/O models

Link-uri către secțiuni utile

Linux

- Linux - multiplexarea I/O
  - select
  - poll
  - epoll
- Linux - generalizarea multiplexării
  - eventfd
  - signalfd
- Linux - operații asincrone
- Zero-copy I/O
- Vectored I/O

Linux - multiplexarea I/O

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

În aceste situații nu pot fi folosite operații obișnuite de citire sau scriere. Folosirea acestor operații are drept consecință blocarea thread-ului curent până la încheierea operației. O posibilă soluție este folosirea de operații non-blocante (spre exemplu folosirea flag-ul `O_NONBLOCK`) și interogarea succesivă a descriptorilor de fișier. Totuși, interogarea succesivă (polling) este o formă de *busy waiting* și este ineficientă.

> Atunci când un fișier este deschis (folosind apelul `open`) cu flag-ul `O_NONBLOCK` operațiile pe acel descriptor de fișier nu se vor bloca în așteptarea terminării operației. În acest caz apelul `read` va întoarce -1 și `errno` este setat la valoarea `EAGAIN` sau `EWOULDBLOCK` dacă nu sunt date de citit. Asemănător în cazul apelului `write`.

Soluția este folosirea unor mecanisme care permit unui thread să aștepte producerea unui eveniment I/O pe un set de descriptori. Thread-ul se va bloca până când unul dintre descriptorii din set poate fi folosit

pentru citire/scriere. Un server care folosește un mecanism de acest tip are, de obicei, o structură de forma:

```
set = setul de descriptori urmăriți
while (true) {
    așteaptă producerea unui eveniment pe unul dintre 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 secvența de pseudocod de mai sus reprezintă structura de bază pentru serverele care folosesc *multiplexarea I/O*.

select

O primă soluție este utilizarea funcțiilor select [http://linux.die.net/man/2/select] sau pselect [http://linux.die.net/man/2/select]. Folosirea acestor funcții conduce la blocarea thread-ului curent până la producerea unui eveniment I/O pe un set de descriptori de fișier, a unei erori pe set sau până la expirarea unui timer.

Funcțiile folosesc un set de descriptori de fișier pentru a preciza fișierele/socketţii pe care thread-ul curent va aștepta producerea evenimentelor I/O. Tipul de date folosit pentru definirea acestui set este `fd_set`, care este, de obicei, o mască de biți.

Funcțiile `select` și `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
);
```

Nu vom insista asupra apelului `select`, căci standardul POSIX specifică un alt apel, `poll`, ce oferă o performanţă mai bună.

Avantaje:

- simplitate;
- portabilitate: funcția select este disponibilă chiar și in API-ul Win32;

Dezavantaje:

- lungimea setului de descriptori este definită cu ajutorul lui FD_SETSIZE, și implicit are valoarea 64;
- este necesar ca seturile de descriptori să fie reconstruite la fiecare apel `select`;
- la apariția unui eveniment pe unul dintre descriptori, toți descriptorii puși în set înainte de `select` trebuie testați pentru a vedea pe care dintre ei a apărut evenimentul;
- la fiecare apel, același set de descriptori este transmis în kernel.

poll

Funcția poll [http://linux.die.net/man/2/poll] consolidează argumentele funcției `select` și permite notificarea pentru o gamă mai largă de evenimente. Funcția se definește 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, semnificația este de așteptare pentru o perioadă nedefinită (“infinit”).

Structura `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 */
};
```

Funcția `poll` permite astfel așteptarea evenimentelor descrise de vectorul `ufds` de dimensiune `nfds`.

În cadrul structurii `pollfd` avem:

- `events` este o mască de biți în care se specifică evenimentele urmărite de `poll` pentru descriptorul `fd` (`POLLIN` - există date ce pot fi citite, `POLLOUT` - se pot scrie date).
- `revents` este, de asemenea, o mască de biți completată de kernel cu evenimentele apărute în momentul în care apelul se întoarce (`POLLIN`, `POLLOUT`) sau cu valori predefinite (`POLLERR`, `POLLHUP`, `POLLNVAL`) pentru situații speciale.

În caz de succes, funcția returnează un număr diferit de zero reprezentând numărul de structuri pentru care `revents` nu e zero (cu alte cuvinte toți descriptorii cu evenimente sau erori). Se returnează 0 dacă a expirat timpul (timeout milisecunde) și nu a fost selectat nici un descriptor. În caz de eroare se returnează -1 și se setează errno. De asemenea, funcția `poll` poate fi întreruptă de semnale, caz în care va întoarce -1 și 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 funcției `select`;
- setul de descriptori nu trebuie reconstruit la fiecare apel `poll`.

Dezavantaje `poll`:

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

epoll

Funcțiile `select` și `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 situații, la fiecare pas, trebuie construită lista de descriptori și apelat `poll` sau `select` care copiază tot setul în kernel. La apariția unui eveniment va fi marcat corespunzător descriptorul. Utilizatorul trebuie să parcurgă tot setul de descriptori pentru a-și 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 și tot mai puțin timp făcând I/O.

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

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

Aceste interfețe rezolvă problemele asociate cu `select` și `poll`, dar și problemele de scalabilitate.

Pentru a folosi epoll [http://linux.die.net/man/4/epoll], trebuie inclus `sys/epoll.h`. Interfața `epoll` oferă funcții pentru:

- crearea unui obiect `epoll` (`epoll_create`);
- adăugarea sau eliminarea de descriptori de fișiere/socketi la obiectul epoll (`epoll_ctl`);
- așteptarea unui eveniment pe unul dintre descriptori (`epoll_wait`).

Crearea unui obiect epoll

Pentru crearea unui obiect `epoll` se folosește funcția epoll_create [http://linux.die.net/man/2/epoll_create]:

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

Apelul `epoll_create` facilitează crearea unui descriptor de fișier ce va fi ulterior folosit pentru așteptarea de evenimente. Descriptorul întors va trebui la final închis folosind apelul `close`.

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

Adăugarea/eliminarea descriptorilor la/de la obiectul epoll

Operațiile de adăugare/eliminare de descriptori se realizează cu ajutorul funcției epoll_ctl [http://linux.die.net/man/2/epoll_ctl]:

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

Apelul `epoll_ctl` permite specificarea evenimentelor care vor fi așteptate.

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

Câmpul `event` descrie evenimentul asociat descriptorului `fd` care poate fi adăugat, șters sau modificat în funcție de valoarea argumentului op:

- `EPOLL_CTL_ADD`: pentru adăugare;
- `EPOLL_CTL_MOD`: pentru modificare;
- `EPOLL_CTL_DEL`: pentru ștergere.

Structura `epoll_event` specifică evenimentele așteptate:

```
typedef union epoll_data {
     void *ptr;              /* Pointer to user-defined data */
     int fd;                 /* File descriptor */
     __uint32_t u32;         /* 32-bit integer */
     __uint64_t u64;         /* 64-bit integer */
} epoll_data_t;

struct epoll_event {
     __uint32_t events;      /* Epoll events (bit mask) */
     epoll_data_t data;      /* User data*/
};
```

Exemple de evenimente:

- `EPOLLIN` - fișierul este disponibil pentru citire,
- `EPOLLOUT` - fișierul este disponibil pentru scriere.

Așteptarea unui eveniment I/O

Thread-ul curent așteaptă producerea unui eveniment I/O la unul dintre descriptorii asociați obiectului `epoll` prin intermediul funcției epoll_wait [http://linux.die.net/man/2/epoll_wait]:

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

Funcția `epoll_wait` este echivalentul funcțiilor `select` și `poll`. Este folosită pentru așteptarea unui eveniment la unul din descriptorii asociați obiectului `epoll`.

La revenirea apelului, utilizatorul nu va trebui să parcurgă toți descriptorii configurați, 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 (așteptare

nedefinită), apelul se va întoarce imediat dacă există evenimente asociate, sau se va bloca până la apariția unui eveniment.

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

Edge-triggered sau level-triggered

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

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

Vom presupune că se citesc 50 de octeți din cei 100 primiți. Diferența între cele două moduri de funcționare 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 relație cu evenimentul, iar level-triggered prezența stării. Modul edge-triggered este implementat mai eficient în kernel, chiar dacă pare mai greu de folosit.

În continuare, în cele două spoilere de mai jos, sunt prezentate câteva reguli care trebuie urmărite cu o metodă sau alta. Pentru ambele metode este recomandată folosirea socketilor în modul non-blocant.

- La apariția unui eveniment EPOLLIN se poate citi oricât, la următorul apel epoll_wait se va raporta din nou EPOLLIN dacă mai sunt date de citit.
- EPOLLOUT nu trebuie configurat inițial pentru un socket pentru că astfel epoll_wait va raporta imediat că este loc de scris în buffer (inițial bufferul de scriere asociat cu socketul este gol). Acesta este o formă deghizată de busy waiting. Folosirea corectă implică scrierea normală pe socket și numai dacă la un moment dat funcțiile de scriere raportează că nu mai este loc de scriere in buffer (EAGAIN), se va activa EPOLLOUT pe descriptorul respectiv și salva ce mai este de scris. Când în sfârșit se face loc, se va raporta EPOLLOUT și atunci se poate încerca să se scrie datele păstrate. Dacă se reușește scrierea lor integrală, trebuie eliminat flagul EPOLLOUT pentru a nu intra într-un nou ciclu de busy-waiting. În concluzie, EPOLLOUT trebuie activat doar când nu se reușește scrierea integrală a datelor și scos imediat după ce acestea au fost scrise.

- La apariția unui eveniment EPOLLIN pe un descriptor, trebuie citit tot ce se poate citi înainte de reapelarea epoll_wait, altfel nu va mai fi raportat EPOLLIN niciodată.
- Pentru scrierea folosind edge-triggered se poate activa de la început EPOLLOUT. Aceasta va cauza apariția unui eveniment EPOLLOUT imediat după apelarea epoll_wait (pentru că bufferul de scriere este gol) care ar trebui ignorat. La următorul apel epoll_wait nu se mai generează EPOLLOUT pentru că nu s-a schimbat starea de la ultimul apel. Dacă la un moment dat se încearcă scrierea unor date pe socket și acestea nu pot fi scrise integral, la următorul epoll_wait se generează EPOLLOUT, pentru că s-a schimbat starea socketului. Mai pe scurt, asta are ca efect faptul că nu mai trebuie activat/deactivat EPOLLOUT ca în cazul level-triggered.

Exemplu folosire epoll

Mai jos este prezentat un exemplu de utilizare a `epoll` echivalent cu exemplele pentru select și poll (server care multiplexează mai multe conexiuni pe socketi și 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 ((ret_ev.data.fd == listenfd) && ((ret_ev.events & EPOLLIN) != 0)) {
        /* TODO ... handle new connection */
    }
    else if ((ret_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 multiplexării

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

eventfd

Pentru a asigura în Linux posibilitatea așteptării de evenimente multiple s-a definit interfața `eventfd`. Cu ajutorul acestei interfețe și combinat cu interfețele de multiplexare I/O existente, kernel-ul poate notifica o aplicație utilizator de orice tip de eveniment.

Interfața `eventfd` permite unificarea mecanismelor de notificare ale kernel-ului într-un descriptor de fișier care va fi folosit de utilizator.

Cele trei apeluri de bază pentru extinderea funcționalității multiplexării I/O sunt: eventfd [http://linux.die.net/man/2/eventfd], signalfd [http://linux.die.net/man/2/signalfd] și timerfd_create [http://www.unix.com/man-page/Linux/2/timerfd_create/].

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

int eventfd(unsigned int initval, int flags);
```

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

int signalfd(int fd, const sigset_t *mask, int flags);
```

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

int timerfd_create(int clockid, int flags);
```

Toate cele trei apeluri întorc un descriptor de fișier pe care se vor putea primi notificări (evenimente, semnale, timere). Operațiile posibile pe descriptorul de fișier 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 și 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 [http://linux.die.net/man/2/signalfd] este folosit în mod similar pentru recepționarea de semnale prin intermediul unui descriptor de fișier. Pentru a putea recepționa un semnal cu ajutorul interfeței signalfd, va trebui blocat în masca de semnale a procesului. La fel ca și 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 aren't handled
     * according to their default dispositions
     */

    sigprocmask(SIG_BLOCK, &mask, NULL);

    /* create signalfd descriptor */
    sfd = signalfd(-1, &mask, 0);

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

        if (fdsi.ssi_signo == SIGINT) {
             printf("Got SIGINT\n");
        } else if (fdsi.ssi_signo == SIGQUIT) {
             printf("Got SIGQUIT\n");
             exit(EXIT_SUCCESS);
        } else {
             printf("Read unexpected signal\n");
        }
    }
    [...]
```

Linux - operații asincrone

În mod clasic, operațiile de lucru cu datele aflate pe suporturi externe înseamnă utilizarea apelurilor sincrone de tipul `read`, `write` și `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 instrucțiuni curent până la terminarea operației cerute.

Pentru fire de execuție care nu au nevoie frecvent de operații de intrare-ieșire, această abordare funcționează. În schimb, pentru aplicații specializate pe lucrul cu memoria externă, folosirea apelurilor sincrone (blocante) încetinește semnificativ execuția programului. Timpul necesar unui acces la memorie (cu atât mai mult memoria externă) depășește cu mult timpul de execuție a unei instrucțiuni strict aritmetice.

Linux AIO

Standardul `POSIX.1b` definește un nou set de operații I/O care pot reduce semnificativ timpul pe care o aplicație îl petrece așteptând pentru I/O. Noile funcții permit unui program să inițieze una sau mai multe operații de I/O și să-și continue lucrul normal în timp ce operațiile de I/O sunt executate în paralel.

Această funcționalitate 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
```

Totodată, programul care folosește acest API trebuie să includă fișierul header `libaio.h` [`http://libaio.sourcearchive.com/documentation/0.3.109-1ubuntu1/libaio_8h_source.html`] și să link-eze biblioteca `libaio`. Toate funcțiile și structurile de care vom vorbi în continuare se pot găsi în acest fișier header. Dacă ați instalat pachetul, fișierul se găsește în `/usr/include/libaio.h`.

Structuri de bază Linux AIO

Structura `iocb` folosită pentru încapsularea unei operații asincrone. Structura este definită în header-ul `libaio.h` [http://libaio.sourcearchive.com/documentation/0.3.109-1ubuntu1/libaio_8h_source.html].

```
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 funcții de inițializare:

- Pentru operații 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 operații 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 aplicație va include `libaio.h` [http://libaio.sourcearchive.com/documentation/0.3.109-1ubuntu1/libaio_8h_source.html]. Un exemplu de inițializare 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 operație sau set de operații Linux AIO sunt identificate printr-o valoare de tipul `io_context_t` ce reprezintă un context de operații asincrone.

Inițializarea, respectiv distrugerea contextului se realizează cu ajutorul funcțiilor io_setup [http://linux.die.net/man/2/io_setup] și io_destroy [http://linux.die.net/man/2/io_destroy]:

```
#include <libaio.h>

int io_setup(unsigned nr_events, aio_context_t *ctxp);
```

```
#include <libaio.h>

int io_destroy(aio_context_t ctx);
```

Un exemplu de inițializare și distrugere a contextului:

```
#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 și anulează toate operațiile I/O asincrone necompletate */

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

Operații AIO

Pentru realizarea unei operații asincrone se folosește funcția io_submit [http://linux.die.net/man/2/io_submit]. Această funcție declanșează pornirea operațiilor asincrone definite în vectorul de pointeri de structuri `iocb` primit ca argument. Această funcție nu blochează procesul curent.

```
#include <libaio.h>

int io_submit(
     aio_context_t ctx_id,
     long nr,
     struct iocb **iocbpp
);
```

Un exemplu de utilizare este:

```
#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 așteptarea încheierii unei operații AIO și obținerea de informații despre rezultatul acesteia se folosește funcția io_getevents [http://linux.die.net/man/2/io_getevents]. Funcția folosește structura `io_event` pentru a obține informații despre încheierea unei operații asincrone.

```
#include <linux/time.h>
#include <libaio.h>

int io_getevents(
     aio_context_t ctx_id,
     long min_nr,
     long nr,
     struct io_event *events,
     struct timespec *timeout
);
```

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`) și pentru așteptarea încheierii operațiilor asincrone. Pentru aceasta, interfața AIO a Linux 2.6 permite integrarea API-ului de operații asincrone cu mecanismul eventfd.

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

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

Apelul `io_getevents` [http://linux.die.net/man/2/io_getevents] este în continuare util pentru a obține informații despre încheierea operațiilor. `eventfd` oferă doar mecanismul de așteptare a acestora.

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

/* creare event cu valoare inițială 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 */
}

printf("%llu operations have completed\n", efd_val);
```

Citirea din descriptorul eventfd reprezintă numărul de operații I/O încheiate. Această valoare va fi, de obicei, folosită ca al doilea și al treilea argument al `io_getevents` [http://linux.die.net/man/2/io_getevents].

Util pentru `Tema 5`: Folosind integrarea operațiilor asincrone cu eventfd și mecanismele de multiplexare I/O (`select`, `poll`, `epoll`) se poate aștepta unificat încheierea unei operații asincrone sau sosirea de date pe socketi.

Zero-copy I/O

Linux - splice

Este un apel de sistem ce permite transferul de date între 2 descriptori de fișier, dintre care cel puțin unul este pipe. Avantajul este că nu se folosește un buffer (byte array) în userspace.

```
#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 să 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 și pentru `fd_out` și `off_out`, la scriere

- parametrul len specifică numărul maxim de octeți transferați
- masca de biți flags poate specifica o operație non-blocantă sau hint-uri pentru nucleu. Citiți pagina de manual a funcției pentru mai multe detalii.

Exemplu de folosire:

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

/* ... deschideri fișiere, creare pipe */

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

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

O altă funcție foarte utilă și care folosește zero-copy este sendfile [http://linux.die.net/man/2/sendfile]

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 ieșire sau citirea de date de la un flux de intrare în mai multe buffere. Bufferele sunt precizate ca un vector de buffere, de unde și denumirea de vectored I/O.

Apelurile din clasa vectored I/O sunt utile în momentul în care datele sunt disparate/dezasamblate în memorie și se dorește "concatenarea" acestora într-un singur flux de scriere sau "desfacerea" acestora dintr-un flux de citire. Un exemplu îl reprezintă pachetele de rețea în care headerele, datele și trailerele se găsesc, de obicei, în locații 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 operație. Nu este nevoie de crearea unui buffer nou cu pachetele concatenate, drept care Vectored I/O poate fi considerat o formă de zero-copy.

Apeluri:

- UNIX: readv [http://linux.die.net/man/2/readv], writev [http://linux.die.net/man/2/writev].
- Windows (fișiere) ReadFileScatter, WriteFileGather.
- Windows (socketi) WSARecv,WSASend.

readv/writev

Funcțiile readv [http://linux.die.net/man/2/readv] și writev [http://linux.die.net/man/2/writev] sunt folosite în sistemele Unix ca operații de tipul vectored I/O. Structura de bază folosită de aceste funcții este `struct iovec`:

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

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

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

ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
```

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

ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

Un apel readv [http://linux.die.net/man/2/readv] sau writev [http://linux.die.net/man/2/writev] va permite recepționarea/transmiterea unui număr de buffere reprezentate de structura `iovec`. Funcțiile întorc numărul total de octeți citiți sau scriși.

Apelul writev [http://linux.die.net/man/2/writev] scrie datele (reprezentate de elementele din `iov` în fișier), 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 */
}
```

Exercițiul 0 - Joc interactiv

- Detalii desfășurare joc [http://ocw.cs.pub.ro/courses/so/meta/notare#joc_interactiv].

Linux

În rezolvarea laboratorului folosiți arhiva de sarcini lab11-tasks.zip [http://elf.cs.pub.ro/so/res/laboratoare/lab11-tasks.zip]

Acest laborator se desfășoară în echipe. O echipă este formată din 2 studenți. În cazul în care numărul de studenți prezenți este impar, o singură echipă va avea dreptul de a avea 3 studenți.

Discutați exercițiile și colaborați pe parcursul întregului laborator. Have fun!

Exercițiul 1 - poll

Intrați în directorul `1-pollpipe`. Parcurgeți fișierul `poll.c` pentru a vedea un exemplu de folosire al funcției `poll`.

Programul creează folosind `fork` o aplicație de test pentru `poll`. Aplicația folosește un server (părintele) și `CLIENT_COUNT` clienți (copiii) ce comunică prin pipe-uri anonime.

Server-ul:

- construiește un vector de pipe-uri (în funcția `main`);
- creează clienții;
- se blochează în așteptarea datelor de la clienți și tipărește datele primite;
- termină execuția după ce a primit date de la fiecare client;

Clienții:

- așteaptă un număr aleator de secunde (mai mic decât 10);
- scriu în pipe-ul corespunzător un șir de MSG_SIZE caractere de forma

```
<pid>:<caracter random> ('a' + random() % 30)
```

- scrierile și citirile în pipe-uri de până la PIPE_BUF octeți (4096 pe Linux) sunt atomice.

Compilați și rulați programul. Pentru nelămuriri puteti consulta secțiunea poll și pipe-uri în Linux.

Exercițiul 2 - epoll

Intrați în directorul 2-epollpipe și parcurgeți fișierul epoll.c. Considerând cerința de la exercițiul anterior, în loc de poll folosiți epoll.

În partea de inițializare realizați următoarele operații:

- Inițializați, înainte de ciclul for, handle-ul de epoll folosing epoll_create.
- Adăugați câte un eveniment pentru fiecare pipe (folosind variabila ev) folosind epoll_ctl cu opțiunea EPOLL_CTL_ADD.
  - Câmpul events al variabilei ev îl veți inițializa la EPOLLIN.
  - Câmpul data.fd al variabilei ev îl veți inițializa la capătul de citire al pipelului.
- Închideți capătul de scriere al pipe-ului.

În partea de așteptare realizați, în bucla while următoarele operații:

- Așteptați un eveniment folosind funcția epoll_wait.
  - Descriptorul indicat în structura întoarsă de epoll_wait (adică ev.data.fd) este capătul de citire al pipe-ului.
- Din pipe citiți mesajul trimis de client, folosind read.
- Eliminați pipe-ul din multiplexor (folosind epoll_ctl cu opțiunea EPOLL_CTL_DEL) și închideți-l folosind close.
- Incremenetați valoarea variabilei recv_msgs.

Exercițiul 3 - eventfd

Pornind de la codul scris pentru execițiul anterior, notificați serverul de terminarea unui client utilizând eventfd. Clientul transmite mesaje către server și, când dorește să închidă comunicația, va trimite un mesaj pe canalul de control reprezentat de eventfd. Acum, epoll este folosit pentru a demultiplexa atât descriptorii de pipe, cât și descriptorul de eventfd.

Decomentați în epoll.c linia cu

```
#define USE_EVENTFD
```

Clientul va scrie mesaje în pipe-ul aferent, iar la sfârșit va genera un mesaj de notificare. Acesta va folosi funcția set_event definită în program. Mesajul de notificare este un număr pe 64 de biți organizat astfel:

- Primii 32 de biți conțin valoarea definită de macro-ul MAGIC_EXIT.
- Ultimii 32 de biți conțin indexul clientului.

Serverul va adăuga descriptorul de eventfd în epoll (cu eveniment de tipul EPOLLIN). Evenimentele vor fi așteptate folosind epoll_wait (așa cum făcea și până acum). Apelul epoll_wait se întoarce când un

descriptor din `epoll` are informații de citit. Descriptorul întors în urma `epoll_wait` (identificat de câmpul `ev.data.fd`) poate fi descriptor de pipe sau poate fi descriptorul de `eventfd`.

Dacă descriptorul întors în urma `epoll_wait` este descriptorul de `eventfd`, atunci veți citi 64 de biți de pe acest descriptor (folosind `read`). Dacă primii 32 biți sunt valoarea descrisă de macro-ul `MAGIC_EXIT`, atunci **scoate** capătul pipe-ului corespunzător din `epoll` și închide acel capăt (adică folosește `epoll_ctl` cu opțiunea `EPOLL_CTL_DEL` și apoi `close`). Pentru a extrage ultimii 32 de biți din mesajul de 64 de biți primit pe descriptorul de `eventfd`, reprezentând indexul clientului, folosiți funcția `get_index` definită local în program.

Exercițiul 4 - async I/O (KAIO)

Intrați în directorul `4-kaio` și parcurgeți fișierul `kaio.c`. Completați zonele lipsă pentru a programa scrierea a 4 fișiere cu numele date de variabila `files`.

Folosiți API-ul KAIO (io_setup, io_destroy, io_submit, io_getevents). Folosiți **doar** io_getevents pentru așteptarea încheierii operațiilor asincrone.

Parcurgeți secțiunea Linux AIO și consultați exemplul lui Davide Libenzi [http://www.xmailserver.org/eventfd-aio-test.c]. Urmăriți comentariile cu *TODO 1*

Compilați și rulați programul. Va trebui să aveți 4 fișiere de dimensiune 8192 octeți create în `/tmp`.

Atenţie! În cazul în care, la compilare, header-ul 'libaio' nu este găsit rulaţi

```
so$ sudo apt-get install libaio1 libaio-dev
```

Exercițiul 5 - async I/O (KAIO)

Folosiți `eventfd` pentru așteptarea operațiilor asincrone. Porniți de la codul scris pentru execițiul anterior.

Decomentați în `kaio.c` linia cu

```
#define USE_EVENTFD
```

La inițializarea structurilor `iocb`, folosiți funcția `io_set_eventfd` pentru a activa folosirea `eventfd`. Completați funcția `wait_aio` pentru a aștepta terminarea operațiilor asincrone folosind `eventfd`.

Parcurgeți secțiunea Linux AIO și urmăriți comentariile cu *TODO 2* . Consultați exemplul lui Davide Libenzi [http://www.xmailserver.org/eventfd-aio-test.c].

Compilați și rulați programul. Va trebui să aveți 4 fișiere de dimensiune 8192 octeți create în `/tmp`.

BONUS

1. `signalfd`
   - Modificați codul de la exercițiul 2 pentru a permite notificarea de terminare a clienților bazată pe semnale.
   - Server-ul:
     - creează un descriptor via signalfd pentru semnalul SIGCHLD și îl adaugă la epoll
     - la primirea unui semnal, prin read(2) pe descriptorul creat, determină PID-ul copilului defunct, afișează un mesaj și scoate pipe-ul din epoll.
   - **Hints**:

- Consultați secțiunea signalfd
- man signalfd [http://linux.die.net/man/2/signalfd]

Soluții

Soluţii laborator 11 [http://elf.cs.pub.ro/so/res/laboratoare/lab11-sol.zip]

so/laboratoare/laborator-11.txt · Last modified: 2019/05/12 21:33 by bogdan.purcareata