Operatii IO avansate (1)

Contents

Linux - multiplexarea I/O

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

În aceste situaii nu pot fi folosite operaii obinuite de citire sau scriere. Folosirea acestor operaii are drept consecina blocarea thread-ului curent până la încheierea operaiei. O posibilă soluie este folosirea de operaii non-blocante (spre exemplu folosirea flag-ul `O_NONBLOCK`) i interogarea succesivă a descriptorilor de fiier. Totui, interogarea succesivă (polling) este o formă de ateptare ocupată (busy waiting) i este ineficientă.

Soluia este folosirea unor mecanimse care permit unui thread să atepte 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 folosete un mecanism de acest tip are, de obicei, o structură de forma:

```
set = setul de descriptori urmării
```

```
while (true) {
    ateaptă 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 secvena de pseudocod de mai sus reprezintă structura de bază pentru serverele care folosesc multiplexarea I/O.

select

O primă soluie este utilizarea funciilor `select` sau `pselect`. Folosirea acestor funcii conduce la blocarea thread-ului curent până la producerea unui eveniment I/O pe un set de descriptori de fiier, a unei erori pe set sau până la expirarea unui timer.

Funciile folosesc un set de descriptori de fiier pentru a preciza fiierele/socketii pe care thread-ul curent va atepta producerea evenimentelor I/O. Tipul de date folosit pentru definirea acestui set este `fd_set`, care este, de obicei, o mască de bii.

Funciile `select` i `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 *timeou
```

Deoarece si apelul **poll** este specificat in standardul POSIX (deci portabilitate mare), dar ofera performante mai bune, **nu** vom insista asupra apelului **select**!

Avantaje:

- simplitate;
- portabilitate: funcia select este disponibilă chiar i in API-ul Win32;

Dezavantaje:

- lungimea setul de descriptori este definită cu ajutorul lui FD_SETSIZE, i implicit are valoarea 64;
- este necesar ca seturile de descriptori sa fie reconstruite la fiecare apel select;
- la apariia unui eveniment pe unul din descriptori, toi descriptorii pui în set înainte de select trebuie testai pentru a vedea pe care din ei a apărut evenimentul;
- la fiecare apel, acelai set de descriptori este transmis în kernel.

poll

Funcia `poll` consolidează argumentele funciei `select` i permite notificarea pentru o gamă mai largă de evenimente. Funcia se definete ca mai jos:

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

 int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

Timeout-ul este specificat in milisecunde. In caz de valoare negativa, semnificatia este de asteptare pentru o perioada 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 */
};
```

Functia `poll` permite astfel asteptarea evenimentelor descrise de vectorul `ufds` de dimensiune `nfds`.

În cadrul structurii `struct pollfd`, câmpul `events` este o mască de bii în care se specifică evenimentele urmărite de `poll` pentru descriptorul `fd`. `revents` este, de asemenea, o mască de bii completată de kernel cu evenimentele apărute în momentul în care apelul se întoarce (`POLLIN`, `POLLOUT`) sau valori predefinite (`POLLERR`, `POLLHUP`, `POLLNVAL`) pentru situaii speciale.

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

Un exemplu de utilizare a `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 funciei `select`;
- setul de descriptori nu trebuie reconstruit la fiecare apel `poll`;

Dezavantaje poll

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

epoll

Funciile `select` i `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 situaii, la fiecare pas, trebuie contruită lista de descriptori i apelat `poll` sau `select` care copiază tot setul în kernel. La apariia unui eveniment va fi marcat corespunzător descriptorul. Utilizatorul trebuie să parcurgă tot setul de descriptori pentru a-i 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 i tot mai puin timp făcând I/O.

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

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

Aceste interfee rezolvă problemele asociate cu `select` i `poll` i rezolvă problemele de scalabilitate.

Pentru a folosi `epoll`, trebuie inclus `sys/epoll.h`. Funciile asociate sunt `epoll_create`, `epoll_ctl` i `epoll_wait`. Interfaa `epoll` oferă funcii pentru crearea unui obiect epoll, adăugarea sau eliminarea de descriptori de fiiere/sockei la obiectul epoll i ateptarea unui eveniment pe unul dintre descriptori.

Crearea unui obiect epoll

Pentru crearea unui obiect epoll se folosete funcia `epoll_create`:

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

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

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

Adăugarea/eliminarea descriptorilor la/de la obiectul epoll

Operaiile de adăugare/eliminare de descriptori se realizează cu ajutorul funciei `epoll_ctl`:

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

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

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

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

Structura `struct epoll_event` specifică evenimentele ateptate

```
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 */
};
```

Tipuri de evenimente care pot fi urmărite sau primite pe un descriptor:

- `EPOLLIN`: descriptorul asociat are date de citit;
- `EPOLLOUT`: descriptorul asociat are loc în buffer-ul asociat pentru a scrie date;
- `EPOLLERR`: a apărut o eroare pe descriptorul asociat.

Union-ul `epoll_data` poate fi folosit pentru a asocia o cheie cu descriptorul urmărit.

Ateptarea unui eveniment I/O

Thread-ul curent ateptă producerea unui eveniment I/O la unul din descriptorii asociai obiectului epoll prin intermediul funciei `epoll_wait`:

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

Funcia `epoll_wait` este echivalentul funciilor `select` i `poll`. Este folosită pentru ateptarea unui eveniment la unul din descriptorii asociai descriptorului `epollfd`.

La revenirea apelului programatorul nu va trebui să parcurgă toi descriptorii configurai 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` (ateptare nedefinită), apelul se va întoarce imediat dacă există evenimente asociate, sau se va ploca până la apariia unui eveniment.

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

Edge-triggered sau level-triggered

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

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

Vom presupune că se citesc 50 de octei din cei 100 primii. Diferena între cele două moduri de funcionare 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 relaie cu evenimentul, iar level-triggered *prezena* 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 sockeilor în modul non-blocant.

- *Level-triggered*
  - La apariia 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 iniial pentru un socket pentru că astfel `epoll_wait` va raporta imediat că este loc de scris în buffer (iniial bufferul de scriere asociat cu socketul este gol). Acesta este o formă deghizată de busy waiting. Folosirea corectă implică scrierea normală pe socket i numai dacă la un moment dat funciile de scriere raportează că nu mai este loc de scriere in buffer (`EAGAIN`), se va activa `EPOLLOUT` pe descriptorul respectiv i salva ce mai este de scris. Când în sfârit se face loc, se va raporta EPOLLOUT i atunci se poate încerca să se scrie datele păstrate. Dacă se reuete 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 reuete scrierea integrală a datelor i scos imediat după ce acestea au fost scrise.

- *Edge-triggered*
  - La apariia unui eveniment `EPOLLIN` pe un descriptor, trebuie citit tot ce se poate citi înainte de reapelarea `poll_wait`, altfel nu va mai fi raportat `EPOLLIN` niciodată.
  - Pentru scrierea folosind edge-triggered se poate activa de la început `EPOLLOUT`. Aceasta va cauza apariia unui eveniment `EPOLLOUT` imediat dupa 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 i 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` i `poll` (server care multiplexează mai multe conexiuni pe sockei i 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 funciilor de multiplexare de mai sus (`select`, `poll`, `epoll`) este aceea că sunt limitate la descriptori de fiier. Altfel spus, se pot atepta doar evenimente asociate cu un fiier/socket: gata de citire, gata de scriere. De multe ori însă se dorete să existe un punct comun de ateptare a unui semnal, a unui semafor, a unui proces, a unei operaii de intrare/ieire, a unui timer. În Windows, acest lucru se poate realiza cu ajutorul funciei `WaitForMultipleObjects` i pe faptul că majoritatea mecanismelor din Windows sunt folosit cu ajutorul tipului de date `HANDLE`.

eventfd

Pentru a asigura în Linux posibilitate ateptării de evenimente multiple s-a definit interfaa **eventfd**. Cu ajutorul aceste interfee i combinat cu interfeele de multiplexare I/O existente, kernel-ul poate notifica o aplicaie utilizator de orice tip de eveniment.

Interfaa **eventfd** este prezentă în nucleul Linux începând cu versiunea 2.6.22. În momentul de faă (glibc 2.7), biblioteca standard C nu oferă suport pentru aceste mecanisme. Suportul va fi oferit în versiunea 2.8 a glibc. Pentru folosirea mecanismelor oferite de kernel se poate folosi interfaa `syscall`:

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

static int eventfd(unsigned int initval, int flags)
{
        return syscall(__NR_eventfd, initval, 0);
}
```

Cele trei apeluri de bază pentru extinderea funcionalităii multiplexării I/O sunt `eventfd`, `signalfd` i `timerfd_create`.

Toate cele trei apeluri întorc un descriptor de fiier pe care se vor putea primi notificări (semnale, timere etc.). Operaiile posibile pe descriptorul de fiier 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 i 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 recepionarea de semnale prin intermediul unui descriptor de fiier. Pentru a putea recepiona un semnal cu ajutorul interfeei `signalfd`, va trebui blocat în masca de semnale a procesului. La fel ca i 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)


/* on par with the current glibc wrapper */
static int signalfd(int fd, const sigset_t *mask, int flags)
{
    return syscall(__NR_signalfd, fd, mask, SIZEOF_SIGSET);
}

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

```
            printf("Read unexpected signal\n");
        }
    }
    [...]
```

Interfaa **eventfd** permite unificarea mecanismelor de notificare ale kernel-ului într-un descriptor de fiier care va fi folosit de utilizator.

În acest moment (kernel 2.6.29, glibc 2.9) a fost unificat subsistemul de operaii I/O al nucleului Linux cu interfaa **eventfd** si exista si suport in biblioteca standard C pentru acest apel (incepand cu 2.8).

ATENTIE! Versiunea glibc instalata in laborator (2.7) nu are suport pentru eventfd, fiind necesara folosirea wrapper-ului de mai sus, ce foloseste syscall.

Windows - I/O asincron (overlapped)

În Windows se folosesc funciile `ReadFile`, `WriteFile` sau `ReadFileEx`, `WriteFileEx` pentru pornirea operaiilor asincrone. Pentru notificarea terminării operaiilor în Windows se pot folosi *evenimente*, *APC*-uri, *Thread Pooling* sau *Completion Port*-uri.

Apeluri pentru transfer asincron (Overlapped I/O)

În Windows, operaiile asincrone se numesc **overlapped I/O** (operaii suprapuse). Pentru folosirea overlapped I/O în cazul fiierelor se folosete ultimul argument transmis funciei `ReadFile`, respectiv `WriteFile`, ca în exemplul de mai jos:

```
HANDLE hFile = CreateFile(...);
char buffer[BUFSIZ];
DWORD bytesRead;
OVERLAPPED ov;

/* init overlapped structure */
ov.Offset = 0;                /* read from beginning of file */

/* TODO ... ov.hEvent = ... - if using an event for notification */

if (ReadFile(
           hFile,
           buffer,
           BUFSIZ,
           &bytesRead,
           &ov) == FALSE) {
    fprintf(stderr, "ReadFile failed with %d\n", GetLastError());
    ExitProcess(-1);
}
[...]
```

Ultimul argument este un pointer la structura `OVERLAPPED` care descrie operaia asincronă de realizat. Pentru ca operaiile asincrone să poată decurge pe fiiere, fiierul trebuie deschis cu flag-ul `FILE_FLAG_OVERLAPPED`:

```
HANDLE hFile;
```

```
/* open file for reading using overlapped I/O */
hFile = CreateFile(
                files[i],
                GENERIC_READ,
                FILE_SHARE_READ,
                NULL,
                OPEN_EXISTING,
                FILE_ATTRIBUTE_NORMAL <PIPE> FILE_FLAG_OVERLAPPED,
                NULL);
```

Interogarea/ateptarea operaiilor asincrone

Pentru determinarea stării operaiei asincrone se poate folosi funcia `GetOverlappedResult`:

```
HANDLE hFile;
OVERLAPPED ov;
DWORD bytesTransferred;

/* TODO ... start overlapped I/O operation */

/* wait for completion */
GetOverlappedResult(hFile, &ov, &bytesTransferred, TRUE);
```

După cum se poate observa, ultimul argument al apelului `GetOverlappedResult` difereniază între interogarea operaiei asincrone (`FALSE`) sau ateptarea încheierii acesteia (`TRUE`).

Notificarea încheierii operaiilor asincrone

Pentru notificarea încheierii unei operaii asincrone se pot folosi:

- *evenimente* - prin completarea corespunzătoare a câmpului `hEvent` al structurii `OVERLAPPED`;
- *APC* - prin folosirea apelurilor `ReadFileEx`/`WriteFileEx` care permit specificarea unei rutine apelată la încheierea operaiei;
- *I/O completion ports* - mecanismul scalabil de ateptare a operaiilor asincrone.

Operatii asincrone pe socketi

La fel ca la fisiere, folosirea operatii asincrone (overlapped I/O) pe socketi presupune folosirea unei structuri specializate (`WSAOVERLAPPED`). Pentru transmiterea/receptia de informatii in format asincron se folosesc functiile `WSARecv`, respectiv `WSASend`.

Un exemplu de folosire a functiei `WSARecv` pentru primirea asincrona a informatiilor este prezentat in continuare. Codul este inspirat din exemplul din pagina MSDN asociata:

```
char buffer[BUFSIZ];
WSABUF DataBuf;
DWORD flags;
DWORD recvBytes;
WSAOVERLAPPED ov;
SOCKET s;
```

```
/* TODO ... connect socket */

DataBuf.buf = buffer;
DataBuf.len = BUFSIZ;
flags = 0;

/* start asynchronous I/O */
WSARecv(s, &DataBuf, 1, &recvBytes, &flags, &ov, NULL);
```

Exerciii

Prezentare

Pentru a urmări mai uor noiunile expuse la începutul laboratorului folosii această prezentare (pdf) (odp).

Exerciii de laborator

Folositi arhiva de sarcini a laboratorului.

Linux

1. **pollpipe (2p)** Creai folosind fork(2) o aplicaie de test pentru poll(2). Aplicaia folosete un server (părintele) i `CLIENT_COUNT` clieni (copiii) ce comunică prin pipe-uri anonime.
    - ♦ server-ul:
        - ◊ construiete un vector de pipe-uri (în funcia `main`);
        - ◊ creează clienii;
        - ◊ blochează în ateptarea datelor pe acestea i tipărete datele primite;
        - ◊ termină execuia după ce a primit date de la fiecare client;
    - ♦ clienii:
        - ◊ ateaptă un număr aleator de secunde (mai mic decât 10);
        - ◊ scriu în pipe-ul corespunzător un ir de `MSG_SIZE` caractere de forma `"<pid>:<caracter random>"` (`'a' + random() % 30`);
        - ◊ scrierile i citirile în pipe-uri de până la `PIPE_BUF` octei (`4096` pe Linux) sunt atomice.

          Hints:
        - ◊ Consultai seciunea poll i Pipe-uri în Linux.
2. **epollpipe (2p)** Modificai codul anterior pentru a folosi epoll.
3. **eventpipe (2p)** Modificai codul anterior pentru a adăuga funcionaliatea de deînregistrare a clienilor.
    - ♦ Se va folosi un descriptor eventfd prin intermediul căruia serverul este notificat de încheierea execuiei unui client.
    - ♦ server-ul:
        - ◊ creează eventfd-ul i îl adaugă la descriptor-ul epoll;
        - ◊ la primirea unui eveniment prin acesta, dacă primii 32 bii sunt `MAGIC_EXIT`, atunci scoate capătul pipe-ului corespunzător din `epoll` şi îl inchide;

- ◊ există 2 opiuni pentru a determina pipe-ul care trebuie scos i închis, pe baza restului de 32 de bii din event:
  - · îi folosii pentru PID-ul client-ului i suplimentar retineti in server o asociere client-pipe
  - · va folositi de mostenirea descriptorilor la fork()
- ◊ face exit dupa ce a primit MAGIC_EXIT de la toti cei CLIENT_COUNT clienti

♦ clientii:
- ◊ fac trimiterea datelor intr-o bucla cu IT_COUNT iteratii
- ◊ la fiecare iteratie, verifica daca o valoare random() este multiplu de IT_COUNT, si daca da trimite un eveniment MAGIC_EXIT si termina executia.

4. **signalpipe (2p)** Inlocuiti in codul anterior notificarea (de terminare a clientilor) bazata pe **eventfd** cu una bazata pe semnale.
   ♦ server-ul:
   - ◊ creează un descriptor via signalfd pentru 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.

Windows

1. **aio (2p)** Scrierea unor date aleatoare în fiiere folosind operaii sincrone i asincrone. Implementai funciile `do_io_sync`, respectiv `do_io_async`.
   ♦ Pentru operaiile sincrone putei folosi funcia `xwrite` definită în fiier.
   ♦ Va trebui să alocai spaiu pentru structurile `OVERLAPPED` pentru toate cele 4 fiiere.
   ♦ Pentru iniializarea structurilor `OVERLAPPED` se recomandă implementarea funciei `init_overlapped`.
   ♦ Folosii `GetOverlappedResult` pentru realizarea operaiilor asincrone pe cele 4 fiiere.
   ♦ Funciile trebuie să scrie coninutul bufferului `g_buffer` în cele 4 fiiere cu numele dat de vectorul `files`.
   ♦ Folosii macro-ul `IO_OP_TYPE` pentru a determina comportamentul programului.

Soluii

Resurse utile

- eventfd
  - ♦ Exemplu de cod integrare eventfd cu Linux AIO
- select/poll/epoll
  - ♦ Linux System Programming - Chapter 2 - File I/O (Multiplexed I/O)
  - ♦ Linux System Programming - Chapter 4 - Advanced File I/O (The Event Poll Interface)
  - ♦ Beginning Linux Programming, 4th Edition - Chapter 15: Sockets (select)
  - ♦ Un articol interesant despre epoll, la un nivel mai inalt
- Windows I/O Completion Ports
  - ♦ MSDN - I/O Completion Ports
  - ♦ Inside I/O Completion Ports
  - ♦ O introducere în Completion Ports
  - ♦ Tutorial IOCP i sockei

- General
  - libevent - bibliotecă de operaii asincrone
  - C10K - Problema celor 10.000 de clieni