Operatii IO avansate (2)

Contents

Windows - I/O Completion Ports

Mecanismul de completion ports este cel mai scalabil dintre toate cele prezentate până acum. Un server care folosete completion ports poate face faă la foarte multe (zeci de mii) conexiuni simultan, fără probleme prea mari. Celelalte metode îi ating limitările cu mult înainte.

Un completion port este un obiect în kernel cu care se asociază ali descriptori (fiiere, sockei) i prin intermediul căruia se transmit notificările de completare a unor operaii asincrone lansate anterior. Un completion port are asociat un pool de worker threads. Aceste threaduri ateaptă să primească notificări de completare a operaiilor asincrone. În momentul în care un thread primete o notificare va deveni activ i va lucra o perioada până se va întoarce din nou ateptând următoarea notificare.

Crearea unui completion port

Pentru crearea unui completion port se folosete funcia `CreateIoCompletionPort` ca în exemplul de mai jos:

```
HANDLE iocp = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, (ULONG_PTR) NULL, 0);
```

Adăugarea unui descriptor la completion port

Pentru adăugarea unui descriptor deschis cu opiunea de overlapped I/O la completion port se folosete tot funcia `CreateIoCompletionPort`. În această situaie primul argument va fi handle-ul fiierului/socketului care se dorete adăugat, iar al doilea handle-ul completion port-ului obinut la crearea acestuia:

```
HANDLE iocp;
HANDLE hFile;

/* create completion port */
iocp = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, (ULONG_PTR) NULL, 0);

/* open file for overlapped I/O */
hFile = CreateFile(..., FILE_FLAG_OVERLAPPED, ...);

/* add file handle to completion port */
CreateIoCompletionPort(hFile, iocp, (ULONG_PTR) hFile /* use handle as key */, 0);
```

După cum se observă, în cazul creării unui completion port al doilea argument este `NULL`. La adăugarea unui handle de fiier la completion port al doilea argument este handle-ul de completion port. Al treilea argument este o cheie care va fi folosită pentru identificarea handle-ului în momentul receptionării unei notificări.

Ateptarea încheierii unei operaii asincrone

Thread-urile worker sunt folosite pentru ateptarea încheierii operaiilor asincrone i prelucrările ulterioare. Thread-urile vor primi notificări de la handle-ul completion port-ului folosind funcia `GetQueuedCompletionStatus`:

```
HANDLE       iocp;
DWORD        bytes;
ULONG_PTR    key;
LPOVERLAPPED op;
[...]
/*
 * [in]  iocp  - HANDLE către un iocp
 * [out] bytes - numărul de bytes care au fost transferai în operaia care s-a încheiat
 * [out] key   - cheia care a fost asociată fiierului la CreateIoCompletionPort
 * [out] op    - structura OVERLAPPED care a fost specificată la pornirea operaiei IO
 */
GetQueuedCompletionStatus(iocp, &bytes, &key, &op, INFINITE);
```

Pe baza cheii obinute se poate determina handle-ul care a generat notificarea.

Exemplu de folosire completion ports

În exemplul de mai jos este prezentată folosirea mecanismului de completion ports în cazul operaiilor asincrone pe sockei. Exemplul este similar cu cel prezentat în seciunile dedicate funciilor de multiplexare I/O pe Linux. Există un thread worker care va atepta primirea notificărilor la completion port, iar thread-ul

principal va fi responsabil cu primirea de cereri de conexiune (apeluri `accept`).

```
HANDLE iocp;

/*** main thread ***/

SOCKET listenfd, sockfd;        /* listener socket; connection socket */

/* create I/O completion port */
iocp = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, (ULONG_PTR) NULL, 0);

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

/* TODO ... create worker thread */

while (1) {        /* server loop */
   /* TODO ... accept connections */

   /* add socket to completion port */
   CreateIoCompletionPort(sockfd, iocp, (ULONG_PTR) sockfd/* use handle as key */, 0);

   /* TODO ... start asynchronous operation */
}


/*** worker thread ***/

DWORD bytes;
ULONG_PTR key;
LPOVERLAPPED op;

while (1) {
    /* wait for notification */
    GetQueuedCompletionStatus(iocp, &bytes, &key, &op, INFINITE);

    /* TODO ... process request */
}
```

Linux - operaii asincrone

În mod clasic, operaiile de lucru cu datele aflate pe suporturi externe înseamnă utilizarea apelurilor sincrone de tipul `read`, `write` i `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 instruciuni curent până la terminarea operaiei cerute.

Pentru fire de execuie care nu au nevoie frecvent de operaii de intrare-ieire, această abordare funcionează. În schimb, pentru aplicaii specializate pe lucrul cu memoria externă, folosirea apelurilor sincrone (blocante) încetinete semnificativ execuia programului. Timpul necesar unui acces la memorie (cu atât mai mult memoria externă) depăete cu mult timpul de execuie a unei instruciuni strict aritmetice.

Linux AIO

Implementarea curentă (glibc 2.7) a operaiilor POSIX asincrone în Linux este realizată în user-space cu ajutorul thread-urilor POSIX. Totui, nucleul Linux 2.6 oferă interfaă (apeluri de sistem) pentru operaii

asincrone implementate la nivelul nucleului. Există, în acest sens, biblioteci specializate care oferă interfaa POSIX AIO peste API-ul nativ al nucleului Linux 2.6 (spre exemplu PAIOL - POSIX Asynchronous I/O for Linux).

Alternativ, se poate folos interfaa syscall care permite accesul la apelurile de sistem expuse de kernel în absena acestora din biblioteca standard C.

Apelurile de sistem care asigură interfaa cu implementarea AIO în Linux sunt prezentate, în forma de wrapper syscall mai jos:

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

long io_setup(unsigned nr_reqs, aio_context_t *ctx)
{
        return syscall(__NR_io_setup, nr_reqs, ctx);
}

long io_destroy(aio_context_t ctx)
{
        return syscall(__NR_io_destroy, ctx);
}

long io_submit(aio_context_t ctx, long n, struct iocb **paiocb)
{
        return syscall(__NR_io_submit, ctx, n, paiocb);
}

long io_cancel(aio_context_t ctx, struct iocb *aiocb, struct io_event *res)
{
        return syscall(__NR_io_cancel, ctx, aiocb, res);
}

long io_getevents(aio_context_t ctx, long min_nr, long nr,
                          struct io_event *events, struct timespec *tmo)
{
        return syscall(__NR_io_getevents, ctx, min_nr, nr, events, tmo);
}
```

Din păcate, documentaia asociată Linux AIO este destul de redusă, paginile de manual fiind principala formă de documentare. Un exemplu complet de utilizare a interfeei Linux AIO este exemplul lui Davide Libenzi.

Structuri de bază Linux AIO

Similară cu `struct aiocb` este structura `struct iocb` folosită pentru încapsularea unei operaii asincrone. Structura este definită în header-ul aio_abi.h. Pentru folosirea acesteia o aplicaia va include `linux/aio_abi.h`. Un exemplu de iniializare a acestei structuri este:

```
#include <linux/aio_abi.h>

        /* ... */

        struct iocb *iocb;

        memset(iocb, 0, sizeof(*iocb));
        iocb->aio_fildes = fd;
        iocb->aio_lio_opcode = IOCB_CMD_PWRITE;
```

```
        iocb->aio_reqprio = 0;
        iocb->aio_buf = (u_int64_t) buf;
        iocb->aio_nbytes = nbytes;
        iocb->aio_offset = offset;
#ifdef USE_EVENTFD
        iocb->aio_flags = IOCB_FLAG_RESFD;
        iocb->aio_resfd = efd;
#else
        iocb->aio_flags = 0;
#endif
```

Comenzi posibile sunt: `IOCB_CMD_PWRITE`, `IOCB_CMD_PREAD`, `IOCB_CMD_PREADV`, `IOCB_CMD_PWRITEV` i altele. Comenzile care se încheie cu `V` folosesc vectored I/O.

Context AIO

Orice operaie sau set de operaii Linux AIO sunt identificate printr-o valoare de tipul aio_context_t ce reprezintă un context de operaii asincrone.

Iniializarea, respectiv distrugerea contextului se realizează cu ajutorul funciilor io_setup i io_destroy:

```
#include <linux/aio_abi.h>

aio_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 i anulează toate operaiile I/O asincrone necompletate */
if (io_destroy(ctx) < 0) {
    /* handle error */
}
```

Operaii AIO

Pentru realizarea unei operaii asincrone se folosete funcia io_submit. Această funcia declanează pornirea operaiilor asincrone definite în vectorul de pointeri de structuri `struct iocb` primit ca argument. Din punct de vedere al funcionalităii, funcia este similară lio_listio. Nu există apeluri similare `aio_read` sau `aio_write`, tipul operaiei fiind descris de câmpul `aio_lio_opcode` al structurii `struct iocb`. Fiind un apel asincron, la fel ca funciile `aio_read`, `aio_write` i `lio_listio`, io_submit nu blochează procesul curent.

```
#include <linux/aio_abi.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 */
```

```
aio_context_t ctx = 0;

/* init context, iocb */

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

/*
 * pune în coada contextului 'ctx' NUM_AIO_OPS operaii asincrone descrise de pointerii din 'piocb
 * Nu se ateaptă terminarea operaiilor.
 */
if (io_submit(ctx, NUM_AIO_OPS, piocb) < 0) {
    /* handle error */
}

/* procesează date în paralel cu execuia asincronă a operaiilor I/O */
```

Pentru ateptarea încheierii unei operaii AIO i obinerea de informaii despre rezultatul acesteia se folosete funcia io_getevents. Funcia folosete structura struct io_event pentru a obine informaii despre încheierea unei operaii asincrone. Un exemplu de utilizare este:

```
#include <linux/aio_abi.h>
#define       NUM_AIO_OPS  10
aio_context_t ctx = 0;
struct        io_event events[NUM_AIO_OPS];     /* aio result array */
/* ... */

/*
 * ateaptă ca exact NUM_AIO_OPS operaii asincrone să se încheie
 * min_nr - numărul minim de operaii asincrone care trebuie să se încheie pentru ca funcia să se
 * max_nr - numărul maxim de operaii asincrone care pot fi întoarse de io_getevents.
 */
if (io_getevents(ctx, NUM_AIO_OPS /* min_nr */, NUM_AIO_OPS  /* max_nr */, events, NULL /* no tim
    /* handle error */
}
```

Integrarea Linux AIO cu eventfd

Este utilă folosirea apelurilor de multiplexare I/O (`select`, `poll`, `epoll`) i pentru ateptarea încheierii operaiilor asincrone. Pentru aceasta, interfaa AIO a Linux 2.6 permite integrarea API-ului de operaii asincrone cu mecanismul `eventfd`.

Pentru aceasta se configurează flag-ul `IOCB_FLAG_RESFD` iar câmpul `resfd` al structurii `struct iocb` va conine un descriptor `eventfd` ce va fi notificat în momentul încheierii operaiei asincrone. Apelul `io_getevents` este în continuare util pentru a obine informaii despre încheierea operaiilor. Mecanismul `eventfd` oferă doar mecanismul de ateptare a acestora.

```
#include <linux/aio_abi.h>
int efd;

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

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

```
/* ... */
/* use eventfd */
iocb->aio_flags = IOCB_FLAG_RESFD;
iocb->aio_resfd = 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 operaii I/O încheiate. Această valoare va fi, de obicei, folosită ca al doilea i al treilea argument al io_getevents.

Folosind integrarea operaiilor asincrone cu `eventfd` i mecanismele de multiplexare I/O (`select`, `poll`, `epoll`) se poate atepta unificat încheierea unei operaii asincrone sau sosirea de date pe sockei. (**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 fiier, din care cel puin unul este pipe. Avantajul este ca nu se folosete un buffer (byte array) în userspace. Pentru o scurtă introducere putei citi descrierea de apelului splice pe Wikipedia, iar pentru o mai bună aprofundare a avantajului oferit de apel, citii descrierea de pe LKML.

```
#define _GNU_SOURCE // trebuie definit pentru că splice este o extensie nespecificată de standard
#include <fcntl.h>
long splice(int fd_in, loff_t *off_in, int fd_out, loff_t *off_out, size_t len, unsigned int flag
```

- 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 i pentru `fd_out` i `off_out`, la scriere
- parametrul `len` specifica numărul maxim de octei transferai
- masca de bii `flags` poate specifică o operaie non-blocantă sau hint-uri pentru nucleu. Citii pagina de manual a funciei pentru detalii.

Exemplu:

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

// ... deschideri fiiere, creare pipe
```

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

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

Windows - TransmitFile

Apelul TransmitFile este folosit pentru a eficientiza transmiterea de fiiere în reea. Transmit File folosete cache-ul sistemului de operare. Este o operaie zero-copy: nu necesită alocarea de buffere în user-space i diminuează numărul de apeluri de sistem.

Pentru a transmite un fiier, acesta trebuie deschis folosind flag-ul `FILE_FLAG_OVERLAPPED`. Apelul TransmitFile primete ca argument socket-ul pe care se realizează comunicaia i handle-ul fiierului. Un exemplu de utilizare a apelului este prezentat mai jos:

```
BOOL       result;
SOCKET     hSocket;
HANDLE     hFile;
OVERLAPPED ov;
/* ... */
result = TransmitFile(hSocket,         /* destination socket handle */
                      hFile,           /* source file handle */
                      0,               /* nr bytes to write. 0 == send entire file */
                      0,               /* block size. 0 == default block size */
                      &ov,             /* overlapped I/O structure */
                      NULL, 0);
if (result == FALSE) {
    /* handle error */
}
```

O funcie similară este funcia TransmitPackets care transmite date stocate în memorie pe un socket folosind cache-ul intern al sistemului de operare. Datele sunt reprezentate de o structură TRANSMIT_PACKETS_ELEMENT.

POSIX - sendfile

Operaia similară în POSIX este asigurată de funcia sendfile.

```
#include <sys/sendfile.h>
ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);

ssize_t bytesSent = sendfile(out_fd, /* destination socket file descriptor */
                             in_fd,  /* source file descriptor */
                             &off,   /* offset în fi. sursă de unde se trasmit date. */
                             count); /* nr bytes to send */
```

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 ieire sau citirea de date de la un flux de ieire în mai

multe buffere. Bufferele sunt precizate ca un vector de buffere, de unde i denumirea de vectored I/O.

Apelurile din clasa vectored I/O sunt utile în momentul în care datele sunt disparate/dezasamblate în memorie i se dorete "concatenarea" acestora într-un singur flux de scriere sau "desfacerea" acestora dintr-un flux de citire. Un exemplu îl reprezintă pachetele de reea în care headerele, datele i trailerele se găsesc, de obicei, în locaii 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 operaie. 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 (fiiere) ReadFileScatter, WriteFileGather.
- Windows (sockei) WSARecv,WSASend.

readv/writev

Funciile readv i writev sunt folosite în sistemele Unix ca operaii de tipul `vectored I/O`. Structura de bază folosită de aceste funcii 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 recepionarea/transmiterea unui număr de buffere reprezentate de structura `struct iovec`. Funciile întorc numărul total de octei citii sau scrii.

writev scrie datele către care trimit elementele din iov în fiier, 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 */
}
```

Exerciii

Exerciii de laborator

- Folosii arhiva de sarcini a laboratorului.
- Putei folosi fiierele tags din directoarele `lin/`, respectiv `win/` pentru o parcurgere rapidă a surselor folosind vim (fiierul `tags`) sau Emacs (fiierul `TAGS`).

Linux

1. (**3 puncte**) Asynchronous I/O (KAIO)
   - ♦ Parcurgei fiierul `kaio.c`.
   - ♦ Completai funciile zonele lipsă pentru a programa scrierea a 4 fiiere cu numele date de variabila `files`.
   - ♦ Folosii API-ul KAIO (`io_setup`, `io_destroy`, `io_submit`, `io_getevents`).
   - ♦ Folosii _doar_ `io_getevents` pentru ateptarea încheierii operaiilor asincrone.
     Hints:
     - ◊ Parcurgei seciunea Linux AIO.
     - ◊ Consultai exemplul lui Davide Libenzi.
   - ♦ Compilai i rulai programul. Va trebui să avei 4 fiiere de dimensiune 8192 octei create în `/tmp`.
2. (**1 punct**) Asynchronous I/O (KAIO) + eventfd
   - ♦ Copiai fiierul `kaio.c` de la exerciiul anterior.
   - ♦ Folosii `eventfd` pentru ateptarea operaiilor asincrone.
     - ◊ Completai funcia `wait_aio`.
     - ◊ Folosii flag-ul `IOCB_FLAG_RESFD` i completai corespunzător câmpul `aio_resfd` al structurii `struct iocb`.
       Hints:
     - ◊ Parcurgei seciunea Linux AIO.
     - ◊ Consultai exemplul lui Davide Libenzi.
   - ♦ Compilai i rulai programul. Va trebui să avei 4 fiiere de dimensiune 8192 octei create în `/tmp`.
3. (**1.5 puncte**) Vectored I/O
   - ♦ Parcurgei fiierele `../sock_util/sock_util.h`, `../sock_util/sock_util.c`, `vectored_client.c` i `vectored_server.c`.
   - ♦ Completai funciile `send_buffers`, respectiv `receive_buffers` pentru a realiza o comunicaie client-server folosind vectored I/O (operaiile readv i writev).
     - ◊ Va trebuie să transmitei/recepionai bufferele `header`, `data`, `trailer`.
   - ♦ Pentru testare, pornii serverul pe o consolă i clientul în altă consolă.
   - ♦ Mesajul transmis de client va trebui să fie identic cu cel primit de server.
     Hints:
     - ◊ Parcurgei seciunea Vectored I/O.

Windows

1. (**1 punct**) I/O completion ports
   - Anlizai coninutul fiierului `include/iocp.h`.
   - Intrai în directorul `iocp/`.
   - Analizai coninutul fiierului `iocp.c`.
   - Completai cele 4 funcii definite în `iocp.c`.
   - Nu compilai fiierul. Acesta va fi compilat la exerciiul următor.

     (**Hints**)

     ◊ Parcurgei seciunea Windows - I/O Completion Ports.
2. (**3 puncte**) Operaii I/O asincrone cu I/O completion ports
   - Intrai în directorul `aio_cp/`.
   - Analizai coninutul fiierului `aio.c`.
   - Scopul exerciiului este folosirea I/O completion ports pentru ateptarea încheierii operaiilor I/O asincrone (overlapped I/O).
   - Implementai funciile `init_io_async`, `do_io_async` i `wait_io_async`.

     Hints:

     ◊ Pentru iniializarea structurilor `OVERLAPPED` se recomandă implementarea funciei `init_overlapped`.

     ◊ Urmării indicaiile din exemplele de cod.

     ◊ Parcurgei seciunea Windows - I/O Completion Ports

     ◊ Parcurgei seciunea Windows - I/O asincron (overlapped) a laboratorului trecut.
   - Compilai i rulai programul.
3. (**1.5 puncte**) Zero-copy (`TransmitFile`)
   - Intrai în directorul `transmit/`.
   - Parcurgei fiierele `sock_util/sock_util.h`, `sock_util/sock_util.c`, `server.c` i `transmit_client.c`.
   - Completai funciile care lipsesc.

     ◊ Clientul face transmitere folosind `TransmitFile`.

     ◊ Serverul recepionează informaiile folosind `recv` i `WriteFile`.

     Hints:

     ◊ Parcurgei seciunea Windows - TransmitFile.

     ◊ Nu este nevoie să folosii câmpul `LPOVERLAPPED` al funciei `TransmitFile` (putei folosi `NULL`).

Extra

1. (**Windows**) Operaii asincrone pe sockei i I/O completion ports.
   - Intrai în directorul `sock_cp/`.
   - Parcurgei fiierele `sock_util/sock_util.h`, `sock_util/sock_util.c`, `client.c`.
   - Analizai coninutul fiierului `iocp_server.c`.
   - Completai fiierul `iocp_server.c`.
   - Scopul exerciiului este crearea unui server care să multiplexeze mai multe conexiuni de la clieni folosind operaii asincrone pe socketi i I/O completion ports. Serverul _doar_ va citi informaii de la clieni pe care le va afia la ieirea standard.
   - Serverul dispune de două thread-uri: un thread acceptă conexiuni pe care le adaugă la I/O completion port i iniiază operaie asincronă de citire; un thread worker ateaptă notificarea de la I/O completion port, o tratează i apoi iniiază o nouă operaie asincronă.

Hints:

- ◊ Folosii structura `struct overlappedContext` pentru a reine informaiile despre un apel asincron.
- ◊ Se recomandă folosirea structurii pe post de cheie pentru a putea demultiplexa notificarea sosită la I/O completion port.

- ♦ Urmării indicaiile din comentariile din cod.
- ♦ Folosii fiierul `Makefile` pentru a obine executabilele `client.exe` i `iocp_server.exe`.
- ♦ Pentru testare pornii serverul i mai multe instane de client.

2. (**Linux**) Funcii de multiplexare I/O
   - ♦ Intrai în directorul `include/`.
     - ◊ Analizai coninutul fiierelor `mpx_types.h`, respectiv `mpx_funs.h`.

       Hints:

       · Funciile descrise în `mpx_funs.h` se doresc a fi generice. Uneori implementarea va fi ineficientă.

       · Separaia `mpx_funs.h` de `mpx_types.h` este puin forată, dar menine o anumită simplitate.
   - ♦ Intrai în directorul `mpx/`.
   - ♦ Analizai coninutul fiierelor `mpx_select.c`, `mpx_poll.c` respectiv `mpx_epoll.c`.
   - ♦ Implementai funciile definite în `mpx_select.c`.

     Hint:

     - ◊ Urmării indicaiile din comentariile asociate funciilor.
   - ♦ Implementai funciile definite în `mpx_poll.c`.

     Hints:

     - ◊ Se presupune că se ateaptă doar notificări pentru citire (`POLLIN`) sau scriere (`POLLOUT`).
     - ◊ Urmării indicaiile din comentariile asociate funciilor.
   - ♦ Implementai funciile definite în `mpx_epoll.c`.

     Hints:

     - ◊ Interfaa oferită peste `epoll` presupune ateptarea unui singur tip de eveniment pe un descriptor (`EPOLLIN` sau `EPOLLOUT`). Funcia `epoll_del_` va elimina complet descriptorul fiierului.
     - ◊ Urmării indicaiile din comentariile asociate funciilor.
3. (**Linux**) Multiplexare I/O peste socketi
   - ♦ Intrai în directorul `sock_mpx/`.
   - ♦ Parcurgei fiierele `../sock_util/sock_util.h`, `../sock_util/sock_util.c`, `client.c`.
   - ♦ Analizai coninutul fiierului `mpx_server.c`.
   - ♦ Completai fiierul `mpx_server.c`.
   - ♦ Scopul exerciiului este crearea unui server care să multiplexeze mai multe conexiuni de la clieni. Serverul _doar_ va citi informaii de la clieni pe care le va afia la ieirea standard.

     Hints:

     - ◊ Serverul va multiplexa socketul listener i socketii de conectare. Dacă socketul listener este gata de citire atunci se acceptă o nouă conexiune; dacă un socket de conectare e gata de citire se citete informaia de la acesta.
     - ◊ Urmării indicaiile din comentariile din cod.
   - ♦ Folosii fiierul `Makefile` pentru a obine executabilele `client` i `mpx_server`.
   - ♦ Pentru testare pornii serverul i mai multe instane de client.

Soluii

- Soluii exerciii laborator 11

Resurse utile

- Linux AIO
  - linux-aio mailing list
  - Biblioteca PAIOL - POSIX Asynchronouos I/O for Linux
  - libaio - O biblioteca (veche) pentru operaii AIO cu suport în nucleul Linux
- 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 Overlapped I/O
  - Syncrhonization and Overlapped Input and Output
  - Synchronous and Asynchronous I/O
  - Programming with Asynchronous Sockets
  - Asynchronous Input/Output and Completion Ports - Windows System Programming, 3rd Edition - Chapter 14
- 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
  - Asynchronous I/O - Wikipedia
  - libevent - bibliotecă de operaii asincrone
  - C10K - Problema celor 10.000 de clieni