Лекція 6
Мультиплексування введення-виведення

Ідея мультиплексування введення-виведення

Якщо потоку треба виконувати введення-виведення для кількох дескрипторів файлів і якщо введення-виведення можуть бути заблоковані на невизначений час через нестачу ресурсу, тоді потік буде ефективно (реально) працювати з кількома дескрипторами файлів почергово. Такий сценарій не завжди прийнятний, оскільки час блокування в системних викликах введення-виведення не визначений.

Мультиплексування введення-виведення – це пасивне очікування можливості виконати введення-виведення без блокування для кількох дескрипторів файлів та виконання введення-виведення без блокування для тих дескрипторів файлів, для яких опитування показало можливість це виконати. Зазвичай ці дії виконуються в одному потоці програми в циклі. Мультиплексування введення-виведення дає змогу одному потоку виконувати введення-виведення для кількох дескрипторів файлів одночасно. Зрозуміло, що один потік не може одночасно виконувати введення-виведення для кількох дескрипторів файлів фізично, він буде працювати з ними почергово, але мультиплексування введення-виведення дає змогу потоку взагалі не блокуватися в системних

викликах введення-виведення через настачу ресурсу. Потік точно знає для якого дескриптора файлу можливо застосувати системний виклик введення-виведення, не бути заблокованим у ньому та отримати результат.

Альтернативний підхід одночасного введення-виведення для кількох дескрипторів файлів в одному потоці полягає в постійних викликах системних викликів введення-виведення без блокування почергово в циклі. Такий підхід призводить до необхідності постійного переходу в режим ядра під час викликів системних викликів, які можуть нічого не виконати через нестачу ресурсів. Такі спроби виконати введення-виведення ефективно (реально) є активним очікуванням можливості виконати введення-виведення для кількох дескрипторів файлів.

Мультиплексування введення-виведення відрізняється від асинхронних дій. Потік, який викликав системний виклик для опитування можливості виконати введення-виведення без блокування для кількох дескрипторів файлів, може бути заблокований на невизначений або на максимальний вказаний тайм-аут (ядро виконає добровільне перемикання контексту). Потік, який ініціював кілька асинхронних дій, не блокується та продовжує своє виконання.

Мультиплексування введення-виведення може бути застосовано для дескрипторів файлів, які асоційовані з різними об’єктами ядра, наприклад, з неіменованими та іменованими каналами, сокетами, але практично нема сенсу використовувати мультиплексування введення-виведення для звичайних файлів, оскільки можливість виконати введення-виведення без блокування для звичайного файлу завжди успішна.

Є різні API для мультиплексування введення-виведення. Ці API можна поділити на стандартний API, який визначений в POSIX, та альтернативні API, які реалізовані в конкретних ОС. Переносна програма має послуговуватися API, який визначений у POSIX. Оптимізована переносна програма визначає для якої ОС відбувається її компіляція та використовує більш оптимальний API, який реалізований у цій ОС, або використовує API, який визначений у POSIX.

POSIX визначає три функції `pselect()`, `select()` та `poll()` для мультиплексування введення-виведення. Функція `pselect()` може працювати з маскою сигналів. Оскільки сигнали не є темою цих лекцій, ця функція не пояснюється. Семантика використання цих системних викликів не передбачає створення сталого контексту, який визначає для яких об’єктів ядра перевірятимуться

стани. Цей контекст створюється ядром кожний раз під час виклику цих системних викликів та звільняється після завершення їхнього виконання. Створення цього контексту потребує часу, який пропорційний кількості дескрипторів файлів, вказаних в аргументах. Для однократного виклику цих системних викликів або для мультиплексування введення-виведення для невеликої кількості дескрипторів файлів це не є проблемою (цей контекст однаково треба створювати), але зазвичай ці системні виклики викликаються в циклі й часто з тою самою або з дещо зміненою множиною дескрипторів файлів, тому створення такого самого або дещо зміненого контексту для кожного виклику є неефективним рішенням.

Деякі ОС надають альтернативні API для мультиплексування введення-виведення і можливо для рішення інших супутніх задач. Семантики цих API передбачають створення сталого контексту в ядрі, який визначає для яких об’єктів ядра перевірятимуться стани. Відповідно ці API дають змогу реалізовувати ефективніші рішення мультиплексування введення-виведення в програмах користувача, ніж API, який визначений у POSIX.

Повільні системні виклики без блокування

Блокування виконання системного виклику відбувається через нестачу деякого ресурсу для продовження його виконання або через семантику системного виклику (також через нестачу ресурсу, який безпосередньо пов’язаний з об’єктом ядра, для якого викликається системний виклик). Таке блокування є нормальним явищем і може відбуватися під час виконання швидкого та повільного системного виклику. Але блокування виконання повільного системного виклику можливо на невизначений час (саме тому він називається *повільним*). Усі причини неможливості продовжити виконання повільного системного виклику через настачу запитаного ресурсу пов’язані із зовнішніми чинниками для ядра. Заблокований повільний системний виклик через нестачу запитуваного ресурсу можна перервати сигналом.

Виконання швидкого системного виклику може бути заблоковано на невизначений час через нестачу запитаного ресурсу, але це блокування не може бути перервано сигналом. Наприклад, системні виклики `read*()` та `write*()` застосовані для звичайного файлу є швидкими, навіть якщо цей файл належить мережевій ФС (дані мережевої ФС відправляються іншим сервером,

тому цей сервер є зовнішнім чинником для ядра, який визначає коли буде наявним запитаний ресурс).

Системний виклик, який застосовується до об’єкта деякого механізму синхронізації, також може бути заблокований на невизначений час через неможливість заволодіти цим об’єктом, але виконання такого заблокованого системного виклику неможливо перервати сигналом, оскільки він вважається швидким. Тобто заволодіти об’єктом деякого механізму синхронізації можна завжди.

Також виконання процесу користувача може бути заблоковано на невизначений час через нестачу деякого ресурсу для вирішення виключною ситуації, яка виникла під час його виконання. Наприклад, вирішення сторінкового промаху, який потребує введення-виведення з мережевою ФС. Таке блокування відбувається неявно для процесу користувача, тому його також неможливо перервати сигналом. Те, що це блокування відбувається неявно, не значить, що воно відбулося не через виконання системного виклику. Наприклад, сторінковий промах міг відбутися під час виконання системного виклику `fstat()`, в якому залучено буфер пам’яті, який належить не відображеній

віртуальній сторінці простору користувача, вміст якої належить файлу мережевої ФС.

Прапорець статусу `O_NONBLOCK` є ознакою об’єкта ядра, який вказує ніколи не блокуватися під час виконання деяких повільних системних викликів, застосованих до цього об’єкту ядра.

Прапорці статусу можна встановити командою `F_SETFL` системним викликом `fcntl()`, тип третього аргументу повинен бути `int`. Прапорці статусу можна отримати командою `F_GETFL` системним викликом `fcntl()`. Прапорці статусу застосовуються до об’єкта ядра, який асоційований з вказаним дескриптором файлу.

Зазвичай використовують функції, які дають змогу додавати та забирати потрібні прапорці статусу з поточної множини прапорців статусу об’єкта ядра.

Приклад реалізації цих функцій на C.

```
#include <fcntl.h>

void
```

```
fd_add_status(int fd, int flags)
{
    int flags_curr;

    flags_curr = fcntl(fd, F_GETFL);
    if (flags_curr < 0)
        exit_err("fd_add_status(): fcntl()");
    if (fcntl(fd, F_SETFL, flags_curr | flags) < 0)
        exit_err("fd_add_status(): fcntl()");
}

void
fd_rem_status(int fd, int flags)
{
    int flags_curr;

    flags_curr = fcntl(fd, F_GETFL);
    if (flags_curr < 0)
        exit_err("fd_rem_status(): fcntl()");
    if (fcntl(fd, F_SETFL, flags_curr & ~flags) < 0)
        exit_err("fd_rem_status(): fcntl()");
}

void
fd_set_nonblock(int fd)
{
    fd_add_status(fd, O_NONBLOCK);
}
```

```
void
fd_clr_nonblock(int fd)
{
    fd_rem_status(fd, O_NONBLOCK);
}
```

Стандартний API

Системний виклик `select()` повертає інформацію про можливість виконати введення-виведення без блокування для об’єктів ядра, які асоційовані з дескрипторами файлів, вказаних в множинах дескрипторів файлів.

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

int select(int nfds, fd_set *restrict readfds, fd_set *restrict writefds,
        fd_set *restrict exceptfds, struct timeval *restrict timeout);
void FD_CLR(int fd, fd_set *fds);
int FD_ISSET(int fd, fd_set *fds);
void FD_SET(int fd, fd_set *fds);
void FD_ZERO(fd_set *fds);
```

Об’єкт типу `fd_set` визначає множину номерів дескрипторів файлів. В об’єкті цього типу можна запам’ятати номери дескрипторів файлів у діапазоні [`0`, `FD_SETSIZE`) (слово «`SIZE`» у назві макросу `FD_SETSIZE` не позначає максимальну кількість номерів дескрипторів файлів). Тип `fd_set` – це структура, тому вміст об’єкта цього типу можна присвоювати, але не можна порівнювати (поля структури не визначені в POSIX, у структурі можуть бути padding bytes, а в

полях структури можуть бути padding bits). Зазвичай ця структура містить масив об’єктів цілочисельного типу, який використовується як бітова карта.

Працювати із вмістом об’єкта типу `fd_set` треба за допомогою `FD_*()`, які є функціями та/або макросами. `FD_CLR()` забирає номер дескриптора файлу `fd` з об’єкту, на який вказує аргумент `fds` (якщо `fd` не був встановлений в об’єкті, тоді це ні на що не впливає). `FD_ISSET()` матиме ненульове значення, якщо номер дескриптора файлу `fd` встановлено в об’єкті, на який вказує аргумент `fds`. `FD_SET()` встановлює номер дескриптора файлу `fd` в об’єкті, на який вказує аргумент `fds` (якщо `fd` вже встановлений в об’єкті, тоді це ні на що не впливає). `FD_ZERO()` ініціалізує об’єкт, на який вказує аргумент `fds`, у цьому об’єкті не буде встановлено жодного номера дескриптора файлу. Якщо `fd` від’ємний, дорівнює або більший ніж `FD_SETSIZE`, не є номером дескриптора файлу або якщо аргументи є виразами з побічними діями, тоді поведінка `FD_*()` не визначена.

Кожний з аргументів `readfds`, `writefds` та `exceptfds` може бути null покажчиком, у цьому випадку відповідний аргумент ігнорується. Ці аргументи вказують на об’єкти, в яких вказані множини номерів дескрипторів файлів, для яких треба перевірити можливість виконувати введення та виведення без блокування, та

наявність виключних ситуацій відповідно. Будуть перевірятися перші `nfds` номерів дескрипторів у цих об’єктах (аргумент `nfds` використовується для оптимізації, щоб ядро можливо не зчитувало непотрібні дані та не перевіряло невстановлені програмою номери дескрипторів файлів у цих об’єктах).

Після успішного виконання `select()`, об’єкти, на які вказують аргументи `readfds`, `writefds` та `exceptfds` будуть модифіковані, вміст цих об’єктів позначатиме номери дескрипторів файлів, які були раніше вказані в запиті і для яких можливо виконання введення та виведення без блокування, та наявність виключних ситуацій відповідно.

Об’єкт, на який вказує аргумент `timeout`, визначає тайм-аут, тобто максимальний час виконання системного виклику. Якщо аргумент `timeout` null покажчик, тоді максимальний час виконання системного виклику необмежений. Нульові значення в об’єкті, на який вказує аргумент `timeout`, призводять до негайного завершення виконання системного виклику незалежно від результату його виконання (системні виклики мають ефект polling’а). Інші значення в об’єкті, на який вказує аргумент `timeout`, визначають тайм-аут. Максимальне значення тайм-ауту має бути щонайменше 31 день. Якщо вказується значення більше,

ніж максимальне значення в реалізації, тоді використовується максимальне значення тайм-ауту реалізації. Також ядро може округляти вказаний тайм-аут до більшого значення (ядро має дискретизацію часу). Системний виклик `select()` може змінювати вміст об’єкта, на який вказує аргумент `timeout`, але як цей вміст змінюється не визначено.

У `struct timeval` є принаймні такі поля: `time_t tv_sec` (секунди), `suseconds_t tv_usec` (знаковий цілочисельний тип достатній для збереження значень у діапазоні [`-1`, `1000000`], мікросекунди).

Системний виклик `select()` після успішного виконання повертає кількість встановлених номерів дескрипторів файлів в об’єктах, на які вказують аргументи `readfds`, `writefds` та `exceptfds`. У разі помилки виконання системного виклику `select()`, об’єкти, на які вказують аргументи `readfds`, `writefds` та `exceptfds`, не змінюються.

Приклад

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/select.h>
```

```
#include <sys/socket.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>

int
main(void)
{
    const time_t timeout_sec = 15;
    struct sockaddr_in srv_sin4 = { 0 };
    in_port_t srv_port = htons(1234);
    struct timeval tv;
    fd_set rset;
    char buf[10];
    ssize_t nread;
    int listenfd, sockfd, nready, maxfd;

    listenfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (listenfd < 0)
        exit_err("socket()");

    srv_sin4.sin_family = AF_INET;
    srv_sin4.sin_port = srv_port;
    srv_sin4.sin_addr.s_addr = INADDR_ANY;
    if (bind(listenfd, (struct sockaddr *)&srv_sin4, sizeof(srv_sin4)) < 0)
        exit_err("bind()");
```

```
if (listen(listenfd, 10) < 0)
    exit_err("listen()");

sockfd = accept(listenfd, NULL, NULL);
if (sockfd < 0)
    exit_err("accept()");
if (sockfd >= FD_SETSIZE)
    exit_errx("sockfd >= %ju", (uintmax_t)FD_SETSIZE);

for (;;) {
    FD_ZERO(&rset);
    FD_SET(sockfd, &rset);
    maxfd = sockfd;
    tv.tv_sec = timeout_sec;
    tv.tv_usec = 0;
    nready = select(maxfd + 1, &rset, NULL, NULL, &tv);
    if (nready < 0)
        exit_err("select()");
    if (nready == 0) {
        printf("No data during %ju seconds\n", (uintmax_t)timeout_sec);
        continue;
    }
    if (FD_ISSET(sockfd, &rset)) {
        nread = read(sockfd, buf, sizeof(buf));
        if (nread < 0)
            exit_err("read()");
        if (nread == 0) {
            printf("End of connection\n");
```

```
                break;
            }
            printf("Received %zd bytes\n", nread);
        }
    }

    if (close(sockfd) < 0 || close(listenfd) < 0)
        exit_err("close()");
}
```

Якщо необхідно мультиплексування введення-виведення за кількома дескрипторами файлів, яке виконується в циклі, тоді є сенс мати окремі об’єкти типу `fd_set`, вміст яких копіюється в інші об’єкти типу `fd_set` безпосередньо перед їхнім використанням у системному виклику `select()`, це дасть змогу не витрачати час на ініціалізацію множини дескрипторів файлів перед перевіркою можливості виконати введення-виведення для них. Ці окремі об’єкти типу `fd_set` також можна модифікувати (додавати або забирати дескриптори файлів) за потреби.

Приклад можливої реалізації таких дій на C++.

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

```
class FdSet {
private:
    fd_set fdset;
    int nfds;
public:
    FdSet()
    {
        FD_ZERO(&this->fdset);
        this->nfds = -1;
    }
    void set(int fd)
    {
        FD_SET(fd, &this->fdset);
        if (this->nfds < fd)
            this->nfds = fd;
    }
    void clr(int fd)
    {
        FD_CLR(fd, &this->fdset);
        if (this->nfds == fd) {
            while (--fd >= 0)
                if (FD_ISSET(fd, &this->fdset))
                    break;
            this->nfds = fd;
        }
    }
    bool isset(int fd) const
    {
```

```
        return FD_ISSET(fd, &this->fdset);
    }
    int get_nfds() const
    {
        return this->nfds;
    }
    fd_set* fdset_ptr()
    {
        return &this->fdset;
    }
};
```

У цій реалізації, якщо в множині дескрипторів файлів у полі `fdset` нема жодного встановленого дескриптора файлу, тоді значення поля `nfds` буде `-1`. Це значить, що для цього значення треба додати `1` перед використанням у системному виклику `select()`, що відповідає семантиці цього системного виклику.

Системний виклик `poll()` повертає інформацію про можливість виконати введення-виведення без блокування для об’єктів ядра, з якими асоційовані номери дескрипторів файлі, вказані в масиві.

```
#include <poll.h>

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

У масиві, на який вказує аргумент `fds`, вказуються дескриптори файлів та дії, які будуть перевірятися. Кількість елементів у масиві вказується в аргументі `nfds` (беззнаковий цілочисельний тип). У `struct pollfd` є принаймні такі поля: `int fd` (номер дескриптора файлу, для якого треба виконати перевірку), `short events` (прапорці, які визначають що треба перевірити), `short revents` (прапорці, які позначають результат).

Якщо значення поля `fd` від'ємне, тоді значення поля `events` ігнорується, а значення поля `revents` буде встановлено в нуль після виконання системного виклику `poll()`. Встановлення значення поля `fd` у від'ємне значення дає змогу ігнорувати елемент масиву `fds`.

У полі `events` можна вказати та в полі `revents` можна перевірити наявність встановлених таких прапорців: `POLLIN` визначає можливість читання даних, відмінних від високопріоритетних (цей прапорець еквівалентний використанню `POLLRDNORM|POLLRDBAND`); `POLLRDNORM` визначає можливість читання нормальних (непріоритетних) даних; `POLLRDBAND` визначає можливість читання пріоритетних даних; `POLLPRI` визначає можливість читання високопріоритетних даних; `POLLOUT`

або еквівалентний POLLWRNORM визначає можливість запису нормальних даних; POLLWRBAND визначає можливість запису пріоритетних даних. Семантика нормальних, пріоритетних та високопріоритетних даних залежить від об'єкта ядра, який асоційований з дескриптором файлу. Пріоритети мають повідомлення у функціональності STREAMS, яку підтримують не всі ОС. У полі revents ще можна перевірити наявність встановлених таких прапорців: POLLERR визначає наявність помилки; POLLHUP позначає від'єднання пристрою; POLLNVAL визначає некоректний номер дескриптора файлу.

Деякі реалізації системного виклику poll() не відповідають POSIX. Відмінність полягає у встановлених прапорцях у полі revents у разі помилки. Наприклад, замість прапорця POLLERR може бути встановлений прапорець POLLNVAL. Тут нема сенсу виписувати комбінації прапорців, які можуть бути встановлені в різних реалізаціях або в різних версіях реалізацій. Переносні програми, які використовують системний виклик poll(), мають сприймати будь-яку комбінацію прапорців відмінних від POLLIN або POLLOUT, встановлених у полі revents, як помилку.

Значення timeout визначає тайм-аут, тобто максимальний час виконання системного виклику. Якщо значення timeout дорівнює -1, тоді максимальний час

виконання системного виклику необмежений. Якщо значення `timeout` дорівнює нулю, тоді це призводить до негайного завершення виконання системного виклику незалежно від результату виконання (системний виклик має ефект polling'а). Інше значення `timeout` визначає тайм-аут у мілісекундах. Також ядро може округляти вказаний тайм-аут до більшого значення (ядро має дискретизацію часу). Максимальне допустиме значення для тайм-ауту не визначено.

Після успішного виконання системний виклик `poll()` повертає кількість елементів масиву `fds`, які мають ненульове значення поля `revents`. Системний виклик `poll()` не змінює значення полів `fd` та `events` у структурах у масиві `fds`.

Приклад

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <poll.h>
#include <stddef.h>
#include <stdio.h>
#include <unistd.h>

int
```

```
main(void)
{
    const int timeout_msec = 15 * 1000;
    struct sockaddr_in srv_sin4 = { 0 };
    in_port_t srv_port = htons(1234);
    struct pollfd fds[1];
    char buf[10];
    ssize_t nread;
    int listenfd, sockfd, nready;

    listenfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (listenfd < 0)
        exit_err("socket()");

    srv_sin4.sin_family = AF_INET;
    srv_sin4.sin_port = srv_port;
    srv_sin4.sin_addr.s_addr = INADDR_ANY;
    if (bind(listenfd, (struct sockaddr *)&srv_sin4, sizeof(srv_sin4)) < 0)
        exit_err("bind()");

    if (listen(listenfd, 10) < 0)
        exit_err("listen()");

    sockfd = accept(listenfd, NULL, NULL);
    if (sockfd < 0)
        exit_err("accept()");

    fds[0].fd = sockfd;
```

```
    fds[0].events = POLLIN;

    for (;;) {
        nready = poll(fds, 1, timeout_msec);
        if (nready < 0)
            exit_err("poll()");
        if (nready == 0) {
            printf("No data during %d seconds\n", timeout_msec);
            continue;
        }
        if (fds[0].revents & POLLIN) {
            nread = read(fds[0].fd, buf, sizeof(buf));
            if (nread < 0)
                exit_err("read()");
            if (nread == 0) {
                printf("End of connection\n");
                break;
            }
            printf("Received %zd bytes\n", nread);
        }
        if (fds[0].revents & (POLLNVAL|POLLHUP|POLLERR))
            exit_errx("poll(): got (POLLNVAL|POLLHUP|POLLERR)");
    }

    if (close(sockfd) < 0 || close(listenfd) < 0)
        exit_err("close()");
}
```

Системні виклики `select()` та `poll()` є повільними системними викликами і їхнє виконання може бути перервано сигналом, тобто вони можуть завершуватися з помилкою `EINTR`.

Альтернативні API

Деякі системи реалізують альтернативні API для мультиплексування введення-виведення. Семантика функцій цих API передбачає створення сталого контексту в ядрі, який визначає для яких об’єктів ядра перевірятиметься стан. Цей контекст потім можна використовувати багаторазово для перевірки станів об’єктів ядра та змінювати його за потреби. Використання системних викликів `select()` та `poll()` потребує створення контексту в ядрі, який визначає для яких об’єктів ядра перевірятимуться стани, під час кожного їхнього виклику. Якщо програмі треба перевіряти стани великої кількості об’єктів ядра, тоді створення контексту в ядрі під час кожного виклику системного виклику для мультиплексування введення-виведення є неефективним рішенням.

Далі надаються ідеї використання API `epoll` в Linux та API `kqueue` в macOS. Дистрибутиви Linux та macOS надають докладну документацію про ці API. Оскільки ці API не визначені в POSIX, вони можуть змінюватися.

Системний виклик `epoll_create()` створює об’єкт ядра `epoll`.

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

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

Сучасні версій Linux ігнорують значення аргументу `size`, але його значення має бути додатне. У разі успішного виконання цей системний виклик повертає номер дескриптора файлу, який асоційований з об’єктом ядра `epoll`.

Системний виклик `epoll_ctl()` модифікує налаштування в об’єкті ядра `epoll`.

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

int epoll_ctl(int epfd, int act, int fd, struct epoll_event *event);
```

Аргумент `epfd` – це номер дескриптора файлу, який відповідає об’єкту ядра `epoll`. Аргумент `act` визначає дію, яку потрібно виконати: `EPOLL_CTL_ADD` (додати налаштування), `EPOLL_CTL_MOD` (змінити раніше додані налаштування), `EPOLL_CTL_DEL` (забрати раніше додані налаштування). Аргумент `fd` – це номер дескриптора файлу, який відповідає об’єкту ядра, для якого вказуються налаштування. Аргумент `event` вказує на об’єкт із налаштуваннями (для дії `EPOLL_CTL_DEL` цей аргумент ігнорується). Якщо всі дескриптори файлів, які асоційовані з об’єктом ядра, для якого були додані налаштування, будуть закриті, тоді відповідні

раніше додані налаштування для цього об’єкта ядра забираються з об’єкта ядра `epoll`.

У `struct epoll_event` є такі поля: `uint32_t events` (біти, які кодують можливості виконати дії та помилки), `epoll_data_t data` (дані, визначені програмою). Програма може вказати будь-які дані в полі `data`, аби в подальшому була можливість визначати для якого об’єкта ядра зі списку опитуваних повернена інформація. Тип `epoll_data_t` – це `union`, який складається з таких полів: `void *ptr`, `int fd`, `uint32_t u32`, `uint64_t u64`. Наприклад, у полі `fd` можна вказувати номер дескриптора файлу. У полі `events` можуть бути вказані такі біти (наводяться тільки кілька з можливих): `EPOLLIN` (те саме, що `POLLIN` у `poll()`), `EPOLLOUT` (те саме, що `POLLOUT` у `poll()`), `EPOLLERR` (те саме, що `POLLERR` у `poll()`), `EPOLLHUP` (те саме, що `POLLHUP` у `poll()`). Об’єкти типу `struct epoll_event` використовуються для вказування налаштувань та для отримання результатів.

Системний виклик `epoll_wait()` перевіряє можливість виконати дії, опитування яких раніше були зареєстровані в об’єкті ядра `epoll`.

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

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

Аргумент `epfd` – це номер дескриптора файлу, який відповідає об’єкту ядра `epoll`. Аргумент `timeout` має такий самий сенс, як у `poll()`. У разі успішного виконання, інформація про можливість виконати дії та помилки повертається в масиві, на який вказує аргумент `events`, кількість заповнених елементів масиву повертається як результат, який буде не більше значення `maxevents`.

Приклад

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/epoll.h>
#include <stddef.h>
#include <stdio.h>
#include <unistd.h>

int
main(void)
{
    const int timeout_msec = 15 * 1000;
    struct sockaddr_in srv_sin4 = { 0 };
```

```
in_port_t srv_port = htons(1234);
struct epoll_event event, events[1];
char buf[10];
ssize_t nread;
int epfd, listenfd, sockfd, nready;

epfd = epoll_create(1);
if (epfd < 0)
    exit_err("epoll_create()");

listenfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (listenfd < 0)
    exit_err("socket()");

srv_sin4.sin_family = AF_INET;
srv_sin4.sin_port = srv_port;
srv_sin4.sin_addr.s_addr = INADDR_ANY;
if (bind(listenfd, (struct sockaddr *)&srv_sin4, sizeof(srv_sin4)) < 0)
    exit_err("bind()");

if (listen(listenfd, 10) < 0)
    exit_err("listen()");

sockfd = accept(listenfd, NULL, NULL);
if (sockfd < 0)
    exit_err("accept()");

event.events = EPOLLIN;
```

```
event.data.fd = sockfd;
if (epoll_ctl(epfd, EPOLL_CTL_ADD, sockfd, &event) < 0)
    exit_err("epoll_ctl()");

for (;;) {
    nready = epoll_wait(epfd, events, 1, timeout_msec);
    if (nready < 0)
        exit_err("epoll_wait()");
    if (nready == 0) {
        printf("No data during %d seconds\n", timeout_msec);
        continue;
    }
    if (events[0].events & EPOLLIN) {
        nread = read(events[0].data.fd, buf, sizeof(buf));
        if (nread < 0)
            exit_err("read()");
        if (nread == 0) {
            printf("End of connection\n");
            break;
        }
        printf("Received %zd bytes\n", nread);
    } else if (events[0].events & EPOLLHUP) {
        exit_errx("epoll_wait(): got error in events");
    } else {
        exit_errx("epoll_wait(): unexpected flag in events");
    }
}
```

```
    if (close(sockfd) < 0 || close(listenfd) < 0 || close(epfd) < 0)
        exit_err("close()");
}
```

Системний виклик kqueue() створює об’єкт ядра kqueue.

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

int kqueue(void);
```

У разі успішного виконання цей системний виклик повертає номер дескриптора файлу, який асоційований з об’єктом ядра kqueue.

Системний виклик kevent() модифікує налаштування та перевіряє можливість виконати дії, опитування яких раніше були зареєстровані в об’єкті ядра kqueue.

```
#include <sys/types.h>
#include <sys/event.h>
#include <sys/time.h>

int kevent(int kqfd, const struct kevent *changelist, int nchanges,
        struct kevent *eventlist, int nevents, const struct timespec *timeout);
EV_SET(&kevent, ident, fileter, flags, fflags, data, udata);
```

Аргумент `kqfd` – це номер дескриптора файлу, який відповідає об’єкту ядра `kqueue`. Масив з `nchanges` елементів, на який вказує аргумент `changelist`, визначає зміни налаштувань. Аргумент `timeout` має такий самий сенс, як у `select()`, тільки інший тип. У разі успішного виконання, інформація про можливість виконати дії та помилки повертається в масиві, на який вказує аргумент `eventlist`, кількість заповнених елементів повертається як результат, який буде не більше значення `nevents`. Кожний з аргументів `changelist` та `eventlist` може бути null покажчиком, у цьому випадку відповідний аргумент ігнорується.

У `struct kevent` є такі поля: `uintptr_t ident` (ідентифікатор, для якого застосовується фільтр), `int16_t filter` (фільтр, який треба застосувати для ідентифікатора), `uint16_t flags` (прапорці), `uint32_t fflags` (прапорці, специфічні для конкретного фільтру), `intptr_t data` (дані, специфічні для конкретного фільтру), `void *udata` (довільні користувальницькі дані). Для зручності ініціалізації налаштувань в об’єкті типу `struct kevent` є макрос `EV_SET()`.

Значення поля `filter` можуть бути такі (наводяться тільки кілька з можливих): `EVFILT_READ` (те саме, що `POLLIN` у `poll()`), `EVFILT_WRITE` (те саме, що `POLLOUT` у `poll()`). Для фільтрів `EVFILT_READ` та `EVFILT_WRITE` у полі `ident` вказується номер дескриптора

файлу. Якщо всі дескриптори файлів, з якими асоційовані об’єкти ядра, для яких були додані налаштування, будуть закриті, тоді відповідні раніше додані налаштування для цього об’єкта ядра забираються з об’єкта ядра kqueue.

Значення прапорців у полі flags можуть бути такі (наводяться тільки кілька можливих): EV_ADD (додати або змінити налаштування), EV_ENABLE (увімкнути раніше вимкнуті налаштування), EV_DISABLE (вимкнути налаштування), EV_DELETE (забрати раніше додані налаштування), EV_ERROR (позначає помилку під час додавання або зміни налаштувань).

Приклад

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/event.h>
#include <sys/time.h>
#include <fcntl.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
```

```
int
main(void)
{
    const time_t timeout_sec = 15;
    struct timespec ts;
    struct sockaddr_in srv_sin4 = { 0 };
    struct kevent events[1];
    in_port_t srv_port = htons(1234);
    char buf[10];
    ssize_t nread;
    int listenfd, sockfd, kqfd, nready;

    kqfd = kqueue();
    if (kqfd < 0)
        exit_err("kqueue()");

    listenfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (listenfd < 0)
        exit_err("socket()");

    srv_sin4.sin_family = AF_INET;
    srv_sin4.sin_port = srv_port;
    srv_sin4.sin_addr.s_addr = INADDR_ANY;
    if (bind(listenfd, (struct sockaddr *)&srv_sin4, sizeof(srv_sin4)) < 0)
        exit_err("bind()");

    if (listen(listenfd, 10) < 0)
```

```
    exit_err("listen()");

sockfd = accept(listenfd, NULL, NULL);
if (sockfd < 0)
    exit_err("accept()");

EV_SET(&events[0], sockfd, EVFILT_READ, EV_ADD, 0, 0, NULL);
if (kevent(kqfd, events, 1, NULL, 0, NULL) < 0)
    exit_err("kevent()");

ts.tv_sec = timeout_sec;
ts.tv_nsec = 0;
for (;;) {
    nready = kevent(kqfd, NULL, 0, events, 1, &ts);
    if (nready < 0)
        exit_err("kevent()");
    if (nready == 0) {
        printf("No data during %ju seconds\n", (uintmax_t)timeout_sec);
        continue;
    }
    if (events[0].filter == EVFILT_READ) {
        nread = read((int)events[0].ident, buf, sizeof(buf));
        if (nread < 0)
            exit_err("read()");
        if (nread == 0) {
            printf("End of connection\n");
            break;
        }
```

```
            printf("Received %zd bytes\n", nread);
        }
    }
    if (close(listenfd) < 0 || close(sockfd) < 0 || close(kqfd) < 0)
        exit_err("close()");
}
```

TCP-сокет та мультиплексування введення-виведення

Якщо системний виклик `connect()` застосовується для TCP-сокета, для якого встановлено прапорець статусу `O_NONBLOCK`, тоді, якщо з'єднання не може бути встановлено відразу, цей системний виклик завершується з помилкою `EINPROGRESS`. Перевірити завершення створення або не створення TCP-з'єднання можна функцією мультиплексування введення-виведення для цього TCP-сокета. Треба перевірити чи можна читати з та чи можна записувати в цей TCP-сокет. Помилку встановлення TCP-з'єднання можна отримати за допомогою значення опції сокета `SO_ERROR`.

Якщо системний виклик `accept()` застосовується для TCP-сокета, для якого встановлено прапорець статусу `O_NONBLOCK`, тоді, якщо нема готового TCP-з'єднання, тоді цей системний виклик завершується з помилкою `EAGAIN` або `EWOULDBLOCK`.

Якщо системний виклик `read()` або відповідний застосовується для TCP-сокета, для якого становлено прапорець статусу `O_NONBLOCK`, тоді, якщо буфер отриманих

даних цього сокета порожній, тоді цей системний виклик завершується з помилкою EAGAIN або EWOULDBLOCK.

Якщо системний виклик write() або відповідний застосовується для TCP-сокета, для якого встановлено прапорець статусу O_NONBLOCK, тоді, якщо буфер даних на відправлення цього сокета заповнений, тоді цей системний виклик завершується з помилкою EAGAIN або EWOULDBLOCK, інакше цей системний виклик записує стільки байт, скільки можна скопіювати в буфер даних на відправлення.

Перевірити наявність встановленого TCP-з'єднання можна функцією мультиплексування введення-виведення для TCP-сокета який приймає нові з'єднання. Треба перевірити можливість чи можна читати з цього TCP-сокета. Оскільки між перевіркою можливості читання та викликом системного виклику accept() є часове вікно, клієнт може розірвати вже встановлене TCP-з'єднання. Семантика системного виклику accept() для цього сценарію залежить від реалізації, тому правильно буде встановити прапорець статусу O_NONBLOCK для цього TCP-сокета, щоб виконання accept() не було заблоковано.

TCP-сокет, з яким асоційоване TCP-з’єднання, готовий для читання, якщо в буфері отриманих даних цього сокета є принаймні кількість отриманих байт, яке визначається значенням опції `SO_RCVLOWAT`.

TCP-сокет, з яким асоційоване TCP-з’єднання, готовий для запису, якщо в буфері даних на відправлення цього сокета є принаймні вільне місце розміром, яке визначається значенням опції `SO_SNDLOWAT`.

Сервер з мультиплексуванням введення-виведення

Програма на C++ приймає TCP з'єднання, клієнт відправляє 4 байтове значення, це кількість байт, потім відправляє значення вказаної кількості байтів. Сервер отримує дані та виводить їх.

```
#include <cassert>
#include <cerrno>
#include <cstdint>
#include <cstring>
#include <iostream>
#include <string>
#include <vector>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>

static std::string
sin4_repr(const struct sockaddr_in& sin4)
{
    std::string repr;
    char addr_repr[INET_ADDRSTRLEN];
    char port_repr[6];
```

```
    if (getnameinfo((struct sockaddr*)&sin4, sizeof(sin4),
        addr_repr, sizeof(addr_repr), port_repr, sizeof(port_repr),
        NI_NUMERICHOST|NI_NUMERICSERV) != 0
    ) {
        addr_repr[0] = '?', addr_repr[1] = '\0';
        port_repr[0] = '?', port_repr[1] = '\0';
    }
    try {
        repr.reserve(sizeof(addr_repr) + sizeof(port_repr));
        repr.append(addr_repr);
        repr.append(":");
        repr.append(port_repr);
    } catch (...) {
        repr.clear();
    }
    return repr;
}

std::ostream&
operator<<(std::ostream& os, const struct sockaddr_in& sin4)
{
    return os << sin4_repr(sin4);
}

static void
socket_close(int sockfd)
{
    if (close(sockfd) < 0)
```

```
        exit_err("close()");
}

static const std::size_t buffer_capacity = 10;
static const std::size_t netio_read_chunk = 2;

class NetIo {
    char buf[buffer_capacity];
    std::size_t capacity{buffer_capacity};
    std::size_t size{0};
    std::size_t offset;
    int sockfd;

public:
    void init(std::size_t size)
    {
        assert(size <= this->capacity);
        assert(this->size == 0);
        assert(size > 0);
        this->size = size;
        this->offset = 0;
    }

    void reset()
    {
        assert(this->size > 0);
        this->size = 0;
    }
```

```
bool inited() const
{
    return this->size != 0;
}

bool completed() const
{
    assert(this->offset <= this->size);
    return this->offset == this->size;
}

ssize_t read_data()
{
    assert(this->offset < this->size);
    size_t size;
    ssize_t nread;
    std::cout << "NetIo::read_data(): sockfd " << this->sockfd << '\n';
    std::cout << "    to read " << (this->size - this->offset) <<
        " of " << this->size << " bytes\n";
    size = this->size - this->offset;
    if (size > netio_read_chunk)
        size = netio_read_chunk;
    nread = read(sockfd, this->buf + this->offset, size);
    if (nread < 0)
        return -1;
    std::cout << "    read " << nread << " bytes\n";
    this->offset += nread;
    return nread;
```

```
    }

    void set_sockfd(int sockfd)
    {
        this->sockfd = sockfd;
    }

    std::size_t get_size() const
    {
        return this->size;
    }

    std::size_t get_capacity() const
    {
        return this->capacity;
    }

    char* buf_ptr()
    {
        return this->buf;
    }
};

enum class FdStatus {
    Read,
    Write,
};
```

```
enum class ClientStatus {
    Header,
    Data,
    Close,
};

static void
print_bytes(const char* buf, std::size_t size)
{
    for (std::size_t i = 0; i < size; ++i)
        std::cout << std::hex << static_cast<unsigned int>(buf[i]) << ' ';
}

class Client {
private:
    std::string descr;
    ClientStatus status{ClientStatus::Header};
    FdStatus fd_status{FdStatus::Read};
    NetIo netio;
    std::size_t data_size;
    std::size_t data_offset;
    int sockfd;

public:
    Client(int sockfd, struct sockaddr_in& sin4) : sockfd(sockfd)
    {
        try {
            auto repr = sin4_repr(sin4);
```

```
        this->descr.reserve(repr.size() + 7);
        this->descr.append("client ");
        this->descr.append(repr);
    } catch (...) {
        this->descr.clear();
    }
    this->netio.set_sockfd(sockfd);
}

void read_data()
{
    std::size_t size;
    ssize_t nread;
    switch (this->status) {
    case ClientStatus::Header:
        std::cout << this << ": receive header\n";
        if (!this->netio.inited())
            this->netio.init(sizeof(std::uint32_t));
        nread = this->netio.read_data();
        if (nread <= 0)
            break;
        if (this->netio.completed()) {
            std::uint32_t data_size;
            std::memcpy(&data_size, this->netio.buf_ptr(),
                sizeof(data_size));
            data_size = ntohl(data_size);
            std::cout << this << ": data size " << data_size << '\n';
            if (data_size == 0) {
```

```
            std::cout << this << ": wants to send zero bytes\n";
            this->status = ClientStatus::Close;
        } else {
            this->data_size = data_size;
            this->data_offset = 0;
            this->status = ClientStatus::Data;
            this->netio.reset();
        }
    }
    break;
case ClientStatus::Data:
    std::cout << this << ": receive data\n";
    if (!this->netio.inited()) {
        size = this->data_size - this->data_offset;
        if (size > this->netio.get_capacity())
            size = this->netio.get_capacity();
        this->netio.init(size);
    }
    nread = this->netio.read_data();
    if (nread <= 0)
        break;
    this->data_offset += (std::size_t)nread;
    if (this->netio.completed()) {
        std::cout << this << ": received bytes\n    ";
        print_bytes(this->netio.buf_ptr(), this->netio.get_size());
        std::cout << '\n';
        if (this->data_offset == this->data_size) {
            std::cout << this << ": data has been received completely\n";
```

```
                this->status = ClientStatus::Close;
            } else {
                this->netio.reset();
            }
        }
        break;
    default:
        exit_errx("Client::read_data(): wrong client status");
    }
    if (nread < 0) {
        warn_err(this, ": read failed");
        this->status = ClientStatus::Close;
    } else if (nread == 0) {
        warn_errx(this, ": TCP connection is half-closed");
        this->status = ClientStatus::Close;
    }
}

FdStatus get_fd_status() const
{
    return this->fd_status;
}

ClientStatus get_status() const
{
    return this->status;
}
```

```
    int get_sockfd() const
    {
        return this->sockfd;
    }

    friend std::ostream& operator<<(std::ostream& os, const Client* client)
    {
        return os << client->descr;
    }
};

using ClientVector = std::vector<Client*>;

class Server {
private:
    ClientVector client_read_vec;
    ClientVector client_write_vec;
    FdSet readfds;
    FdSet writefds;
    std::size_t client_num;
    std::size_t client_conn = 0;
    int listenfd;

    void client_pop(ClientVector& vec, std::size_t i)
    {
        auto last_i = vec.size() - 1;
        if (i != last_i)
            vec[i] = vec[last_i];
```

```
        vec.resize(last_i);
        if (this->client_conn-- == this->client_num)
            this->readfds.set(this->listenfd);
    }

    void client_free(ClientVector& vec, std::size_t i)
    {
        auto client = vec[i];
        this->client_pop(vec, i);
        socket_close(client->get_sockfd());
        delete client;
    }

public:
    Server(int listenfd, std::size_t client_num)
    {
        try {
            client_read_vec.reserve(client_num);
            client_write_vec.reserve(client_num);
        } catch (...) {
            exit_err("std::vector::reserve()");
        }
        this->readfds.set(listenfd);
        this->listenfd = listenfd;
        this->client_num = client_num;
    }

    const FdSet& get_readfds() const
```

```
{
    return this->readfds;
}

Client* get_client_read(std::size_t i)
{
    assert(i <= this->client_read_vec.size());
    return i < this->client_read_vec.size() ?
        this->client_read_vec[i] : nullptr;
}

void client_new(int sockfd, struct sockaddr_in& sin4)
{
    try {
        auto client = new Client(sockfd, sin4);
        std::cout << client << ": connected\n";
        this->client_read_vec.push_back(client);
        this->readfds.set(sockfd);
        if (++this->client_conn == client_num)
            this->readfds.clr(this->listenfd);
    } catch (...) {
        warn_err("new");
        socket_close(sockfd);
    }
}

bool client_read(std::size_t i)
{
```

```
        auto client = this->client_read_vec[i];
        client->read_data();
        if (client->get_status() == ClientStatus::Close) {
            std::cout << client << ": close connection\n";
            this->readfds.clr(client->get_sockfd());
            this->client_free(this->client_read_vec, i);
            return false;
        }
        return true;
    }
};

int
main()
{
    struct sockaddr_in srv_sin4{};
    struct sockaddr_in cln_sin4;
    FdSet readfds;
    const std::size_t client_num = 10;
    in_addr_t srv_addr = INADDR_ANY;
    in_port_t srv_port = htons(1234);
    int backlog = 10;
    socklen_t addrlen;
    int listenfd, nfds, nready, sockfd;

    sigpipe_ignore();

    srv_sin4.sin_family = AF_INET;
```

```
srv_sin4.sin_port = srv_port;
srv_sin4.sin_addr.s_addr = srv_addr;
listenfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (listenfd < 0)
    exit_err("socket()");
if (listenfd >= FD_SETSIZE)
    exit_errx("listenfd >= ", FD_SETSIZE);
if (bind(listenfd, (struct sockaddr *)&srv_sin4, sizeof(srv_sin4)) < 0)
    exit_err("bind()");
if (listen(listenfd, backlog) < 0)
    exit_err("listen()");
fd_set_nonblock(listenfd);

Server server(listenfd, client_num);

std::cout << "Accept on " << srv_sin4 << '\n';
std::cout << "Multiplexing input/output\n";
for (;;) {
    readfds = server.get_readfds();
    nfds = readfds.get_nfds();
    nready = select(nfds + 1, readfds.fdset_ptr(), NULL, NULL, NULL);
    if (nready < 0)
        exit_err("select()");
    if (readfds.isset(listenfd)) {
        addrlen = sizeof(cln_sin4);
        sockfd = accept(listenfd, (struct sockaddr *)&cln_sin4, &addrlen);
        if (sockfd < 0) {
            if (system_error())
```

```
            exit_err("accept()");
        if (errno != EWOULDBLOCK && errno != EAGAIN)
            warn_err("accept()");
    } else if (sockfd >= FD_SETSIZE) {
        warn_errx("sockfd >= ", FD_SETSIZE);
        socket_close(sockfd);
    } else {
        fd_set_nonblock(sockfd);
        server.client_new(sockfd, cln_sin4);
    }
    if (--nready == 0)
        continue;
}
for (std::size_t i = 0;;) {
    auto client = server.get_client_read(i);
    if (client == nullptr)
        break;
    if (readfds.isset(client->get_sockfd())) {
        std::cout << client << ": read possible\n";
        if (server.client_read(i))
            ++i;
        if (--nready == 0)
            break;
    } else {
        ++i;
    }
}
if (nready == 0)
```

```
            continue;
        }
    }
```