Лекція 8
Асинхронне введення-виведення

Ідея асинхронних дій

Виконання будь-яких дій на комп’ютері потребує часу. Програма користувача може запобігти генерації сторінкових промахів для частини свого адресного простору, може впливати на планування виконання своїх потоків, може використовувати мультиплексування введення-виведення, може надавати ядру інформацію щодо сценаріїв доступу до свого адресного простору та сценаріїв доступу до вмісту файлів, може створювати додаткові потоки користувача, може враховувати властивості піраміди пам’яті. У такі способи програма користувача може впливати на час свого виконання. Програма користувача в загальному випадку не може впливати на час виконання системних викликів (за винятком залученого простору користувача в системних викликах). Системні виклики виконуються ядром, швидкість їхнього виконання залежить від навантаження на систему та від зовнішніх чинників для ядра.

Асинхронна дія – це дія, яку один об’єкт планування запитує виконати інший об’єкт планування та не чекає на завершення її виконання, потім можна отримати інформацію про результат її виконання. Асинхронні дії теоретично можуть бути будь-які, але практично реалізація підтримування виконання

асинхронних дій визначає, що можна робити в асинхронних діях. Дія є асинхронною для об’єкта планування, який запитав її виконання, але є синхронною для об’єкта планування, який її виконує. Підтримування виконання асинхронних дій може бути реалізовано в ядрі і в програмі користувача. Ці реалізації не є тотожними та не є взаємовиключними.

Ця дисципліна присвячена системному програмуванню на рівні програм користувача, тому об’єкт планування, який запитує виконати асинхронну дію, – це потік користувача (потік, який був створений програмою користувача). Об’єкт планування, який виконує запитану асинхронну дію, може бути процесом ядра або потоком користувача (залежить від реалізації підтримування виконання асинхронних дій).

Тип асинхронних дій, виконання яких процес користувача може запитати в ядра, наперед визначений. Зазвичай це дії, які мають відношення до введення-виведення. Введення-виведення в сучасних ядрах має асинхронний KPI, і ядра можуть надавати відповідний API для асинхронного введення-виведення програмам користувача. Виконання інших типів дій асинхронно в режимі ядра за запитом програми користувача не має сенсу, оскільки в режимі ядра

виконується тільки те, що не може бути виконано в режимі користувача (взагалі в режимі ядра можна виконувати що завгодно, але зазвичай ядро не виконує те, що можна виконати в режимі користувача).

Виконання асинхронних введень-виведень, запитаних процесами користувача, в ядрі реалізовано в кількох спеціально призначених для цього процесах ядра (процес ядра є об’єктом планування в ядрі). Створювати новий процес ядра для виконання кожного асинхронного введення-виведення, запитаного процесом користувача, не має сенсу, оскільки система однаково не може виконати більше певних дій, ніж деяку максимальну кількість. Тобто або буде черга процесів ядра, які виконують по одному асинхронному введенню-виведенню, або буде черга запитів асинхронного введення-виведення в кожного процесу ядра. Начебто нема різниці, однаково є черга, але для створення та завершення процесу ядра витрачається час, також витрачається час на планування виконання цих об’єктів планування.

Програма користувача може реалізувати підтримку яких завгодно асинхронних дії, тобто програма користувача може підтримувати асинхронне виконання довільних своїх функцій. Виконання асинхронних дій у програмі користувача

реалізовано в спеціально призначених для цього потоках користувача (потік користувача є об’єктом планування в програмі користувача). Чи буде програма користувача створювати новий потік користувача для виконання асинхронних дії або використовувати спеціально призначені потоки користувача для цього залежить від типу дії, оскільки тип асинхронної дії в програмі користувача не обмежується тільки введенням-виведенням.

Програма користувача має перевіряти результати виконання асинхронних дій. Якщо програма користувача не перевіряє результати виконання запитаних асинхронних дій, тоді це так само неправильно, як не перевіряти результат виконання будь-яких інших дій у програмі. Зазвичай програма користувача має обмеження на максимальну кількість «одночасно» запитаних асинхронних дій і в циклі запитує виконання нової асинхронної дії, якщо була завершена асинхронна дія, запитана в якомусь попередньому запиті (черговість виконання асинхронних дій зазвичай не визначена). Якщо програма користувача запитує забагато асинхронних дій для системи в деякий проміжок часу, тоді вони однаково не будуть виконуватися одночасно фізично через обмежені ресурси

комп’ютера. Також реалізація підтримування виконання асинхронних дій може мати власні обмеження на кількість «одночасно» виконуваних асинхронних дій.

POSIX визначає запити тільки для асинхронних дій, які мають відношення до введення-виведення. Відповідні функції POSIX позначаються, як *асинхронне введення-виведення* (*asynchronous input/output*, AIO). Насправді майже будь-яке введення-виведення є певною мірою асинхронним. Ядро підтримує буферний кеш для файлів, буфери для неіменованих та іменованих каналів, буфери для сокетів і т. ін. Якщо є можливість зберегти дані, які відправляються або отримуються, у буфері, тоді ядро збереже ці дані в буфері, так виконуючи введення-виведення асинхронно. Відмінність запитів AIO в тому, що потік ніколи не блокується в запиті AIO та не чекає на завершення виконання запитаного в ньому введення-виведення.

Стандартний API

Запит AIO у функціях асинхронного введення-виведення POSIX визначається значенням покажчика на об’єкт типу `struct aiocb`, який визначений в `<aio.h>`. Цей об’єкт називається *блоком управління* (*control block*). Якщо значення покажчика на блок управління стає недійсним або значення покажчиків у його полях стають недійсними до завершення виконання дії, ініційованої в запиті AIO, тоді поведінка не визначена. Одночасне використання одного й того блоку управління в кількох функціях асинхронного введення-виведення POSIX призводить до невизначених результатів, але якщо був запитаний результат виконання завершеної дії, ініційованої певним запитом AIO, тоді цей блок управління можна використовувати для нового запиту AIO. Оскільки запит AIO визначається значенням покажчика на блок управління, програма може ініціювати однакові дії в запитах AIO одночасно використовуючи різні блоки управління (якщо в цьому є потрібність).

У `struct aiocb` є принаймні такі поля: `int aio_fildes` (номер дескриптора файлу), `off_t aio_offset` (зміщення в об’єкті, з яким асоційований вказаний номер дескриптора файлу), `volatile void *aio_buf` (покажчик на буфер), `size_t aio_nbytes`

(кількість байтів у запиті), `int aio_reqprio` (значення зменшення пріоритету запиту), `struct sigevent aio_sigevent` (спосіб сповіщення про завершення виконання запиту), `int aio_lio_opcode` (дія запиту). Кожна функція асинхронного введення-виведення POSIX використовує значення тільки деяких полів блоку управління, значення інших полів блоку управління ігноруються.

У `struct sigevent` є принаймні такі поля: `int sigev_notify` (тип сповіщення), `int sigev_signo` (номер сигналу), `union sigval sigev_value` (супутнє значення для сигналу), `void (*sigev_notify_function)(union sigval)` (покажчик на функцію-сповіщення зі значенням `sigev_value` в аргументі), `pthread_attr_t *sigev_notify_attributes` (покажчик на об’єкт з атрибутами функції-сповіщення). Тип сповіщення `SIGEV_NONE` визначає відсутність асинхронного сповіщення про завершення дії, ініційованої запитом AIO. Тип сповіщення `SIGEV_SIGNAL` визначає надсилання сигналу із супутнім значенням. Тип сповіщення `SIGEV_THREAD` визначає виклик функції-сповіщення з вказаними атрибутами.

Для того, щоб відправлений сигнал `sigev_signo` був доданий до черги сигналів, треба вказувати прапорець `SA_SIGINFO` для його обробника. Насправді прапорець `SA_SIGINFO` для обробника сигналу є сенс вказувати, оскільки це є єдиний спосіб

отримати надіслане супутнє значення `sigev_value` в обробнику сигналу. Це значення буде доступно в об’єкті типу `siginfo_t`, покажчик на який передається в аргументі обробника сигналу, у значенні поля `si_value`, а поле `si_code` буде мати значення `SI_ASYNCIO`. POSIX нічого не визначає щодо черги сигналів для сигналів, які будуть відправлятися як сповіщення про завершення дій, ініційованих запитами AIO.

Функція-сповіщення `sigev_notify_function` викликається в такому середовищі, якби це була початкова функція нового потоку процесу з вказаними атрибутами. Якщо значення поля `sigev_notify_attributes` null покажчик, тоді поведінка така, як би цей новий потік створювався з атрибутом `PTHREAD_CREATE_DETACHED`. Реалізація може не створювати новий потік для виклику функції-сповіщення, може бути пул потоків для викликів функцій-сповіщень. Оскільки невідомо скільки буде виконуватися функція-сповіщення потрібен саме пул потоків та за необхідності створення нового потоку. Вказування атрибуту для функції-сповіщення, що можна очікувати завершення виконання її потоку, призводить до невизначеної поведінки, тому пул потоків для викликів функцій-сповіщень є коректним рішенням.

Якщо функції асинхронного введення-виведення POSIX реалізовано в режимі користувача, тоді в програмі користувача нема можливості визначити чи буде достатньо ресурсів для додавання відправленого сигналу до черги сигналів або виклику функції-сповіщення в майбутньому. Тому дія, ініційована в запиті AIO, може бути успішно завершена, а відправлення сигналу не вдалось (системний виклик `sigqueue()` завершився помилкою через нестачу ресурсу) або не вдалось викликати функцію-сповіщення (виклик функції `pthread_create()` завершився помилкою через нестачу ресурсу або не вдалось отримати пам’ять в алокатора пам’яті). Так само дія, ініційована в запиті AIO, може бути завершена з помилкою, але відправлення сигналу не вдалось або не вдалось викликати функцію-сповіщення. Відповідно результат виконання дії, ініційованої в запиті AIO, може бути позначений помилкою, яка має відношення до помилки, які виникла під час сповіщення про завершення його виконання. POSIX не визначає яку саме помилку треба вказувати для завершеного дії, ініційованої запитом AIO, у цьому випадку. Теоретично таке саме може бути якщо функції асинхронного введення-виведення POSIX реалізовано в ядрі, якщо ядро не резервує потрібні ресурси для майбутнього додавання сигналу в

чергу сигналів або виклику функції-сповіщення, але це більше помилка реалізації, ніж особливість реалізації.

Функції `aio_read()` та `aio_write()` ініціюють асинхронне читання та запису даних, визначених вказаним блоком управління, відповідно.

```
#include <aio.h>

int aio_read(struct aiocb *aiocb);
int aio_write(struct aiocb *aiocb);
```

В об’єкті, на який вказує аргумент `aiocb`, використовуються значення всіх полів, значення поля `aio_lio_opcode` ігноруються. Асинхронна дія буде виконуватися для об’єкта ядра, з яким асоційований дескриптор файлу з номером `aio_fildes`, покажчик на буфер визначається в полі `aio_buf`, кількість байтів визначається в полі `aio_nbytes`. Якщо функція `aio_write()` застосовується до звичайного файлу, для якого не було вказано прапорець `O_APPEND`, тоді зміщення визначається в полі `aio_offset`, інакше дані записуються (додаються) у кінець файлу. Якщо функції `aio_read()` та `aio_write()` застосовується до об’єкта, який не дозволяє виконувати введення-виведення за довільним зміщенням, тоді значення поля `aio_offset`

ігнорується. Після успішного ініціювання запиту AIO зміщення в дескрипторі відкритого файлу буде невизначене.

Значення поля `aio_reqprio` визначає значення зменшення пріоритету запиту, це значення віднімається від базового пріоритету планування. Тобто пріоритет запиту можна тільки зменшити. Більший номер пріоритету (менше значення `aio_reqprio`) значить більш високий пріоритет. Якщо виконання дій, вказаних у запитах AIO, до одного й того ж файлу заблоковані через нестачу ресурсу, тоді черговість виконання дій буде відповідно до пріоритетів, вказаних у запитах AIO. Базовий пріоритет планування – це базовий пріоритет планування процесу або потоку, який викликає запит AIO (залежить від реалізації).

Функція `sysconf()` з аргументом `_SC_AIO_PRIO_DELTA_MAX` повертає максимальне дозволене значення зменшення пріоритету запиту AIO. У `<limits.h>` може бути визначено макрос `AIO_PRIO_DELTA_MAX`.

Якщо для файлу було вказано прапорець `O_APPEND` або якщо значення поля `aio_offset` ігнорується для об’єкта ядра, тоді черговість виконання дій, вказаних у запитах AIO, застосованих для цього об’єкта, може бути довільна. Теоретично

такого не має бути, але POSIX таке дозволяє і реалізація має надавати умови, за яких таке може бути (використання багатопроцесорного комп'ютера, врахування пріоритетів у запитах AIO).

Якщо синхронізація введення та виведення застосована для дескриптора відкритого файлу, з яким асоційовано дескриптор файлу `aio_fildes`, тоді виконання дії, ініційованої в запиті AIO, виконується з гарантіями, які визначені в завершенні синхронізованого введення-виведення з гарантією цілісності даних або в завершенні синхронізованого введення-виведення з гарантією цілісності файлу.

Якщо функція `aio_read()` або `aio_write()` успішно ініціювала дію, ініційовану в запиті AIO, тоді вона повертає `0`, інакше вона повертає `-1`. Якщо номер помилки `EAGAIN`, тоді функцію не вдалось виконати через нестачу ресурсу для створення запиту AIO.

Функція `aio_fsync()` ініціює асинхронну синхронізацію для файлу, визначеним вказаним блоком управління, також для всіх дій, ініційованих у запитах AIO для цього файлу, які є в черзі на момент виклику функції.

```
#include <aio.h>

int aio_fsync(int op, struct aiocb *aiocb);
```

В об’єкті, на який вказує аргумент `aiocb`, використовуються значення полів `aio_fildes` та `aio_sigevent`, значення інших полів ігноруються. Значення аргументу `op` можуть бути такі: `O_DSYNC` (синхронізація виконується так, як визначено у `fdatasync()`), `O_SYNC` (синхронізація виконується так, як визначено у `fsync()`).

Якщо функція `aio_sync()` успішно ініціювала дію, визначеною запитом AIO, тоді вона повертає `0`, інакше вона повертає `-1`. Якщо номер помилки `EAGAIN`, тоді функцію не вдалося виконати через нестачу ресурсу для створення запиту AIO.

Функція `lio_listio()` ініціює синхронне або асинхронне виконання дій, визначених вказаними блоками управління.

```
#include <aio.h>

int lio_listio(int mode, struct aiocb *restrict const list[restrict],
       int nent, struct sigevent *restrict sigevent);
```

Назва цієї функції позначає «list directed input/output». Ця функція дає змогу ініціювати читання та запис, тому її можна використовувати замість функцій `aio_read()` та `aio_write()`. Але виклик цієї функції не є заміною кільком викликам системних викликів `*read*()` та `*write*()`, вказані запити AIO можуть виконуватися в довільному порядку (див. інформацію вище про прапорець `O_APPEND` та ігнорування поля `aio_offset`).

Значення покажчиків на блоки управління вказуються в масиві `list`, кількість елементів вказується в аргументів `nent`. Масив `list` може мати значення null покажчиків, відповідно ці елементи ігноруються. Значення поля `aio_lio_opcode` кожного блоку управління визначає дію запиту: `LIO_NOP` вказує ігнорувати цей блок управління; `LIO_READ` вказує використовувати цей блоку управління так, як би була викликана функція `aio_read()` для нього з дотриманням усієї її семантики; `LIO_WRITE` вказує використовувати цей блоку управління так, як би була викликана функція `aio_write()` для нього з дотриманням усієї її семантики.

Аргумент `mode` має мати одне з двох значень: `LIO_NOWAIT` вказує не чекати на завершення дій, ініційованих у запитах AIO; `LIO_WAIT` вказує чекати на завершення виконання всіх дій, вказаних у запитах AIO, у цьому випадку

аргумент `sigevent` ігнорується, оскільки виконання дій буде синхронне. Якщо аргумент `sigevent` не null покажчик, тоді він вказує на об’єкт, який визначає спосіб сповіщення коли всі дії, ініційовані в запитах AIO, будуть завершені. Значення поля `aio_sigevent` кожного кожного блоку управління визначає спосіб сповіщення про завершення виконання запиту. Тобто можна вказувати спосіб сповіщення про виконання всіх вказаних дій, ініційованих у запитах AIO, та/або спосіб сповіщення про виконання кожної дії, ініційованої в кожному запиту AIO, індивідуально.

Якщо аргумент `mode` має значення `LIO_NOWAIT`, тоді функція `lio_listio()` повертає `0`, якщо вона успішно ініціювала всі дії, вказані в запитах AIO, інакше вона повертає `-1`. Якщо аргумент `mode` має значення `LIO_WAIT`, тоді функція `lio_listio()` повертає `0`, якщо були успішно виконані всі дії, вказані в запитах AIO, інакше вона повертає `-1`.

Деякі номери помилок для `lio_listio()` мають відношення до вказаних запитів AIO. Якщо номер помилки `EAGAIN`, тоді функції не вистачило ресурсів для створення всіх вказаних запитів AIO. Якщо номер помилки `EIO`, тоді принаймні одну дії, ініційовану вказаним запитом AIO, не було виконано (має сенс якщо

аргумент mode має значення LIO_NOWAIT). Якщо номер помилки EINTR, тоді виконання функції було перервано сигналом, коли вона чекала на завершення виконання всіх дій, ініційованих запитами AIO (має сенс якщо аргумент mode має значення LIO_WAIT), цей сигнал може бути надісланий у результаті сповіщення завершення одної із дій, ініційованих вказаними запитами AIO, водночас інші дії, ініційовані вказаними запитами AIO, не скасовуються. Якщо lio_listio() завершується з помилкою EAGAIN, тоді деякі дії у вказаних запитах AIO могли бути успішно ініційовані, виконання кожної дії у вказаних запитах AIO треба перевіряти індивідуально. Якщо lio_listio() завершується з помилкою не EAGAIN, EIO або EINTR, тоді жодної дії не було ініційовано.

Функція sysconf() з аргументом _SC_AIO_LISTIO_MAX повертає максимальне дозволену кількість елементів списку в одному виклику функції lio_listio(). У <limits.h> може бути визначено макрос AIO_LISTIO_MAX та визначено макрос _POSIX_AIO_LISTIO_MAX.

Функція sysconf() з аргументом _SC_AIO_MAX повертає максимальну дозволену кількість одночасно виконуваних асинхронних дій (реалізація можуть мати власне розуміння щодо того, що є одночасне виконання асинхронних дій). У <limits.h> може бути визначено макрос AIO_MAX та визначено макрос _POSIX_AIO_MAX.

Функція `aio_cancel()` пробує скасувати одну асинхронну дію, визначену вказаним блоком управління, або кілька асинхронних дій, які були ініційовані для вказаного номера дескриптора файлу.

```
#include <aio.h>

int aio_cancel(int fd, struct aiocb *aiocb);
```

Якщо аргумент `aiocb` null покажчик, тоді скасовуються всі асинхронні дії, ініційовані для номера дескриптора файлу, який вказується в запиті AIO. Якщо аргумент `aiocb` не null покажчик і якщо аргумент `fd` не дорівнюється значенню поля `aio_fildes` вказаного блока управління, тоді результат не специфіковано. Реалізація визначає які асинхронні дії можна скасувати (насправді, якщо реалізація дає змогу скасувати певну асинхронну дію, то це не значить, що її завжди можна скасувати, навіть якщо вона ще не виконується, оскільки вона вже може бути запланована на виконання в деякому драйвері). Якщо були скасовані якісь асинхронні дії, тоді для них виконується сповіщення, яке було визначено в їхніх блоках управління.

Функція `aio_cancel()` повертає такі значення: `AIO_CANCELED` якщо всі асинхронні дії були скасовані; `AIO_NOTCANCELED` якщо принаймні одна асинхронна дія не була скасована; `AIO_ALLDONE` якщо всі асинхронні дії вже виконані (тобто вони вже не можуть бути скасовані); `-1` якщо її виконання завершилося з помилкою.

Функція `aio_suspend()` чекає на завершення виконання принаймні одної із дій, визначених вказаними блоками управління.

```
#include <aio.h>

int aio_suspend(const struct aiocb *const list[], int nent,
        const struct timespec *timeout);
```

Значення покажчиків на блоки управління вказуються в масиві `list`, кількість елементів вказується в аргументі `nent`. Масив `list` може мати значення null покажчиків, відповідно ці елементи ігноруються. Якщо в масиві `list` є покажчики на блок управління, який не був використаний для ініціювання запиту AIO, тоді ефект невизначений. Якщо аргумент `timeout` не null покажчик, тоді він вказує на об’єкт, який визначає максимальний час очікування.

Функція `aio_suspend()` повертає `0`, якщо принаймні одна із дій, визначених вказаним блоком управління, була завершена, інакше вона повертає `-1`. Якщо номер помилки `EAGAIN`, тоді жодна із дій, визначених вказаними блоками управління, не була завершена (має сенс якщо аргумент `timeout` null покажчик). Якщо номер помилки `EINTR`, тоді виконання функції було перервано сигналом, коли вона чекала на завершення виконання принаймні однієї із дій, визначених вказаними блоками управління, цей сигнал може бути надісланий у результаті сповіщення завершення однієї із дій, визначених вказаними блоками управління.

Функція `aio_error()` повертає статус виконання асинхронної дії, ініційованої вказаним блоком управління. Функція `aio_return()` повертає статус виконання завершеної асинхронної дії, ініційованої вказаним блоком управління.

```
#include <aio.h>

int aio_error(const struct aiocb *aiocb);
ssize_t aio_return(struct aiocb *aiocb);
```

Функція `aio_error()` повертає такі значення: `0` якщо асинхронну дію було успішно завершено; `EINPROGRESS` якщо виконання асинхронної дії ще не завершено; `ECANCELED`

якщо виконання асинхронної дії було скасовано; більше ніж `0` якщо виконання асинхронної дії було завершено з помилкою, повертається відповідний номер помилки (помилки `EINPROGRESS` та `ECANCELED` не можуть повертатися як помилки виконання дії); `-1` якщо її виконання завершилося з помилкою. Якщо аргумент `aiocb` вказує на блок управління, який не був застосований для ініціювання асинхронної дії, тоді результат виконання функції `aio_error()` не визначений.

Функція `aio_return()` повертає значення, яке позначає статус виконання асинхронної дії, або `-1` у разі помилки під час свого виконання. Якщо функція `aio_error()` для блоку управління повернула статус виконання `EINPROGRESS`, тоді результат, який повертає функція `aio_return()` для цього ж блоку управління, не визначено. Якщо функція `aio_error()` для блоку управління повернула статус виконання `ECANCELED`, тоді функція `aio_return()` повертає для цього ж блоку управління `-1`. Якщо функція `aio_error()` для блоку управління повернула статус виконання, який позначає помилку, тоді функція `aio_return()` повертає для цього ж блоку управління `-1`. Функцію `aio_return()` можна викликати тільки один раз для одного блоку управління і тільки тоді, коли відомо про завершення виконання асинхронної дії, визначеної цим блоком управління. Після цього цей блок

управління можна використовувати повторно для ініціювання нової асинхронної дії.

Якщо асинхронну дію не було виконано через помилку, тоді номер помилки такий самий, який визначений у номерах помилок у `pread()` або `read()` для `aio_read()`, `pwrite()` або `write()` для `aio_write()` та `fdatasync()` або `fsync()` для `aio_fsync()`.

Функція `aio_error()` завершується з помилкою `EINVAL`, якщо для завершеної асинхронної запиту AIO вже було викликано функцію `aio_return()`. Також виконання запиту AIO може завершитися з помилкою `EINVAL`. Тобто неможливо відрізнити до чого має відношення ця помилка, хоча викликати функцію `aio_error()` для завершеного запиту AIO, для якого була викликана функція `aio_return()` є логічною помилкою в програмі.

Наявність двох функцій `aio_error()` та `aio_return()` в API є надлишковим. Таку саму семантику можна реалізувати однією функцією, яка буде встановлювати значення двох додаткових полів у блоці управління.

Використання кваліфікатора `volatile` для поля `aio_buf` у `struct aiocb` не має сенсу. Відповідно до стандарту POSIX програма користувача не має якось особливо

працювати із вмістом буфера, який залучений у блоці управління. Зрозуміло, що це має сенс тільки для функції `aio_read()`, виконання якої призводить до модифікації вмісту буфера. Якщо покажчик на цей буфер був переданий у функцію `aio_read()` через значення поля блоку управління, тоді компілятор не може використовувати кешований вміст цього буфера після виклику будь-якої функції, реалізацію якої він не бачить, оскільки невідомо чи ця функція змінює вміст буфера. Навіть якщо `aio_error()` та/або `aio_return()` реалізовані як inline функції (якщо функції асинхронного введення-виведення POSIX реалізовано в режимі користувача), тоді в реалізації цих функцій мають бути залучені механізми синхронізації, які також будуть мати бар'єр компілятора. Якщо в реалізації функції `aio_return()` не використовується механізм синхронізації і ця функція викликається в іншому потоці, ніж у потоці, в якому була викликана функція `aio_error()` для того самого блоку управління, тоді там буде працювати механізм транзитивності в моделі пам'яті (потік має повідомити інший потік про результат виконання функції `aio_error()`).

Системи можуть реалізовувати нестандартні функції асинхронного введення-виведення, які можна використовувати разом із запитами AIO, визначених у

POSIX. Ці нестандартні функції принципово нічого не змінюють у семантиці стандартних запитів AIO.

Приклад. Програма відправляє на вказану адресу та номер порту вміст вказаного файлу, змінюючи маленькі літери на великі літери використовуючи TCP.

```
#include <sys/socket.h>
#include <sys/stat.h>
#include <aio.h>
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <netdb.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdio.h>
#include <unistd.h>

#define BUFSZ 100

struct request {
    char buf[BUFSZ];
    struct aiocb aiocb;
};
```

```
static void
transform(char *buf, size_t n)
{
    for (size_t i = 0; i < n; ++i)
        buf[i] = (char)toupper(buf[i]);
}

int
main(int argc, char *argv[])
{
    struct request requests[2];
    struct stat statbuf;
    const char *pathname = NULL;
    const char *nodename, *servname;
    const struct aiocb *aiocb_list[2];
    struct addrinfo ai_hints = { 0 };
    struct addrinfo *ai, *ai_result;
    void *buf;
    ssize_t nbytes;
    off_t offset;
    int filefd, sockfd, opt, rv, error;
    bool readpending_flag, writepending_flag, last_flag;

    opterr = 0;
    while ((opt = getopt(argc, argv, "f:")) != -1) {
        switch (opt) {
        case 'f':
            pathname = optarg;
```

```
            break;
        default:
            exit_errx("wrong option -%c", optopt);
        }
    }
    if (pathname == NULL)
        exit_errx("specify the -f option");
    if (optind != argc - 2)
        exit_errx("wrong number of command line arguments");
    nodename = argv[optind];
    servname = argv[optind + 1];

    ai_hints.ai_family = AF_UNSPEC;
    ai_hints.ai_socktype = SOCK_STREAM;
    ai_hints.ai_protocol = IPPROTO_TCP;
    rv = getaddrinfo(nodename, servname, &ai_hints, &ai_result);
    if (rv != 0) {
        if (rv == EAI_SYSTEM)
            exit_err("getaddrinfo(): %s", gai_strerror(rv));
        exit_errx("getaddrinfo(): %s", gai_strerror(rv));
    }

    for (ai = ai_result; ai != NULL; ai = ai->ai_next) {
        sockfd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
        if (sockfd < 0) {
            if (system_error())
                exit_err("socket()");
            continue;
```

```
    }
    if (connect(sockfd, ai->ai_addr, ai->ai_addrlen) < 0) {
        if (system_error())
            exit_err("connect()");
        if (close(sockfd) < 0)
            exit_err("close()");
        continue;
    }
    break;
}
freeaddrinfo(ai_result);
if (ai == NULL)
    exit_errx("cannot create any socket with needed protocol");

filefd = open(pathname, O_RDONLY);
if (filefd < 0)
    exit_err("open()");
if (fstat(filefd, &statbuf) < 0)
    exit_err("fstat()");

sigpipe_ignore();

aiocb_list[0] = aiocb_list[1] = NULL;
requests[0].aiocb.aio_buf = requests[0].buf;
requests[0].aiocb.aio_fildes = filefd;
requests[0].aiocb.aio_reqprio = 0;
requests[0].aiocb.aio_sigevent.sigev_notify = SIGEV_NONE;
requests[1].aiocb.aio_buf = requests[1].buf;
```

```
requests[1].aiocb.aio_fildes = sockfd;
requests[1].aiocb.aio_offset = 0;
requests[1].aiocb.aio_reqprio = 0;
requests[1].aiocb.aio_sigevent.sigev_notify = SIGEV_NONE;

offset = 0;
readpending_flag = writepending_flag = last_flag = false;
for (;;) {
    if (readpending_flag) {
        error = aio_error(&requests[0].aiocb);
        if (error < 0)
            exit_err("aio_error()");
        if (error > 0) {
            if (error != EINPROGRESS)
                exit_errn(error, "request in aio_read()");
        } else {
            nbytes = aio_return(&requests[0].aiocb);
            if (nbytes < 0)
                exit_err("aio_return()");
            if ((size_t)nbytes != requests[0].aiocb.aio_nbytes)
                exit_errx("short read, %zd of %zu bytes",
                    nbytes, requests[0].aiocb.aio_nbytes);
            printf("Read completed\n");
            transform((char *)requests[0].aiocb.aio_buf,
                requests[0].aiocb.aio_nbytes);
            aiocb_list[0] = NULL;
            readpending_flag = false;
        }
```

```
    }
    if (writepending_flag) {
        error = aio_error(&requests[1].aiocb);
        if (error < 0)
            exit_err("aio_error()");
        if (error > 0) {
            if (error != EINPROGRESS)
                exit_errn(error, "request in aio_write()");
        } else {
            nbytes = aio_return(&requests[1].aiocb);
            if (nbytes < 0)
                exit_err("aio_return()");
            if (nbytes == 0)
                exit_errx("short write, %zd of %zu bytes",
                    nbytes, requests[1].aiocb.aio_nbytes);
            if ((size_t)nbytes != requests[1].aiocb.aio_nbytes) {
                warn_errx("short write, %zd of %zu bytes",
                    nbytes, requests[1].aiocb.aio_nbytes);
                requests[1].aiocb.aio_buf =
                    (char *)requests[1].aiocb.aio_buf + nbytes;
                requests[1].aiocb.aio_nbytes -= (size_t)nbytes;
                if (aio_write(&requests[1].aiocb) < 0)
                    exit_err("aio_write()");
            } else {
                printf("Write completed\n");
                aiocb_list[1] = NULL;
                writepending_flag = false;
            }
```

```
    }
}
if (!readpending_flag && !writepending_flag) {
    buf = (char *)requests[0].aiocb.aio_buf;
    requests[0].aiocb.aio_buf = requests[1].aiocb.aio_buf;
    requests[1].aiocb.aio_buf = buf;
    if (offset > 0 && !last_flag) {
        printf("AIO write request\n");
        requests[1].aiocb.aio_nbytes = requests[0].aiocb.aio_nbytes;
        if (aio_write(&requests[1].aiocb) < 0)
            exit_err("aio_write()");
        aiocb_list[1] = &requests[1].aiocb;
        writepending_flag = true;
    }
    nbytes = (ssize_t)(statbuf.st_size - offset);
    if (nbytes > BUFSZ)
        nbytes = BUFSZ;
    if (nbytes > 0) {
        printf("AIO read request\n");
        requests[0].aiocb.aio_offset = offset;
        requests[0].aiocb.aio_nbytes = (size_t)nbytes;
        if (aio_read(&requests[0].aiocb) < 0)
            exit_err("aio_read()");
        offset += (off_t)nbytes;
        aiocb_list[0] = &requests[0].aiocb;
        readpending_flag = true;
    } else {
        last_flag = true;
```

```
            }
        }
        if (!readpending_flag && !writepending_flag)
            break;
        if (aio_suspend(aiocb_list, 2, NULL) < 0)
            exit_err("aio_suspend()");
    }

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