Podstawy programownia (w języku C++)

I/O (operacje wejścia-wyjścia)

Marek Marecki

Polsko-Japońska Akademia Technik Komputerowych

4 października 2021

Overview

I/O

I/O czyli „operacje wejścia wyjścia” (ang. *input-output*) pozwala programom na interakcję ze światem zewnętrznym.
Bez I/O nie byłoby możliwe napisanie nawet tak prostego programu jak klasyczne „Hello, World!”.

C++: I/O streams

Biblioteka standardowa języka C++ dostarcza tzw. „I/O streams”: struktury danych, które opakowują mechanizmy I/O dostarczane przez system operacyjny. Są one proste w użyciu (np. `std::cout`, `std::cin`) i oferują spójny interfejs „strumienia” również dla typów, które udają I/O, np. `std::ostringstream` i `std::istringstream`.

I/O streams jednak sprawdzają się średnio w sytuacjach kiedy potrzeba jest zrelizować coś co wykracza poza ich „happy path”, oraz nie oferują nic w zakresie I/O z wykorzystaniem sieci.
Z tego powodu pominiemy je i zajmiemy się mechanizmami I/O oferowanymi przez system operaycjny Linux.

Linux: file descriptors

Linux jest systemem Unixopodobnym (ang. **nix-like*) i zapewnia mechanizmy I/O opisane przez standard POSIX.

Oprócz mechanizmów I/O narzuconych przez standard POSIX, Linux oferuje swoje własne API (np. `io_uring`), jednak w tym wykładzie nie będzie ono omawiane.

POSIX I/O: Everything is a file

Systemy POSIX działają zgodnie z zasadą „everything is a file” czyli „wszystko jest plikiem”. Dzięki temu możemy używając jednego, spójnego interfejsu manipulować sporą ilością rzeczy w systemie.

Pod kątem tego wykładu ważniejsze jednak będzie to, że ta zasada oznacza, że bardzo podobny kod pozwoli nam obsłużyć I/O na plikach jak i na innych metodach przenoszenia danych.

Overview

API

1. `open(2)`
2. `read(2)`
3. `write(2)`
4. `lseek(2)`
5. `stat(2)`, `lstat(2)`
6. `close(2)`

Otwieranie i zamykanie pliku

`open(2)`, `close(2)`

Stworzenie pliku i z prawami do zapisu i odczytu przez użytkownika, a następnie (po użyciu) zamknięcie file descriptora, który na niego wskazuje:

```
auto name = std::string{"./example.txt"};
auto fd = open(name.c_str(), O_CREAT|O_RDWR, S_IRUSR|S_IWUSR);

// do something with fd

close(fd);
```

Funkcja `open(2)` może zwrócić wartość -1 w przypadku błędu[1]!

[1] Zdecydowana większość funkcji z API POSIX zwraca -1 w przypadku błędu, i ustawia globalną zmienną `errno` na wartość informującą o tym co dokładnie poszło nie tak.

Zapis do deskryptora

`write(2)`

Zapis danych polega na podaniu funkcji `write(2)` wskaźnika do pewnego obszaru pamięci, oraz podaniu ilości bajtów, która ma być z tego obszaru zapisana do file descriptora:

```
auto buf = std::string{"Hello, World!\n"};
auto n = write(fd, buf.data(), buf.size());
if (n == -1) {
    perror("write(2)");
}
```

Odczyt z deskryptora

read(2)

Odczyt danych polega na podaniu funkcji `read(2)` wskaźnika do pewnego obszaru pamięci, oraz podaniu ilości bajtów, która powinna być do niego wczytana[2]:

```
std::array<char, 4096> buf { 0 };
auto const n = read(fd, buf.data(), buf.size());
if (n == -1) {
    perror("read(2)");
} else {
    std::cout << std::string{buf.data(), static_cast<size_t>(n)};
}
```

[2] należy pamiętać, żeby nie podać wartości większej niż rozmiar bufora

Częściowy sukces

`write(2)`, `read(2)`

Dla funkcji `write(2)` i `read(2)` w przypadku sukcesu wynikiem jest ilość zapisanych lub odczytanych bajtów. Nie ma *absolutnie żadnej gwarancji*, że ta ilość będzie dokładnie równa ilości przekazanej jako argument[3].

To ile bajtów jest możliwe do zapisu lub odczytu nie zależy tylko od naszego programu. Mają na to wpływ inne uruchomione procesy, rozmiary buforów systemu operacyjnego, scheduler I/O, prędkość dysku i łącza, itd.

[3] ta sama uwaga dotyczy I/O wykonywanego po sieci

Informacje o pliku

fstat(2)

Używając funkcji `fstat(2)` można odczytać np. rozmiar pliku, żeby móc określić ile bajtów potrzeba żeby wczytać jego całość:

```
struct stat info;
memset(&info, 0, sizeof(info));
auto const r = fstat(fd, &info);
if (r == -1) {
    perror("stat(2)");
} else {
    std::cout << info.st_size << " byte(s)\n";
}
```

Zadanie: zapis studenta

Korzystając ze struktury danych `Student` z poprzednich zajęć napisz program, który zapisze ją do pliku `student.txt`. Niech każde pole będzie zapisane w osobnej linijce (tj. zakończone znakiem nowej linii).
Kod źródłowy: `s06-store-student.cpp`

Przykładowy plik:

```
Student Studencki
s12345
1
4.25
```

Zadanie: odczyt studenta

Korzystając ze struktury danych `Student` z poprzednich zajęć napisz program, który wczyta ją z pliku `student.txt` (niech wczyta to co zapisał program z poprzedniego zadania).
Kod źródłowy: `s06-load-student.cpp`

Zadanie: drukowanie pliku

Napisz program, który pobierze jako argument (`argv`) nazwę pliku, oraz wydrukuje zawartość tego pliku na standardowe wyjście. Obsłuż sytuację, w której plik nie istnieje.
Kod źródłowy: `s06-cat.cpp`

Overview

API

1. `socket(2)`
2. `bind(2)`
3. `listen(2)`
4. `connect(2)`
5. `read(2)`
6. `write(2)`
7. `shutdown(2)`
8. `close(2)`
9. `signal(2)` (blokada `SIGPIPE`)
10. `ip(7)`, `ipv6(7)`
11. `tcp(7)`, `udp(7)`

Utworzenie socketu TCP/IP

`socket(2)`

```
auto sock = socket(AF_INET, SOCK_STREAM, 0);
```

Stworzy to „goły” socket TCP/IP, który można potem wykorzystać do nasłuchiwania na połączenia (server), albo do nawiązania połączenia ze zdalną maszyną (client).

Zamknięcie połączenia

`shutdown(2)`

```
shutdown(sock, SHUT_RDWR);
```

Zamknie to kanały komunikacyjne i w poprawny sposób zakończy połączenie. Zasoby (file descriptor i socket) nie będą zwolnione do systemu operacyjnego.

Zamknięcie socketu

`close(2)`

```
close(sock);
```

Zamknięcie socketu i file descriptora, oraz oddanie zasobów do systemu operaycjnego.

Określenie adresu

`inet_pton(2)`, `sockaddr_in`

```
auto const ip = std::string{"192.168.0.24"};
auto const port = uint16_t{8080};

sockaddr_in addr;
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htobe16(port);
inet_pton(addr.sin_family, ip.c_str(), &addr.sin_addr);
```

W przypadku serwera - na jakim adresie będzie nasłuchiwać na połączenia. W przypadku klienta - na jaki adres spróbuje nawiązać połączenie.

Przywiązanie do adresu

BIND(2)

```
sockaddr_in addr;
// prepare the address

bind(sock, reinterpret_cast<sockaddr*>(&addr), sizeof(addr));
```

Nasłuchiwanie na połączenia

LISTEN(2)

```
listen(sock, SOMAXCONN); // or 0 for implementation-default value
```

Argument podany jako SOMAXCONN określa maksymalną liczbę połączeń oczekujących na odebranie.

Odebranie połączenia

`accept(2)`

```
auto client_sock = accept(sock, nullptr, nullptr);
```

Zmienna `client_sock` zawiera file descriptor wskazujący na socket, który umożliwia komunikację z klientem.
Dodatkowe dwa parametry wskazują na strukturę danych (i jej rozmiar), do której system operacyjny powinien zapisać adres klienta. Jeśli ta informacja nie jest potrzebna, można przekazać `nullptr`.

Nawiązanie połączenia

connect(2)

```
auto server_sock = connect(
    sock
    , reinterpret_cast<sockaddr*>(&addr)
    , sizeof(addr)
);
```

Zmienna `server_sock` zawiera file descriptor wskazujący na socket, który umożliwia komunikację z serwerem.

Komunikacja

TCP/IP

Socket, który w ten sposób został stworzony reprezentuje *połączenie* między serwerem, a klientem, które jest *strumieniowe*.
Oznacza to, że dane zapisane jednym wywołaniem `write(2)` na serwerze, nie muszą wcale być możliwe do odczytania jednym wywołaniem `read(2)` na kliencie, i *vice versa* – połączenie strumieniowe nie ma pojęcia „wiadomości”. Musi to być zaimplementowane w programie (np. definiując schemat danych, które są wysyłane – protokół).

Opóźnienia i błędy

TCP/IP

Należy pamiętać, że komunikacja po sieci może być **wolna** i **zawodna**. Bardzo istotna jest poprawna obsługa możliwych błędów.

Współcześnie używana pamięć masowa potrafi przekazywać dane z prędkościami sięgającymi kilku GB/s (*gigabajtów* na sekundę). Połączenia sieciowe rzadko osiągają prędkość 1 Gb/s (*gigabitu* na sekundę).

Co więcej, sieć to nie połączenie SATA lub PCIe, tylko kable, kabelki i fale radiowe. Ta infrastruktura może zawieść - fale mogą być zagłuszone, a kable przerwane, co może wymagać użycia wolniejszej ścieżki lub doprowadzić do zerwania połączenia.

Zadanie: serwer echo

Napisz serwer, który będzie oczekiwać na połączenia od klientów, odczytywać od nich jakieś dane (zakończone znakiem nowej linii), oraz odsyłać im te dane z powrotem. Pamiętaj o obsłużeniu sytuacji, w której klient się rozłącza!
Kod źródłowy: `s06-echo-server.cpp`

Zadanie: serwer plików (odczyt)

Napisz serwer, który będzie oczekiwać na połączenia od klientów, odczytywać od nich dane (zakończone znakiem nowej linii) reprezentujące ścieżki do plików.
Po odebraniu ścieżki serwer powinien odczytać plik, i wysłać do klienta ścieżkę, znak nowej linii, oraz zawartośc pliku. Jeśli plik nie istnieje serwer powinien odesłać smutną buźkę: `:-(`
Kod źródłowy: `s06-file-server.cpp`

Zadanie: wymiana wiadomości

Napisz serwer, który będzie umożliwiał klientom wysłanie wiadomości do serwera. Serwer powinien te wiadomości zapisywać w pamięci, i odpowiadać losową wiadomością (jedną z tych które otrzymał już od innych klientów).
Kod źródłowy: `s06-random-note-server.cpp`

Zadanie: client

Napisz program, który będzie umożliwiał połączenie się z wybranym serwerem, wysyłanie do niego linijek tekstu, i odbieranie od niego danych. Użyj tego klienta do komunikacji z serwerami z poprzednich zadań.
Kod źródłowy: `s06-client.cpp`

Overview

API

1. `select(2)`
2. `poll(2)`
3. `epoll(7)` (Linux)
4. `aio(7)`

Podane strony manuala zawierają przykładowy kod w języku C, który można przepisać, skompilować, i sprawdzić jak działa.

System Linux oferuje jeszcze interfejs `io_uring`, który jest na tyle nowy, że nie ma jeszcze strony w manualu – informacji o nim trzeba szukać w Internecie.

Zastosowanie

I/O zajmuje czas i może zablokować nasz wątek. Jeśli istnieje potrzeba uniknięcia takiej sytuacji, to można poradzić sobie z tym na kilka sposobów:

1. utworzyć wątek dedykowany operacjom I/O, który może być blokowany
2. użyć *nonblocking I/O*, czyli wywołań z gwarancją nieblokowania wątku (ten styl wiąże się z dodatkowymi wyzwaniami)
3. użyć *asynchronous I/O*, czyli wywołań, które powodują wykonanie operacji I/O „w tle” przez system operacyjny

Przykłady

Wykonywanie kilku operacji I/O naraz, lub operacji I/O i czegoś jeszcze, jest bardzo częstą sytuacją - np.:

1. serwer obsługuje kilka klientów naraz
2. program odczytuje kopiuje pliki i drukuje pasek postępu
3. gra wczytuje poziom i pokazuje użytkownikowi „tip of the day”
4. czat oczytuje wiadomości, ale jednocześnie drukuje powiadomienia „ktoś pisze...”

Overview

Podsumowanie

Student powinien umieć:

1. prowadzić operacje I/O na plikach
2. prowadzić operacje I/O z wykorzystaniem połączenia sieciowego używając protokołu TCP/IP
3. znać możliwości i przykłady zastosowania API dostarczanego przez standard POSIX dla asychronicznych i nieblokujących operacji I/O

Zadania

Podsumowanie

Zadania znajdują się na slajdach 14, 15, 16, 29, 30, 31, 32.