Лекція 4
TCP-сокети, частина 1

Характеристики TCP

Протокол TCP (Transmission Control Protocol, протокол управління передачею) – це транспортний протокол у комунікаційному домені Internet, визначений у RFC 9293. TCP потребує встановлення логічного з’єднання між двома сторонами, забезпечує дуплексне надійне впорядковане доставлення потоку байтів з перевіркою на помилки у режимі мережевої адресації unicast (за потреби режим передачі даних у TCP-з’єднанні може бути змінений на симплексний), підтримує варіант out-of-band даних. Зазвичай під час встановлення TCP-з’єднання є активна сторона (клієнт), яка відправляє запит на встановлення з’єднання, та є пасивна сторона (сервер), яка очікує запитів на встановлення з’єднань, після встановлення TCP-з’єднання ролі сторін у загальному розумінні є умовними.

Терміни, які визначають режим мережевої адресації. Unicast (one-to-one). Multicat (one-to-many-of-many). Широкомовний (broadcast, one-to-all). Anycast (one-to-one-of-many).

Терміни, які визначають режим передачі даних. *Дуплексний* (*duplex*) або *повнодуплексний* (*full-duplex*) режим передачі даних дає змогу відправляти та отримувати дані в будь-який момент часу. *Напівдуплексний* (*half-duplex*) режим передачі даних дає змогу відправляти дані тільки в одному напрямку в один момент часу. *Симплексний* (*simplex*) режим передачі даних дає змогу відправляти дані тільки в одному напрямку, іноді такий режим визначають як *напівдуплексний* (тобто може бути плутанина в термінах).

Режим передачі даних, визначений для транспортного протоколу (наприклад, дуплексний або симплексний для TCP), не значить, що такий самий режим передачі даних повинен бути на канальному рівні передачі даних (наприклад, напівдуплексний для Ethernet). Власне ідея мережевої моделі OSI або мережевої моделі TCP/IP полягає в тому, щоб розподілити мережеві протоколи за рівнями, щоб зменшити залежність між мережевими протоколами, які використовуються разом.

TCP забезпечує надійне доставлення даних, але ця надійність є умовною. Деякі характеристики цієї надійності можна поліпшити практично, а деякі характеристики не можна вдосконалити принципово. Про це йдеться далі.

TCP не гарантує, що відправлені дані будуть отримані, оскільки таку гарантію не може забезпечити жодний мережевий протокол. Мережа є ненадійним середовищем передачі даних, оскільки на мережу впливають зовнішні чинники, які не контролює жодна зі сторін комунікації (фізична неможливість передавати відправлені дані, відмова спрямовувати відправлені дані іншими системами на шляху від відправника до отримувача і т. ін.).

У TCP-з'єднанні є дві сторони. Одна сторона – це програма, яка реалізує TCP, або програма, яка використовує TCP. Зазвичай TCP реалізує ядро, а TCP використовує процес користувача або процес ядра. Що мається на увазі під «стороною з'єднання» конкретно зрозуміло з контексту або вказується.

Реалізація TCP відправляє дані *сегментами* (*segment*) та вимагає від іншої сторони TCP-з'єднання відправляти *підтвердження* (*acknowledgment*) отримання відправлених даних. Сегменти з даними та сегменти з підтвердженнями можуть об'єднуватися, завдяки наявності необхідних полів у заголовку TCP.

Якщо підтвердження про отримання відправлених даних не отримано, тоді реалізація TCP відправляє відповідний сегмент знову, кожний раз збільшуючи проміжок часу між повторними відправленнями. Це дає змогу не перенавантажувати мережу трафіком, оскільки перенавантаження мережі трафіком може бути причиною неотримання відповідного відправленого сегмента або неотримання відповідного підтвердження. Якщо підтвердження про отримання відправлених даних не отримано, тоді через деякий час (кілька хвилин, конкретне значення визначає реалізація) реалізація TCP позначає помилку в з'єднанні. Реалізація TCP постійно вираховує час необхідний для отримання підтвердження (round-trip time, RTT) використовуючи різні алгоритми, тому може динамічно адаптуватися до характеристик мережі, яка може постійно змінюватися. RTT може бути визначений як кілька мілісекунд або десятки секунд. Реалізація TCP використовує різні алгоритми розрахунку RTT, які враховують різні сценарії поведінки трафіку в мережі.

У TCP є *контрольна сума* (*checksum*), яка обчислюється для значень полів псевдозаголовка IP (кілька полів заголовка IPv4 або IPv6), значень полів заголовка TCP та даних (payload) сегмента та зберігається в заголовку TCP.

Заголовок TCP та вміст сегмента не шифруються як усталено, тому заголовок TCP та/або вміст сегмента треба шифрувати іншими засобами за потреби. Алгоритм обчислення контрольної суми відомий та використовує тільки дані в IP пакеті, що відправляється, тому отримувач даних, відправлених через TCP, не знає чи ці дані були змінені разом із контрольною сумою сторонніми системами під час спрямування цього пакету в мережі. Також алгоритм обчислення контрольної суми не є надійним порівняно з іншими алгоритмами обчислення контрольних сум, але достатній для практичного використання.

Сторона TCP-з'єднання точно знає про позначення помилки в з'єднанні, але ця помилка не значить, що відправлені раніше дані не були отримані іншою стороною. Будь-яка сторона TCP-з'єднання може точно знати про то, що відправлені дані були отримані іншою стороною (процесом), якщо отримає від іншої сторони повідомлення про отримання даних. Сторона TCP-з'єднання точно знає, що відправлені дані були успішно отримані іншою стороною (системою), у разі успішного завершення цього з'єднання.

Відправлені байти даних у TCP нумеруються, тому реалізація TCP може визначити отримання даних не по порядку (out of order) та зібрати отримані

сегменти в правильному порядку. Реалізація TCP ігнорує отримання дубльованих даних, але підтверджує їхнє отримання (попередні підтвердження могли бути не отриманні іншою стороною TCP-з'єднання). Відправлені дані можуть бути отримані не по порядку та можуть бути дубльовані, оскільки протокол мережевого рівня IP не визначає нічого, щоб забезпечити порядок отримання відправлених даних та забезпечити відсутність дубльованих даних, також інша сторона TCP-з'єднання може відправляти кілька сегментів поспіль та повторно відправляти сегменти, для даних, для яких не отримав підтвердження. Завдяки нумерації відправлених байтів реалізація TCP може підтверджувати отримання відразу кількох сегментів та завдяки відповідній опції в заголовку TCP може підтверджувати отримання кількох сегментів вибірково. *Вибіркове підтвердження* (*selective acknowledgment*) значно підвищує продуктивність TCP у ненадійних мережах проти кумулятивного підтвердження, оскільки вказує іншій стороні TCP-з'єднання які сегменти (тобто дані) уже було отримано і їх не треба відправляти знову.

TCP має *управління потоком* (*flow control*). Реалізація TCP постійно повідомляє іншу сторону з'єднання скільки байт даних бажано відправляти, ця

кількість байт у термінах TCP називається *розмір вікна* (*window size*). Якщо розмір вікна дорівнює нулю, тоді відправник має почекати перед відправленням наступних сегментів. Звісно, інша сторона TCP-з'єднання може ігнорувати значення отриманого розміру вікна, але тоді наступні відправлені дані просто будуть ігноруватися. Розмір вікна визначає розмір місця в буфері отриманих даних TCP-з'єднання. Якщо попередні дані з буфера отриманих даних TCP-з'єднання не були забрані програмою, яка використовує це з'єднання, тоді нема сенсу приймати наступні дані, оскільки вони також можуть бути не забрані цією програмою.

TCP-з'єднання ідентифікується кортежем з чотирьох значень. Перші два значення – це IP адреса відправника (source address), вказується в полі заголовка IP, та номер порту відправника (source port), вказується в полі заголовка TCP. Другі два значення – це IP адреса отримувача (destination address), вказується в полі заголовка IP, та номер порти отримувача (destination port), вказується в полі заголовка TCP. Ці пари значень на кожній стороні TCP-з'єднання міняються місцями.

Підтримування TCP зазвичай реалізує ядро. Програма користувача використовує TCP послуговуючись відповідними функціями API між програмою користувача та ядром. Для програми, яка використовує TCP, дані передаються як потік байтів. TCP не розмежовує відправлені дані, тому програми, які потребують розділяти дані на повідомлення, мають заздалегідь домовитися про протокол рівня застосунку, який вони будуть використовувати під час комунікації.

Програма користувача може сама реалізувати протокол схожий на TCP використовуючи UDP (User Datagram Protocol, протокол датаграм користувача), але реалізувати всі оптимізації, які закладені в стандартах TCP потребує часу. Також програма користувача не може знати характеристики інших мережевих з’єднань у системі, тому не може використовувати їх для динамічної зміни характеристик власного протоколу. Відправлення та отримання даних між програмою користувача та ядром майже завжди вимагатиме копіювання даних між адресними просторами користувача та ядра в системних викликах, повторне відправлення даних вимагатиме виклику відповідного системного

виклику. Тому програмі користувача може бути вигідно використовувати TCP, якщо властивостей TCP достатньо для рішення потрібних завдань.

Програма користувача може створити сокет у комунікаційному домені Internet для можливого використання його в TCP, тобто створити TCP-сокет, системним викликом `socket(domain, SOCK_STREAM, IPPROTO_TCP)`, значення `domain` має бути `AF_INET` для IPv4 або `AF_INET6` для IPv6. Якщо в значенні протоколу вказати нуль, тоді типове значення протоколу для типу сокета `SOCK_STREAM` буде `IPPROTO_TCP` у комунікаційному домені Internet. У сучасних програмах користувача, краще явно вказувати протокол для комунікаційних доменів, які дозволяють таке робити, для запобігання можливої двозначності.

Призначення адреси сокету, частина 1

Деякі протоколи вимагають призначення адреси сокету, а деякі протоколи дозволяють призначити адресу сокету перед його використанням у комунікації. Адреса сокета вказується в структурі адреси сокета відповідного комунікаційного домену. Структури адрес сокетів різні в різних комунікаційних доменах. Функції API, які працюють з адресами сокетів, отримують в аргументі покажчик на `struct sockaddr`. Відповідно покажчик на об’єкт, який містить структуру адреси сокета в конкретному комунікаційному домені, треба привести до покажчика на `struct sockaddr`.

Процес користувача може призначити сокету wildcard мережеву адресу. Ця мережева адреса може бути в подальшому змінена ядром або не змінена, залежить від ролі сторони TCP-з’єднання (клієнт або сервер). Процес користувача може призначити сокету нульовий номер порту, тим самим роблячи запит ядру призначити сокету вільний випадковий номер (ephemeral) порту. Зазвичай це буде номер порту більший за номер 1023. Зазвичай процес користувача має мати відповідні привілеї, щоб призначити сокету номер порту менший за 1024.

Номери портів діляться на три діапазони. Номери портів від 1–1023 – це *широко відомі порти* (*well-known ports*), які контролює та призначає сервісами IANA (Internet Assigned Numbers Authority), номери портів від 1024–49151 – це *зареєстровані порти* (*registered ports*), які IANA не контролює, але реєструє їх для зручності, номери портів 49152–65535 – це *приватні порти* (*private ports*), які IANA не контролює та не реєструє. Значення 49152 – це (65536 × 3) ÷ 4.

Якщо номер порту отримувача належить діапазону значень 1–1023, тоді це не може бути використано в загальному випадку для авторизації користувача, від імені якого виконується процес-отримувач. По-перше, інша система може не обмежувати процеси і її процеси можуть призначати будь-які номери портів сокетам. По-друге, інша система може бути скомпрометована, тому не можна довіряти її діям. Якщо відомо завдяки якимось механізмам, що інша система не скомпрометована і вона вимагає відповідних привілеїв процесів для призначення номерів портів у діапазоні значень 1–1023, тоді належність номера порту отримувача до діапазону значень 1–1023 може бути використана для авторизації користувача, від імені якого виконується процес-отримувач. Але якщо трафік між системами не шифрується, тоді така авторизація

користувача нічого не значить. Тут під шифруванням розуміється саме шифрування трафіку, шифрування даних протоколу рівня застосунку недостатньо, оскільки системи, які спрямовують відправлені дані на шляху від відправника до отримувача, можуть змінювати вміст заголовків TCP (номери портів у тому числі).

Системний виклик `bind()` призначає адресу сокету, асоційованого з вказаним дескриптором файлу.

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

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Аргумент `sockfd` – це дескриптор файлу, з яким має бути асоційований сокет. Аргумент `addr` для комунікаційного домену Internet має вказувати на об’єкт типу `struct sockaddr_in`, `struct sockaddr_in6` або `struct sockaddr_storage`. Аргумент `addrlen` має дорівнюватися розміру відповідної структури адреси сокета. Сокету можна призначити конкретну мережеву адресу або wildcard адресу, сокету неможливо призначити список мережевих адрес. Цей системний виклик не змінює вміст

об’єкта на який вказує аргумент `addr` не тільки тому, що це покажчик на константні дані, але ще тому, що зазвичай у цьому нема сенсу.

Системний виклик `getsockname()` повертає призначену адресу сокету, який асоційований з вказаним дескриптором файлу.

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

int getsockname(int sockfd, struct sockaddr *restrict addr,
        socklen_t *restrict addrlen);
```

Аргумент `sockfd` – це дескриптор файлу, з яким має бути асоційований сокет. Аргумент `addr` для комунікаційного домену Internet має вказувати на об’єкт типу `struct sockaddr_in`, `struct sockaddr_in6` або `struct sockaddr_storage`. Значення `*addrlen` має дорівнюватися розміру відповідної структури адреси сокета, після виконання системного виклику це значення дорівнюватиметься розміру збереженої адреси сокета в об’єкті, на який вказує аргумент `addr`. Якщо розміру наданого об’єкта, на який вказує аргумент `addr` не достатньо, тоді ядро збереже частину вмісту адреси сокета в ньому. Якщо сокету не було призначено ім’я, тоді вміст об’єкта, на який вказує аргумент `addr`, не визначено. Використання слова

«name» у назві системного виклику є оманливим, сокету може бути призначена адреса сокета, а не ім’я (мережеве ім’я, чи будь-яке інше).

Приклад програми на C, яка дає змогу вказати текстове представлення IPv4 адреси та/або номера порту в аргументах командного рядка, призначає вказані значення або wildcard адресу та/або нульовий порт TCP-сокета, потім виводить призначену адресу сокету. У прапорцях функції `getaddrinfo()` вказано `AI_PASSIVE`, для того, щоб поверталася wildcard адреса в результаті.

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

static void
print_sin4(const char *msg, const struct sockaddr_in *sin4)
{
    char repr[INET_ADDRSTRLEN];

    if (getnameinfo((struct sockaddr *)sin4, sizeof(*sin4), repr,
```

```
        sizeof(repr), NULL, 0, NI_NUMERICHOST) != 0
    ) {
        repr[0] = '?';
        repr[1] = '\0';
    }
    printf("%s %s:%" PRIu16 "\n", msg, repr, ntohs(sin4->sin_port));
}

int
main(int argc, char **argv)
{
    struct sockaddr_in sin4;
    const char *addr_str = NULL;
    const char *port_str = NULL;
    socklen_t addrlen;
    int opt, sockfd;

    opterr = 0;
    while ((opt = getopt(argc, argv, "a:p:")) != -1) {
        switch (opt) {
        case 'a':
            addr_str = optarg;
            break;
        case 'p':
            port_str = optarg;
            break;
        default:
            exit_errx("wrong option -%c", optopt);
```

```
    }
}
if (optind < argc)
    exit_errx("wrong number of command line arguments");

if (addr_str != NULL || port_str != NULL) {
    struct addrinfo ai_hints = { 0 };
    struct addrinfo *ai_res;
    int error;

    ai_hints.ai_flags = AI_PASSIVE | AI_NUMERICHOST | AI_NUMERICSERV;
    ai_hints.ai_family = AF_INET;
    ai_hints.ai_socktype = SOCK_STREAM;
    ai_hints.ai_protocol = IPPROTO_TCP;
    error = getaddrinfo(addr_str, port_str, &ai_hints, &ai_res);
    if (error != 0)
        exit_errx("getaddrinfo(): %s", gai_strerror(error));
    if (ai_res->ai_next != NULL)
        exit_errx("wrong algorithm: getaddrinfo() "
            "returned multiple results");
    sin4 = *(struct sockaddr_in *)ai_res->ai_addr;
    freeaddrinfo(ai_res);
} else {
    sin4 = (struct sockaddr_in){ 0 };
    sin4.sin_family = AF_INET;
    sin4.sin_addr.s_addr = INADDR_ANY;
}
print_sin4("Argument", &sin4);
```

```
    sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sockfd < 0)
        exit_err("socket()");
    if (bind(sockfd, (struct sockaddr *)&sin4, sizeof(sin4)) < 0)
        exit_err("bind()");
    addrlen = sizeof(sin4);
    if (getsockname(sockfd, (struct sockaddr *)&sin4, &addrlen) < 0)
        exit_err("getsockname()");
    print_sin4("Bound   ", &sin4);

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

// Запускаємо програму з різними опціями та з різними привілеями.

```
$ ./a.out
Argument 0.0.0.0:0
Bound    0.0.0.0:34423
$ ./a.out -p 123
Argument 0.0.0.0:123
FAIL: bind(): Permission denied
$ ./a.out -p 1234
Argument 0.0.0.0:1234
Bound    0.0.0.0:1234
$ ./a.out -a 1.2.3.4
Argument 1.2.3.4:0
FAIL: bind(): Cannot assign requested address
$ ./a.out -a 127.0.0.1
```

```
Argument 127.0.0.1:0
Bound    127.0.0.1:34593
$ ./a.out -a 127.0.0.1 -p 1234
Argument 127.0.0.1:1234
Bound    127.0.0.1:1234
$ su root -c ./a.out -p 123
Argument 0.0.0.0:123
Bound    0.0.0.0:123
$ su root -c ./a.out -p 53
Argument 0.0.0.0:53
FAIL: bind(): Address already in use
$
```

Ця програма демонструє таке: неможливо призначити не wildcard IPv4 адресу сокету, якщо нема мережевого інтерфейсу в системі з такою мережевою адресою; неможливо призначити деякі номери портів сокету, якщо процес користувача немає відповідних привілеїв; неможливо призначити номер порту сокету, якщо цей номер порту вже зайнятий у комунікаційному домені (уточнення буде в наступній лекції).

Встановлення з’єднання

TCP потребує встановлення з’єднання між клієнтом та сервером (відправлення сегментів із прапорцями SYN), тому дії, які мають бути виконанні на стороні клієнта та на стороні сервера для встановлення TCP-з’єднання, мають бути різними, інакше не було б різниці між сервером та клієнтом у TCP-з’єднанні. Процес користувача може призначити адресу TCP-сокета, але це не є обов’язковим. Якщо мережеву адресу не призначено TCP-сокету або призначено wildcard адресу в системному виклику `bind()`, тоді ядро потім може призначити мережеву адресу сокету само. Значення цієї мережевої адреси буде залежати від ролі сторони TCP-з’єднання (клієнт або сервер). Якщо номер порту не призначено TCP-сокету або призначено нульовий номер порту в системному виклику `bind()`, тоді ядро призначить сокету вільний випадковий номер порту.

Системний виклик `listen()` позначає сокет, асоційований з вказаним дескриптором файлу таким, що буде приймати з’єднання, та обмежує кількість з’єднань, які очікуватимуть прийняття запиту на з’єднання в черзі. Цей

системний виклик має викликати програма, яка має роль сервера під час створення з’єднання.

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

int listen(int sockfd, int backlog);
```

Аргумент `backlog` надає підказку (hint) ядру щодо обмеження кількості завершених з’єднань, які очікуватимуть прийняття запиту на з’єднання в черзі. Але це значення не точне, ядро може його збільшити або зменшити, також ядро може враховувати незавершені з’єднання в цьому обмеженні. Від’ємне значення цього аргументу має такий самий сенс як нульове значення. Нульове значення цього аргументу не обов’язково призводить до заборони встановлювання нових з’єднань, тому, якщо процесу потрібно заборонити встановлення нових з’єднань, тоді треба закрити сокет (закрити всі дескриптори файлів, з якими асоційований цей сокет). Максимальне значення для аргументу `backlog` дорівнює значенню макросу `SOMAXCONN`, який визначений у `<sys/socket.h>`. Наявність обмеження кількості завершених з’єднань у черзі зумовлено обмеженим розміром пам’яті ядра й будь-якої іншої реалізації. Якщо сторона з’єднання ще не прийняла вже встановлені з’єднанні, тоді нема сенсу

встановлювати нові, вони також можуть бути не прийняті або прийняті невідомо коли.

Цей системний виклик можна застосовувати для щойно створеного TCP-сокета на стороні сервера. TCP-з’єднання створюються прозоро для процесу користувача, це виконує ядро, яке реалізує TCP. Якщо черга заповнена, тоді запит на створення нового TCP-з’єднань (був отриманий сегмент із прапорцем SYN) ігноруються, а ініціатор з’єднання відправлятиме повторні сегменти з прапорцем SYN, оскільки не отримає підтвердження на свій попередній запит створення нового з’єднання. Після чергового неотримання підтвердження з прапорцем SYN, ініціатор TCP-з’єднання завершіть спробу встановити з’єднання з помилкою `ETIMEDOUT`.

Сокет, для якого був застосований системний виклик `listen()`, може бути використаний тільки для створення (прийняття) з’єднань на стороні сервера. Його не можна використовувати для відправлення або отримання даних.

Системний виклик `accept()` забирає перше завершене з’єднання із черги або чекає на появу завершеного з’єднання для сокета, асоційованого з вказаним

дескриптором файлу. Цей системний виклик має викликати програма, яка має роль сервера під час створення з’єднання.

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

int accept(int sockfd, struct sockaddr *restrict addr,
        socklen_t *restrict addrlen);
```

Аргумент `sockfd` – це дескриптор файлу, з яким має бути асоційований сокет, для якого раніше був викликаний системний виклик `listen()` (сервер). Якщо аргумент `addr` не null покажчик, тоді в об’єкт, на який вказує цей покажчик, буде збережено адресу сокета іншої сторони з’єднання (клієнта). Аргументи `addr` та `addrlen` використовуються так само, як у системному виклику `getsockname()`. Якщо аргумент `addr` null покажчик, тоді аргумент `addrlen` також має бути null покажчиком. Якщо завершене з’єднання було забрано із черги, тоді цей системний виклик повертає ще не зайнятий дескриптор файлу з найменшим можливим номером, з яким асоційований новий сокет. Цей новий сокет можна використовувати для роботи зі з’єднанням, а сокет, асоційований з дескриптором файлу `sockfd`, можна продовжувати використовувати надалі в системному виклику `accept()`.

Цей системний виклик можна застосовувати для TCP-сокета, для якого був застосований системний виклик `listen()`, тобто на стороні сервера. Якщо сокету була призначена wildcard мережева адреса, тоді ядро призначить новому сокету, який повертає системний виклик `accept()`, мережеву адресу, яка була вказана ініціатором TCP-з'єднання (клієнтом), як мережева адреса отримувача (сервера). Зрозуміло, що це буде одна з мережевих адрес системи, інакше система б не отримала запит на встановлення TCP-з'єднання.

Системний виклик `accept()` для TCP є повільним, його виконання може бути перервано сигналом. Номер помилку буде `EINTR`.

Системний виклик `connect()` встановлює з'єднання із сервером, адреса якого визначається аргументом, використовуючи сокет, асоційований з вказаним дескриптором файлу. Цей системний виклик має викликати програма, яка має роль клієнта під час створення з'єднання.

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

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Аргумент `sockfd` – це дескриптор файлу, з яким має бути асоційований сокет, для якого раніше не був викликаний системний виклик `listen()` (клієнт). Об’єкт, на який вказує аргумент `addr`, визначає адресу сервера. Аргументи `addr` та `addrlen` використовуються так само, як у системному виклику `bind()`.

Цей системний виклик можна застосовувати для щойно створеного TCP-сокета на стороні клієнта. Якщо сокету була призначена wildcard мережева адреса, тоді ядро призначить цьому сокету мережеву адресу одного з локальних мережевих інтерфейсів системи, який можна використати для встановлення TCP-з’єднанням із сервером. Цей системний виклик чекає завершення встановлення TCP-з’єднання.

Системний виклик `connect()` для TCP є повільним, його виконання може бути перервано сигналом.

Системний виклик `getpeername()` повертає адресу іншої сторони з’єднання для сокета, асоційованого з вказаним дескриптором файлу.

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

int getpeername(int sockfd, struct sockaddr *restrict addr,
        socklen_t *restrict addrlen);
```

Аргументи `addr` та `addrlen` використовуються так само, як у системному виклику `getsockname()`.

Приклад TCP-сервера на C, який приймає запити на створення TCP-з’єднань на адресі `0.0.0.0:1234` та виводить адреси клієнтів, з якими були створені з’єднання.

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <errno.h>
#include <stdbool.h>
#include <unistd.h>

static bool
system_error(void)
{
    switch (errno) {
    case EBADF: case EFAULT: case ENOTSOCK: case EINVAL:
        return true;
```

```
    default:
        return false;
    }
}

int
main(void)
{
    struct sockaddr_in srv_sin4 = { 0 };
    struct sockaddr_in cln_sin4;
    in_addr_t srv_addr = INADDR_ANY;
    in_port_t srv_port = htons(1234);
    int backlog = 10;
    socklen_t addrlen;
    int listenfd, connfd;

    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 (bind(listenfd, (struct sockaddr *)&srv_sin4, sizeof(srv_sin4)) < 0)
        exit_err("bind()");
    if (listen(listenfd, backlog) < 0)
        exit_err("listen()");
    print_sin4("Accept at", &srv_sin4);
    for (;;) {
```

```
        addrlen = sizeof(cln_sin4);
        connfd = accept(listenfd, (struct sockaddr *)&cln_sin4, &addrlen);
        if (connfd < 0) {
            if (system_error())
                exit_err("accept()");
            warn_err("accept()");
            continue;
        }
        addrlen = sizeof(srv_sin4);
        if (getsockname(connfd, (struct sockaddr *)&srv_sin4, &addrlen) < 0) {
            if (system_error())
                exit_err("getsockname()");
            warn_err("getsockname()");
        } else {
            print_sin4("Server at", &srv_sin4);
            print_sin4("Client at", &cln_sin4);
        }
        // Data transfer.
        if (close(connfd) < 0)
            exit_err("close()");
    }

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

Приклад TCP-клієнта на C, який ініціює TCP-з’єднання із сервером на `127.0.0.1:1234` та виводить адресу свого сокета, якому явно не було призначено адресу, після встановлення з’єднання.

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

#ifndef INADDR_LOOPBACK
# define INADDR_LOOPBACK ((in_addr_t)0x7f000001)
#endif

int
main(void)
{
    struct sockaddr_in srv_sin4 = { 0 };
    struct sockaddr_in cln_sin4;
    in_addr_t srv_addr = htonl(INADDR_LOOPBACK);
    in_port_t srv_port = htons(1234);
    socklen_t addrlen;
    int sockfd;

    srv_sin4.sin_family = AF_INET;
    srv_sin4.sin_port = srv_port;
    srv_sin4.sin_addr.s_addr = srv_addr;
```

```
    print_sin4("Server at", &srv_sin4);
    sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sockfd < 0)
        exit_err("socket()");
    if (connect(sockfd, (struct sockaddr *)&srv_sin4, sizeof(srv_sin4)) < 0)
        exit_err("connect()");
    addrlen = sizeof(cln_sin4);
    if (getsockname(sockfd, (struct sockaddr *)&cln_sin4, &addrlen) < 0) {
        if (system_error())
            exit_err("getsockname()");
        warn_err("getsockname()");
    } else {
        print_sin4("Client at", &cln_sin4);
    }
    // Data transfer.
    if (close(sockfd) < 0)
        exit_err("close()");
}
```

Запускаємо сервер та клієнт у різних терміналах.

```
$ ./server
Accept at 0.0.0.0:1234
Server at 127.0.0.1:1234
Client at 127.0.0.1:33532
Server at 127.0.0.1:1234
Client at 127.0.0.1:33548
```

```
^C
$

$ ./client
Server at 127.0.0.1:1234
Client at 127.0.0.1:33532
$ ./client
Server at 127.0.0.1:1234
Client at 127.0.0.1:33548
$
```

Сервер завершує своє виконання через помилку або отриманий сигнал. Не всі помилки, з якими можуть завершитися системні виклики в цих програмах, є системними помилками, тому наведені програми перевіряють номери помилок для прийняття рішення чи треба завершувати своє виконання.

Зазвичай TCP-сервер призначає сокету, який буде використовуватися для прийняття з’єднань, номер порту, який відомий клієнтам. Але якщо в сервера є деякий механізм зареєструвати номер порту свого сокета, а в клієнта є деякий механізм отримати зареєстрований номер порту сокета сервера, тоді сервер може використовувати будь-який номер порту для свого сокета.

POSIX визначає такий API для TCP-з’єднань, що сервер може приймати запити від клієнтів на створення TCP-з’єднань на одних IP адресі та номері порту, потім цю IP адресу та номер порту успадковує TCP-сокет, який створює системний виклик `accept()`. Так само клієнт може використовувати одні й ті самі IP адресу та номер порту для створення TCP-з’єднань із різними серверами (сервера мають мати різні IP адреси та/або різні номери портів). Реалізація TCP відрізняє кілька одночасних TCP-з’єднань, оскільки одне з’єднання ідентифікується кортежем з чотирьох значень (IP адреса та номер порту сервера, IP адреса та номер порту клієнта).

Відправлення та отримання даних, частина 1

У цій темі надається інформація про системні виклики для введення-виведення, які можна застосовувати для TCP-сокета, для якого було створено TCP-з’єднання. Ці системні виклики також можна також застосовувати для звичайних файлів, неіменованих та іменованих каналів, але їхнє застосування для TCP-сокетів має деякі особливості. Це не всі можливі системні виклики, є ще інші (продовження в наступній лекції).

Системний виклик `read()` зчитує дані з об’єкта ядра, асоційованого з вказаним дескриптором файлу. Системний виклик `write()` записує дані в об’єкт ядра, асоційований з вказаним дескриптором файлу.

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Аргумент `fd` – це номер дескриптора файлу, системний виклик застосовується для об’єкта ядра, асоційованого з цим дескриптором файлу. Аргумент `buf` вказує на буфер даних в адресному просторі процесу, дані копіюються в або

копіюються із цього буфера відповідно. Аргумент `count` дорівнює кількості байт, яку треба прочитати або записати відповідно.

Системні виклики `readv()` та `writev()` подібні до `read()` та `write()` відповідно, але дають змогу вказати кілька буферів, замість одного (так звані «scatter read» та «gather write», або «scatter/gather input/output», scatter – розкидати, gather – збирати).

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

ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

Ці системні виклики доцільно використовувати у випадках, коли треба прочитати або записати дані одним викликом системного виклику. У `struct iov` є принаймні такі поля: `void *iov_base` (покажчик на буфер), `size_t iov_len` (кількість байтів у буфері). Кількість елементів у масиві, на який вказує аргумент `iov`, вказується в аргументі `iovcnt`, який має бути більше нуля. Буфери вказані в масиві, на який вказує аргумент `iov`, використовуються один за одним, поки вміст одного буфера не буде використано, системний виклик не використовує наступний буфер.

Системні виклики `read()` та `write()` повертають результат типу `ssize_t` (знаковий цілочисельний тип), для того, щоб позначити можливу помилку значенням `-1`. Тому значення аргументу `count` не має бути більшим, ніж значення `SSIZE_MAX`. Така сама вимога є для системних викликів `readv()` та `writev()`, але сума значень полів `iov_len` усіх об’єктів, на які вказує аргумент `iov`, не має бути більша, ніж значення `SSIZE_MAX`.

Функція `sysconf()` з аргументом `_SC_IOV_MAX` повертає максимальну кількість елементів дозволену в масиві, на який вказує `iov` у системних викликах `readv()` та `writev()`. Макрос `IOV_MAX` може бути визначений і макрос `_XOPEN_IOV_MAX` визначений у `<limits.h>`.

Будь-який системний виклик для введення-виведення з TCP-сокетом є повільним, як усталено. Ядро для кожного TCP-сокета має буфер даних на відправлення та буфер отриманих даних, розміри цих буферів обмежені через обмежений розмір пам’яті ядра. Наявність цих буферів дає змогу ядру асинхронно відправляти дані інший стороні TCP-з’єднання та асинхронно отримувати дані від іншої сторони TCP-з’єднання (асинхронно для процесу, який працює із цим з’єднанням). Буфер отриманих даних необхідний для

реалізації керування потоком у TCP. Отже, потік процесу, який виконує введення-виведення з TCP-сокетом, не завжди блокується у відповідних системних викликах.

Далі надається семантика системних викликів `read()` та `write()` для TCP-сокетів (семантика системних викликів `readv()` та `writev()` застосованих для TCP-сокетів така сама). Ця семантика не зовсім відповідає POSIX, але дасть змогу коректно працювати із цими системними викликами в реальних системах, які можуть реалізовувати цю семантику по різному, тобто не зовсім відповідно до POSIX.

Системні виклики `read()` та `write()` застосовані для TCP-сокета можуть завершитися через помилку, пов’язану з помилкою в з’єднанні. Варіанти номерів помилок такі: `ECONNRESET` (інша сторона надіслала сегмент із прапорцем RST, тобто розірвала з’єднання або повідомила, що такого з’єднання вже нема); `ETIMEDOUT` (відбувся тайм-аут у з’єднанні); `ECONNABORTED`, `ENETDOWN`, `ENETRESET`, `EHOSTUNREACH`, `ENETUNREACH` (різні варіанти помилок перерваного з’єднання через проблеми в мережі). Можуть бути інші номери помилок, навіть нестандартні. Які саме номери помилок повертаються залежить від реалізації. Якщо

системний виклик для введення-виведення для TCP-з’єднання повертає помилку, тоді програма користувача не має продовжувати працювати із цим з’єднанням, це з’єднання вже не буде працювати. Наступні виклики системних викликів `read()` та `write()` для цього TCP-з’єднання можуть завершуватися з помилкою з іншим номером помилки або `read()` може завершуватися без помилки та зчитувати нуль байтів. Як варіант, у разі помилки в TCP-з’єднанні програма користувача має закрити дескриптор файлу, з яким асоційований TCP-сокетом. Інтерпретувати номер помилки в загальному випадку нема сенсу, оскільки різні реалізації можуть повертати різні номери помилок. Помилка під час виконання поточного системного виклику `read()` або `write()` може бути помилкою в TCP-з’єднанні, яка була отримана під час виконання попереднього системного виклику `read()` (помилка під час відправленні сегментів з підтвердженнями) або `write()` (помилка під час відправлення сегментів з даними з буфера даних на відправлення).

Якщо буфер отриманих даних TCP-сокета порожній, тоді `read()` чекає на данні будь-якого розміру в буфері, тобто потік, який викликав цей системний виклик, блокується. Якщо дані в буфері є, тоді `read()` зчитує не більше даних, ніж було

вказано в його аргументі. Якщо виконання `read()` було перервано сигналом, тоді цей системний виклик завершується з помилкою з номером помилки `EINTR` якщо не було зчитано жодних даних або повертає розмір уже зчитаних даних. Тобто `read()` для TCP-з'єднання може прочитати менше даних, ніж вказано в його аргументі, навіть якщо його виконання не було перервано сигналом.

Якщо інша сторона TCP-з'єднання змінила режим передачі даних на симплексний у напрямку отримання даних (інша сторона вже не буде відправляти дані, був отриманий сегмент із прапорцем FIN) або завершила це з'єднання (був отриманий сегмент із прапорцем FIN, але з'єднання не було перервано іншою стороною, не був отриманий сегмент із прапорцем RST), тоді `read()` зчитує дані з буфера отриманих даних і потім зчитує нуль байтів. Наступний виклик `read()` також може прочитати нуль байтів або може завершитися з помилкою. Якщо програма користувача не відправляє дані в TCP-з'єднання, тоді програма не знає чи було це з'єднання завершено. Як варіант програма користувача для TCP-з'єднання може вказати ядру надсилати keep-alive сегменти (буде пояснено), але ядро може відправити перший keep-alive сегмент не раніше, ніж через дві години (рекомендоване

значення) неактивності цього з’єднання (коли нічого не відправляється в обох напрямках з’єднання). Тобто, якщо `read()` прочитав нуль байт, тоді отримати дані із цього TCP-з’єднання вже неможливо (відправлених даних уже не буде), але невідомо чи інша сторона завершила це з’єднання (немає можливості дізнатися чи режим передачі даних у цьому з’єднанні було змінено іншою стороною на симплексний або чи це з’єднання було завершено іншою стороною, для перевірки треба відправити дані).

Якщо програма користувача вказала в `read()` прочитати нуль байтів для TCP-сокета, тоді цей системний виклик зчитує нуль байтів або може завершитися з помилкою.

Якщо в буфері даних на відправлення TCP-сокета нема достатнього вільного місця, тоді `write()` чекає на вільне місце достатнього розміру, щоб скопіювати дані в буфер, тобто потік, який викликав цей системний виклик, блокується. Якщо виконання `write()` було перервано сигналом, тоді цей системний виклик завершується з помилкою з номером помилки `EINTR` якщо не було записано жодних даних або повертає розмір уже записаних даних. Реалізації можуть записати менше даних у TCP-сокет, ніж вказано в аргументі системного

виклику `write()`, навіть якщо виконання системного виклику не було перервано сигналом. Зазвичай це відбувається через закриття або переривання TCP-з’єднання іншою стороною.

Якщо програма користувача змінила режим передачі даних на симплексний у напрямку отримання даних (закрила свій TCP-сокет на запис) або якщо це з’єднання було завершено або перервано іншою стороною, тоді виклик системного виклику `write()` призведе або може призвести до відправлення процесу синхронного сигналу `SIGPIPE`. Якщо цей сигнал ігнорується, опрацьовується процесом або блокується потоком, який отримує синхронний сигнал `SIGPIPE`, тоді виконання `write()` завершиться з помилкою з номером помилки `EPIPE`. Усталена поведінка в разі отримання сигналу `SIGPIPE` – це завершення виконання процесу.

Успішне виконання системного виклику `write()`, застосованого для TCP-сокета, не значить, що дані були успішно доставлені системі іншої сторони TCP-з’єднання. Помилка під час виконання системного виклику `write()` не значить, що раніше відправлені дані не були успішно доставлені системі іншої сторони

TCP-з’єднання. POSIX не визначає поведінку функції `write()`, застосованої для TCP-сокетів, якщо програма вказує нуль байтів в аргументі.

POSIX не визначає поведінку паралельних викликів функцій введення-виведення для дескриптора файлу, з яким асоційований сокет, але для TCP-сокета це немає сенсу, оскільки TCP не розмежовує відправлені дані. Паралельні виклики функцій введення-виведення можуть відбуватися в потоках одного процесу для одного дескриптора файлу або в потоках одного процесу або в різних процесах для дубльованого дескриптора файлу.

Зазвичай у книгах наводять реалізації функцій `readn()` та `writen()`, які спрощують використання системних викликів `read()` та `write()` відповідно, застосованих для TCP-сокета. Функції `readn()` та `wriiten()` можна застосовувати для TCP-сокета, для якого було встановлено ознаку не блокуватися в системному виклику (буде пояснено), а функцію `readn()` можна застосовувати для TCP-сокета, для якого не було встановлено ознаку не блокуватися (усталена поведінка). Наведемо реалізацію цих функцій на C, які дещо відрізняються від реалізацій, які зазвичай наводять у книгах: якщо в аргументі вказано нуль байтів, тоді відповідний системний виклик завжди викликається; якщо `write()` записав нуль

байтів, тоді функція `writen()` завершується без помилки (передбачається, що реалізації системного виклику `write()`, які повертали нульове значення через певну помилку вже не використовуються, навіть якщо вони використовуються, тоді функція `writen()` усе одно працюватиме коректно, а рішення про нуль записаних батів прийматиме програма, яка викликає функцію `writen()`); функція завершує своє виконання, якщо системний виклик було перервано сигналом (опрацювання сигналів не є темою цих лекцій).

```
#include <errno.h>
#include <unistd.h>

ssize_t
readn(int fd, void *buf, size_t n)
{
    size_t total;
    ssize_t nread;

    total = 0;
    do {
        nread = read(fd, (char *)buf + total, n - total);
        if (nread < 0) {
            if (errno == EINTR)
                break;
            return -1;
```

```
        }
        if (nread == 0)
            break;
        total += (size_t)nread;
    } while (total != n);
    return (ssize_t)total;
}

ssize_t
writen(int fd, const void *buf, size_t n)
{
    size_t total;
    ssize_t nwritten;

    total = 0;
    do {
        nwritten = write(fd, (char *)buf + total, n - total);
        if (nwritten < 0) {
            if (errno == EINTR)
                break;
            return -1;
        }
        if (nwritten == 0)
            break;
        total += (size_t)nwritten;
    } while (total != n);
    return (ssize_t)total;
}
```

Приклад функції на C для ігнорування сигналу `SIGPIPE`.

```
#include <signal.h>
#include <stddef.h>

void
sigpipe_ignore(void)
{
    struct sigaction sigact;

    sigact.sa_handler = SIG_IGN;
    sigact.sa_flags = 0;
    sigemptyset(&sigact.sa_mask);
    if (sigaction(SIGPIPE, &sigact, NULL) < 0)
        exit_err("sigpipe_ignore(): sigaction()");
}
```

Завершення з’єднання, частина 1

Завершення TCP-з’єднання значить, що сторона з’єднання закрила відповідний сокет та відправила іншій стороні з’єднання сегмент з прапорцем FIN. Це можна виконати застосувавши системний виклик `close()` для дескриптора файлу, з яким асоційований відповідний сокет, або завершивши виконання процесу (ядро закриє всі дескриптори файлів процесу), якщо цій сокет більше не асоційований з іншими дескрипторами файлів.

Завершення TCP-з’єднання як усталено відбувається асинхронно для потоку, який застосував системний виклик `close()` для відповідного сокета, або в разі завершення процесу. Тобто системний виклик `close()` завершується та завершення процесу виконується не чекаючи на результат завершення відповідного TCP-з’єднання.

Стан завершеного TCP-з’єднання буде визначатися черговістю завершення цього з’єднання в кожній стороні з’єднання. Про це йдеться в наступній лекції.

Багатопроцесний ітеративний сервер

Найпростіший варіант побудови сервера, який працює з клієнтами через TCP-з’єднання – це ітеративний сервер, який опрацьовує запити одного клієнта повністю, перед тим, як почати опрацьовувати запити наступного клієнта. Приклад коду такого сервера для TCP-з’єднань був наведений раніше в темі про встановлення з’єднання.

Невідомо скільки часу потрібно для комунікації з клієнтом та опрацювання запитів клієнта. Зазвичай сервер має працювати з кількома клієнтами, тому можна зробити багатопроцесний ітеративний сервер. Ідея цього сервера полягає в тому, що процес сервера заздалегідь створює нові процеси (pre-fork) для опрацювання запитів клієнтів, кожний дочірній процес є ітеративним сервером.

Системний виклик `fork()` створює новий процес. Вміст пам’яті нового процесу є копією вмісту пам’яті оригінального процесу. Дескриптори файлів у новому процесу дублюються з дескрипторів файлів оригінального процесу. Обидва процеси продовжують своє виконання повертаючись з виклику `fork()`.

Системний виклик `fork()` повертає `-1` у разі помилки, `0` у новому процесі, PID нового процесу в оригінальному процесі.

Сервер створює TCP-сокет, який прийматиме з'єднання, та створює кілька процесів використовуючи системний виклик `fork()`. Нові процеси отримують дубльовані дескриптори файлів оригінального процесу, у такий спосіб вони отримують дескриптор файлу, з яким асоційований TCP-сокет, який прийматиме з'єднання. Дочірні процеси викликають системний виклик `accept()` для цього дескриптора файлу (можливо одночасно), а ядро вибере який із цих системних викликів отримує новий дескриптор файлу, з яким буде асоційований сокет для першого завершеного TCP-з'єднання з черги. Отримавши дескриптор файлу дочірній процес опрацьовує запити клієнта повністю, перед тим як почати опрацьовувати запити наступного клієнта. Якщо запити від клієнтів є незалежними, тоді не треба реалізовувати взаємодію між усіма процесами, які реалізуються сервер.

Приклад реалізації такого сервера на C++. У цій програмі не реалізовано підтримування зупинки всіх процесів сервера, тому що це приклад реалізації і сигнали та управління процесами не є темами цих лекцій. Для завершення всіх

процесів сервера, достатньо натиснути Ctrl-C у терміналі, де запускається сервер. Дочірній процес отримує копію стандартного потоку виведення та копію вмісту його буфера. Якщо цей буфер непорожній, тоді дані, які зберігаються в ньому, можуть бути виведені два рази (в оригінальному та в дочірньому процесі). Щоб вирішити цю проблему, треба заборонити буферизацію для стандартного потоку виведення або вивести вміст його буфера в оригінальному процесі перед викликом `fork()`.

```
#include <cstdlib>
#include <iostream>
#include <vector>

#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>

std::ostream&
operator<<(std::ostream& os, const struct sockaddr_in& sin4)
{
    char repr[INET_ADDRSTRLEN];

    if (getnameinfo((struct sockaddr*)&sin4, sizeof(sin4), repr, sizeof(repr),
```

```
        NULL, 0, NI_NUMERICHOST) != 0
    ) {
        repr[0] = '?';
        repr[1] = '\0';
    }
    return os << repr << ':' << htons(sin4.sin_port);
}

class Server {
private:
    int listenfd;
    pid_t pid;

    void handle_client(int connfd) const
    {
        ssize_t nread;
        unsigned char ch;

        nread = read(connfd, &ch, 1);
        if (nread < 0) {
            if (system_error())
                exit_err("Server::handle_client(): read()");
            warn_err("Server::handle_client(): read()");
        } else if (nread == 0) {
            warn_errx("Process ", this->pid, " TCP connection is half-closed");
        } else {
            std::cout << "Process " << this->pid << " read byte " <<
                static_cast<unsigned int>(ch) << " from TCP connection\n";
```

```
        }
    }

public:
    Server(int listenfd)
        : listenfd(listenfd), pid(getpid())
    {
    }
    ~Server()
    {
        if (close(this->listenfd) < 0)
            exit_err("Server::~Server(): close()");
    }

    void run()
    {
        struct sockaddr_in srv_sin4, cln_sin4;
        socklen_t addrlen;
        int connfd;

        for (;;) {
            addrlen = sizeof(cln_sin4);
            connfd = accept(this->listenfd, (struct sockaddr*)&cln_sin4,
                &addrlen);
            if (connfd < 0) {
                if (system_error())
                    exit_err("Server::run(): accept()");
                warn_err("Server::run(): accept()");
```

```
                continue;
            }
            addrlen = sizeof(srv_sin4);
            if (getsockname(connfd, (struct sockaddr*)&srv_sin4,
                &addrlen) < 0
            ) {
                if (system_error())
                    exit_err("Server::run(): getsockname()");
                warn_err("Server::run(): getsockname()");
            } else {
                std::cout << "Process " << this->pid << ", server at " <<
                    srv_sin4 << '\n';
            }
            std::cout << "Process " << this->pid << ", client at " <<
                cln_sin4 << "\n";
            this->handle_client(connfd);
            if (close(connfd) < 0)
                exit_err("Server::run(): close()");
        }
    }
};

int
main()
{
    struct sockaddr_in srv_sin4{};
    std::vector<pid_t> pid_vec;
    in_addr_t srv_addr = INADDR_ANY;
```

```
in_port_t srv_port = htons(1234);
int backlog = 10;
unsigned int server_num = 3;
int listenfd;

try {
    pid_vec.reserve(server_num);
} catch (...) {
    exit_err("std::vector::reserve()");
}

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 = srv_addr;
if (bind(listenfd, (struct sockaddr*)&srv_sin4, sizeof(srv_sin4)) < 0)
    exit_err("bind()");
if (listen(listenfd, backlog) < 0)
    exit_err("listen()");
std::cout << "Accept at " << srv_sin4 << '\n';
std::cout.flush();

for (unsigned int i = 0; i < server_num; ++i) {
    pid_t pid = fork();
    if (pid < 0) {
        warn_err("fork()");
```

```
        } else if (pid > 0) {
            pid_vec.push_back(pid);
        } else {
            Server(listenfd).run();
            return EXIT_SUCCESS;
        }
    }
    if (close(listenfd) < 0)
        exit_err("close()");
    if (pid_vec.size() == 0)
        exit_errx("cannot create any new process");
    pause();
}
```

Багатопроцесний паралельний сервер

Ідея багатопроцесного паралельного сервера полягає в тому, що сервер створює нові процеси для опрацювання запитів нових клієнтів. Сервер приймає TCP-з'єднання та створює новий процес використовуючи системний виклик `fork()`. Новий процес отримує дубльовані дескриптори файлів оригінально процесу, так він отримує дескриптор файлу, з яким асоційований сокет встановленого TCP-з'єднання. Далі оригінальний процес має закрити дескриптор файлу, з яким асоційований сокет TCP-з'єднання, а дочірній процес має закрити дескриптор файлу, з яким асоційований сокет, який використовується для прийняття TCP-з'єднань. Дочірній процес опрацьовує запити від клієнта та завершує своє виконання.

Сервер, який створює нові процеси для опрацювання запитів нових клієнтів, повинен мати обмеження на максимальну кількість дочірніх процесів. Якщо постійно приймати нові TCP-з'єднання та створювати нові процеси, тоді кількість створених процесів може бути занадто великою для виконання на конкретному комп'ютері. Нові процеси використовують пам'ять, дескриптори файлів та їх треба виконувати (використовується процесорний час).

Пам’ять нового процесу є копією пам’яті оригінального процесу. Зазвичай ядро налаштовує адресні простори оригінального та нового процесів так, щоб вони використовували за можливості спільну пам’ять, відтерміновуючи фізичне копіювання пам’яті (так зване *ледаче копіювання*). Для реалізації цього ядро послуговується властивостями віртуальної пам’яті. Регіони адресної карти оригінального процесу, які відображаються без права модифікації (це коди та read-only дані з файлів програми) будуть відображені з такими самими правами доступу в новому процесі в ту саму фізичну пам’ять (це буде спільна пам’ять для двох процесів). Регіони адресної карти оригінального процесу, які відображаються з правом модифікації (це дані та стек програми, це *анонімна пам’ять*, її вміст процес створює під час свого виконання) будуть відображені в обох процесах із правами доступу read-only (це буде спільна пам’ять для двох процесів). Під час модифікації даних у цій пам’яті (ця модифікація не вдасться через неможливість модифікувати вміст пам’яті, яка відображена з правами доступу read-only, буде згенерована відповідна виключна ситуація) ядро виконає фізичне копіювання вмісту відповідної фізичної сторінки у вільну фізичну сторінку, змінить відображення та права доступу на read-write

відповідної віртуальної сторінки. Технологія такого відтермінованого копіювання вмісту пам’яті називається *copy-on-write* (COW).

Якщо сервер створив певну кількість дочірніх процесів для опрацювання TCP-з’єднань, тоді сервер має почекати, поки якийсь із цих процесів не завершить своє виконання використовуючи системний виклик `waitpid()`, і тільки потім приймати нові TCP-з’єднання.

Може здатися, що створення нового процесу для опрацювання запитів від одного клієнта є рішенням, яке взагалі немає сенсу. Насправді опрацювання запитів від одного клієнта в окремому процесі підвищує безпеку сервера. Якщо в коді сервера є помилка і якщо цю помилку може використати клієнт, тоді клієнт може скомпрометувати тільки один процес, який безпосередньо опрацьовує запити від цього клієнта. Ще більше підвищити безпеку сервера можна так. Дочірній процес викликає системний виклик `execve()` та змінює свій образ (програму користувача, яка буде виконуватися в контексті процесу). Після зміни образу процесу дескриптори файлів залишаються відкритими як усталено. Номер дескриптора файлу, з яким асоційований сокет TCP-з’єднання, можна передати в аргументі командного рядка новій програмі. Якщо

змінити образ процесу, тоді в пам’яті процесу взагалі не буде жодного вмісту пам’яті оригінального процесу (сервера, який приймає TCP-з’єднання).

Приклад реалізації такого сервера на C.

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static void
handle_client(int connfd)
{
    ssize_t nread;
    unsigned char ch;

    nread = read(connfd, &ch, 1);
    if (nread < 0) {
        if (system_error())
            exit_err("read()");
        warn_err("read()");
    } else if (nread == 0) {
        warn_errx("TCP connection is half-closed");
```

```
    } else {
        printf("Process %jd: read byte %hhu from TCP connection\n",
            (intmax_t)getpid(), ch);
    }
}

int
main(void)
{
    struct sockaddr_in srv_sin4 = { 0 };
    struct sockaddr_in cln_sin4;
    in_addr_t srv_addr = INADDR_ANY;
    in_port_t srv_port = htons(1234);
    int backlog = 10;
    pid_t pid;
    socklen_t addrlen;
    int listenfd, connfd;

    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 = srv_addr;
    if (bind(listenfd, (struct sockaddr *)&srv_sin4, sizeof(srv_sin4)) < 0)
        exit_err("bind()");
    if (listen(listenfd, backlog) < 0)
        exit_err("listen()");
```

```
print_sin4("Accept at", &srv_sin4);

for (;;) {
    addrlen = sizeof(cln_sin4);
    connfd = accept(listenfd, (struct sockaddr *)&cln_sin4, &addrlen);
    if (connfd < 0) {
        if (system_error())
            exit_err("accept()");
        warn_err("accept()");
        continue;
    }
    addrlen = sizeof(srv_sin4);
    if (getsockname(connfd, (struct sockaddr *)&srv_sin4, &addrlen) < 0) {
        if (system_error())
            exit_err("getsockname()");
        warn_err("getsockname()");
    } else {
        print_sin4("Server at", &srv_sin4);
    }
    print_sin4("Client at", &cln_sin4);
    for (;;) {
        pid = waitpid(-1, NULL, WNOHANG);
        if (pid < 0) {
            if (errno != ECHILD)
                exit_err("waitpid()");
            break;
        }
        if (pid == 0)
```

```
                break;
            printf("Process %jd exited\n", (intmax_t)pid);
        }
        fflush(stdout);
        pid = fork();
        if (pid < 0) {
            warn_err("fork()");
        } else if (pid == 0) {
            if (close(listenfd) < 0)
                exit_err("close()");
            handle_client(connfd);
            if (close(connfd) < 0)
                exit_err("close()");
            return EXIT_SUCCESS;
        }
        if (close(connfd) < 0)
            exit_err("close()");
    }
    if (close(listenfd) < 0)
        exit_err("close()");
}
```

Програма чекає на завершення дочірніх процесів за допомогою системного виклику `waitpid()`. Вказані аргументи значать таке: перший аргумент `-1` вказує, чекати на завершення будь-якого дочірнього процесу; null покажчик у другому аргументі вказує не повертати інформацію про статус завершеного дочірнього

процесу (ця інформація в цій програмі не потрібна); третій аргумент `WNOHANG` вказує не блокуватися в системному виклику, якщо не буде жодного завершеного дочірнього процесу. Для вказаних аргументів системний виклик `waitpid()` повертає: `-1` у разі помилки, номер помилки `ECHILD` позначає відсутність дочірніх процесів; `0` якщо жодний дочірній процес не було завершено; PID завершеного дочірнього процесу.