Лекція 3

Дескриптори файлів та сокети

Дескриптори файлів

Назва *дескриптор файлу* може позначати різні речі. Іноді ця назва позначає щось, що взагалі не має відношення до файлів. Далі надано варіант кількох програмних абстракцій, кожну з яких називають *дескриптор файлу* або чимось схожим, та зв’язків між ними. Імена структур та полів у цих структурах умовні та відповідають наведеному далі прикладу.

Кожний процес користувача має *таблицю дескрипторів файлів* (*file descriptor table*) в ядрі. Для того, щоб процес міг працювати з файлом, ядро має знайти вільний елемент у його таблиці дескрипторів файлів, встановити покажчик у цьому елементі на *дескриптор відкритого файлу* (*open file description*, переклад українською неточний) та повернути індекс цього елемента процесу. Цей індекс називають *дескриптор файлу*. Програма користувача використовує ці дескриптори файлів в аргументах системних викликів. Зазвичай ядро шукає вільний елемент у таблиці дескрипторів файлів із найменшим можливим індексом або процес може запитати конкретний номер дескриптора файлу в деяких системних викликах.

Елементи таблиці дескрипторів файлів представляють не тільки відкриті файли. Ці дескриптори файлів також представляють сокети, кінці неіменованих каналів та інші програмні абстракції, які доступні в API між програмою користувача та ядром. Кожний елемент таблиці дескрипторів файлів також містить прапорці, які впливають на семантику роботи з дескриптором файлу.

Таблиця дескрипторів файлів не обов’язково має бути масивом в ядрі, це може бути будь-яка структура даних, яка дає змогу ядру ефективно знаходити елемент за його індексом, знаходити вільний елемент із найменшим індексом та представляти елементи з непослідовними індексами в пам’яті.

Ядро, під час надання процесу користувача дескриптора файлу з його таблиці дескрипторів файлів, створює об’єкт типу `struct file` для визначення стану відкритого файлу або іншого об’єкта, асоційованого з дескриптором файлу в API. Ця структура має такі поля: `f_type` визначає тип об’єкта, який представляє ця структура; `f_flags` визначає прапорці відкритого файлу; `f_offset` визначає зміщення для наступного читання або запису для файлу; `f_vnode` вказує на об’єкт ядра, який представляє всю інформацію про файл. Ця структура є *дескриптор відкритого файлу*. Об’єкт цього типу асоційований з одним або кількома

дескрипторами файлів, навіть із різних таблиць дескрипторів файлів. Тобто кілька процесів користувача можуть сумісно працювати з одним і тим самим дескриптором відкритого файлу. Кілька дескрипторів файлів посилаються на один дескриптор відкритого файлу, якщо вони дубльовані (пояснено далі).

Ядро, під час першого звернення до файлу, створює об’єкт типу `struct vnode`. Кілька об’єктів в ядрі (наприклад, поля `f_vnode` кількох об’єктів типу `struct file`) можуть посилатися на один об’єкт цього типу. У цій структурі є покажчик `v_data`, який вказує на дескриптор файлу в драйвері ФС. Ця структура є *дескриптор файлу*, який абстрагує файл для всіх підсистем ядра.

Кожний драйвер ФС має *дескриптор файлу* для кожного файлу (`struct fs_inode`), тільки драйвер використовує цей дескриптор файлу. Драйвер ФС бере частину вмісту цієї структури зі структур даних ФС на носії інформації. Цей дескриптор файлу асоційований тільки з одним об’єктом типу `struct vnode` з боку коду ядра, але не обов’язково з боку коду драйвера ФС.

Структури даних ФС на носії інформації мають *дескриптор файлу* для кожного файлу (`struct inode`). Цей дескриптор файлу не обов’язково має містити всю

інформацію про файл на носії інформації, але її має бути достатньо для отримання всієї потрібної інформації.

Програма користувача може реалізувати власну програмну абстракцію для роботи з файлом, яка може мати назву *дескриптор файлу*.

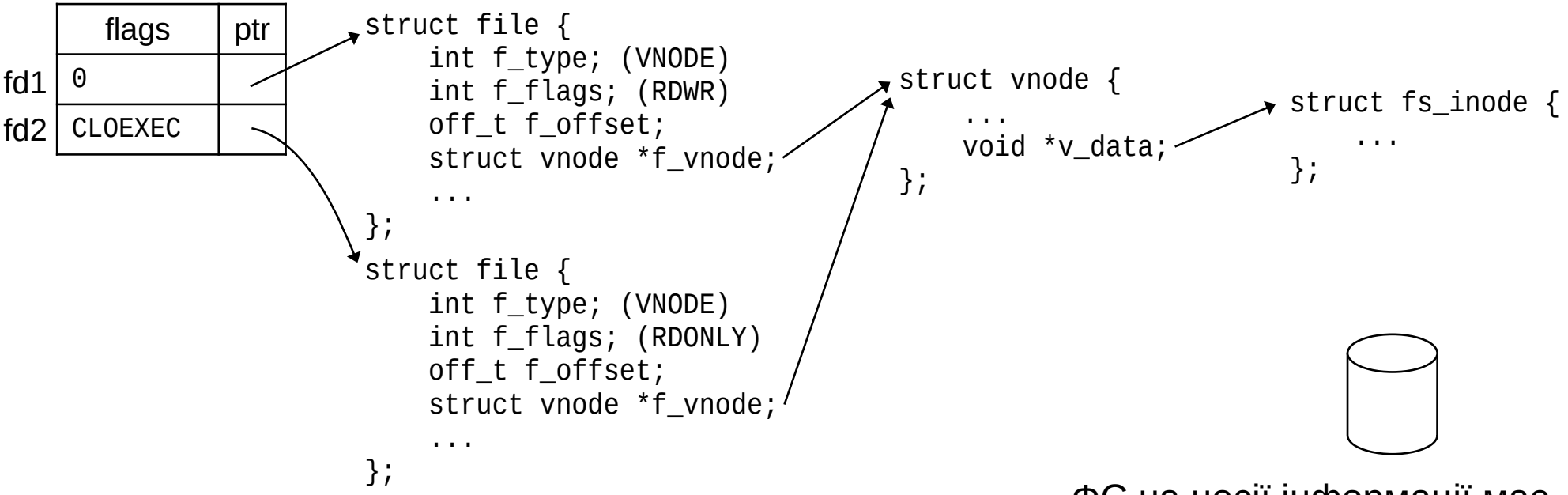
Назва *open file description* використана в POSIX для позначення будь-якої програмної абстракції, яка доступна в API через *file descriptor* (невід'ємне число). Якщо для представлення різних програмних абстракцій ядра використані об'єкти різних типів (не тип `struct file`, як у прикладі), то в таблиці дескрипторів файлів має бути вказано тип об'єкта, на який посилається дескриптор файлу. Це деталі реалізації, які не впливають на API. POSIX взагалі не визначає як та де щось має бути реалізовано.

Імовірніше в назвах *дескриптор файлу* та *дескриптор відкритого файлу* є слово «файл», оскільки спочатку API між ядром та процесом користувача давав змогу працювати тільки з файлами, потім він був розширений для роботи з об'єктами ядра інших типів. Використання дескрипторів файлів для ідентифікації об'єктів ядра різних типів дає змогу програмам користувача більш

менш узагальнити введення-виведення з об’єктами ядра різних типів. Наприклад, деякі системні виклики для введення-виведення можна майже однаково застосовувати до файлів, кінців неіменованих каналів та сокетів.

В `<unistd.h>` визначені макроси `STDIN_FILENO` (значення `0`), `STDOUT_FILENO` (значення `1`), `STDERR_FILENO` (значення `2`), значення яких – це дескриптори файлів, які відповідають standard input, standard output та standard error відповідно.

Функція `sysconf()` з аргументом `_SC_OPEN_MAX` повертає значення на одиницю більше, ніж максимальний можливий номер дескриптора файлу. У `<limits.h>` може бути визначено макрос `OPEN_MAX` та визначено макрос `_POSIX_OPEN_MAX` (тут та далі сенс таких макросів такий самий, як у відповідних макросів, які були надані в лекції 2).



ФС на носії інформації має дескриптор файлу `struct inode`

```
int fd1 = open("file.txt", O_RDWR);
int fd2 = open("file.txt", O_RDONLY|O_CLOEXEC);
pause();
```

Цей рисунок показує об’єкти, які створює ядро в разі успішного виконання системних викликів `open()`, які були викликані цією програмою за умови, що ім’я файлу **file.txt** не було забрано, а потім створено знов у часовому вікні між двома викликами системного виклику `open()`.

Сокети

Сокет (socket) – це об’єкт ядра, який можна використовувати для різноманітних дій, що мають відношення до мережевого програмування. Сокет асоційований принаймні з одним дескриптором файлу.

Системний виклик `socket()` створює сокет у вказаному комунікаційному домені, вказаного типу та протоколу.

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

int socket(int domain, int type, int protocol);
```

Аргумент `domain` визначає комунікаційний домен сокета, аргумент `type` визначає тип сокета, аргумент `protocol` визначає протокол у комунікаційному домені. Якщо значення аргументу `protocol` дорівнює нулю, ядро вибере типовий протокол для вказаного комунікаційного домену для вказаного типу сокета. Якщо новий сокет було створено, то цей системний виклик поверне номер дескриптора файлу з найменшим можливим номером, з яким буде асоційований створений сокет.

Аргумент `domain` має мати одне зі значень `AF_*`, але не `AF_UNSPEC`. Деякі реалізації мають макроси `PF_*`. Літери `PF` означають protocol family (сімейство протоколів). Ідея була в тому, щоб одне сімейство протоколів могло підтримувати декілька сімейств адрес. Відповідно макроси `PF_*` мали використовуватися під час створення сокетів, а макроси `AF_*` мали використовуватися в структурах адрес сокетів. Насправді це не було реалізовано, тому макроси `PF_*` можуть бути визначені в `<sys/socket.h>` та мають такі самі значення як відповідні за ім'ям макроси `AF_*`. У `struct addrinfo` є тільки поле зі значенням сімейства адрес, хоча мало б бути також поле зі значенням сімейства протоколів. SUSv4 визначає префікс `PF_` зарезервованим, але не визначає жодного макросу із цим префіксом. У деяких вихідних кодах ядер та програм користувача буває плутанина з використанням цих макросів, назв відповідних полів та/або в коментарях, які мають відношення до використання цих макросів.

Системний виклик `close()` закриває дескриптор файлу.

```
#include <unistd.h>

int close(int fd);
```

Деякі системи мають нестандартні системні виклики `closefrom()` та/або `close_range()`, які закривають усі дескриптори файлів, починаючи з вказаного або у вказаному діапазоні. Зрозуміло, що ядро використовує оптимальний спосіб для закриття відкритих дескрипторів файлів у вказаному діапазоні.

Програма може мати багато відкритих дескрипторів файлів у певному діапазоні, які їй треба закрити. Замість виклику системного виклику `close()` у циклі для кожного відкритого дескриптора файлу (кожний раз режими виконання буде змінюватися), можна викликати один із цих системних викликів.

Програма може не знати, які дескриптори файлів відкриті, і їй треба закрити всі відкриті дескриптори файлів, починаючи з вказаного. Програмі треба викликати `sysconf(_SC_OPEN_MAX)`, щоб отримати максимальний номер дескриптора файлу. Але ця функція може повернути `-1`, доведеться вгадувати максимальний номер дескриптора файлу або закривати всі можливі дескриптори файлів починаючи з вказаного. Замість виклику системного виклику `close()` для всіх можливих відкритих та не відкритих дескрипторів файлів, починаючи із вказаного, можна викликати один із цих системних викликів.

Якщо системний виклик `close()` застосовано до дескриптора файлу, з яким асоційований сокет, і більше нема дескрипторів файлів, які посилаються на цей сокет, то успішне виконання цього системного виклику може не означати успішне відправлення даних, які раніше були відправлені в цей сокет (лекція 4 та лекція 9).

Атрибути об’єкта ядра, асоційованого з дескриптором файлу, можна отримати за допомогою системного виклику `fstat()`. Мабуть, тільки поле `st_mode` у `struct stat` може бути в пригоді в програмі, яка відповідає POSIX. Значення цього поля з маскою `S_IFMT` кодує тип об’єкта ядра. Значення `S_IFSOCK` позначає сокет, також для значення цього поля можна застосовувати макрос `S_ISSOCK()`.

Приклад програми на C, яка створює TCP-сокет у комунікаційному домені Internet IPv4 та перевіряє чи повернутий дескриптор файлу посилається на сокет.

```c
#include <sys/socket.h>
#include <sys/stat.h>
#include <stdio.h>
#include <unistd.h>
```

```
int
main(void)
{
    struct stat statbuf;
    int fd;

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

    if (fstat(fd, &statbuf) < 0)
        exit_err("fstat()");
    printf("fd %d refers to ", fd);
    if (S_ISSOCK(statbuf.st_mode))
        printf("a socket\n");
    else
        printf("some another kernel object\n");

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

// Запускаємо програму.

```
$ ./a.out
fd 3 refers to a socket
$
```

Дублювання дескриптора файлу

Один дескриптор відкритого файлу асоційований принаймні з одним дескриптором файлу, але він може бути асоційований із кількома дескрипторами файлів, навіть різних процесів. Процес користувача може явно запросити ядро дублювати дескриптор файлу. Після успішного дублювання обидва дескриптори файлів посилатимуться на той самий дескриптор відкритого файлу. Серед дескрипторів файлів, які посилаються на один дескриптор відкритого файлу, нема першого чи головного дескриптора файлу, процес користувача навіть не може визначити який дескриптор файлу з них є оригінальний, а який є дубльований.

Процесу користувача може бути потрібно явно дублювати дескриптор файлу для того, щоб він мав або не мав певний номер. Зазвичай системні виклики, які повертають дескриптор файлу, повертають дескриптор файлу з найменшим можливим номером (це визначено в POSIX). Процес користувача має напередвизначені дескриптори файлів `STDIN_FILENO`, `STDOUT_FILENO` та `STDERR_FILENO`, (визначені в `<unistd.h>`) які дорівнюють першим трьом номерам дескрипторів файлів. Зазвичай процес користувача дублює дескриптори файлів із цими

номерами або в дескриптори файлів із цими номерами. Процес користувача може мати якийсь (або всі) із цих дескрипторів файлів закритим, відповідно системний виклик `open()` поверне один із цих дескрипторів файлів. Програма може не працювати зі standard input, standard output або standard error, але це може робити якась функція бібліотеки. Це не є правильним, але іноді функції бібліотек можуть щось виводити в standard error у разі помилки.

Системні виклики `dup()` та `dup2()` дублюють вказаний дескриптор файлу.

```
#include <unistd.h>

int dup(int fd);
int dup2(int fd1, int fd2);
```

Системний виклик `dup()` дублює дескриптор файлу `fd` у вільний дескриптор файлу з найменшим можливим номером. У разі успішного виконання цей номер буде повернуто та новий дескриптор файлу посилатиметься на дескриптор відкритого файлу, на який посилається дескриптор файлу `fd`.

Системний виклик `dup2()` дублює дескриптор файлу `fd1` у дескриптор файлу `fd2`. Якщо `fd1` дорівнює `fd2`, то дескриптор файлу `fd2` не буде закрито, а прапорці

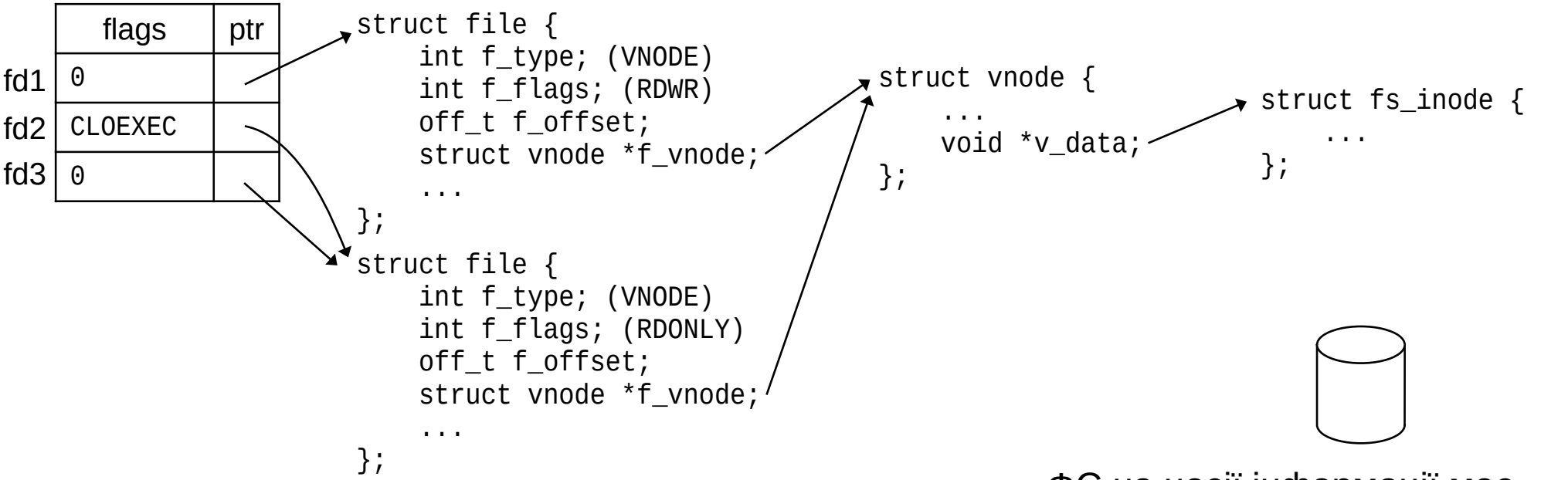
дескриптора файлу `fd2` не буде змінено (лекція 6). Якщо `fd1` не дорівнює `fd2`, то дескриптор файлу `fd2` буде закрито перед дублюванням, а прапорці дескриптора файлу `fd2` будуть скидані. Якщо не вдасться закрити дескриптор файлу `fd2`, то він буде посилатися на той самий дескриптор відкритого файлу. Якщо `fd1` є некоректний дескриптор файлу, то дескриптор файлу `fd2` не буде закрито. У разі успішного виконання значення `fd2` буде повернуто та дескриптор файлу `fd2` посилатиметься на дескриптор відкритого файлу, на який посилається дескриптор файлу `fd1`.

Системний виклик `fcntl()` застосовує вказану команду до вказаного дескриптора файлу або до об’єкта ядра, асоційованого з вказаним дескриптором файлу.

```c
#include <fcntl.h>

int fcntl(int fd, int cmd, ...);
```

Системний виклик `fcntl()` з командою `F_DUPFD` (перший аргумент) дублює дескриптор файлу, вказаний у другому аргументі. Дескриптор файлу, номер якого вказаний у другому аргументі, буде дубльовано у вільний дескриптор файлу з номером, який більший, ніж або дорівнює значенню третього

аргументу, яке має мати тип `int`. У разі успішного виконання номер дубльованого дескриптора файлу буде повернуто та цей дескриптор файлу посилатиметься на дескриптор відкритого файлу, на який посилається дескриптор файлу, вказаний у другому аргументі. Відповідно виклик `dup(fd)` є еквівалентним виклику `fcntl(fd, F_DUPFD, 0)`.




```
int fd1 = open("file.txt", O_RDWR);
int fd2 = open("file.txt", O_RDONLY|O_CLOEXEC);
int fd3 = dup(fd2);
pause();
```

ФС на носії інформації має дескриптор файлу `struct inode`

Цей рисунок показує об’єкти, які створює ядро в разі успішного виконання системних викликів `open()` та `dup()`, які були викликані цією програмою за умови, що ім’я файлу **file.txt** не було забрано, а потім створено знов у часовому вікні між двома викликами системного виклику `open()`.

Ядро дублює дескриптори файлів під час створення нового процесу системним викликом `fork()` (пояснено далі) та за певних умов під час запуску програми функціями `posix_spawn*()`.

Дескриптор файлу також можна відправити іншому процесу в допоміжних даних (ancillary data) через сокет комунікаційного домену Unix (лекція 11). Таке відправлення дескриптора файлу є запитом ядру дублювати дескриптор файлу в іншому процесі. Відправлення дескриптора файлу іншому процесу є єдиним способом дублювати дескриптор файлу в непов’язаному (unrelated) процесі.

Програма на C, яка показує використання дубльованого дескриптора файлу, який посилається на відкритий файл.

```c
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>

static void
print_file_offset(const char *msg, int fd)
{
    off_t offset;
```

```
    offset = lseek(fd, 0, SEEK_CUR);
    if (offset < 0)
        exit_err("lseek()");
    printf("%-30s: %7jd\n", msg, (intmax_t)offset);
}

int
main(void)
{
    const char *pathname = "/tmp/file.txt";
    const char text1[] = "some text\n";
    const char text2[] = "another text\n";
    int fd1, fd2;

    fd1 = open(pathname, O_CREAT|O_TRUNC|O_RDWR, S_IRUSR|S_IWUSR);
    if (fd1 < 0)
        exit_err("open() for %s", pathname);
    print_file_offset("Offset for fd1 after open()", fd1);

    if (write(fd1, text1, sizeof(text1) - 1) < 0)
        exit_err("write()");
    print_file_offset("Offset for fd1 after write()", fd1);

    fd2 = dup(fd1);
    if (fd2 < 0)
        exit_err("dup()");
    print_file_offset("Offset for fd2 after dup()", fd2);
```

```
    if (write(fd2, text2, sizeof(text2) - 1) < 0)
        exit_err("write()");
    print_file_offset("Offset for fd1 after write()", fd1);

    if (close(fd1) < 0 || close(fd2) < 0)
        exit_err("close()");
}
```

// Запускаємо програму.

```
$ ./a.out
Offset for fd1 after open()   :       0
Offset for fd1 after write()  :      10
Offset for fd2 after dup()    :      10
Offset for fd1 after write()  :      23
$ cat /tmp/file.txt
some text
another text
$
```

Створення та завершення процесу

Управління процесами не є темою лекцій із мережевого програмування, але створення та завершення процесу потрібні для реалізації багатопроцесних серверів. Також сигнали не є темою лекцій із мережевого програмування, але процес може бути завершено через отримання сигналу. Тут інформацію про управління процесами надано спрощено та не повністю, але достатньо для реалізацій багатопроцесних серверів.

Ідентифікатор процесу (process identifier, PID) – це унікальне додатне ціле число, яке ядро призначає кожному процесу під час його створення. Значення PID можна зберегти в об’єкті типу `pid_t` (знаковий цілочисельний тип), визначений у `<sys/types.h>`. У деяких системних викликах від’ємним значенням PID можна кодувати об’єкт системи, до якого застосовують системний виклик. Після звільнення інформації про завершений процес, ядро може призначити його PID новому процесу.

Системний виклик `getpid()` повертає PID процесу. Системний виклик `getppid()` повертає PID батьківського процесу (parent process ID, PPID).

```c
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

Ці системні виклики завжди виконуються успішно.

Процес користувача може створити новий процес відповідним системним викликом. Створення нового процесу полягає в створенні нової структури процесу, призначенні йому унікального PID, створенні таблиці дескрипторів файлів, ініціалізації налаштувань опрацювання сигналів, створенні адресної карти. Під час виконання цих дій стан нового процесу є *Новий*, він не може виконуватися. Далі ядро налаштовує контекст виконання для нового процесу та додає його структуру процесу до RunQ. Ядро змінює стан нового процесу на *Готовий до роботи*, він може виконуватися. Ядро може вибрати його структуру процесу та виконати перемикання контексту на нього.

Яка програма буде виконуватися в новому процесі залежить від налаштувань адресної карти процесу. Адресна карта містить інформацію про діапазони адрес (регіони) адресного простору процесу та про джерела вмісту для цих діапазонів адрес. Наприклад, з яких файлів брати код, з яких файлів брати

початкові дані, де міститься стек і т. ін. (тут використано слово «файли», оскільки код процесу користувача може складатися з коду програми користувача та кодів використаних динамічних бібліотек). Ядро використовує адресну карту процесу для налаштування його адресного простору, ця дія є машинозалежною. В Unix під створенням нового процесу розуміють створення нового процесу, який є майже точною копією оригінального процесу.

Системний виклик `fork()` створює новий процес.

```c
#include <unistd.h>

pid_t fork(void);
```

Атрибути нового процесу (дочірній процес, child process) є копіями атрибутів оригінального процесу (батьківський процес, parent process) за винятком (перелічені тільки деякі атрибути): унікальний PID, інший PPID (це PID оригінального процесу). Новий процес матиме тільки один потік, який його створював. Слово «fork» перекладають як «роздвоєння», «розгалуження».

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. Це так зване ледаче копіювання (lazy copying). Для реалізації цього ядро послуговується властивостями сторінкової пам’яті. Регіони адресної карти оригінального процесу, які відображені без дозволу на модифікацію (це код та read-only дані файлів програми) будуть відображені з такими самими дозволами на доступу в новому процесі в ту саму фізичну пам’ять (це буде спільна пам’ять для двох процесів). Власне, саме так ядро відображає вміст будь-якого файлу з дозволами на доступ read-only або read-execute у будь-якому адресному просторі. Регіони адресної карти оригінального процесу, які відображені з дозволами на модифікацію (це дані та стек програми, це анонімна пам’ять (anonymous memory) її вміст процес створює під час свого виконання) будуть відображені в обох процесах із правами дозволами на 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 дозволами на доступу read-only, буде згенерована відповідна виняткова ситуація) ядро виконає фізичне копіювання вмісту відповідної фізичної сторінки у вільну фізичну сторінку, змінить відображення та дозволи на доступ на read-write

відповідної віртуальної сторінки. Реалізацію ледачого копіювання для сторінкової пам’яті називають copy-on-write (COW).

Новий процес має копію таблиці дескрипторів файлів оригінального процесу. Тобто кожний дескриптор файлу нового процесу вказує на той самий дескриптор відкритого файлу, на який вказує відповідний дескриптор файлу (дескриптор файлу з таким самим номером) оригінального процесу. Під час виконання `fork()` ядро дублює дескриптори файлів оригінального процесу, так само як це робить системний виклик `dup2()`, тільки дубльовані дескриптори файлів будуть міститися в таблиці дескрипторів файлів іншого процесу.

У разі успішного виконання системний виклик `fork()` повертає нуль у новому процесі та PID нового процесу в оригінальному процесі, інакше повертає `-1`. Можна сказати, що програма викликає системний виклик `fork()` один раз, але можуть бути два повернення з нього (виконання кожного процесу продовжується з повернення з виклику `fork()`).

Функція `sysconf()` з аргументом `_SC_CHILD_MAX` повертає максимальну дозволену кількість процесів у системі для одного реального user ID (максимальна

кількість процесів, які мають однаковий реальний user ID). У `<limits.h>` може бути визначено макрос `CHILD_MAX` та визначено макрос `_POSIX_CHILD_MAX`.

Процес завершує своє виконання в таких випадках: повернення з функції `main()`; процес викликає одну з функцій `exit()`, `_exit()`, `_Exit()`; повернення з початкової функції останнього потоку процесу; останній потік процесу викликає функцію `pthread_exit()`; останній потік процесу відповідає на запит на завершення (cancellation request), який був запитаний іншим потоком функцією `pthread_cancel()`; відбувається доставлення деякого сигналу процесу, що призводить до його завершення.

Ядро для процесу, виконання якого завершується, виконує такі дії (перелічені тільки деякі дії): закриває всі дескриптори файлів; звільняє пам’ять процесу; реєструє статус завершення процесу та змінює стан процесу на стан *Зомбі*.

Якщо стан процесу *Зомбі*, то його PID усе ще йому призначений та ядро має невелику кількість інформації про цей процес і статус його завершення.

Функція `exit()`, системні виклики `_exit()` та `_Exit()` завершують процес.

```c
#include <stdlib.h>

void _Exit(int status);
void exit(int status);

#include <unistd.h>

void _exit(int status);
```

Системні виклики `_exit()` та `_Exit()` еквівалентні. Функція `exit()` викликає всі функції, які були зареєстровані за допомогою функції `atexit()`, та завершує роботу з усіма потоками файлів (записує ще не записані дані, які зберігаються в буферах потоків файлів), далі викликає системний виклик `_exit()` або `_Exit()`.

Аргумент `status` визначає число, яке буде статусом нормального завершення процесу. За домовленістю значення `0` або `EXIT_SUCCESS` (нуль, визначено в `<stdlib.h>`) позначає успішне виконання процесу (щоб це не значило), а будь-яке інше значення, наприклад, `EXIT_FAILURE` (не нуль, визначено в `<stdlib.h>`), позначає якусь помилку.

Повернення з функції `main()` є еквівалентним виклику `exit()` зі значенням, яке повертає `main()`.

Якщо процес завершується через повернення з початкової функції останнього його потоку, якщо останній його потік викликає функцію `pthread_exit()` або якщо останній його потік відповідає на запит на завершення, то процес завершується, якби була викликана функція `exit(0)`.

Ядро зберігає інформацію про статус завершення процесу, для того, щоб його батьківський процес міг запитати цю інформацію. Це необхідно для синхронізування дій процесів, один із яких створює процес, чекає на його завершення та перевіряє його статус завершення. Якби ядро не надавало такий механізм, то програмам користувача довелося б його реалізовувати самими, використовуючи файли або засоби міжпроцесної взаємодії. Якщо батьківський процес запитав статус завершеного процесу, ядро звільнить усю інформацію про цей процес, оскільки синхронізацію дій завершено.

Системний виклик `waitpid()` повертає статус одного з дочірніх процесів.

```c
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *wstatus, int options);
```

Цей системний виклик чекає на завершення, зупинку або продовження виконання вказаного дочірнього процесу, та повертає його PID у разі успішного виконання. Якщо аргумент `pid` додатний, то це PID дочірнього процесу, для якого запитано статус. Якщо аргумент `pid` дорівнює `-1`, значить запитано статус для будь-якого дочірнього процесу.

Прапорці в аргументі `options` можуть бути такі: `WCONTINUED` вказує повертати статус дочірнього процесу, виконання якого було продовжено; `WNOHANG` вказує не блокуватися, якщо для жодного дочірнього процесу не можна відразу повернути статус, у цьому випадку системний виклик повертає нуль; `WUNTRACED` вказує повертати статус зупиненого дочірнього процесу.

Системний виклик `waitpid()` зберігає статус дочірнього процесу в об’єкті, на який вказує аргумент `wstatus`, якщо він не null покажчик. Якщо значення об’єкта, на який вказує аргумент `wstatus`, позначити `value`, то інформацію про статус дочірнього процесу можна отримати так: якщо `WIFEXITED(value)` не нуль, дочірній процес завершив своє виконання нормально, `WEXITSTATUS(value)` дорівнює вісьмом молодшим бітам статусу нормального завершення виконання дочірнього процесу; якщо `WIFSIGNALED(value)` не нуль, дочірній процес був завершений

сигналом, `WTERMSIG(value)` дорівнює номеру цього сигналу; якщо `WIFSTOPPED(value)` не нуль, виконання дочірнього процесу було зупинено сигналом, `WSTOPSIG(value)` дорівнює номеру цього сигналу; якщо `WIFCONTINUED(value)` не нуль, виконання зупиненого дочірнього процесу було продовжено. Також реалізація може підтримувати нестандартний макрос для визначення чи ядро зберегло образ пам’яті процесу, завершеного сигналом: якщо `WCOREDUMP(value)` не нуль і виконання дочірнього процесу було завершено сигналом, ядро зберегло образ пам’яті цього процесу.

Якщо нема такого процесу, для якого може бути отриманий запитуваний статус, то `waitpid()` завершиться з помилкою, номер помилки буде `ECHILD`.

Якщо процес постійно створює нові процеси й «забуває» опитати статуси завершених процесів, то в системі буде багато процесів у стані *Зомбі*. Для кожного такого процесу витрачається пам’ять для його структури процесу та PID. Якщо в ядрі обмежена кількість PID (ця кількість завжди обмежена через кінцеву кількість значень у цілочисельному типі, але ядро може мати заздалегідь визначену максимальну кількість структур процесів), то процеси в

стані *Зомбі* можуть використати всі доступні PID і ядро не зможе створити жодного нового процесу.

Програма на C, яка демонструє використання системних викликів `fork()` та `waitpid()`. Виклик `fflush(stdout)` у цій програмі пояснено далі.

```c
#include <sys/wait.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

static void
print_wstatus(pid_t pid, int wstatus)
{
    printf("Process PID %jd\n", (intmax_t)pid);
    if (WIFEXITED(wstatus)) {
        printf("  Exited, exit status %d", WEXITSTATUS(wstatus));
    } else if (WIFSIGNALED(wstatus)) {
        printf("  Terminated by signal %d: %s", WTERMSIG(wstatus),
            strsignal(WTERMSIG(wstatus)));
#ifdef WCOREDUMP
        if (WCOREDUMP(wstatus))
            printf(" (core dump generated)");
#endif
```

```
    } else {
        printf("  Stopped or continued");
    }
    printf("\n");
}

static void
run_process(bool req_fail)
{
    pid_t pid;
    int wstatus;

    fflush(stdout);
    pid = fork();
    if (pid < 0)
        exit_err("fork()");
    if (pid == 0) {
        if (req_fail) {
            char *ptr = nullptr;
            *ptr = 'x';
        }
        exit(123);
    }
    printf("Forked process PID %jd\n", (intmax_t)pid);
    pid = waitpid(-1, &wstatus, 0);
    if (pid < 0)
        exit_err("waitpid()");
    print_wstatus(pid, wstatus);
```

```
}

int
main(void)
{
    run_process(true);
    run_process(false);
}
```

// Запускаємо програму.

```
$ ./a.out
Forked process PID 4463
Process PID 4463
  Terminated by signal 11: Segmentation fault (core dumped)
Forked process PID 4467
Process PID 4467
  Exited, exit status 123
```

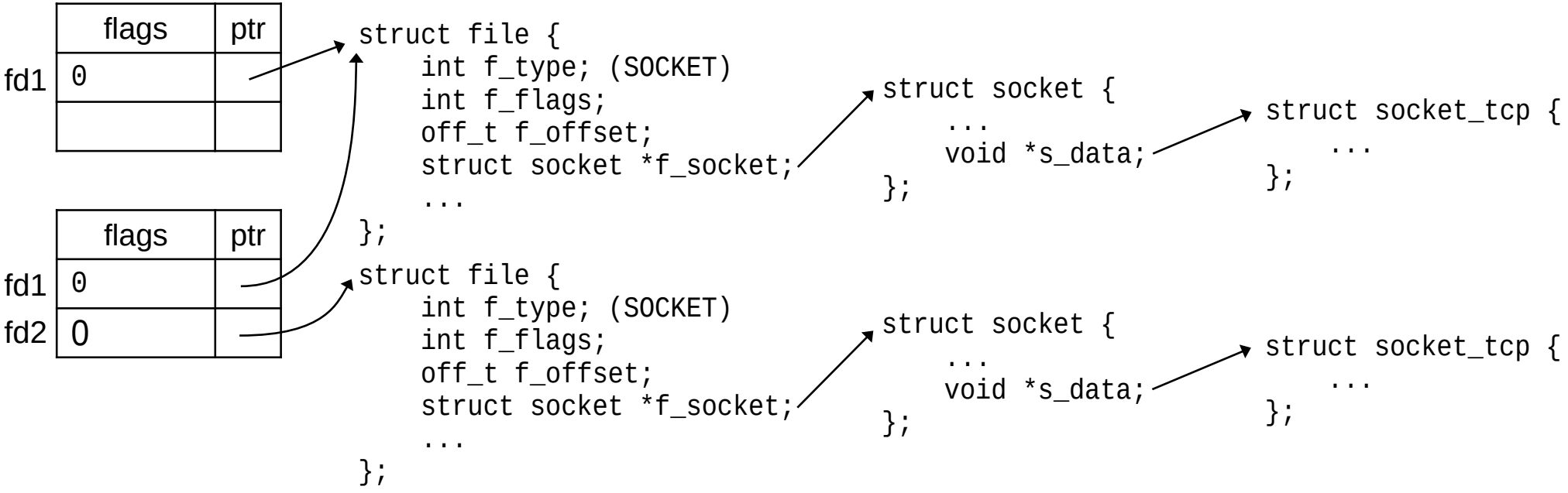
Якщо процес завершує своє виконання до завершення виконання свого дочірнього процесу, то батьківським процесом цього дочірнього процесу буде деякий системний процес. Цей системний процес запитує статуси завершених своїх (власних та успадкованих) дочірніх процесів, тому завершені процеси не будуть у стані *Зомбі*.

Перевіримо це в програмі на C.

```c
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>

int
main(void)
{
    pid_t pid1, pid2, ppid;

    pid1 = getpid();
    printf("Original process PID %jd\n", (intmax_t)pid1);

    pid2 = fork();
    if (pid2 < 0)
        exit_err("fork()");

    if (pid2 == 0) {
        for (;;) {
            ppid = getppid();
            printf("New process PPID %jd\n", (intmax_t)ppid);
            if (ppid != pid1)
                break;
            sleep(2);
        }
    } else {
        sleep(1);
    }
}
```

// Запускаємо програму.

```
$ ./prog
Original process PID 5705
New process PPID 5705
$ New process PPID 1718
```




```
int fd1 = socket(AF_INET, SOCK_STREAM, 0);
if (fork() == 0)
     int fd2 = socket(AF_INET, SOCK_STREAM, 0);
pause();
```

Цей рисунок показує об’єкти, які створює ядро в разі успішного виконання системних викликів `fork()` та `socket()`, які були викликані цією програмою.

Перевіримо правильність цього рисунку в програмі. Програма (оригінальний процес) відкриває або створює файл, змінює його розмір на нуль. Потім оригінальний процес записує дані у файл. Далі програма створює новий процес, в якому дубльовані всі дескриптори файлів оригінального процесу. Новий процес відкриває той самий файл. Потім він записує дані в дубльований дескриптор файлу та зчитує дані зі щойно відкритого файлу. Оригінальний процес чекає на завершення нового процесу та записує дані у файл.

```c
#include <sys/stat.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <stddef.h>
#include <unistd.h>

int
main(void)
{
    const char *pathname = "/tmp/file.txt";
    const char data1[] = {'1'};
    const char data2[] = {'2'};
    const char data3[] = {'3'};
    pid_t pid;
    int fd1, fd2;
    char buf[sizeof(data1) + sizeof(data2)] = {'\0'};
```

```
    fd1 = open(pathname, O_CREAT|O_TRUNC|O_RDWR, S_IRUSR|S_IWUSR);
    if (fd1 < 0)
        exit_err("open()");
    if (write(fd1, data1, sizeof(data1)) < 0)
        exit_err("write()");
    pid = fork();
    if (pid < 0)
        exit_err("fork()");
    if (pid == 0) {
        fd2 = open(pathname, O_RDONLY);
        if (fd2 < 0)
            exit_err("open()");
        if (write(fd1, data2, sizeof(data2)) < 0)
            exit_err("write()");
        if (read(fd2, buf, sizeof(buf)) < 0)
            exit_err("read()");
        if (close(fd1) < 0 || close(fd2) < 0)
            exit_err("close()");
        write(STDOUT_FILENO, buf, sizeof(buf));
    } else {
        if (waitpid(-1, nullptr, 0) < 0)
            exit_err("waitpid()");
        if (write(fd1, data3, sizeof(data3)) < 0)
            exit_err("write()");
        if (close(fd1) < 0)
            exit_err("close()");
    }
}
```

// Запускаємо програму.

```
$ ./a.out
12$
```

// Виводимо вміст файлу.

```
$ cat /tmp/file.txt
123$
```

Ця програма показує, що дубльований дескриптор файлу в новому процесі дає змогу модифікувати файл. Також зміщення, де були модифікування файлу в обох процесах, є такими, які були встановлені попередніми модифікуваннями файлу, як в оригінальному процесі, так і в новому процесі. Тобто `fd1` в оригінальному процесі та `fd1` у новому процесі – це дескриптори файлів, які «вказують» на той самий об’єкт типу `struct file` в ядрі. Файл, який був відкритий новим процесом, має власне зміщення, яке дорівнює нулю після відкриття файлу. Тобто `fd2` в новому процесі «вказує» на інший об’єкт типу `struct file` в ядрі. Оскільки записані дані у файл (який був відкритий першим викликом `open()`) та прочитані дані з файлу (який був відкритий другим викликом `open()`) збігаються, зрозуміло, що два об’єкти типу `struct file` посилаються на один об’єкт типу `struct vnode` в ядрі.

Якщо оригінальний процес працює з потоком файлу та створює новий процес, то це треба враховувати. Ядро копіює потік файлу та вміст його буфера оригінального процесу в дочірній процес (ядро копіює вміст пам’яті оригінального процесу, а потік файлу є програмною абстракцією режиму користувача). Якщо цей буфер не порожній, то дані, які зберігаються в ньому, можуть бути виведені в потоки файлів кілька раз (в оригінальному та в дочірніх процесах). Щоб розв'язувати цю проблему, треба заборонити буферизацію для цього потоку файлу, або викликати `fflush()` для нього в оригінальному процесі перед викликом `fork()`, або викликати `fflush()` з аргументом `nullptr` в оригінальному процесі, або викликати `_exit()` для завершення дочірнього процесу.

Ось програма на C, яка пояснює в чому проблема.

```c
#include <stdio.h>
#include <unistd.h>

static void
fork_and_exit(void (*exit_func)(int))
{
    pid_t pid;
```

```
    pid = fork();
    if (pid < 0)
        exit_err("fork()");
    if (pid == 0)
        exit_func(EXIT_SUCCESS);
}

int
main(void)
{
    printf("some text 1_1 ");
    fork_and_exit(exit);
    printf("some text 1_2\n");
    fork_and_exit(exit);

    printf("some text 2\n");
    fork_and_exit(exit);

    printf("some text 3\n");
    fflush(stdout);
    fork_and_exit(exit);
    printf("some text 4\n");
    fork_and_exit(_exit);
}
```

// Запускаємо програму два рази.

```
$ ./a.out
some text 1_1 some text 1_2
```

```
some text 2
some text 1_1 some text 3
some text 4
$ ./a.out > /tmp/output.txt
$ cat /tmp/output.txt
some text 1_1 some text 1_1 some text 1_2
some text 1_1 some text 1_2
some text 2
some text 3
some text 1_1 some text 1_2
some text 2
some text 4
$
```

Перший раз програма виводить «коректний» результат тільки для рядків, які завершуються символом нового рядка. Це відбувається через те, що стандартна бібліотека C визначила, що standard output програми асоційований з інтерактивним пристроєм (з терміналом) та увімкнула буферизацію для `stdout` для рядків (стандартна бібліотека C виводить рядок, якщо він завершується символом нового рядка). Так зазвичай роблять реалізації стандартної бібліотеки C.

Другий раз standard output програми було перенаправлено у звичайний файл і стандартна бібліотека C увімкнула повну буферизацію для `stdout`, це визначено в POSIX. Деякі рядки, збережені у файлі, дубльовані кілька разів. Це відбувається через те, що непорожній буфер `stdout` оригінального процесу було скопійовано в кількох дочірніх процесах, які потім вивели його вміст.