Міністерство освіти і науки України
Житомирський державний університет імені Івана Франка
Кафедра комп'ютерних наук та інформаційних технологій

Андрій Зіновчук, Світлана Постова

Системне програмування

Частина 1

Навчально-методичний посібник

Житомир
Вид-во ЖДУ ім. І. Франка
2026

УДК 004.75 : 004.9
З 81

Затверджено на засіданні вченої ради Житомирського державного університету імені Івана Франка, протокол № 11 від 29.05.2026р.

Рецензенти:
Оксана НАКОНЕЧНА – кандидат технічних наук, доцент, доцент кафедри інформаційних технологій Одеського державного аграрного університету.

Ірина КОЛЕСНІКОВА – заступниця директора з науково-методичної роботи, кандидатка педагогічних наук, професорка кафедри суспільно-гуманітарних дисциплін КЗ «Житомирський ОІППО» ЖОР.

Олена УСАТА – кандидат педагогічних наук, доцент, завідувачка кафедри комп'ютерних наук та інформаційних технологій Житомирського державного університету імені Івана Франка.

З 81

Зіновчук А. В., Постова С.А. Системне програмування: навчально-методичний посібник. Житомир: Вид-во ЖДУ ім. Івана Франка, 2026. 43 с.

Навчально-методичний посібник, що пропонується, може бути рекомендованим здобувачам вищої освіти спеціальності F3 Комп'ютерні науки, А4 Середня освіта (предметна спеціальність А4.09 Середня освіта (Інформатика)), А5 Професійна освіта (спеціалізація А5.39 (Цифрові технології)), а також вчителям інформатики для підготовки до уроків та факультативних занять.

УДК 004.75 : 004.9

ЗМІСТ

ВСТУП

Системним програмуванням називається процес розробки програм для управління компонентами комп'ютера: процесором, оперативною пам'яттю та периферійними пристроями. Такі програми безпосередньо взаємодіють з системним програмним забезпеченням або операційною системою (ОС). Системне програмування відрізняється від прикладного тим, що результатом системного програмування є програми, які обслуговують операційну систему або апаратне забезпечення. Оскільки різні ОС відрізняються як внутрішньою архітектурою, так і способами взаємодії з периферійними пристроями, то принципи системного програмування для різних ОС є відмінними.

В цьому посібнику ми описуємо основні елементи програмного інтерфейсу Unix-подібних ОС на базі ядра Linux (Ubuntu, Debian, Mint). Це інтерфейс використовується (прямо, чи опосередковано) в кожному застосунку, що працює під ОС Linux. Він містить системні виклики, що дають можливість програмувати файлове введення/виведення, створення та видалення файлів і каталогів, створення нових процесів та потоків, виконання програм, обмін даними між процесами та потоками на одному комп'ютері та обмін даними між процесами, що знаходяться на різних комп'ютерах, підключених через мережу. Однією з особливостей системного програмування в ОС Linux є переважно низькорівневий підхід. Ми використовували "рідну" для Linux і перевірену роками мову С. Однак більшість із прикладів і завдань, що при приведені тут, можна реалізувати на мові С++.

Основною метою даного навчально-методичного посібника є знайомство з системним програмним інтерфейсом та оволодіння основними прийомами системного програмування в Unix-подібних ОС. В посібнику розглянуті прийоми створення, компіляції та збірки програм, системні виклики в стандартах С та POSIX, управлінні процесами та концепція багатозадачності, міжпроцесна комунікація, управління та синхронізація програмних потоків. За межами розгляду залишилися такі розділи як файлова система, сокети, мережеве програмування, так як вони більше відносяться до мережевого адміністрування. Посібник може бути корисним для здобувачів спеціальностей F3 Комп'ютерні науки, А4 Середня освіта (предметна спеціальність А4.09 Середня освіта (Інформатика)), А5 Професійна освіта (спеціалізація А5.39 (Цифрові технології)) при вивченні освітніх компонент, пов'язаних з операційними системами та основами системного програмування.

1. ОСНОВИ СИСТЕМНОГО ПРОГРАМУВАННЯ. СТВОРЕННЯ, КОМПІЛЯЦІЯ ТА ЗБІРКА ПРОГРАМ. СТАНДАРТИ POSTIX ТА СИСТЕМНИЙ API

1.1. Компіляція програми

Створення будь-якої програми починається з постановки завдання, проектування та написання вихідного коду (*source code*). Зазвичай вихідний код програми записується в один або кілька файлів, які називають вихідними файлами або вихідними кодами.

Традиційно процес компіляції складається з двох етапів: трансляції вихідного коду програми на машинну мову і наступного компонування (іноді використовується термін "лінкування" - від англ. *link*). Раніше для отримання коду програми використовувалися дві програми: транслятор, на виході якого ми отримували об'єктний файл (з розширенням obj), і компонувальник (*linker*), що створює з об'єктного файлу запускний файл. Сучасні компілятори поєднують обидва ці етапи, і на виході ми відразу отримуємо код програми. Звичайно, будь-який сучасний компілятор може створити об'єктний файл, якщо це є необхідно.

В ОС Ubuntu або Debian найпростішим варіантом встанови одразу все необхідне для системного програмування на мові С або С++ є використання пакету *build-essential*. Встановити його можна за допомогою команди

```
sudo apt install build-essential
```

На етапі встановлення компілятор запитує дозвіл на встановлення певних пакетів. Введіть **Y**, щоб надати дозвіл. Ви також помітите багато команд для розпакування та налаштування пакетів. Після того як пакет буде встановлено, ви можете перевірити версію GCC за допомогою такої команди:

```
gcc --version
```

Для С++ компілятора

```
g++ --version
```

Щоб переконатися, що пакет *build-essential* було успішно встановлено, напишемо та скомпілюємо найпростішу програму на С. Для написання можна вибрати довільний текстовий редактор (наприклад Vim чи nano, якщо ви працюєте безпосередньо в Ubuntu, або VScode чи Notepad ++ , якщо ви розгортаєте Ubuntu через віртуальну машину під Windows). Напишемо наступний фрагмент коду:

```
#include <stdio.h>
int main(void) {
     printf("Hello, world!\n");
     return 0;
}
```

Заголовочний файл `stdio.h` робить доступними механізми введення-виведення стандартної бібліотеки мови С. Нам він потрібний для виклику функції `printf`.

Власне програма починається з функції `main`, у тілі якої рядок "Hello, world!" виводиться на екран функцією `printf`. Інструкція `return 0` здійснює вихід із програми.

Тепер запустимо компілятор:

```
gcc hello.c
```

Якщо ви не зробили помилок в цій простій програмі, то ви нічого не побачите і компілятор завершить роботу. В поточному каталозі з'явиться файл `a.out`. Це і є ваша скомпільована програма, запустіть її, щоб побачити результат роботи.

В наведеному вище прикладі, ми викликали основну функцію як `main(void)` . Однак, загальному випадку ця функція може мати аргументи, що передаються програмі через командний рядок. Оголошення `main` можна представити наступним чином:

```
int main(int argc, char *argv[])
```

або

```
int main(int argc, char **argv)
```

`int argc` – ціле число, що рівне числу аргументів командного рядка, що передаються програмі;
`char *argv[]` – масив вказівників на аргументи командного рядка, кожен з яких має тип `char`. Перший аргумент `argv[0]`зазвичай містить назву програми. `argv[1]` – вказівник на перший переданий програмі аргумент, `argv[2]` – вказівник на другий переданий програмі аргумент і т. д. до елемента `argv[argc-1]`. Список вказівників в `argv` закінчується NULL-вказівником (`argv[argc]=NULL`).

1.2. Опцій компілятора

Загальний формат виклику компілятора `gcc` наступний

```
gcc [parameters] file
```

Опцій (або їх ще називають параметрами) у `gcc` дуже багато. Ви можете прочитати призначення тієї чи іншої опції в довідковій системі (команда `man gcc`). Одна з найбільш корисних для нас в даний момент опцій, це `-o`, що дає можливість задати ім'я виконуваного файлу (за замовчуванням дається ім'я `a.out`). Опція `-v` виводить додаткову інформацію у процесі компіляції. Перекомпілюємо нашу програму використовуючи параметр `-o`:

```
gcc -v -o hello hello.c
```

Опція `-v` виведе повну інформацію, починаючи від розташування заголовочних файлів і версії компілятора, до списку використовуваних бібліотек. Навіть для такої простої програми опція `-v` згенерує багато тексту:

процес компіляції зовсім не такий простий, як здається на перший погляд. Опція `-o` створить файл `hello` замість `a.out`. Запустити `hello` можна аналогічно:

```
./hello
```

1.3. Багатофайлові проєкти

Сучасні програмні проєкти рідко обмежуються одним вихідним файлом. Розподіл вихідного коду програми на кілька файлів має низку суттєвих переваг перед однофайловими проєктами. Зазвичай процес складання багатофайлового проєкту здійснюється за таким алгоритмом:

1. Створюються та готуються вихідні файли. Кожен файл має бути цілісним, тобто не повинен містити незавершені конструкції. Функції та структури не повинні розриватися. Якщо у межах проєкту передбачається створення виконуваної програми, то одному з вихідних файлів має бути функція `main`.

2. Створюються та готуються заголовні файли. У заголовних файлів особлива роль: вони встановлюють угоди щодо використання спільних ідентифікаторів (імен) у різних частинах програми. Якщо, наприклад, функція `func` реалізована у файлі `aaa.c`, а викликається у файлі `bbb.c`, то обидва файли потрібно включити директивою `#include` головний файл, що містить оголошення виклик `main`.

3. Кожен вихідний файл окремо компілюється із опцією `-c`. Внаслідок цього з'являється набір об'єктних файлів (`file.o`).

4. Отримані об'єктні файли з'єднуються компонувальником в один виконуваний файл.

Якщо необхідно скомпонувати декілька об'єктних файлів (`file1.o`, `file2.o` і т.д.), то застосовують наступний простий шаблон:

```
gcc -o OUTPUT file1.o file2.o ...
```

1.4. Автоматична збірка програм

Під час збірки програми потрібно вказувати багато опцій компілятора у командному рядку. В результаті виходить довга команда, набираючи яку щоразу після виправлення чергової помилки, можна легко припуститися помилки. У Linux (та інших UNIX-системах), для збирання складних проєктів прийнято використовувати утиліту `make`. Суть полягає в наступному: ви створюєте так званий `Makefile`, в якому вказуються цілі для збірки програми, бібліотек чи об'єктного файлу).

Автоматична збірка програм на мові С зазвичай здійснюється за наступним алгоритмом:

1. Підготовляються вихідні та заголовочні файли.

2. Підготовляється `make`-файл, що містить відомості про проєкт. Навіть великі проєкти можна зібрати одним `make`-файлом. `make`-файл може мати довільну назву, але, зазвичай, вибирають одне з трьох стандартних імен: `Makefile`, `makefile` або `GNUmakefile`, які розпізнаються утилітою для збірки `make` автоматично.

3. Викликається утиліта `make`, яка збирає проєкт на підставі даних, одержаних з `make`-файлу.

Отже, щоб працювати з `make` необхідно створити файл з ім'ям `Makefile`. У `make`-файлах можуть бути такі конструкції:

- Коментарі. У `make`-файлах допустимі однорядкові коментарі, які починаються символом # і діють до кінця рядка.
- Оголошення констант. Константи в `make`-файлах служать для встановлення. Вони багато в чому схожі з константами препроцесора мови С.
- Цільові зв'язки. Ці елементи мають основне значенні в `make`-файлі. За допомогою цільових зв'язок задаються залежності між різними частинами програми, а також визначаються дії, які будуть виконуватися при складанні програми. У будь-якому `make`-файлі має бути хоча б одна цільова зв'язка.

  Кожна цільова зв'язка складається з наступних компонентів:
- Ім'я мети. Якщо метою є файл, то зазначається його ім'я. Після імені мети слідує двокрапка.
- Список залежностей. Тут просто перераховуються (через пробіл) імена файлів або імена проміжних цілей. Якщо мета ні від чого не залежить, цей список буде порожнім.
- Інструкції. Це команди, які мають виконуватися задля досягнення мети. Кожна інструкція пишеться на новому рядку і починається із символу табуляції (саме табуляції, а не групи пробілів). Іноді цільова зв'язка не передбачає виконання команд, а покликана лише встановити залежності. У такому разі список інструкцій залишають порожнім.

У `make`-файлах для параметризації процесу збірки можна використовувати константи. Для оголошення та ініціалізації констант передбачено наступний шаблон:

```
NAME=VALUE
```

Тут NAME – це ім'я константи, VALUE – її значення. Ім'я константи не повинне починатися з цифри. Значення може містити будь-які символи, включаючи пробіли. Ознака закінчення константи – кінець рядка. Інакше кажучи, будь-які символи, що стоять між знаком "=" та символом перенесення рядка, будуть значенням константи.

Значення константи можна підставити в будь-яку частину `make`-файлу (крім коментаря). Якщо ім'я константи складається з одного символу, для підстановки достатньо додати перед ім'ям символ $. Коли ім'я складається з кількох символів, для підстановки використовується наступний шаблон:

```
$(NAME)
```

Для прикладу, створимо `make`-файл що збирає бінарник із трьох файлів `main.c`, `obj1.c` та `obj2.c`. Цей файл містить п'ять цілей: перша (вона називається основною) ціль, `target`, виконує збірку. Другі, третя та четверта

цілі first_object, second_object та main) виконують компіляцію вихідних файлів obj1.c, obj2.c та main.c. П'ята мета clean виконує очищення всіх об'єктних файлів, що утворилися під час компіляції. Приклад такого make-файлу може бути наступним:

```
CC=gcc
CLEAN=rm -f
PROGRAM_NAME=final
# Головна ціль
target: main first_object second_object
     $(CC) -o $(PROGRAM_NAME) main.o obj1.o obj2.j

# Ціль для компіляції файла obj1.c
first_object: obj1.c
     $(CC) -c obj1.c

# Ціль для компіляції файла obj2.c
second_object: obj2.c
     $(CC) -c obj2.c

# Ціль для компіляції основного файла main.c
main: main.c
     $(CC) -c main.c

# Очистка проєкта
clean:
     $(CLEAN) *.o
```

Основна ціль target вимагає виконання додаткових цілей, тому вона містить список залежностей. Ціль clean не має списку залежностей. Після зберігання make-файлу викликаємо утиліту make з вказанням цілі, яку треба виконати:

```
$ make target
```

Взагалі, при запуску make ім'я мети можна не вказувати. Тоді основною метою буде вважатися перша мета в Makefile. Отже, у нашому випадку, щоб зібрати проєкт, достатньо викликати make без аргументів. При цьому ціль clean не виконуватиметься. Реалізацію цієї цілі треба запускати окремо.

```
$ make clean
```

1.5. Змінні оточення

Щоб зрозуміти, що таке оточення, потрібно трохи забігти наперед і розібратися в базових термінах багатозадачності Linux. В основі багатозадачності лежить поняття "процес". В самому простому розумінні, процес – це щось, що виконує в системі якісь дії (більш детально про процеси буде іти мова в наступних розділах). Із кожною запущеною програмою пов’язаний процес. Один процес може створити інший. У такому разі процес, що створює, називають батьківським процесом, а створений процес – дочірнім.

Усі процеси у системі є елементами однієї ієрархії, які називають деревом процесів. На вершині цієї ієрархії знаходиться процес `init`. Це єдиний процес, який не має батьківського процесу. Щоразу, коли ви запускаєте у командному рядку якусь програму, наприклад `ls`, у системі народжується новий процес, для якого командна оболонка є батьківським процесом.

Оточення (*environment*) – це набір специфічних для конкретного користувача пар `ЗМІННА = ЗНАЧЕННЯ`. Кожен процес має в своєму розпорядженні і вільно розпоряджається своєю копією оточення, яку він отримує від батьківського процесу. Таким чином, якщо процес спочатку модифікує своє оточення, а потім створить новий процес, нащадок отримає копію вже зміненого оточення.

У сучасних Linux-дистрибутивах оточення представлено десятками змінних, але широко використовуються лише деякі з них:

- `USER` – ім'я користувача;
- `HOME` – домашній каталог;
- `PATH` – список каталогів, у яких здійснюється пошук виконуваних файлів;
- `SHELL` – командна оболонка, що використовується за замовчуванням;
- `PWD` – поточний каталог.

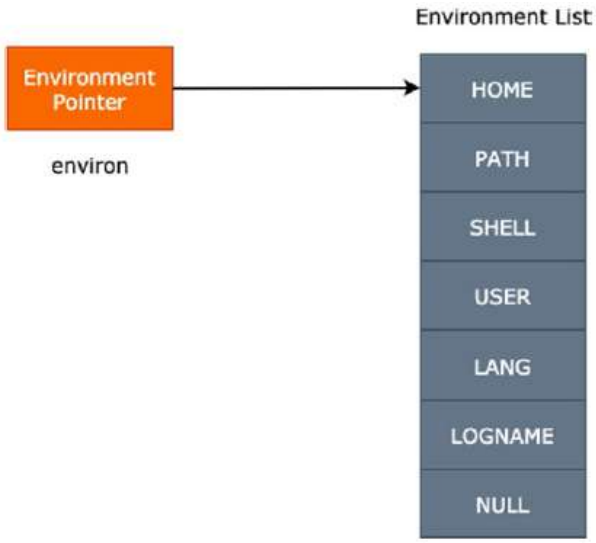


Рис.1. Приклад списку змінних оточення

В програмах на мові С доступ до оточення може бути забезпечений двома способами. Перший спосіб це використання зовнішньої змінної `extern char *environ[]` (або `char **environ`). Вона представляє собою масив вказівників на рядки `ЗМІННА=ЗНАЧЕННЯ` із завершальним `NULL`-вказівником. Так само як і масив `argv[]`, `environ[]` закінчується `NULL`-вказівником. Ця змінна оголошена в заголовочному файлі `unistd.h`. Другий спосіб це використання функції `getenv( )`, що оголошена в `stdlib.h`.

```
char *getenv(const char *name);
```

повертає вказівник на змінну оточення `name`, або `NULL`, якщо такої змінної немає.

Для модифікації змінних оточення використовуються наступні функції (всі оголошені в `stdlib.h`).

```
int setenv(const char *name, const char *value, int overwrite);
```

повертає 0 (успішне виконання) або -1 (помилка)

Ця функція створює (або змінює значення) змінну з іменем `name`, значення цієї змінної задано параметром `value`. Опція `overwrite` – це параметр перезапису. Якщо він дорівнює 0, існуюча змінна `name`, не буде перезаписана, а якщо містить значення, відмінне від 0, змінній `name` буде присвоєно нове значення `value`. Наступна функція

```
int putenv(char *string);
```

повертає 0 (успішне виконання) або ненульове число (помилка)

Виконує те саме, що і `setenv`, але для завдання змінної оточення використовується рядок виду `name = value` на який посилається вказівник `*string`. Для видалення змінної `name` із оточення використовується функція `unsetenv`.

```
int unsetenv(const char *name);
```

повертає 0 (успішне виконання) або -1 (помилка)

1.6. Стандарти POSTIX та ISO C

POSIX – це абревіатура від *Portable Operating System Interface* для операцій на базі Unix-подібних ОС. Під цією назвою розуміють це сімейство стандартів, розроблених організацією IEEE (*Institute of Electrical and Electronics Engineers*). Основний інтерес представляє стандарт 1003.1, мета якого полягає у підвищенні переносимості додатків між різними версіями UNIX. В літературі цей стандарт називають також POSIX.1. Цей стандарт визначає набір служб, які має надавати ОС, якщо вона претендує на звання POSIX-сумісної. Хоча стандарт POSIX.1базується на операційній системі UNIX, він не обмежується UNIX та UNIX-подібними операційними системами. Оскільки стандарт POSIX.1 визначає інтерфейс, а не реалізацію, між системними викликами та бібліотечними функціями немає ніяких відмінностей. На сьогодні відомо декілька розширень оригінального POSIX.1. Вони відомі під назвами • POSIX.1b, POSIX.1c та POSIX.2.

Стандарт POSIX.1 стосується основних служб усіх моделей операційних систем. Нижче наведені функції, включені в цей стандарт.

- Створення та контроль процесу
- Тригери процесу
- Операції з файлами та каталогами
- Помилки сегментації
- Помилки пам'яті

• Винятки з плаваючою комою
• Канали
• Сигнали
• Реалізація стандартної бібліотеки С
• Стандартний інтерфейс введення/виведення та керування

Розширення оригінального стандарту, яке відоме під назвою POSIX.1b, включає програмні інтерфейси з наступних тем.

• Алгоритми планування ЦП
• Передача повідомлень
• Спільна пам'ять
• Семафор
• Інтерфейси блокування пам'яті
• Синхронні та асинхронні інтерфейси передачі даних

Стандарт POSIX.1c включає основні функції, пов’язані з багатопотоковістю.

• Створення потоку
• Керування потоками
• Видалення теми
• Синхронізація потоків
• Планування потоків

Стандарти POSIX.2 стосуються основних функцій ОС.

• uname
• tty
• cd
• ls
• mkdir
• echo
• cp
• rm
• mv

Мета стандарту ISO (*International Organization for Standardization*) С забезпечити переносимість програм написаних на мові С між різними ОС. Цей стандарт визначає не тільки синтаксис та семантику мови, але також склад стандартної бібліотеки. Ця бібліотека має велике значення, тому що всі сучасні версії UNIX-подібних ОС, зобов'язані надавати бібліотеки функцій, які визначаються стандартом мови С. Бібліотеку ISO С можна розбити на 24 розділи, виходячи з імен заголовочних файлів, визначених стандартом. Стандарт POSIX.1 включає ці файли і, крім того, визначає ряд додаткових заголовочних файлів. У таблиці 1 наведено декілька заголовочних файлів, які будуть часто зустрічатися у прикладах програм цього посібника.

Таблиця 1

Бібліотека	Функціональність
stdio.h	містить усі стандартні функції операцій введення та виведення: 10 макросів та 41 функція. Найбільш популярними функціями є printf і scanf.
stdlib.h	стандартна бібліотека в C, яка в основному використовується для загального призначення.
unistd.h	забезпечує стандартний інтерфейс для POSIX API.
sys/types.h	містить стандартні та похідні типи даних, які корисні на системному рівні програмування.
signal.h	функції для обробки дії сигналів в операційній системі.
time.h	підтримує роботу з датою та часом.
sys/stat.h	визначає стан і активність файлової системи.
fcntl.h	є частиною POSIX API, яка маніпулює файлами, наприклад змінює дозволи.
sys/ipc.h	функції для міжпроцесної взаємодії (черги повідомлень, семафори та спільна пам’ять).
sys/msg.h	працює з бібліотекою sys/ipc.h для роботи з IPC.
semaphore.h	функції для виконання дій семафорів в ОС. Це також а частина бібліотеки POSIX.
sys/shm.h	функції для виконання дій зі спільною пам'яттю.
sys/wait.h	функції для переведення процесу у стан очікування.

2. КОНЦЕПЦІЇ ВВЕДЕННЯ/ВИВЕДЕННЯ

Введення/виведення (Input/Output, I/O) – поняття досить абстрактне. Загалом, введенням/виведенням вважається взаємодія обробника інформації (у нашому випадку ним буде програма) із зовнішнім світом (з ОС та іншими програмами). У цьому посібнику ми розглядатимемо лише файлове введення/виведення. Реалізувати введення/виведення у програмі можна двома способами: Перший – це використання бібліотечних функцій мови С. Другий – це використання системних викликів ядра Linux. Кожен спосіб має свої переваги. Програма, яка використовує бібліотечні функції, може бути легко перенесена в іншу систему, де є компілятор С. Зовсім інша справа, якщо програма використовує системні виклики. Така програма читає та записує файли, звертаючись до ядра Linux. У цьому випадку введення/виведення називається низькорівневим. Програма буде працювати швидше, ніж та, що використовує бібліотечних функцій мови С (це особливо буде помітно при обробці великих обсягів інформації), але не може бути перенесена в ОС, де немає ядра Linux (наприклад Windows).

2.1. Файлове введення/виведення за допомогою системних викликів ядра Linux

Функції, описані у ядрі Linux, часто називають функціями небуферизованного файлового введення/виведення. Термін небуферизований означає, що кожна операція читання або запису звертається до системного виклику. Функції небуферизованого введення/виведення є частиною стандарту POSIX.1. До операцій файлового введення/виведення належать відкриття файлу, читання з файлу, запис у файл і т. д. Більшість операцій файлового введення/ виведення в UNIX-подібних ОС можна виконати за допомогою п'яти функцій: `open`, `read`, `write`, `lseek` та `close`, що оголошені в заголовочному файлі `fcntl.h`.

Усі відкриті файли представлені у ядрі файловими дескрипторами. У самому простому випадку дескриптор файлу використовуються для звичайних файлів на диску. Однак, в UNIX-подібних ОС, дескриптори файлів також можуть бути використані для позначення всіх інших типів файлів, включаючи канали, сокети, термінали, зовнішні пристрої. Кожен процес має власний набір дескрипторів файлів. Файловий дескриптор – це невід’ємне ціле число. Коли процес відкриває існуючий файл або створює новий, ядро повертає йому файловий дескриптор. Щоб виконати запис у файл чи читання з нього, потрібно передати функції read чи write його файловий дескриптор, отриманий викликом функції `open` чи `creat`.

За домовленістю, командні оболонки UNIX-подібних ОС асоціюють файловий дескриптор 0 з пристроєм стандартного введення процесу, 1 - з пристроєм стандартного виведення і 2 - з пристроєм стандартного виведення повідомлень про помилки. Хоча значення цих дескрипторів визначені стандартом POSIX.1, у програмах замість фактичних значень 0, 1 і 2 слід

використовувати константи `STDIN_FILENO`, `STDOUT_FILENO` і `STDERR_FILENO`, що визначені в у заголовному файлі `unistd.h`.

Як правило, стандартне введення пов'язане з драйвером клавіатури, а стандартне виведення та стандартний потік помилок пов'язаний з дисплеєм комп'ютера. Але можна перенаправити стандартне виведення, наприклад, зв'язавши його із драйвером принтера (який також представлений у вигляді файлу).

Виклик `open`

Файл створюється або відкривається а допомогою виклику open, що оголошений в заголовочному файлі `fcntl.h`

```
int open(const char *path, int flag, mode_t mode)
```

повертає дескриптор файлу (успішне виконання) або -1 (помилка)

Аргумент `path` представляє ім'я файлу, який буде відкрито або створено. Аргумент `flag` визначає опції відкриття файлу, а аргумент `mode` – режим доступу до файлу. Значення аргументу `flag` можна задати однією або декількома константами (об’єднаними побітовим OR “|”), що наведені в таблиці 2.

Таблиця 2

константа	**значення**
`O_RDONLY`	Відкрити тільки для читання
`O_WRONLY`	Відкрити тільки для запису
`O_RDWR`	Відкрити для читання і запису
`O_CREAT`	Створити файл, якщо він ще не був створений
`O_DIRECT`	Дозволити введення/виведення файлів в обхід буферного кешу.
`O_DIRECTORY`	Згенерувати помилку, якщо `pathname` не є каталог
`O_EXCL`	Ця константа використовується в поєднанні з `O_CREAT`, щоб вказати, що якщо файл вже існує, його не слід відкривати.
`O_LARGEFILE`	Відкрити файл із підтримкою великих файлів. Використовується в 32-розрядних системах щоб працювати з великими файлами
`O_NOATIME`	Не оновлювати час останнього доступу до файлу під час читання з цього файлу.
`O_NOCTTY`	Не допускати, щоб файл став керуючим терміналом, якщо файл, що відкривається, є терміналом.
`O_TRUNC`	Очистити файл, знищивши всі наявні дані, якщо файл уже існує і є звичайним файлом.
`O_APPEND`	Записувати в кінець файлу.
`O_ASYNC`	Генерувати сигнал, коли введення/виведення можливе
`O_DSYNC`	Забезпечити синхронізовану цілісність даних введення-виведення

O_NONBLOCK	Відкрити в неблокуючому режимі
O_SYNC	Зробити запис файлів синхронним

Оскільки режим доступу до файлу в UNIX-подібних ОС має особливе значення, то порядок завдання аргументу mode буде показано детально в наступному розділі. Якщо під час виклику open аргумент flag не містись O_CREAT, то аргумент режиму файлу mode можна не вказувати.

Виклик creat

Новий файл можна також створити за допомогою виклику creat

```
int creat(const char *path, mode_t mode);
```

повертає дескриптор файлу (успішне виконання) або -1 (помилка)

Системний виклик creat створює та відкриває новий файл із заданим шляхом path або, якщо файл уже існує, відкриває файл і видаляє всі наявні дані. Аргументи path і mode мають той самий зміст, що і у виклику open. Виклик creat еквівалентний наступному виклику open:

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

Оскільки аргумент flag виклику open забезпечує більший контроль над тим, як відкривається файл, open більш частіше використовується в порівнянні з creat.

Виклик read

Системний виклик read читає дані з відкритого файлу, на який посилається дескриптор fd.

```
ssize_t read(int fd, void *buffer, size_t nbytes);
```

повертає: кількість прочитаних байтів (успішне виконання), 0 – якщо досягнуто кінець файлу (EOF), -1 – якщо виникла помилка

Тип даних ssize_t (введений в POSIX.1) означає цілий тип із знаком. Операція читання починається з поточної позиції у файлі. У разі успіху поточна позиція збільшується число фактично прочитаних байтів. Аргумент nbytes визначає максимальну кількість байтів для читання. Тип даних size_t є беззнаковим цілим типом. Аргумент buffer надає адресу буфера пам'яті, до якого мають бути розміщені вхідні дані. Цей буфер має містити як мінімум nbytes байтів.

При читанні зі звичайного файлу, коли кінець файлу (EOF) зустрівся до того, як було прочитано заявлену кількість байтів nbytes, виклик повертає кількість фактично прочитаних байтів. Наприклад, якщо до кінця файлу залишилося 50 байт, а заявлено 70, функція read поверне число 50. При наступному виклику вона поверне 0. Цей механізм зручно використовувати для організації циклічного доступу до читання і записування у файли. Наприклад цикл while для читання стандартного вводу (STDIN_FILENO)

```
int n;
char buf[BUFFSIZE];
```

```
   while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
```

Як приклад використання `read` можна привести наступний код для введення ряду символів із терміналу:

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

#define MAX_READ 20

int main(int argc, char **argv) {

  char buffer[MAX_READ];
  if (read(STDIN_FILENO, buffer, MAX_READ) == -1){
      perror("open");
      exit(EXIT_FAILURE);
   }
  printf("The input data was: %s\n", buffer);

  return 0;
}
```

В останньому прикладі, для обробки помилок при системному виклику `read`, були використані дві функції `perror` та `exit`, які ще не розглядалися нами. Функція `perror` буде дописана в цьому ж розділі нижче, а `exit` ми будемо розглядати при вивченні процесів.

Виклик `write`

Системний виклик `write` записує дані у відкритий файл, на який посилається дескриптор `fd`.

```
ssize_t write(int fd, void *buffer, size_t nbytes);
```

повертає кількість записаних байтів (успішне виконання) або -1 (помилка)

Аргументи для `write` подібні до аргументів для `read`: `buffer` – це адреса даних, які потрібно записати; `nbytes` – кількість байтів для запису з буфера пам'яті. Для звичайних файлів запис починається з поточної позиції у файлі. Якщо файл відкрито з `flag` = `O_APPEND`, перед початком кожної операції запису поточна позиція встановлюється на кінець файлу. Після закінчення запису значення поточної позиції збільшується кількість фактично записаних байтів.

Виклик `close`

Для закриття відкритого файлу використовується виклик `close`.

```
int close(int fd);
```

повертає 0 (успішне виконання) або -1 (помилка)

Після завершення процесу всі відкриті файли автоматично закриваються ядром. Однак, правильним стилем програмування вважається закривати непотрібні дескриптори файлів явно, оскільки це робить наш код більш читабельним і надійним перед наступними змінами. Крім того, дескриптори

файлів є витратним ресурсом, тому неможливість закрити дескриптор файлу може призвести до того, що у процесу закінчаться дескриптори. Це особливо важлива проблема під час написання програм, які мають справу з кількома файлами, такими як оболонки або мережеві сервери. Також правильним стилем вважається включення виклику `close` (як і будь-якого іншого системного виклику) у “дужки” з перевіркою помилок:

```
if (close(fd) == -1){
    perror("close ");
    exit(EXIT_FAILURE);
}
```

Виклик `lseek`

З будь-яким відкритим файлом пов'язане таке поняття як поточна позиція у файлі. Як правило, це ціле число, яке означає зміщення байтах від початку файлу. Операції читання і запису починають виконуватися з поточної позиції файлі і збільшують її значення на кількість прочитаних чи записаних байтів. За замовчанням, при відкритті файлу поточна позиція ініціалізується числом 0, якщо не було встановлено `flag = O_APPEND`. Явна зміна поточної позиції у файлі виконується за допомогою функції `lseek`

```
off_t lseek(int fd, off_t offset, int whence);
```

повертає нову поточну позицію у файлі (успішне виконання) або -1 (помилка)

Тип даних `off_t` – це цілочисельний тип зі знаком, визначений POSIX.1. Аргумент `offset` визначає кількість байтів на яку необхідно змістити поточну позицію. Аргумент `whence` вказує на базову точку, з якої має відбуватися зміщення. Інтерпретація аргументу `offset` залежить від того яке значення має аргумент `whence`. Аргумент `whence` може приймати наступні значення:

- може бути як додатним так і від’ємним).
- `SEEK_SET` – тоді `offset` інтерпретується як зміщені від початку файлу.
- `SEEK_CUR` – тоді `offset` інтерпретується як зміщені від поточної позиції у файлі (в цьому випадку `offset` може бути як додатним так і від’ємним).

`SEEK_END` – тоді `offset` інтерпретується як зміщені від кінця файлу (в цьому випадку `offset` Прикладом використання `lseek` може бути визначення поточної позиції в даний момент часу. Для цього треба `lseek` передати аргумент `offset=0`

```
off_t currentps;
currentps = lseek(fd, 0, SEEK_CUR);
```

2.2. Введення/виведення за допомогою бібліотеки C

Далі розглянемо стандартну бібліотеку C введення/виведення. Функції цієї бібліотеки оголошені в `stdio.h`. Ця бібліотека визначена стандартом ISO C, тому що реалізована у багатьох операційних системах, що не належать до

сімейства UNIX. Механізми введення/виведення стандартної бібліотеки С називають буферизованими. Буферизація даних у великі блоки для зменшення кількості системних викликів (`read` та `write`) – це саме те, що виконується функціями введення/виведення бібліотеки С при роботі з дисковими файлами. Таким чином, використання цієї бібліотеки звільняє нас від завдання ручної буферизації даних для виведення за допомогою `write` або введення через `read`. Стандартна бібліотека С сама проводить розміщення буферів і виконує операції введення/виведення блоками оптимального розміру, що позбавляє необхідності замислюватися про правильність вибору.

Усі функції введення/виведення з ядра ОС працюють з файлами у вигляді дескрипторів. При обговоренні стандартної бібліотеки введення/виведення вживають термін потік введення/виведення, або просто потік. Одразу зауважимо, потік введення/виведення (*stream*) це зовсім не те що розуміють під програмним потоком (*thread*), який ми будемо розглядати в даному посібнику пізніше. Відкриваючи або створюючи файл засобами стандартної бібліотеки, говорять, що із файлом зв’язують потік. У зв’язку із цим механізми введення/виведення за допомогою стандартної бібліотеки С ще називають *потоковими*. При відкритті потоку функції стандартної бібліотеки повертають вказівник на об'єкт `FILE`. Цей об'єкт є структурою з усією інформацією, необхідною для керування засобами стандартної бібліотеки: дескриптор файлу, покажчик на буфер потоку, розмір буфера, лічильник символів, що знаходяться у буфері , параметр помилки і т.д.

Для будь-якого процесу автоматично створюється три визначені потоки: стандартний потік введення, стандартний потік виведення та стандартний потік помилок. Ці потоки пов'язані з тими самими файлами, як і дескриптори `STDIN_FILENO`, `STDOUT_FILENO` і `STDERR_FILENO`, які згадувалися раніше при обговоренні низькорівневих викликів. Ці три потоки доступні за допомогою глобальних (`extern`) вказівників на файли `stdin`, `stdout` та `stderr`. Визначення файлових вказівників знаходяться у заголовочному файлі `stdio.h`:

```
extern FILE * stdout; /* Стандартне виведеня */
extern FILE * stdin; /* Стандартне введеня */
extern FILE * stderr; /* Стандартный потік помилок */
```

Відкриття файлу (потоку)

Функції fopen, freopen відкривають потік файлового введення/виведення

```
FILE *fopen(const char *pathname, const char *mode);
FILE *freopen(const char *pathname, const char *mode, FILE *fp);
```

повертають вказівник на файл (успішне виконання) або `NULL` (помилка)

Функція `fopen` відкриває файл, що задається шляхом `pathname`. Функція `freopen` відкриває файл `pathname` і пов'язує його із зазначеним потоком `fp`, попередньо закриваючи потік, якщо він уже був відкритий.

Можливі значення аргументу `mode` задані в таблиці 3

Таблиця 3

mode	значення
r або rb	Відкрити для читання
w або wb	Очистити файл або створити і відкрити для запису
a або ab	Відкрити для запису в кінець файлу або створити для запису
r+, або r+b, або rb+	Відкрити для читання і для запису
w+, або w+b, або wb+	Очистити файл або створити і відкрити для читання і запису
a+, або a+b, або ab+	Відкрити або створити для читання і для запису в кінець файлу

Закриття файлу (потоку)

Закривається відкритий потік з допомогою функції `fclose`.

```
int fclose(FILE *fp);
```

повертає 0 (успішне виконання) або `NULL` (помилка)

Перед закриттям потоку всі дані, що знаходяться у буфері виведення, скидаються. Усі дані, що знаходяться у буфері введення, будуть втрачені. Якщо пам'ять під буфер була виділена бібліотекою введення/виведення, вона звільняється автоматично.

Після відкриття потоку можна вибрати один із двох типів неформатного введення/виведення: посимвольне та рядкове введення/виведення.

Функції посимвольного введення/виведення

Наступна функція дає можливість читати по одному символу за одне звертання із файлового потоку `fp`.

```
int fgetc(FILE *fp);
```

повертають символ, зчитаний як `int` (успішне виконання) або `EOF` (помилка)

Функція `fgetc` дає можливість дізнатися про свою адресу, що дозволить передати її як аргумент до іншої функції. Зазначимо, `fgetc` повертає не символ, а код символу (`unsigned char` приведений до `int`). При виведенні зчитаного символу на екран, в програмах необхідно використовувати або специфікатор `%c` або механізм приведення (*typecasting*) в формі `(char)(n)`. Наступний приклад показує як можна вивести на екран перший символ із текстового файлу `a.txt`.

```
#include <stdio.h>

int main() {
   FILE *file = fopen("a.txt", "r");

   int ch = fgetc(file);
   if (ch != EOF) {
      printf("The first character: %c\n", ch);
   } else {
```

```
      printf("Error of reading\n");
   }
   fclose(file);
   return 0;
}
```

Кожній із двох описаних функції введення відповідають дві функції виведення, які записують один символ `c` у файловий потік `fp`.

```
int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
```

повертають значення аргументу `c` (успішне виконання) або `EOF` (помилка)

Функції рядкового введення/виведення

Рядкове введення забезпечується функціями

```
char *gets(char *srt);
char *fgets(char *srt, int size, FILE *fp);
```

повертають рядок `srt` (успішне виконання) або `EOF` (помилка)

Обом функціям передається адреса буфера `*srt` для розміщення прочитаного рядка із потоку `fp`. Функція `gets` читає із стандартного потоку введення, функція `fgets` – із вказаного аргументом `fp` потоку. Для функції `fgets` вказується розмір приймального буфера `size`. Ця функція читатиме вхідні дані в буфер доти, доки не зустріне символ переходу рядка, але не більше `size-1` символів. В кінці прочитаного рядка додається нульовий символ. Якщо довжина рядка, включаючи символ переходу рядка, становить більше `size-1` символів, функція поверне лише частину рядка, але до кінця буфера все одно буде доданий завершальний нульовий символ. Наступний виклик `fgets` поверне залишок рядка.

Рядкове виведення забезпечується функціями

```
int puts(const char *str);
int fputs(const char *str, FILE *fp);
```

повертають невід'ємне ціле (успішне виконання) або `EOF` (помилка)

Функція `fputs` записує рядок, що завершується нульовим символом у зазначений потік. Нульовий символ у потік не записується. Функція `puts` записує рядок, що завершується нульовим символом у потік стандартного виведення.

Позиціонування в потоці

Існує три способи позиціонування у потоці введення/виведення.

1. За допомогою функцій `ftell` і `fseek`. Вони припускають, що позиція файлу може бути представлена у вигляді довгого цілого `long`.

```
long int ftell(FILE *fp);
```

повертає поточну позицію у файлі (успішне виконання) або `-1` (помилка)

```
int fseek(FILE *fp, long offset, int whence);
```

повертає 0 (успішне виконання) або `-1` (помилка)

Поточна позиція вимірюється в байтах відносно початку файлу. Функція `ftell` повертає позицію даного байта. Щоб встановити позицію у файлі за допомогою функції `fseek`, потрібно вказати в аргументі `offset` зміщення байта і як це усунення інтерпретується за допомогою `whence`. Значення аргументу `whence` вибирається точно так, як для функції `lseek`.

2. За допомогою функцій `ftello` та `fseeko`. Вони визначені для випадків, коли довгого цілого недостатньо для представлення позиції файлу. Замість даних `long int` вони використовують тип `off_t`.

```
off_t ftello(FILE *fp);
```

повертає поточну позицію у файлі (успішне виконання) або `-1` (помилка)

```
int fseeko(FILE *fp, off_t offset, int whence);
```

повертає 0 (успішне виконання) або `-1` (помилка)

3. За допомогою функцій `fgetpos` та `fsetpos`. Для представлення позиції файлі ці функції використовують абстрактний тип даних `fpos_t`. Цей тип може бути збільшений настільки, наскільки це необхідно для представлення позиції файлу.

```
int fgetpos(FILE *restrict fp, fpos_t *restrict pos);
int fsetpos(FILE *fp, const fpos_t *pos);
```

повертають 0 (успішне виконання) або ненульове значення (помилка)

Функції форматного введення/виведення

Форматне виведення здійснюється за допомогою функцій із сімейства printf.

```
int printf(const char *format);
int fprintf(FILE *fp, const char *format);
```

повертають кількість виведених символів (успішне виконання) або від'ємне ціле (помилка)

Функція `printf` здійснює запис стандартний потік виведення, `fprintf` - в потік, заданий `fp`. Формат виведення задається за допомогою специфікатора формату `format`, який починається із символу відсотка (%). Специфікатор формату включає чотири необов'язкові компоненти, які нижче показані у квадратних дужках:

```
%[flags][fldwidth][precision][lenmodifier]convtype
```

Значення обов'язкового параметра `convtype` наведені в таблиці 4

Таблиця 4

convtype	значення
d, i	Десяткове число із знаком
o	Вісімкове число без знаку
u	Десяткове число без знаку
x	Шістнадцяткове число без знаку
f	Число з плаваючою комою
e	Число з плаваючою комою подвійної точності в експоненціальній формі
a	Число з плаваючою комою подвійної точності в шістнадцятковій експоненціальній формі
c	Символ
s	Рядок
p	Вказівник типу void
n	Вказівник на ціле число із знаком

Форматне введення здійснюється з допомогою функцій із сімейства `scanf`.

```
int scanf(const char *format);
int fscanf(FILE *fp, const char *format);
```

повертають кількість введених символів (успішне виконання) або `EOF` (помилка)

Функції сімейства `scanf` використовуються для аналізу вхідного рядка та перетворення послідовностей символів на змінні зазначених типів. Аргументи, що йдуть за рядком формату, містять адреси змінних, куди будуть записані результати перетворень. Специфікація формату `format` управляє порядком перетворення. Усі символи в рядку формату, за винятком специфікаторів формату та пробілів, повинні збігатися з символами, що вводяться. Якщо виявиться якась невідповідність, обробка введення зупиниться і решта вхідного рядка залишається непрочитаною. Специфікатор формату включає чотири необов'язкові компоненти, які нижче показані у квадратних дужках: `%[fldwidth][m][lenmodifier]convtype`. Значення обов’язкового параметра `convtype` наведені в таблиці 4.

2.3. Обробка помилок системних викликів

Майже кожен системний виклик повертає певний тип значення статусу, що вказує на те, чи виклик був успішним чи невдалим. Як ми вже зазначали вище, це значення статусу слід завжди перевіряти під час написання програм, щоб перевірити успішність викликів. Якщо цього виклик був невдалим, слід вжити відповідних заходів, наприклад, програма має вивести повідомлення про помилку, чи попередження про те, що сталося щось неочікуване. Зазвичай

більшість системних викликів повертають –1, якщо виклик був невдалим. Наприклад, розглянути нами виклик `open` можна перевірити за допомогою такого коду:

```
fd = open(pathname, flags, mode);
 if (fd == -1) {
  /* Повідомлення про помилку */
  printf("open was interrupted\n");
 }
```

Для визначення типу помилки можна використовувати глобальну цілочисельну змінну `errno`. Коли системний виклик завершується помилкою, змінна `errno` приймає додатного значення, яке визначає конкретну помилку. Змінна `errno`, та набір констант для різних номерів помилок оголошені в заголовочному файлі `errno.h`. Його треба включати в програмні заголовки для того, що мати можливість обробляти помилки за допомогою змінної `errno`. Усі назви символічних констант помилок починаються з `E` (наприклад `EINTR`). Ось простий приклад використання `errno` для діагностики помилки системного виклику `read`:

```
cnt = read(fd, buffer, numbytes);
 if (cnt == -1) {
   if (errno == EINTR)
     printf("read was interrupted\n");
}
```

Є декілька системних викликів, які повертають –1 навіть у випадку успішного завершення. Щоб визначити, чи виникає помилка під час таких викликів, можна встановити `errno` рівним 0 перед викликом, а потім перевірити його після. Якщо виклик повертає –1 і `errno` відмінне від нуля, тоді сталася помилка. Зазвичай після невдалого системного виклику потрібно вивести повідомлення про помилку на основі значення `errno`. Це можна зробити за допомогою двох функцій `perror` або `strerror`. Функція `perror`, оголошена в `stdio.h`, призначена для виведення повідомлення про помилку в стандартний потік помилок (`stderr`).

```
void perror(const char *msg);
```

Аргумент `msg` – вказівник на рядок, який буде виведено перед повідомленням про помилку. Якщо цей рядок не `NULL` і не порожній, то при виведенні після нього буде двокрапка та пробіл. Цей рядок зазвичай вказує на контекст або функцію, де сталася помилка. Ми вже декілька разів використовували цю функцію для обробки помилок в цьому розділі (розгляді викликів `read` та `close`).

Функція `strerror` (оголошена в `string.h`) повертає вказівник на рядок повідомлення про помилку, що відповідає номеру помилки, який задається аргументом `errnum`. Зазвичай `errnum` співпадає з `errno`.

```
char *strerror(int errnum);
```

повертає вказівник на рядок помилки, що відповідає аргументу

Приклад обробки помилок для виклику `fopen`, з використанням `strerror`.

```
fp = fopen("file.txt","r");
if( fp == NULL ) {
   printf("Error: %s\n", strerror(errno));
}
```

3. ФАЙЛИ ТА РЕЖИМ ДОСТУПУ ДО НИХ

У попередньому розділі ми розглянули функції, що виконують операції файлового введення/виведення. Під час розгляду механізмів створення файлів за допомогою функцій ядра Linux `creat` та `open`, ми бачили, важливе значення має такий атрибут файлу, як режим доступу `mode_t mode`. В цьому розділі розглянемо детальніше цей атрибут. На відміну від Windows, UNIX-подібні ОС дуже чітко розмежовують права доступу до файлів (або режим доступу файлу) для різних користувачів. Причому, це робиться вже з самого моменту створення нових файлів. Почнемо із розгляду функції `stat`, яка повертає детальну інформацію про файл

```
int stat(const char *pathname, struct stat *buf);
```

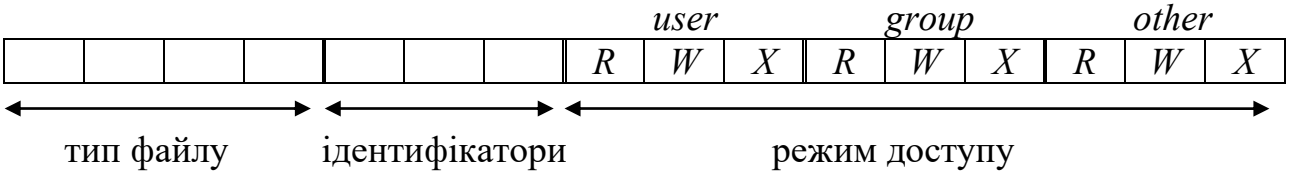
повертає 0 (успішне виконання) або -1 (помилка)

Функція `stat` повертає структуру `buf` з інформацією про файл, абсолютний шлях до якого, заданий в аргументі `pathname`.

Вигляд структури `buf` виглядає наступним чином:

```
struct stat {
 mode_t    st_mode;  /*тип файлу і режим (права) доступу*/
 ino_t     st_ino;   /*номер індексного вузла*/
 dev_t     st_dev;   /*номер пристрою*/
 dev_t     st_rdev;  /*номер пристрою для спеціальних файлів */
 nlink_t   st_nlink; /*кількість посилань*/
 uid_t     st_uid;   /*ідентифікатор власника*/
 gid_t     st_gid;   /*ідентифікатор групи власника*/
 off_t     st_size;  /*розмір в байтах, для звичайних файлів*/
 struct    timespec st_atim; /*час останнього звернення до файлу*/
 struct    timespec st_mtim; /*час останньої зміни файлу*/
 struct    timespec st_ctim; /*час останньої зміни стану файлу*/
 blksize_t st_blksize; /*оптимальний розмір блоку в/в*/
 blkcnt_t  st_blocks;  /*кількість занятих дискових блоків*/
};
```

Перша колонка структури вказує тип даних, а друга – назву файлового атрибуту. Всі ці типи даний, так само як і функція `stat` оголошені у бібліотеці `sys/stat.h`. Інформація про тип файлу і режим доступу містяться у атрибуті `st_mode` типу `mode_t`. Це тип даних містить 16 біт, як показано на схемі нижче.



Перша група із чотирьох біт визначає тип файлу. Ця група визначає не характер інформації, що міститься в ньому (текст, зображення), а особливості

низькорівневого введення/виведення. За цим критерієм ядро поділяє всі файли на такі типи:

1) Звичайний файл (*regular file*) – ці файли призначені для зберігання даних. Бінарники та бібліотеки також належать до звичайних файлів.
2) Спеціальний файл символьного пристрою (*character device*).
3) Спеціальний файл блочного пристрою (*block device*).
4) Каталог (в Linux каталоги розглядаються спеціальний вид файлів).
5) Символічне посилання (*symbolic link*) – вказують на інші файли.
6) Канал FIFO – іменовані канали FIFO беруть участь у міжпроцесній взаємодії.
7) Сокет (*socket*) – універсальний засіб міжпроцесної взаємодії між різними ОС.

Тип файлу можна визначити за допомогою групи макросів, що подана в таблиці 5.

Таблиця 5

макрос	**тип файлу**
`S_ISREG()`	Звичайний файл
`S_ISDIR()`	Каталог
`S_ISCHR()`	Спеціальний файл символьного пристрою
`S_ISBLK()`	Спеціальний файл блочного пристрою
`S_ISFIFO()`	Канал FIFO
`S_ISLNK()`	Символічне посилання
`S_ISSOCK()`	Сокет

В якості аргументу кожному із макросів треба передавати значення поля `st_mode` структури `buf`. Наприклад

```
if (S_ISREG(buf.st_mode))
  printf("Звичайний файл ");
else if (S_ISDIR(buf.st_mode))
  printf("Каталог ");
else if (S_ISCHR(buf.st_mode))
  printf("файл символьного пристрою ");
else if (S_ISBLK(buf.st_mode))
  printf("файл блочного пристрою ");
else if (S_ISFIFO(buf.st_mode))
  printf("Канал FIFO ");
else if (S_ISLNK(buf.st_mode))
  printf("Символічне посилання ");
else if (S_ISSOCK(buf.st_mode))
  printf("Сокет ")
else
  printf("невідомий тип файлу ");
```

Друга група із трьох біт визначає ідентифікатори користувача та груп користувача, що пов’язані з запущеним процесом (процеси будуть розглядатися

детально в наступних розділах). І найважливіша для нас, третя група з останніх дев'яти бітів визначає режим доступу до файлу. Щоб розмежувати повноваження користувачів щодо файлів, в Linux реалізована концепція прав доступу, згідно з якою кожен файл має власника (*user*) і групу (*group*) Власник у файлу завжди один. Крім того, виділяють також поняття інший користувач (*other*), тобто той, хто не має прав суперкористувача (*root*), не є власником і не перебуває у зазначеній групі. Для кожної з перерахованих категорій встановлюються індивідуальні права доступу на основі трьох критеріїв.

1) Право на читання (*Read*) – надає користувачу можливість читати дані з файлу. Якщо файл є каталогом, то право на читання інтерпретується як можливість переглядати його вміст.

2) Право на запис (*Write*) – надає користувачу можливість записувати дані у файл.

3) Право на виконання (*eXecute*) – надає користувачу можливість запускати файл.

Права доступу та категорію власності файлу можна визначити за допомогою команди `ls -l`. Наприклад:

```
$ ls -l a.out
-rwxr-x---    1   user            15945    Dec 30 12:10 a.out
```

У рядку "`rwxr-x---`" перші три символи `rwx` показують права доступу власника (має всі дозволи), другі три `r-x` показують права групи (можуть лише читати і запускати), і треті три `---` показують права сторонніх користувачів (не можуть нічого роботі з даним файлом)

Під час написання програм, для завдання прав доступу використовують спеціальні іменовані константи (та їхні поєднання за допомогою побітового OR "|"), оголошені в `sys/stat.h` (Таблиця 6).

Таблиця 6

Константа	**вісімкове значення**	**значення**
S_IRUSR	0400	*user* read
S_IWUSR	0200	*user* write
S_IXUSR	0100	*user* execute
S_IRGRP	040	*group* read
S_IWGRP	020	*group* write
S_IXGRP	010	*group* execute
S_IROTH	04	*other* read
S_IWOTH	02	*other* write
S_IXOTH	01	*other* execute

Наприклад, для створення нових файлів за допомогою викликів `creat` або `open`, аргумент `mode` може бути заданий наступним чином

```
#define f_mode (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)
#define f_flag (O_WRONLY|O_CREAT|O_TRUNC)

n = open(MYFILE, f_mode);
fd = open(MYFILE, f_flag, f_mode);
```

Константа `f_mode` у цьому прикладі визначає наступний режим доступу "`rw-rw-rw-`" . Також в літературі можна зустріти випадки, коли атрибут `mode` задається за допомогою числа в вісімковій формі, наприклад 0022, 0666 чи 0200.

Режим доступу файлу, що вводиться за допомогою аргументна `mode`, може бути змінений за допомогою так званої маски процесу `umask`. `umask` – це атрибут процесу, який визначає, які біти дозволів слід завжди ***вимикати***, коли процес створює нові файли. Процес, що запускається, зазвичай використовує `umask`, який він успадковує від командної оболонки. Файли ініціалізації для більшості оболонок за замовчуванням встановлюють вісімкове значення `umask` рівне 022 ("`----w--w-`"). Це значення вказує, що дозвіл на запис завжди має бути вимкнено для групи та сторонніх користувачів. Якщо аргумент `mode` у виклику `open` заданий так, що дозволяє читання та запис для всіх користувачів, то, все одно, файл створиться з дозволами на читання та запис для власника, але лише з дозволом на читання як для групи, так і для сторонніх користувачів ("`rw-r—r--`").

Програмно маску процесу можна змінити за допомогою функції

```
mode_t umask(mode_t mask);
```

завжди повертає значення маски, що було пере викликом.

Аргумент `mask` може бути заданий як вісімковим значенням, так і набором констант із таблиці 6, що об'єднуються побітовим OR.

4. ПРОГРАМИ ТА ПРОЦЕСИ. УПРАВЛІННЯ ПРОЦЕСАМИ. КОНЦЕПЦІЯ БАГАТОЗАДАЧНОСТІ

4.1. Програми та процеси. Отримання інформації про процеси

Поряд і з файлом, процес є одною з найбільш фундаментальних абстракцій у UNIX-подібних ОС. Процес – це екземпляр виконуваної програми. Програма – це файл, що містить ряд інформації, що описує, як створити процес під час виконання. Одна програма може бути використана для побудови багатьох процесів або, навпаки, багато процесів можуть запускати ту саму програму. Можна також дати наступне визначення процесу: процес – це абстрактна сутність, визначена ядром, якій виділяються системи ресурси для виконання програми. З точки зору ядра, процес складається з пам'яті простору користувача, що містить програмний код і змінні, що використовуються цим кодом, а також ряд даних ядра структури, що зберігають інформацію про стан процесу. Інформація записана в структурах даних ядра включає різні номери ідентифікаторів (ідентифікатори), пов'язані з процесом, таблицями віртуальної пам'яті, таблицею дескрипторів файлів, інформація, що стосується доставки та обробки сигналу, поточний робочий каталог та іншої інформації.

Будь-який процес має унікальний ідентифікатор процесу, який являє собою ціле позитивне число. Існує ряд спеціальних процесів, що визначаються конкретною реалізацією. Процес з ідентифікатором 0 - це, як правило, процес-планувальник, який часто називають swapper (програма підкачування). Цьому процесу не відповідає програма на диску, оскільки він є частиною ядра і вважається системним процесом. Процес з ідентифікатором 1 це зазвичай процес init, який запускається ядром наприкінці процедури початкового завантаження. Цей процес відповідає за запуск операційної системи після завантаження ядра. Зазвичай init читає системні файли ініціалізації. Процес init ніколи не "вмирає". Процес, який запускає інший процес, називається батьківським; новий процес, є дочірнім.

Системний виклик `getpid` повертає ідентифікатор процесу, який його викликає. Цей виклик описаний у заголовочному файлі `unistd.h`.

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

повертає ідентифікатор процесу, який викликає функцію

Системний виклик `getppid()` повертає ідентифікатор батьківського процесу, по відношенню до процесу, який викликає:

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

повертає ідентифікатор батьківського процесу

Тип даних `pid_t`, який використовується для значення, що повертається `getpid` та `getppid`, є цілим типом визначеним у заголовочному файлі `sys/types.h` з метою зберігання ідентифікаторів процесу.

Для отримання інформації про процеси призначена програма `ps`, що підтримує велику кількість опцій. Деякі з цих опцій є стандартними для Unix-

подібних систем, інші залежать від конкретної реалізації. Розглянемо деякі приклади виклику програми `ps`. Якщо викликати `ps` без аргументів, то на екрані з'явиться список процесів, запущених під поточним терміном. Як що більш точно, то з'явиться таблиця, що складається з чотирьох стовбців з назвами `PID`, `TTY`, `TIME` і `CMD`. Перший стовбець (`PID`) це і є ідентифікатор процесу. Якщо викликати програму `ps` з опціями `-e` і `-H`, то можна побачити дерево процесів.

4.2.Створення та завершення процесів

Для створення нового процесу призначений системний виклик `fork`, оголошений у заголовному файлі `unistd.h` наступним чином:

```
pid_t fork (void);
```

у батьківському процесі: повертає ідентифікатор дочірнього процесу (успішне виконання) або –1 (помилка);
у дочірньому процесі: 0 (успішне виконання)

Ключовим моментом для розуміння `fork` є те, що після того, як він завершив свою роботу, існують два процеси, і, як батьківський, так і дочірній процеси продовжують виконання програми з інструкції, що слідує за викликом функції `fork`.

Дочірній процес є точною копією батьківського процесу. Два процеси виконують однаковий текст програми, але вони мають окремі копії сегментів стека, даних і кучі. Важливо, що це саме копії; батьківський і дочірній процеси не використовують спільно одні й самі області пам'яті. Дочірній стек, дані та сегменти кучі спочатку є точними копіями відповідних частин батьківської пам'яті. Після `fork` кожен процес може змінювати змінні у своєму стеку, даних і сегментах кучі, не впливаючи на інший процес. У коді програми ми можемо розрізнити два процеси за допомогою значення, яке повертає `fork`. Для батьківського процесу `fork` повертає ідентифікатор новоствореного дочірнього процесу. В загальному випадку, ніколи не можна сказати точно, який із двох процесів, дочірній чи батьківський, першим отримає управління безпосередньо після виклику функції `fork`. Це залежить від алгоритму планування ядра ОС. Саме тому, важливою є синхронізація, яка забезпечується механізмами міжпроцесної взаємодії. Вони будуть розглянуті в останніх розділах цього ресурсу. Запустивши на виконання приклад коду, що приведений нижче, можна переконатися, що батьківський процес не завжди отримує управління першим.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
pid_t status = fork();
int main (void) {
    if (status == 0)
       printf("Дочірній процес, PID= %d\n", getpid());
    else
       printf("Батьківський процесс, PID= %d\n", getpid());
 return 0;
}
```

Процес може завершитися у нормальний спосіб (не аварійно) двома шляхами. Перший це звичайний вихід із функції `main`. Ще один спосіб завершення процесу, це використання викликів `_exit` чи `exit`. Виклик `_exit` є системним в UNIX-подібних ОС

```
void _exit(int status);
```

Аргумент `status`, визначає статус завершення процесу, який доступний для батьківського процесу, коли він викликає `wait` (описаний нижче по тексту). За домовленістю, якщо статус завершення рівний 0, це означає, що процес завершився успішно. Ненульове значення статусу означає, що процес завершився не успішно. Досить часто аргументу статус передають одну із констант `EXIT_SUCCESS` або `EXIT_FAILURE`, що оголошені у `stdlib.h`. Як правило, програми не викликають `_exit` безпосередньо, а викликають функцію стандартної бібліотеки С `exit`, яка виконує декілька додаткових дії (наприклад очищенні буфера пам'яті) перед викликом `_exit`.

Функція `exit` має дуже схожий з `_exit` вигляд

```
void exit(int status);
```

Код завершення процесу `status` вважається невизначеним, якщо:

1) будь-яка з цих функцій викликана без коду завершення процесу;
2) функція `main` викликає оператор `return` без аргументу;
3) функція `main`, оголошена як `int` не повертає ціле значення.

Повернення цілого значення з функції `main` еквівалентне виклику функції `exit` з тим самим значенням. Тобто, `exit(0)` означає те саме що і `return 0`.

4.3.Очікування завершення процесів

У багатьох програмах, де батьківський процес створює дочірні процеси, для батьківського процесу корисно мати можливість контролювати дочірні процеси, щоб дізнатися, коли та як вони завершуються. Цю можливість надають системні виклики `wait` та `waitpid`. Ці функції оголошені в заголовочному файлі `sys/wait.h`. Системний виклик `wait` очікує на завершення одного з дочірніх процесів і повертає його ідентифікатор та статус завершення в буфері, на який вказує вказівник `status`.

```
pid_t wait(int *status);
```

повертає ідентифікатор дочірнього процесу, що завершився (успішне виконання) або –1 (помилка)

Системний виклик `wait` виконує наступне:

1. Якщо жоден дочірній процес викликаючого процесу ще не завершився, виклик блокується, доки один із дочірніх пристроїв не завершиться. Якщо якийсь із дочірніх процесів вже закінчив роботу до моменту виклику, `wait` негайно повертається.

2. Якщо `status` не `NULL`, повертається інформація про те, як дочірній процес завершився (ціле число, на яке вказує `status`).

Далі наведений простий приклад використанні виклику `wait`

```
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
int main() {

  pid_t status = fork();
  if (status == 0) {
    printf("Child process start\n");
    printf("Child process is over\n");
  }
  else if(status > 0){
    printf("Parent process\n");
    wait(NULL);
    printf("Parent process has been over\n");
  }

return 0;
}
```

У цій програмі спочатку виконується батьківський процес, так як після успішного виклику `fork` змінна `status` у батьківському процесі набуває значення `> 0`. Потім виклик `wait` переводить його у стан очікування. Коли батьківський процес переходить у стан очікування, дочірній процес починає роботу (в дочірньому процесі `status=0`). Після того, як дочірній процес завершився, батьківський процес продовжує виконання завдань, що стоять безпосередньо після виклику `wait`.

Системний виклик `wait` має низку обмежень, які були усунуті у `waitpid`. Якщо батьківський процес створив кілька дочірніх процесів, то за допомогою `wait` можна лише чекати завершення наступного дочірнього процесу, а не якогось конкретного. Якщо жоден дочірній процес ще не закінчився, `wait` завжди блокує батьківський процес.

Функція `waitpid` має вигляд:

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

повертає ідентифікатор дочірнього процесу, 0 (успішне виконання) або -1 (помилка)

Значення, що повертається та аргумент `status` функції `waitpid` такі самі, як і для функції `wait`. Аргумент `pid` дозволяє вибрати дочірній процес наступним чином:

- Якщо `pid` більше 0 – чекає на завершення дочірнього процесу, ідентифікатор якого дорівнює `pid`.
- Якщо `pid` дорівнює 0 – чекає будь-якого дочірнього процесу в тій самій групі процесів, що і батьківський.
- Якщо `pid` менше –1 – чекає будь-який дочірній процес, ідентифікатор групи процесу якого дорівнює абсолютному значенню `pid`.

- Якщо `pid` дорівнює –1 – чекає будь-який дочірній процес.

Виклику `wait(*status)` еквівалентний виклику `waitpid(-1,*status,0)`.

Аргумент `options` у виклику `waitpid` може дорівнювати наступним константам:

- WCONTINUED – також поверніть інформацію про статус завершених процесів, які були завершені передачею сигналу SIGCONT.
- WUNTRACED – окрім повернення інформації про завершений процес, також повертає інформацію, коли процес зупиняють за сигналом. WNOHANG – якщо жоден дочірній елемент, визначений `pid`, ще не змінив стан, тоді відбувається негайне повернення замість блокування.

Значення `status`, яке повертає `wait` і `waitpid`, дозволяє розрізнити наступні події для дочірнього процесу:

- Дочірній елемент завершився викликом `_exit` (або `exit`), вказавши цілочисельний статус виходу.
- Дочірній елемент було припинено через доставку необробленого сигналу.
- Дочірній елемент був зупинений сигналом, і виклик `waitpid` з опцією `WUNTRACED`.
- Дочірній процес було відновлено за допомогою сигналу `SIGCONT`, і `waitpid` було викликано з опцією `WCONTINUED`.

Для перевірки коду завершення `status`, що повертається функціями `wait` та `waitpid` використовуються наступні макрокоманди з таблиці 7. Як приклад роботи функцій очікування, можна привести користувацьку функцію, яка перевіряє код завершення процесу.

Розглянемо ще один випадок завершення, коли дочірній процес закінчує роботу раніше батьківського. Якщо дочірній процес повністю зникне, батьківський процес не зможе отримати код завершення, коли це буде потрібно. Ядро зберігає деякий обсяг інформації про кожен процес, що завершився, щоб вона була доступна, коли батьківський процес викличе функцію `wait` або `waitpid`. У найпростішому випадку ця інформація складається з ідентифікатора процесу, коду завершення та кількості процесорного часу, витраченого процесом. Ядро може звільнити всю пам'ять, яку займає процес, і закрити його відкриті файли. У термінології Linux, процес, що завершився, але при цьому його батьківський процес не уловив цього моменту, називають *зомбі*. Команда `ps(1)` виводить у полі стану процесу-зомбі символ Z. Якщо написати довго працюючу програму, яка породжує безліч дочірніх процесів, вони будуть перетворюватися на зомбі, якщо програма не чекатиме отримання від них кодів завершення.

Таблиця 7

макрокоманда	Опис
`WIFEXITED(status)`	Повертає true, якщо код `status` отримано внаслідок нормального завершення дочірнього процесу.
`WIFSIGNALED(status)`	Повертає true, якщо код `status` отримано в результаті ненормального (аварійного) завершення дочірнього процесу через сигнал, який не було перехоплено. У цьому випадку можна дізнатися номер сигналу, що спричинив завершення дочірнього процесу: `WTERMSIG(status)`
`WIFSTOPPED(status)`	Повертає true, якщо код status отримано внаслідок зупинки дочірнього процесу за сигналом. У цьому випадку можна дізнатися номер сигналу, який викликав зупинку процесу за допомогою макросу `WSTOPSIG(status)`
`WIFCONTINUED(status)`	Повертає true, якщо код status отримано для дочірнього процесу, який продовжив роботу після зупинки

4.4. Виконання програм. Завантаження програм у процеси

Функція `fork` часто використовується для створення нового процесу, який потім запускає іншу програму за допомогою однієї із функцій сімейства `exec`. Існує сім різних функцій `exec`, які вирішують одна і те саме завдання. Далі ми будемо використовувати функцію `execve`, як найбільш загальну із всього сімейства. Коли процес викликає `execve`, він повністю заміщається іншою програмою, і ця нова програма починає виконання своєї функції `main`. Ідентифікатор процесу при цьому не змінюється, оскільки функція `execve` не створює новий процес. Вона просто замінює поточний процес – його сегмент коду, сегмент даних, динамічну область пам'яті та сегмент стеку – іншою програмою. Функцій сімейства `exec` завершують перелік механізмів UNIX-подібних ОС, призначених для управління процесами. За допомогою функції `fork` можна створювати нові процеси, за допомогою функції `execve` – запускати нові програми. Функція `exit` та функції сімейства `wait` обслуговують процедури виходу та очікування завершення. Ці механізми – все, що потрібне для управління процесами.

Найчастіше `execve` використовується в дочірньому процесі, створеному викликом `fork`.

```
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

нічого не повертає (успішне виконання) або -1 (помилка)

Аргумент `pathname` містить шлях до нової програми, яка буде завантажена в пам’ять процесу. Цей шлях може бути абсолютним (із символами “слеш” /) або відносним до поточного робочого каталогу процесу. Аргумент `argv[]` визначає аргументи командного рядка, які будуть передані новій програмі. Цей масив відповідає другому аргументу функції `main` і має

таку саму форму (список вказівників на рядки, що закінчуються NULL). Як ми вже згадували вище, значення argv[0] відповідає назві команди. Останній аргумент envp[], визначає список змінних оточення для нової програми. Аргумент envp[],це те саме, що і масив environ (список вказівників на рядки символів у вигляді ім'я=величина, закінчений NULL). Далі приведений приклад використання функції execve, яка запускає якусь нову програму new_program (що знаходиться у поточному каталозі процесу) у дочірньому процесі.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main (void) {
char *a[] = {"start","new_prorgam",NULL};
/*список аргументів для програми new_prorgam */
pid_t r = fork();
if (r == 0) {
   execve("new_program", a, NULL);
    }
return 0;
}
```

Розглянемо ще один виклик, що частково спрощує запуск команд. Функція system дозволяє програмі, що викликає, виконувати довільну команду оболонки. Ця функція дає зручний спосіб виконання команд всередині програми.

```
int system(const char *command);
```

Оскільки функція system реалізована на основі функцій fork, exec і waitpid, вона може повертати значення трьох типів.

1. –1: якщо функція fork зазнає невдачі або waitpid повертає код помилки, відмінний від EINTR.
2. exit(127): якщо функція exec повертає помилку (це означає, що командна оболонка не може бути запущена).
3. код завершення командної оболонки у форматі функції waitpid: якщо звернення до всіх трьох функцій fork, exec та waitpid закінчується успіхом.

Наступний код використовує system для запуску команди sleep. Зверніть увагу, що разом із sleep також переданий і аргумент 5. Тому виклик system допускає не тільки передачу команд, а і їх аргументів.

```
#include <stdlib.h>
#include <stdio.h>
int main (void)  {
   system ("sleep 5");
   printf ("end-of-program\n");
 return 0;
}
```

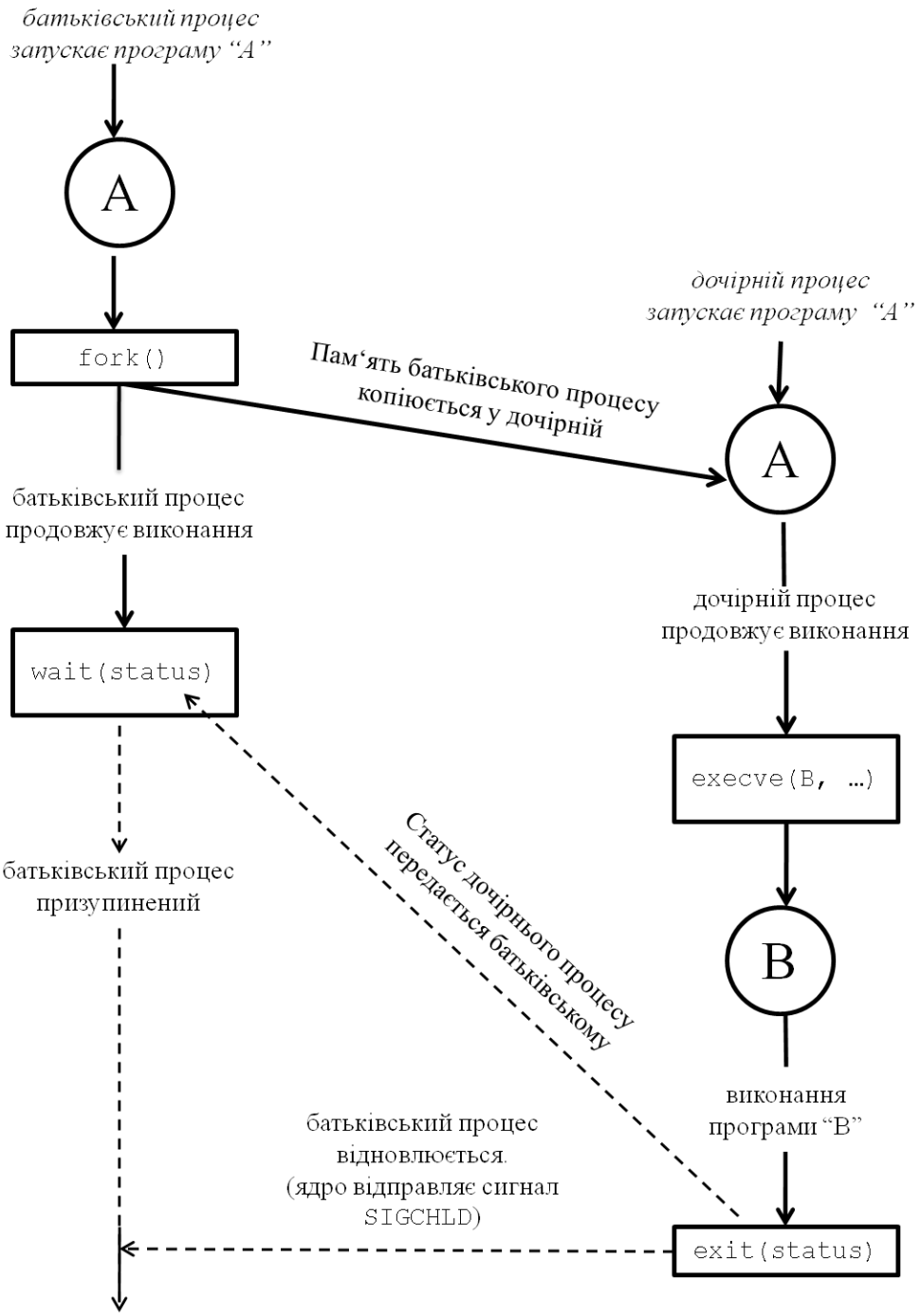


Рис.2 Схема створення дочірнього процесу за допомогою `fork()`.

5. ЛАБОРАТОРНІ РОБОТИ

Робота №1. Створення. компіляція та збірка програм. Робота з оточенням

Мета. Набути умінь та навичок компіляції та збірки програм в UNIX-подібних системах, навчитися читати змінні оточення програми

Програмне забезпечення: VitualBox, розгорнута UNIX-подібна система, набір GNU компіляторів (C/C++).

Робота має бути заархівована у ZIP-архів із ім'ям Прізвище_Lab_номер_v_номер.zip.

Компіляція та збірка

1. Створіть текстовий файл myclock.c в якому буде міститися функція для відображення поточного часу (використовувати системні функції time() і ctime()).
2. Скомпілювати файл myclock.c за допомогою gcc, та запустити скомпільований об'єктний файл.
3. Створити багатофайловий проект. Першим створити заголовочний файл print_up.h, в якому буде розмішена об'ява функції print_up(). Ця функція перетворює символи стрічки в верхній регістр і виводить результати на екран. Потім створіть файл print_up.c, в якому буде міститися сама функція print_up(). Третій файл файл main.c, буде містити функцію main (int argc, char *argv), що необхідна для компанування та запуска програми.
4. Зібрати проект. Спочатку компілюємо кожен .c файл окремо (без заголовочного .h файла). В результаті компіляції в робочій папаці проекту зявився два обєктні файли print_up.o и main.o. Потім об'єднуємо обєктні файли в один і пробуємо запустити на виконання.

Автозбірка проекту

5. Автозбірки програм за допомогою утиліти make. Створити текстовий файл під назвою Makefile, для збірки проекту попереднього пункту. Викликаемо утиліту make для збірки ($ make printup)

Змінні оточення

6. Читання змінних оточення програми. Вивести на екран оточення програми, використовуючи змінну (масив) environ (розміщенна в заголовочному файлі unistd.h).
7. Написати програму, що виводить на екран значення однієї із змінних оточення, ім'я якої читається із першого аргумента командного рядка (argv[1]). Для цього використати функцію getenv() попередньо підключивши до програми заголовочний файл stdlib.h.
8. Додати нову змінну оточення. Для цього написати програму, що читає назву нової змінної оточення і її значення з командного рядка та використовує функцію setenv() (із stdlib.h). Параметр OV функції setenv()

має бути рівним 0 для запобігання переписуванню вже існуючих змінних оточення.

9. Змінити змінну оточення PWD (поточний каталог). Для цього написати програму, що читає назву змінної оточення і її нове значення з командного рядка та використовує функцію setenv() для зміни. Параметр OV функції setenv() має бути рівним 1 можливості переписати вже існуючі змінні оточення.

Контрольні запитання.

1. Написання вихідного коду на C.
2. Компіляція і компоновка файлів.
3. Автозбірка.
4. Базовий синтаксист Makefile.
5. Змінні оточення.

Робота №2. Концепції та базові операції введення і виведення UNIX-подібних системах.

Мета. Набути умінь та навичок здійснювати базові операції введення і виведення.

Програмне забезпечення: VitualBox, розгорнута UNIX-подібна система, набір GNU компіляторів (C/C++).

Робота має бути заархівована у ZIP-архів із ім'ям Прізвище_Lab_номер_v_номер.zip.

Механізми системного введення і виведення

1. Написати програму, яка створює текстовий файл text.txt за допомого системного виклику creat(), який міститься в заголовочному файлі fcntl.h. У файлі програми підключити загловочні файли sys/types.h, sys/stat.h, які містять важливі системні константи і типи.
2. За допомогою текстового редактора записати довільний текст в створений файл. Здійснити посимвольне виведення цього файлу на екран за допомогою системного виклику open( ) та read().
3. Написати програму, що перевіряє можливість переміщення поточної позиції у файлі text.txt на задану кількість байтів. Використати виклик lseek(). Результатом має бути виведене повідомлення про можливість або неможливість переміщення.
4. Записати останні десять символів створеного текстового файлу text.txt в інший файл, використовуючи системний виклик lseek().
5. Створіть досить великий (більше 100 Мб) файл (txt або dat) з довільними даним (наприклад, з великою кількістю випадкових чисел, заповнених по 100 чисел в рядку). Організуйте копіювання даного файлу в інший файл

за допомогою системного виклику read() та write( ), використовуючи різні розміри буферу (параметр buffer у виклику read()). Запустіть програму для buffer=1; 2; 4; 8; 16; 32 …..262 144. За допомогою системної функції time() з time.h визначити час, необхідний для копіювання всіх значень buffer. Побудуйте залежність часу (ефективності) операції від величини buffer.

Функцій стандартної бібліотеки С

6. Створити невеликий текстовий файл в довільному редакторі.Написати програму, яка виведе на екран вміст заданого файл, за допомогою функцій стандартної бібліотеки С (fopen(), fprintf(), fseek(), fputc(), fclose()). Дана програма відкриває файл, що вказаний їй в якості першого параметра командного рядка.
7. Скопіювати вміст заданого файлу в інший файл: спочатку програма відкриває два файли, потім посимвольно читає вихідний файл та записує прочитане у кінцевий файл.
8. Вивести на екран вміст заданого файлу на виворіт (із кінця в початок). Програма відкриває файл, що вказаний їй в якості першого параметра командного рядка і виводить на екран вміст файлу, починаючи з останнього символу.
9. Написати програму, яка спочатку закриває стандартний потік виведення (stdout), а потім перенаправляє його на виведення у текстовий файл. В кінці програми додайте виклик printf ("Якесь повідомлення\n"). Запустіть програму і подивіться як буде працювати виклик printf.
10. Використовуючи файл, створений у пункті 4, виконати його копіювання у інший файл використовуючи спочатку посимвольне копіювання (виклики fgetc(), fputc()), а потім рядкове копіювання (виклики fgets(), fputs()). Зверніть увагу на час, необхідний на виконання операції.

Контрольні запитання.

1. Низькорівневі системні виклики
2. Механізми введення і виведення бібліотеки С.
3. Ефективність виконання операцій введення виведення

Робота №3. Керування процесами в UNIX- подібних системах.

Мета. Набути умінь та навичок роботи з процесами (створення і знищення, керування) та їх аналіз

Програмне забезпечення: VitualBox, розгорнута UNIX-подібна система, набір GNU компіляторів (C/C++).

Робота має бути заархівована у **ZIP-архівом** із ім’ям Прізвище_Lab_номер_v_номер.zip.

Функція system( )

1. Написати програму, що створює новий процес за допомогою функції system( ) (описана в заголовочному файлі stdlib.h). В якості єдиного аргументу передати функції систем аргумент “uname”.
2. Організуйте виведення на екран якогось повідомлення через 30 секунд після поточного моменту часу. Для цього використайте виклик system ("sleep 30");
3. Запустіть один із об’єктних файлів, створених в попередніх роботах, на виконання за допомогою system ( ). Для цього в аргумент system передайте місцезнаходження об’єктного файлу (наприклад “ /etc/passwd”).
4. Організуйте періодичне (щоденне) видалення в домашньому каталозі усіх файлів з розширенням *.tmp.

Отримання інформації про процеси

5. Отримати інформацію про запущені процеси в даний момент часу(використати команду ps). Представити розширену інформацію про процеси (використати команду ps з атрибутом -f). Представити інформацію про процеси у вигляді дерева (використати команду ps з атрибутом –eH, або –e --forest).
6. Вивести інформацію про процеси, що пов’язані з даним терміналом (артибут –t “ідентифікатор теорміналу” (tty1 наприклад)) та процеси, що пов’язані з даним користувачем (атрибут –u “імя користувача”)
7. Написати програму getpinfo.c, яка виводить на екран ідентифікатор процесу самої програми getpinfo. Використовувати функції getpid(), getppid(),getuid() для отримання даних про процес (функції оголошені в заголовочному файлі unistd.h). Вивести також ім’я користувача, з яким пов'язаний процес, за допомогою функції getpwuid(), що оголошена у заголовочному файлі pwd.h. (struct passwd *getpwuid (uid_t UID), де UID – числовий ідентифікатор користувача).

Аварійне завершення процесу

8. Виконати аварійне завершення одного із процесів. Запускаємо довільний процес (наприклад mc (Midnight Commander)). Дізнаємося номер ідентифікатора процесу PID (завдання 7), і завершуємо його командою kill PID. Перевіряємо всі запущені процеси і переконуємося, що завершений процес відсутній.
9. Встановити процес, що завантажує процесів найбільше. Використати команду top (для виходу із top використовуйте клавішу Q)
10. Змінити приорітет виконання програми top на найнижчий (renice –n <приоритет> -p PID, де <приоритет> -рівень приорітету (-20:19); PID – ідентифікатор процесу)

Контрольні запитання.

1. Бібліотечний C виклик system( ).
2. Ієрархія процесів та отримання інформації про процеси
3. Аварійне завершення процесу

Робота №4. Створення процесів.

Мета. Набути практичних навичок застосування системних викликів у програмах для створення процесів у UNIX-подібних системах.

Програмне забезпечення: VitualBox, розгорнута UNIX-подібна система, набір GNU компіляторів (C/C++).

Робота має бути заархівована у ZIP-архів із ім'ям Прізвище_Lab_номер_v_номер.zip.

Системний виклик fork( ).

1. Написати програму, що виводить на екран якесь повідомлення. Далі додайте в програму системний виклик fork( ) для клонування процесу, запущеного програмою. Додайте також затримувач на 30 секунд sleep(30). Запустіть програму у фоновому режимі. Виведіть не екран запущені процеси. Переконайтеся в наявності двох процесів з однаковою назвою, але різними ідентифікаторами.
2. Модифікуйте програму таким чином, щоб батьківський та дочірній процеси виводили на екран різні повідомлення. Для цього використайте вивід функції fork( ) (тип змінної pid_t описаний в файлі unistd.h): fork( ) повертає в поточному процесі ідентифікатор (PID) клонованого (дочірного) процесу (або –1, коли виникає помилка), а в дочірній процес повертає 0. Зверніть увагу на черговість виводів обох процесів. Запустіть програму знову. Знову відмітьте черговість виводів.
3. Зробіть ще одну модифікацію програми, вклавши в неї циклічний вивід повідомлень батьківського і дочірнього процесів (наприклад організуйте 10 виводів.). Знову відмічайте черговість отриманих на екрані результатів.

Системні виклики сімейства функцій exec() (*В стандартній бібліотеці С передбачено шість викликів сімейства exec(). Для виконання завдань підійде будь-який з них, але рекомендуємо використовувати execve( ), як найбільш загальний*)

4. Написати програму, що запускає іншу програму, наприклад переглядач вмісту катологу ls. Для цього використовуємо виклик fork( ), який створює дочірній процес, а також execve( ), який вказує на запуск ls в доцірньому процесі (для визначення дочірнього процесу використовуємо вивід функції fork( )).

Організація послідовності процесів

5. Напишіть програму, яка створює дочірній процес за допомогою fork. Батьківський процес виводить повідомлення, шо він почав виконуватися. Далі, за допомогою виклику wait() передаємо виконання на дочірній процес (в дочірньому процесі також виводимо кілька повідомлень, щоб

розуміти, що від працює). Після виклику wait() в батьківському процесі, виводимо повідомлення, що довічній процес завершився.

6. В попередній програмі замість виклику wait() записати виклик exit() завершення процесу без виконання всієї іншої частини коду. Порівняти виведення на екрані з попереднім запуском програми. Додати exit() в самий початок дочірнього процесу. Знову відмітити відмінності у виведенні на екран.
7. Напишіть програму, яка створює дочірній процес за допомогою fork. Оголосіть якусь числову змінну і надайте їй довільного значення. В дочірньому процесі виконайте якусь операцію із цією змінною (наприклад ++), і виведіть її значення на екран. У батьківському процесі поставте виклик wait() і після нього виведіть значення цієї змінної. Порівняйте значення виводів.

Контрольні запитання.

1. Системний виклик fork( ).
2. Системні виклики сімейства функцій exec()
3. Організація послідовності процесів, системі виклики wait() та exit()

СПИСОК ЛІТЕРАТУРИ

1. Love R. Linux System Programming : Talking Directly to the Kernel and C Library. 2nd ed. Sebastopol : O'Reilly Media, 2020. 456 p.
2. Kerrisk M. The Linux Programming Interface : A Linux and UNIX System Programming Handbook. Updated ed. San Francisco : No Starch Press, 2021. 1552 p.
3. Ward B. How Linux Works : What Every Superuser Should Know. 3rd ed. San Francisco : No Starch Press, 2021. 464 p.
4. Billimoria K. N. Linux Kernel Programming. 2nd ed. Birmingham : Packt Publishing, 2021. 814 p.
5. Molloy D. Hands-On System Programming with Linux : Explore Linux System Programming Interfaces and the Linux API. Birmingham : Packt Publishing, 2021. 684 p.
6. Graziano C. Linux Device Driver Development : Everything You Need to Know about Writing Drivers for Embedded Linux Systems. Birmingham : Packt Publishing, 2022. 518 p.
7. Negus C. Linux Bible. 10th ed. Indianapolis : Wiley Publishing, 2022. 944 p.
8. Blum R., Bresnahan C. Linux Command Line and Shell Scripting Bible. 4th ed. Indianapolis : Wiley Publishing, 2020. 912 p.
9. Rice L. Learning eBPF : Programming the Linux Kernel for Enhanced Observability, Networking, and Security. Sebastopol : O'Reilly Media, 2024. 386 p.
10. Weiss S. System Programming in Linux : A Hands-On Introduction. San Francisco : No Starch Press, 2025. 1048 p.
11. Byrne L. Linux Kernel Programming for System Engineers : A Hands-On Approach. New York : Liam Byrne Publishing, 2025. 604 p.
12. Marlow K. R. Modern Linux Programming : Build, Debug, and Optimize System Software. New York : Kalen R. Marlow Publishing, 2025. 374 p.
13. Yaghmour K. Building Embedded Linux Systems. 3rd ed. Sebastopol : O'Reilly Media, 2021. 520 p.
14. Chapman S., Mitchell J. Linux for Developers : Jumpstart Your Linux Programming Skills. Birmingham : Packt Publishing, 2022. 356 p.
15. Авраменко В. С., Авраменко А. С. Основи операційних систем. Навчальний посібник. Черкаси: ЧНУ імені Богдана Хмельницького, 2018. 524 с.
16. Горбань Г. В. Операційні системи: підготовка до виконання лабораторних робіт. Миколаїв: Вид-во ЧНУ ім. Петра Могили, 2021. 148 с.
17. Мосіюк О. О., Федорчук А. Л. Операційні системи та системне програмування: навч.-метод. посіб. Житомир: Вид-во ЖДУ ім. Івана Франка, 2022. 76 с.