МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
імені І. І. МЕЧНИКОВА
ФАКУЛЬТЕТ МАТЕМАТИКИ, ФІЗИКИ
ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА МАТЕМАТИЧНОГО ЗАБЕЗПЕЧЕННЯ КОМП'ЮТЕРНИХ СИСТЕМ

СИСТЕМНЕ ПРОГРАМУВАННЯ

Методичні вказівки
до виконання лабораторних робіт
здобувачами першого (бакалаврського) рівня вищої
освіти спеціальності 123 – Комп′ютерна інженерія

Одеса
ОЛДІ+
2023

УДК 004.451.9:004.424 072)
С409

Укладачі:
Н. Ф. Трубіна, старший викладач кафедри математичного забезпечення комп'ютерних систем;
І. М. Лісіцина, старший викладач кафедри математичного забезпечення комп'ютерних систем.

Рецензенти:
О. С. Антоненко, фізико-математичних наук, доцент кафедри математичного забезпечення комп'ютерних систем Одеського національного університету імені І. І. Мечникова;
А. В. Камєнєва, кандидат технічних наук, доцент, доцент кафедри комп'ютерних систем та технологій Одеського національного університету імені І. І. Мечникова.

Рекомендовано до друку вченою радою
факультету математики, фізики та інформаційних технологій
ОНУ імені І. І. Мечникова.
Протокол № 2 від 30.10.2023 р.

С409 **Системне** програмування : методичні вказівки до виконання лабораторних робіт здобувачами першого (бакалаврського) рівня вищої освіти, спец. 123 – Комп'ютерна інженерія / уклад.: Н. Ф. Трубіна, І. М. Лісіцина. – Одеса : Олді+, 2023. – 86 с.

Пропоновані методичні вказівки стануть у нагоді при виконанні лабораторних робіт з обов'язкової дисципліни «Системне програмування», яка викладається студентам першого (бакалаврського) рівня вищої освіти спеціальності 123 «Комп'ютерна інженерія» факультету математики, фізики та інформаційних технологій Одеського національного університету імені І. І. Мечникова.

УДК 004.451.9:004.424 072)

ЗМІСТ

ВСТУП.....5
Порядок виконання лабораторних робіт.....7
Лабораторна робота №1. Створення програм в середовищі UNIX.....8
Мета роботи.....8
Вміст роботи.....8
Теоретичні відомості.....8
Завдання до лабораторної роботи.....14
Контрольні питання.....15
Лабораторна робота №2. Автоматизація пакетних завдань.....16
Мета роботи.....16
Вміст роботи.....16
Теоретичні відомості.....16
Завдання до лабораторної роботи.....26
Контрольні питання.....28
Лабораторна робота №3 Зв'язок з операційною системою. Змінні середовища. Програмування системних утиліт.....29
Мета роботи.....29
Вміст роботи.....29
Теоретичні відомості.....29
Завдання до лабораторної роботи.....37
Варіанти завдань.....39
Контрольні питання.....40
Лабораторна робота № 4. Використання системних викликів роботи з файлами.....41
Мета роботи.....41
Вміст роботи.....41
Теоретичні відомості.....41
Варіанти завдань.....48
Контрольні питання.....49

Лабораторна робота № 5. Програмування обходу файлового дерева операційної системи 50
Мета роботи 50
Вміст роботи 50
Теоретичні відомості.....50
Варіанти завдань.....54
Контрольні питання.....55
Лабораторна робота № 6. Обробка переривань та сигналів 56
Мета роботи 56
Вміст роботи 56
Теоретичні відомості.....56
Варіанти завдань.....71
Контрольні питання.....72
Лабораторна робота № 7. Створення процеса-демону 73
Мета роботи 73
Вміст роботи 73
Теоретичні відомості.....73
Варіанти завдань.....77
Контрольні питання.....78
Рекомендована література.....79
ДОДАТОК А Тексти прикладів до лабораторной роботи №1 80
ДОДАТОК Б Функції, рекомендовані для використання в лабораторній роботі №3 82
Додаток В Сигнали 84

ВСТУП

Навчальна дисципліна «Системне програмування» належить до циклу підготовки бакалаврів з напряму 123 «Комп'ютерна інженерія». Дисципліна викладається на другому курсі. Для вивчення курсу цієї дисципліни відведено 5 кредитів (32 години лекційних та 48 годин лабораторних занять). Формою контролю є іспит.

Дисципліна «Системне програмування» орієнтована на вивчення основних принципів створення системних програм.

Мета курсу — отримання як теоретичних знань, так і практичних навичок в створенні та використанні компонент системного програмного забезпечення, що повинне забезпечити вміння оперувати з простими і складними структурами даних, користуватися стандартними системними засобами вводу та виводу, керувати розподілом пам'яті, файлами, процесами, здійснювати захист інформації від несанкціонованого доступу.

Завданнями дисципліни є:

- отримання теоретичних знань і практичних навичок, достатніх для проектування и програмування системного програмного забезпечення сучасних комп'ютерів;
- отримання теоретичних знань и практичних навичок, необхідних для експлуатації операційних систем;
- вивчення і реалізація основних алгоритмів, покладених в основу операційних систем.

Оскільки різні операційні системи відрізняються як внутрішньою архітектурою, так і способами взаємодії з апаратним та програмним забезпеченням, то принципи системного програмування для різних операційних систем є відмінними. Даний лабораторний курс призначений для вивчення особливостей програмування у середовищі операційної системи *UNIX*, яка застосовується сьогодні практично в усіх сферах інформаційних систем.

У результаті вивчення навчальної дисципліни студент повинен

ЗНАТИ:

- засоби та задачі системного програмування;
- програмний інтерфейс операційних систем;
- використання і програмування стандартних функцій мов програмування та інтерфейсу системних викликів;
- мови високого рівня в системному програмуванні;

- програмування типових елементів системних програм;
- організацію взаємодії між процесами;
- засоби створення системних служб.

ВМІТИ:

- складати, відлагоджувати і виконувати системну програму в середовищі операційної системи UNIX, проаналізувати результат її виконання;
- використовувати інтерфейс системних викликів та стандартну бібліотеку С для складання програм;
- вирішувати проблеми, пов'язані із взаємодією між процесами та захистом даних;
- організовувати роботу програми у вигляді системної служби.

Методичні вказівки містять опис 7 лабораторних робіт. Кожна з наведених робіт містить короткі теоретичні відомості, питання для самоконтролю та варіанти індивідуальних завдань для виконання їх на комп'ютері.

ПОРЯДОК ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ

При проведенні лабораторних робіт студент повинен показати творчий підхід до розробки модулів програмного забезпечення, грамотне використання існуючого програмного забезпечення навички програмування на мові *C*.

Завдання передбачають створення програми, яка написана з використанням системних викликів операційної системи *UNIX*.

Виконання роботи включає наступні етапи:

1) Підготовчий етап (до проведення лабораторної роботи):
    а) отримання завдання згідно номера варіанту і вимог викладача;
    б) вивчення теоретичного матеріалу по темі лабораторної роботи;
    в) розробка алгоритму програми;
2) Безпосереднє створення програми в комп'ютерному класі:
    а) проходження допуску до лабораторної роботи;
    б) написання програми;
    в) налагодження і тестування програми.
3) Виконання звіту і захист лабораторної роботи.

Звіт повинен містити:

- титульний лист з найменуванням лабораторної роботи і даними виконавця;
- мета роботи;
- завдання;
- опис алгоритму вирішення поставленого завдання;
- текст програми з коментарями;
- результати роботи програми і їх аналіз;
- висновки.

Допускається представлення звіту в електронній формі.

ЛАБОРАТОРНА РОБОТА №1.
СТВОРЕННЯ ПРОГРАМ В СЕРЕДОВИЩІ UNIX

Мета роботи

Отримання базових навичок роботи з компілятором *GCC*.

Вміст роботи

- ознайомитися з призначенням, можливостями та особливостями роботи з набором компіляторів *GCC*;
- навчитися створювати і використовувати статичні і динамічні бібліотеки;
- ознайомитися з утилітами для роботи з об'єктними та виконуваними файлами;
- відповісти на контрольні питання.

Теоретичні відомості

В UNIX-системах для компіляції програм традиційно використовується компілятор *GCC*, вільно доступний компілятор для мов *C*, *C++*, *Ada* 95, *Java*, *Objective–C*, *Fortran* і *Chill*. Його версії існують для різних реалізацій ОС UNIX (а також VMS, OS/2 та інших систем), і дозволяють генерувати код для великої кількості процесорів.

Компілятор *GCC* можна використовувати як для компіляції програм в об'єктні модулі, так і для компонування об'єктних модулів в єдину програму. Компілятор здатний аналізувати імена файлів, що передаються йому в якості аргументів, і визначати, які дії необхідно виконати. Файли з іменами типу *name.c* розглядаються, як файли на мові С, а файли виду *name.o* вважаються об'єктним представленням.

Для компіляції програм на мові *C* є команда *gcc*, а для програм на мові *C++* – команда *g++*. Синтаксис виклику в них однаковий.

Для компіляції простої програми, що складається з одного модулю можна використовувати команди вигляду:

```
gcc <ім'я вхідного файлу>,
g++ <ім'я вхідного файлу>,
```

в результаті чого виконується компіляція, а потім компоновка.

Якщо процес компіляції пройде успішно, то результатом буде виконуваний файл *a.out*, який можна запустити командою:

```
./a.out
```

При компіляції можна задати назву вихідного файлу, відмінну від *a.out*, виконавши команду с ключом *–o*:

```
gcc - o <ім'я вихідного файлу> <ім'я вхідного файлу>
```

У випадку програм з декількох модулів можна компілювати усі файли разом командою:

```
gcc –o <ім'я виконуваного файлу> <список вхідних файлів>
```

Наприклад,

```
gcc main.c test.c
```

або

```
gcc main.c test.c -o result
```

Тепер розглянемо, що робить програма *gcc*. Її робота включає чотири етапи:

- ***Обробка препроцесором***: Створення коду, в якому немає директив, тобто коду на «чистому» *C*.
- ***Компіляція***: Проводяться лексичний та синтаксичний аналіз коду на «чистому» *C,* результатом є код на асемблері.
- ***Асемблювання:*** Код на асемблері перетворюється в один або декілька об'єктних файлів, які містять коди команд.
- ***Компоновка***: Трансформує об'єктні файли в виконувані.

Компілятор *gcc* надає опції для зупинення на будь-якій стадії обробки. Для цього призначені опції:

-c – Зупиняє по закінченню асемблювання. Результатом є об'єктний файл. За замовчуванням ім'я об'єктного файлу будується з імені вхідного заміною суфікса на суфікс *.o*.

-E – Зупиняє по закінченню роботи препроцесора. Результат поміщається в стандартний потік виведення. Щоб помістити результат в файл використовується опція *–o* <ім'я файлу>. Файли, що містять результат обробки препроцесором повинні мати суфікс *.i*

-S – Зупиняє після компіляції. Результатом є код на асемблері. За замовчуванням ім'я файлу результату будується з імені вхідного заміною. суфікса на суфікс *.s*.

Приклад:

1. Виведення результату роботи препроцесора на екран:
   ```
   gcc -E test.c
   ```
2. Виведення результату роботи препроцесора в файл:
   ```
   gcc -E –o test.i test.c
   ```
3. Отримання коду модуля на асемблері:
   ```
   gcc -S test.c
   ```
4. Компіляція програми з двох модулів *main.c* і *test.c* з метою отримати два об'єктних файли *main.o* та *test.o*:
   ```
   gcc –c main.c test.c
   ```
5. Компоновка програми з двох об'єктних файлів *main.o* і *test.o*:
   ```
   gcc –o result main.o test.o
   ```

Під оптимізацією розуміється процес поліпшення продуктивності програми: зменшення обсягу виконуваного файлу, а також скорочення часу роботи програми. Для цього в компіляторі присутні наступні ключі (шаблон – *O*<рівень>, де рівень змінюється від 0 до 3):

– *O*0 – вимикає оптимізацію, основна мета – забезпечити високу швидкість компіляції, а також передбачуваність результатів налагодження. Цей ключ використовується за замовчуванням;

– *O*1 – м'яка оптимізація, незначне збільшення швидкості процесу компіляції, а також зменшення розміру цільової програми;

– *O*2 – використовує практично всі доступні компілятору методи оптимізації, як швидкості виконання, так і розміру програми;

– *O*3 – максимальна оптимізація, що включає такі методи оптимізації як розгортка циклів то автоматичне вбудовування функцій.

– *Os* – дозволяє оптимізувати розмір програми.

Бібліотеки

Бібліотека об'єктних файлів – це файл, що містить декілька об'єктних файлів, які будуть використовуватися разом на стадії компоновки програми. Окрім файлів бібліотека містить довідкову інформацію, необхідну для їх супроводу.

Об'єктні бібліотеки по способу використання поділяються на два види:

1) статичні бібліотеки;

2) динамічні бібліотеки.

Для створення статичних бібліотек існує спеціальна проста програма *ar* (від *archiver* – архіватор). Вона використовується для створення, модифікації і перегляду об'єктних файлів в статичних бібліотеках, які в дійсності являють собою прості архіви.

```
ar <опції> lib<ім'я бібліотеки>.a [<список об'єктних файлів>]
```

Опції *ar* означають наступне:

c – створити бібліотеку;

r – замінити співпадаючі об'єктні файли всередині бібліотеки зазначеними новими (в разі, якщо бібліотека існувала);

s – створити індекс всередині бібліотеки;

d – видалити файли з бібліотеки;

t – роздрукувати вміст бібліотеки;

x– витягти файли з бібліотеки;

V – друкувати на екрані всі дії програми *ar*;

u –при використанні з опцією "*r*" замінюються лише файли, версії яких в архіві відрізняються від нових.

Наприклад, команда

```
ar src libmylib.a func.o
```

бібліотеку *mylib* та розміщує в ній файл *func.o*. На екран нічого не виводиться.

Команда

```
ar rv libmylib.a func.o
```

добавляє в *libmylib* файл *func.o*, при цьому друкується відповідне повідомлення.

Команда

```
ar t libmylib.a
```

показує вміст бібліотеки.

Помітимо, що в деяких версіях утиліт *ar* породжуються бібліотечні файли, які не можуть безпосередньо оброблятися компоновником при створенні виконуваних файлів. Попередньо архів потрібно обробити програмою *ranlib*:

```
ranlib lib<ім'я бібліотеки>.a
```

Після цього бібліотека готова для компоновки.

Варто відзначити, що на деяких системах програма *ar* автоматично створює індекс, і використання *ranlib* не має ніякого ефекту. Незважаючи на це, утиліту *ranlib* слід запускати в будь-якому випадку.

Для підключення статичної бібліотеки необхідно при компіляції задати опцію *–L* для вказання путі до каталогу с бібліотекою (щоб добавити декілька каталогів, потрібно використовувати ключ *-L* декілька раз) і опцію *–l* для вказання імені бібліотеки, причому ім'я бібліотеки задається без префіксу *lib*. Опція *–L* обов'язково повинна передувати опції *–l*.

Приклад:

```
gcc –o result main.o –L . –l mylib
```

Створенням динамічних бібліотек займається сам компілятор *gcc*. Для створення динамічної бібліотеки потрібно використовувати ключ *-shared*:

```
gcc -shared -o <ім'я бібліотеки>.so <список об'єктних файлів>
```

Приклад:

```
gcc -shared -o lib_1d.so test2.o
```

Тепер, щоб отримати виконуваний файл с використанням динамічної бібліотеки нам потрібно виконати команду:

```
gcc -c main.c
gcc main.o -L. –l _1d -o result
```

Опції *–L* і *–l* використовуються тим же чином, що і для статичних бібліотек.

Однак цього недостатньо для повноцінної роботи, оскільки завантажувач динамічних бібліотек не зможе знайти бібліотеку, оскільки він шукає бібліотеки тільки в відомих йому каталогах, а каталог нашої програми йому не відомий.

Для того, щоб додати каталог з бібліотекою в список відомих каталогів потрібно відредагувати файл */etc/ld.so.conf*, в якому однак може знаходитись посилання на інший каталог, де і знаходяться потрібні файли для редагування. Додавши потрібний каталог, необхідно виконати команду *ldconfig*.

Іншим способом прописати путь до бібліотеки є використання змінної середовища *LD_LIBRARY_PATH*, в якій перелічуються усі каталоги, що містять динамічні бібліотеки користувачів. Для того, щоб додати до списку каталогів в цій змінній каталог /*root* в командному процесорі *bash* потрібно виконати всього дві команди:

```
LD_LIBRARY_PATH=/root:${LD_LIBRARY_PATH}
export LD_LIBRARY_PATH
```

Допоміжні інструменти для аналізу об'єктних файлів

Утиліта *nm* виводить список символів з об'єктних файлів. Для кожного символу *nm* показує:

– Значення символу, в системі числення обраної опціями, або в шістнадцятковій за замовчуванням.

– Тип символу. Завжди використовуються перелічені далі тип; інші, як правило, залежать від формату об'єктного файлу. Якщо символ написаний маленькими буквами, то він локальний. Інакше він глобальний (зовнішній).
  - *A* *Absolute* (абсолютний).
  - *B* *BSS* (не ініціалізовані дані).
  - C Common (загальний).
  - *D* Ініціалізовані дані.
  - *I* Непряме посилання.
  - *T* *Text* (код програми).
  - *U* Undefined (невизначений).
– Ім'я символу.

Утиліта *objdump* видає інформацію про один або декілька об'єктних файлів, перелічених у командному рядку. Що саме можна отримати визначається набором опцій. Наприклад, можна дизасемблювати програму (опція *–D*), показати всі заголовки програм, зокрема файлів, секцій та ін. (опція *–x*), можна показати вміст всіх секцій (опція *–s*), динамічні переміщувані дані (опція *–R*) та багато іншого.

Приклад:

```
objdump –D ./you_prog
```

Утиліта *readelf* призначена для отримання інформації по вмісту *ELF*-файлу. Дозволяє переглянути інформацію по секціям, символам, динамічним залежністям.

Приклади:

1) Прочитати заголовок файлу:

```
readelf -h ./you_prog
```

2) Прочитати інформацію про сегменти та секції:

```
readelf -l -W ./you_prog
```

3) Прочитати інформацію о символах:

```
readelf -s -W simple
```

Утиліта *size* показує розмір секцій *.text, .data і .bss* в об'єктному або виконуваному *ELF*-файлі.

Завдання до лабораторної роботи

1. Компіляція програм з одного модулю.

Відкомпілюйте та виконайте найпростішу програму на мові С (*example1a.c*). Усі лістінги програм наведені в додатку А.

Відкомпілюйте та виконайте найпростішу програму на мові *C++* (*example1b.cpp*). Порівняйте розміри отриманих виконуваних файлів.

2. Вивчення опцій, що управляють стадіями компіляції

Отримайте результат роботи препроцесору для програми *example1a.c*, помістите результат роботи в файл.

Для більше докладного вивчення роботи препроцесора розглянемо більш складний приклад (лістинг *example2.c*) с використанням директив препроцесору:

Відкомпілюйте цей файл зі зупиненням після роботи препроцесору двома способами без зазначення та з зазначенням того, що макроконстанта *HI* визначена (Використовувати опцію *–D*). Порівняйте результат.

Замініте в файле *example2.c* рядок

```
printf("%d\n", square_sum (A,B));
```

на рядок

```
printf("%d\n", square_sum (A+1,B-3))
```

Відкомпілюйте та виконайте програму. Поясніть отриманий результат. Виправіть визначення макровизначення *square_sum.*

Отримайте код на асемблері для файлів *example1a.c* і *example1b.cpp*. Порівняйте результат.

Отримайте виконуваний файл з файлу на асемблері. Порівняйте результати при використанні різних рівнів оптимізації.

Отримайте об’єктні файли *example1a.c* і *example1b.cpp*. Порівняйте розміри отриманих об’єктних файлів.

Отримайте виконувані файли з об’єктних файлів *example1a.o* і *example1b.o.* Порівняйте розміри отриманих виконуваних файлів.

3. Компоновка об’єктних файлів

Тепер розглянемо об'єднання програми з декількох модулів на простому прикладі. Програма містить два модулі: головну програму (*example3_main.c*) та реалізацію допоміжної функції (*example3_func.c*).

В цьому випадку для отримання виконуваного коду необхідно виконати наступні шаги:

- компіляція і асемблювання модулів;
- компоновка модулів в один виконуваний файл.

Відкомпілюйте модулі роздільно.

Отримайте виконуваний файл з об'єктних файлів.

4. Вивчення об'єктних і виконуваних файлів

Получить з виконуваного файлу список символів для компоновника за допомогою утілити *nm*. Знайдіть символи, що відповідають функціям з розробленої програми.

Для будь-якої функції програм получить за допомогою утілити *objdump* її бінарний файл і текст на асемблері з відповідними рядками на *C*.

За допомогою утілити *objdump*, визначити в якій секції зберігаються символьні рядки, і в якій виконуваний код програми.

Виведіть *ELF*-заголовок програми за допомогою утілити *readelf.*

Отримайте відповідні розміри секцій *.text, .data і .bss* для об'єктних і виконуваних файлів всіх прикладів.

5. Створення статичної бібліотеки і її підключення

Вивчіть опції утілити *ar.*

Помістите файл *example3_func.o* в статичну бібліотеку. Виконайте індексування бібліотеки.

Виведіть на екран вміст бібліотеки.

Отримайте виконуваний файл шляхом компоновки об'єктного файлу *example3_main.o* та статичної бібліотеки.

6. Створення динамічної бібліотеки і її підключення

Помістите файл *example3_func.o* в динамічну бібліотеку.

Отримайте виконуваний файл шляхом компоновки об'єктного файлу *example3_main.o* і динамічної бібліотеки.

Контрольні питання

1. Перелічите етапи створення програми на мові *C* в команді *gcc*?
2. Які рівні оптимізації є в *GCC*, і чим вони характеризуються?
3. Яким чином створюються динамічні бібліотеки.

ЛАБОРАТОРНА РОБОТА №2.
АВТОМАТИЗАЦІЯ ПАКЕТНИХ ЗАВДАНЬ

Мета роботи

Отримання базових навичок роботи з утилітою *make*

Вміст роботи

- ознайомитись з призначенням, можливостями та особливостями роботи з утилітою *make*;
- ознайомитися із завданням до лабораторної роботи;
- випробувати різні варіанти *make*-файлів;
- для вказаного варіанту виконати модифікацію програми на мові *C*, що реалізовує завдання;
- внести відповідні зміни у *make*-файл*;*
- виконати тестування складеної програми;
- захистити лабораторну роботу, відповівши на контрольні питання.

Теоретичні відомості

Утиліта make

Утиліта *make* традиційно використовується в ОС UNIX, причому, поряд зі стандартною, існують інші реалізації, що декілька відрізняються по можливостям та синтаксису. Найбільш відома утиліта *make* з проекту *GNU*, істотно більш розвинена, ніж стандартна.

Утиліта *make* дозволяє задати всі залежності між файлами, з яких складається програмний проект, вплив модифікацій одних файлів на інші, точну послідовність дій, необхідних для породження нової версії. Ця інформація за допомогою текстового редактора розміщується в файл з описами (так званий *make*-файл). Спираючись на вміст цього файлу, *make* визначає, які команд необхідно виконати, щоб гарантувати автоматичне формування кінцевого продукту.

Формат команди *make*:

```
make [опції] [макроозначення] [цілеві_файли]
```

При запуску програма *make* зчитує вказаний *make*-файл та виконує необхідні дії на основі його вмісту. Якщо опція *–f* не вказана, використовується ім'я за замовчуванням – *Makefile* або *makefile*, у відповідності з угодами по іменуванню таких файлів (в різних реалізаціях *make* можуть перевірятися інші файли, наприклад, *GNUmakefile*).

Структура* make*-файлу

Найпростіший *make*-файл містить синтаксичні конструкції всього двох типів: правил і макроозначень.

В *make*-файле допускаються однорядкові коментарі, які починаються символом # і діють до кінця рядка.

Загальний вид правила:

```
<ціль> ...: [<Залежність> ...][; команда]
[    команда]
. . .
```

Компоненти в квадратних дужках можна не задавати.

Ціль – це бажаний результату, спосіб досягнення якого описаний в правилі. Ціль зазвичай є ім'ям файлу, що генерується програмою *make*. Вона може також бути ім'ям деякої дії, яку потрібно виконати.

Залежність – це файл, який використовується як вхід для породження цілі. Часто ціль залежить від декількох файлів.

Команда – це дія, яку виконує програма *make*. Команди можуть бути вказані після крапки з комою в рядку залежностей, або в рядках, що починаються з табуляції та мітяться відразу за рядком залежностей. Правило може мати більш ніж одну команду – кожну в своєму рядку. Кожний рядок, що містить команди, повинен починатися з символу табуляції. Для екранування кінця рядка застосовується символ "\".

Правило

```
rm main.o func.o print.o \
program
```

еквівалентно правилу:

```
rm main.o func.o print.o program
```

Припускаються правила, які містять команду без зазначення залежностей. Наприклад, можна створити правило *clean*, що видаляє об'єктні файли проекту.

Декілька цілей в одному правилі вказуються виключно з метою скорочення запису.

Правило

```
x.o y.o: defs.h
```

еквівалентно парі:

```
x.o: defs.h
y.o: defs.h
```

Приклад простого *make*-файлу

```
#Програма містить три модулі: main.c, func.c, print.c
#Всі модулі використовують заголовний файл defs.h
#Результуюча програма має ім'я program

#Правило для трансляцаї програми

program: main.o func.o print.o
        gcc –o program main.o func.o print.o

#Правила для отримання об'єктних файлів
main.o: main.c defs.h
        gcc -c main.c

func.o: func.c defs.h
        gcc -c func.c

print.o: print.c defs.h
        gcc -c print.c

#Видалення файлів, що створюються утилитою make
clean:
        rm main.o func.o print.o \
           program
```

Спрощення *make*-файлу за допомогою вбудованих неявних правил

Неявні правила (*implicit rules*) вказують утиліті *make* на деякі "стандартні" прийоми обробки файлів, щоб користувач міг використовувати їх, не займаючись кожний раз детальним описом способу обробки. Так, наприклад, існує неявне правило для компіляції вхідних файлів на мові *C*. Питання про запуск тих чи інших правил вирішується виходячи з імен файлів, що обробляються. Наприклад, при компіляції програм на *C*, з "вхідного" файлу з суфіксом «*.c*» виходить файл з тим же ім'ям та суфіксом «*.o*». Неявне правило дозволяє не задавати команду компіляції для отримання цілі в подобних випадках. Тобто, можна не задавати явно команди компіляції в правилах, що

описують побудову об’єктних файлів. Крім того файл з суфіксом «*.c*» автоматично включається в список залежностей "свого" об’єктного файлу. Отже, файли с суфіксом «*.c*» можна видалити з списків залежностей об’єктних файлів.

Приклад *make*-файлу с використанням неявних правил

```
#Програма містить три модулі: main.c, func.c, print.c
#Всі модулі використовують заголовний файл defs.h
#Результуча програма має ім’я program

#Правило для трансляції програми
program: main.o func.o print.o
        gcc –o program main.o func.o print.o

#Правило для отримання об’єктних файлів
#Оскільки в правилі не вказана команда, всі правила
#для отримання об’єктних файлів об’єднані в одно
main.o func.o print.o: defs.h
# за замовчуванням кожен з модулів
# буде компилюватися с ключом -c

#Видалення файлів, що створюються утилитою make
clean:
        rm main.o func.o print.o program
```

Макрозначення (макрос)

Загальний вид:

```
<змінна> = <значення>
```

Значення може бути довільною послідовністю символів, яка може включати пропуски та звернення к значенням *make*-файлу.

Приклад:

```
2 = aaaa
abc = -ll -ly -lm
LIBES =
```

Макрозначення можна не тільки включати в файл описів, але і задавати в якості аргументів командного рядка, Наприклад:

```
make abc="-ll -ly -lm"
```

Всі змінні оточення (HOME, LOGNAME, TERM, PATH і т. д.) також обробляються як макроозначення.

При зверненні до макросу перед його ім’ям вказується символ $. Імена макросів, що складаються більше ніж з одного символу, потрібно брати в дужки.

Приклад коректних звернень к макросам (два останніх рядка еквівалентні):

```
$(LIBES)
$2
$(2)
```

Деякі макроозначення є вбудованими:

```
MAKE=make
CFCLAGS=-0
LDFLAGS=
CC=gcc
```

Різні способи визначення макросу мають різні пріоритети. Наведемо їх в порядку збільшення пріоритету:

- вбудовані макроозначення;
- змінні оточення;
- макроозначення в *make*-файлі;
- макроозначення в командному рядку.

Таким чином, вбудований макрос *CC* можна перевизначити в *make*-файле, а макрос, що визначений в *make*-файлі, можна перевизначити в командного рядку.

Макроси мають одне і теж значення у всіх частинах *make*-файлу. Його не можна перевизначити для частини правил. Порядок макрозначень, як і порядок правил, не грає ніякої ролі.

Приклад *make*-файлу використанням макроозначень

```
#Програма містить три модулі: main.c, func.c, print.c
#Всі модулі використовують заголовний файл defs.h
#Результуюча програма має ім'я program

OBJS = main.o func.o print.o
TARGET = program

#Правило для трансляції програми
$(TARGET): $(OBJS)
        $(CC) -o $(TARGET) $(OBJS)

#Правила для отримання об'єктних файлів
$(OBJS): defs.h

#Видалення файлів, що створюються утилитою make
clean:
        rm $(OBJS) $(TARGET)
```

Автоматичні змінні

Змінні зі спеціальними іменами "автоматично" приймають певні значення перед виконанням команд правила (значення таких змінних діють тільки у тілі правила). Автоматичні змінні можна використовувати для уніфікації запису правил

Найчастіше користуються наступними правилами:

$@ – повне ім'я поточної цілі;

$< – ім'я першої залежності правила;

$^ – список залежностей правила.

$? – список залежностей правила, які є "більш новими", ніж ціль.

Так, наприклад, правило:

```
program: $(OBJS)
        $(CC) -o prog $(OBJS)
```

з використанням автоматичних змінних можна записати наступним чином:

```
prog: $(OBJS)
        $(CC) $^ -o $@
```

Приклад *make-файлу* з використанням автоматичних змінних

```
##Програма містить три модулі: main.c, func.c, print.c
#Всі модулі використовують заголовний файл defs.h
#Результуча програма має ім'я program
OBJS = main.o func.o print.o
TARGET = program

#Правило для трансляії програми
$(TARGET): $(OBJS)
        $(CC) $^ -o $@

#Правила для отримання об'єктних файлів
$(OBJS): defs.h

#Видалення файлів, що створюються утилитою make
clean:
        rm $(OBJS) $(TARGET)
```

Абстрактні цілі та імена файлів

Імена дій, що не є іменами файлів, утиліта *make* відрізняє так: спочатку виконується пошук файлу з ім'ям дії, якщо файл знайдено, вважається, що ціль або залежність є ім'ям файлу; інакше вважається, що ім'я є ім'ям неіснуючого файлу або ім'ям дії (ці два варіанти обробляються однаково).

Такий підхід має ряд недоліків:

1) утиліта *make* не раціонально витрачає час, виконуючи пошук неіснуючих імен файлів, які насправді є іменами дій;

2) імена дій не повинні співпадати з іменами будь-яких файлів або каталогів, інакше *make*-файл буде виконуватися помилково.

Деякі версії *make* мають свої варіант рішення цієї проблеми. Наприклад, в *GNU make* є спеціальна ціль *.PHONY*, за допомогою якої можна вказати, що дане ім'я є ім'ям дії.

В нашому випадку в попередній приклад потрібно додати рядок

```
.PHONY: clean
```

Пошук залежностей по каталогам

Для великих систем буває корисним зберігати бінарні файли програми і файли с її вхідними текстами в різних каталогах.

Утиліта *make* може реалізувати таку методики за допомогою механізму автоматичного пошуку залежностей по каталогам.

Змінна *VPATH*: список каталогів для пошуку залежностей

Значення змінної *VPATH* вказує утиліті *make* список каталогів, в яких належить виконувати пошук файлів. Зазвичай, цей путь є переліком каталогів с файлами, які є залежностями якихось правил і знаходяться не в поточному каталозі.

Якщо файл, який є ціллю або залежністю, не знайдено в поточному каталозі, *make* буде шукати його в каталогах, що перелічені в змінній *VPATH.* Тому в правилах імена залежностей записуються так, ніби вони знаходяться в поточному каталозі.

В змінній *VPATH* каталоги відокремлюються один від одного пробілами або двокрапкою. При пошуку, *make* перебирає каталоги в порядку їх слідування в змінної *VPATH.*

Наприклад, запис:

```
VPATH = src:../headers
```

вказує , що путь пошуку складається з двох каталогів: *src* і *../headers.*

Директива *vpath*

Директива *vpath* дозволяє задати путь пошуку для деякої групи файлів, а саме, файлів, імена яких відповідають певному шаблону. Таким чином, можна задати деякий список каталогів пошуку для однієї групи файлів, і зовсім інший список – для інших файлів.

Існують три форми запису директиви *vpath*:

1. Задати путь пошуку *каталоги* для файлів, імена яких відповідають шаблону. Імена каталогів в списку відокремлюються двокрапками або пробілами.
   ```
   vpath <шаблон> <список каталогів>
   ```
2. Очистити путі пошуку для шаблона.
   ```
   vpath <шаблон>
   ```
3. Очистити всі путі пошуку, що були задані директивою *vpath*
   ```
   vpath
   ```

Шаблон – це рядок, який містить символ «%». Ім'я файлу повинне відповідати цьому шаблону, причому символ «%» означає будь-яку послідовність символів (в тому числі і пусту). Наприклад, шаблону «*%.h*» задовольняють імена файлів з суфіксом .h. Шаблон, шо не містить символу «%» означає ім'я конкретного файлу. Спеціальне значення символу «%» в шаблоні може бути відмінено за допомогою передуючого йому символу «\».

Якщо залежності немає в поточному каталозі, а її ім'я задовольняє шаблону директиви *vpath,* робиться спроба знайти його в каталогах, що вказані в цій директиві. Пошук проходить аналогічно пошуку в каталогах, що перелічені в змінної *VPATH*.

Наприклад, запис

```
vpath %.h ../headers
```

інструктує *make* проводити пошук залежностей с суфіксом *.h* в каталозі *../headers*, якщо вони не можуть бути знайдені в поточному каталозі.

Якщо ім'я залежності підходить відразу під декілька шаблонів, що вказані в директивах *vpath*, то *make* обробляє ці директиви по черзі, проводячи пошук во всіх каталогах, що перелічені в кожній з директив. Окремі директиви *vpath* обробляються в том порядку, в якому вони розташовані в *make*-файлі.

Наприклад, в випадку:

```
vpath %.c src
vpath %  bin
vpath %.c mysrc
```

пошук файлу с суфіксом «*.c*» буде проводитися в каталозі *src*, потім *bin*, та, нарешті *mysrc*, а в випадку

```
vpath %.c src: mysrc
vpath %   bin
```

пошук такого файлу буде проводиться в каталозі *src*, потім *mysrc* і потім *bin*.

Процедура пошуку по каталогам

Якщо файл, що є залежністю, знайдено за допомогою пошуку в каталогах, знайдений путь к цьому файлу не завжди буде присутній в імені залежності. В деяких випадках цей путь "відкидається" і не використовується. При цьому *make* діє за наступним алгоритмом:

1. Якщо файл не може бути знайдено у каталозі, який вказано в *make*-файле, проводиться його пошук по каталогам.
2. Якщо пошук завершився успішно, знайдений путь запам’ятовується і цільовий файл тимчасово позначається як знайдена ціль.
3. Всі залежності даної цілі перевіряються с використанням цього же метода.
4. Після обробки всіх залежностей для цілі може знадобиться її оновлення:
    a) Якщо ціль не потрібує оновлення, то використовується каталог, який було знайдено в процесі пошуку по каталогам.
    b) Якщо ціль *потрібує* оновлення (є застарілою), то знайдений в процесі пошуку по каталогам путь *відкидається* і ціль оновлюється з використанням тільки її імені.

Інші версії *make* використовують простіший алгоритм: якщо файлу немає в поточному каталозі, але він був знайдений в процесі пошуку по каталогам, встановлений таким чином путь к цьому файлу використовується завжди, незалежно від того, потрібує ціль оновлення чи ні.

Якщо бажано саме така поведінка *make* по відношенню до деяких (або всіх) каталогів, можна використовувати змінну *GPATH*.

Для змінної *GPATH* використовується такий же синтаксис, що і для *VPATH*. Якщо "застарілий" цільовий файл був знайдений в результаті пошуку по каталогам, і знайдений путь присутній в списку *GPATH*, він не буде "відкинутий". Ціль буде обновлена саме по цьому путі.

Написання команд з урахуванням пошуку по каталогам

Той факт, що залежність була знайдена за допомогою пошуку по каталогам, ніяк не впливає на виконувані команди правила – вони будуть виконані саме в тому вигляді, як записані в *make*-файлі. Тому слідує уважно віднестись к написанню команд – файли, які є залежностями, повинні братися командами з тих каталогів, де вони були знайдені програмою *make*. Це можна зробити с використанням автоматичних змінних, таких як $^.

Часто в список залежностей попадають файли, які не потрібно передавати в виконувану команду (наприклад, заголовні файли). В такому разі можна використовувати автоматичну змінну $<:

```
VPATH = src:../include
func.o : func.c defs.h glob.h
        cc -c $(CFLAGS) $< -o $@
```

Пошук в каталогах і неявні правила

Пошук в каталогах, вказаних за допомогою *VPATH* або *vpath* виконується також і при використанні неявних правил.

Наприклад, якщо для файлу *func.o* не існує явних правил, то *make* спробує використати неявні правила, зокрема, правило, яке говорить що для отримання *func.o*, потрібно скомпілювати файл *func.c*. Якщо такого файлу немає в поточному каталозі, то *make* вживає його пошук по каталогам. Якщо файл *func.c* буде знайдений в якомусь з каталогів, то к ньому буде застосовано відповідне неявне правило для компіляції програм на мові *C*.

Командам з неявних правил "по необхідності" приходиться користуватися автоматичними змінними, отже вони використовувати імена файлів, отриманих в результаті пошуку по каталогам.

Пошук в каталогах при підключенні бібліотек

Пошук в каталогах може проводитися спеціальним образом для файлів, що є бібліотеками. Ця специфічна особливість набуває чинності для залежностей, ім'я яких має спеціальну форму *–l<ім'я>*.

Такі залежності *make* обробляє спеціальним образом, проводячи пошук файлу з ім'ям *libім'я.so* спочатку в поточному каталозі, потім в каталогах, що перелічені у відповідних директивах *vpath*, каталогах з *VPATH*, і, нарешті, в каталогах */lib, /usr/lib,* і *prefix/lib).* Якщо файл не знайдено, робиться спроба знайти файл *lib<ім'я>.a*.

Так, якщо в системі є бібліотека */usr/lib/libcurses.a* та відсутній файл */usr/lib/libcurses.so*, то в наступному прикладі:

```
program : program.c -lcurses
        cc $^ -o $@
```

виконана команда

```
cc program.c /usr/lib/libcurses.a -o program,
```

якщо *program* "старша" ніж *program.c* або */usr/lib/libcurses.a*.

Стандартні правила

До стандартних правил відносяться:

- *all* — основна задача, компіляція програм.

- *install* — копіює виконувані коди програм, бібліотеки настройки і все, що необхідно для наступного використання;
- *uninstall* — видаляє компоненти програм з системи
- *clean* — видаляє з каталогу проекту всі тимчасові та допоміжні файли.

Завдання до лабораторної роботи

1. ***Створення простого make-файлу***

Для заданої програми з трьох модулів випробувати роботу всіх варіантів *make*-файлу, що наведені в теоретичній частині.

Програма розбиває введений рядок на слова, поміщує слова в масив, виводить вміст масиву слів в стандартний файл виведення. Програма складається з трьох модулів *main.c, func.c, print.c*. Всі ці модулі використовують заголовний файл *defs.h.* Тексти модулів та заголовного файлу *defs.h* наведені в додатку А.

Заголовні файли *print.h, func.h* створити самостійно. Обов'язково використовувати стражі включення.

Створити файл с ім'ям *clean*. Виконати команду *make clean.* Усунути проблему.

2. ***Створення make-файлу с розподілом модулів по каталогам***

Перетворити проект так, щоб він відповідав наступній структурі:

```
lab2
├── bin
│       └── makefile
├── include
│       ├── defs.h
│       ├── func.h
│       └── print.h
└── src
          ├── func.c
          ├── main.c
          └── print.c
```

Зробити відповідні зміни в make-файл. Підказка: використовувати змінні *VPATH*, *GPATH* і директиву *vpath*.

3. Додавання модуля до проекту

Доповнити програму рішенням двох задач над масивом слів. Рішення оформити в вигляді функцій, помістити їх в додатковий модуль (*wordfunc.c*). Перевірку на відповідність умові обов'язково оформить в виде функції, яка в

якості аргументу отримує адрес слова і повертає 1 в випадку успіху і 0 в випадку невдачі. Додати модуль до проекту.

Варіанти завдань

У припущенні, що масив містить слова в формате <ім'я=значення>:

1. Вивести поле «ім'я» для всіх слів, у яких в імені зустрічається заданий підрядок;
2. Вивести поле «значення» для всіх слів, імена яких починаються з заданого символу;
3. Вивести поле «значення» для всіх слів, розмір імен яких дорівнює заданому числу;
4. Вивести поле «значення» для всіх слів, розмір імен яких більше заданого числа;
5. Підрахувати число слів, у яких ім'я закінчується заданим підрядком;
6. Підрахувати сумарний розмір слів, у яких ім'я починається с заданого підрядка;
7. Вивести імена всіх слів, у яких поле «значення» пусто;
8. Вивести поле «значення» для всіх слів, розмір імен яких менше заданого числа;
9. Підрахувати число слів, у яких ім'я закінчується заданим підрядком;
10. Вивести поле «значення» для всіх слів, в іменах яких є малі літери;
11. Підрахувати сумарний розмір слів, в імена яких є цифри;
12. Вивести поле «ім'я» для всіх слів, імена яких симетричні;
13. Вивести поле «значення» для всіх слів, в іменах яких літери впорядковані за алфавітом;
14. Підрахувати число слів, імена яких мають максимальний розмір;
15. Підрахувати сумарний розмір слів, в іменах яких немає повторюваних букв;
16. Вивести поле «ім'я» для всіх слів в іменах яких кожна буква входить не менше двох раз;
17. Вивести поле «значення» для всіх слів, в іменах яких немає цифр;
18. Підрахувати число слів, в іменах яких голосні букви (a, e, i, o, u) чергуються з приголосними;
19. Підрахувати сумарний розмір слів, імена яких містять найбільшу кількість голосних букв (a, e, i, o, u);
20. Вивести поле «ім'я» для всіх слів, імена яких містять найменшу кількість приголосних букв (всі латинські букви окрім a, e, i, o, u)

4. Додаткові завдання

1. Функція *split*() реалізована нераціонально (багатократне звернення до функцій виділення пам’яті). Переписати її таким чином, щоб пам'ять для масиву слів виділялась один раз.
2. Створити *make*-файл, що використовує автоматичну побудову списку залежностей. Вказівка: дослідити опції *gcc* для сумісної роботи с *make* (*-M, -MM*, …)

Контрольні питання

1. Для чого використовується утиліта *make*?
2. Як вказати, що ціль в утиліті make є абстрактною?
3. Нижче наведені найпростіші приклади *make*-файлів. Що буде виведено на екрані після виконання *make*-файлу?

   Завдання 1

   ```
   var1 = один
   var2 = $(var1) two
   var1 = три
   all:
   @echo $(var2)
   ```

   Завдання 2

   ```
   var1 = один
   var1 = $(var1) two
   all:
   @echo $(var1)
   ```

4. Неявно задане правило це:
   а) правило, що має ціль *phony*;
   б) правило, створюване автоматично відповідно до метаправила, де задана не конкретна ціль, а якийсь патерн імені цілі;
   в) правило, що використовує у командах змінні;
   г) правило, в якому задана мета та реквізити, але не задано жодних команд.

ЛАБОРАТОРНА РОБОТА №3
ЗВ'ЯЗОК З ОПЕРАЦІЙНОЮ СИСТЕМОЮ. ЗМІННІ СЕРЕДОВИЩА. ПРОГРАМУВАННЯ СИСТЕМНИХ УТИЛІТ

Мета роботи

Отримання базових навичок оформлення службових програм

Вміст роботи

- ознайомитися з способами взаємодії програми з середовищем виконання в ОС *UNIX*;
- ознайомитись з правилами формування і способами розбору командних рядків;
- ознайомитися із завданням до лабораторної роботи;
- вибрати набір системних викликів, що забезпечують рішення задачі;
- для вказаного варіанту скласти програму, що реалізовує завдання;
- виконати тестування складеної програму;
- захистити лабораторну роботу, відповівши на контрольні питання.

Теоретичні відомості

Взаємодія програми з середовищем виконання

При запуску будь-яка програма *UNIX* отримує від процесу, що її викликає, два набори даних: аргументи командного рядка і змінні середовища оточення. У програмах на мові *C* обидва набору представлено у вигляді масивів покажчиків, причому останній покажчик в кожному з масивів має значення *NULL*. Крім того, програма отримує лічильник, що містить кількість елементів в масиві аргументів.

Крім того кожна програма повинна повернути код завершення. Нульове значення означає нормальне завершення, а ненульове значення свідчить або про аварійне завершення, або про те, що програма не достигла своєї мети.

Список аргументів

Для запуску програми досить ввести її ім'я в командному рядку. Додаткові інформаційні елементи, що передаються програмі, також задаються в командному рядку і відокремлюються від імені програми та один від одного пропусками. Під пропуском розуміється пробіл або знак табуляції. Такі елементи називаються аргументами командного рядка. (Аргумент, що містить пропуск, повинен екрануватися.)

Коли програма запускається з командного рядка, список аргументів охоплює весь вміст рядка, включаючи ім'я програми і будь-які присутні аргументи.

Наприклад, викликається програма *ls*, що відображує вміст кореневого каталогу і розміри відповідних файлів:

```
%ls  -s   /
```

В даному випадку список аргументів програми *ls* складається з трьох елементів. Перший — це ім'я самої програми, вказане в командному рядку, а саме *ls*. Другий і третій елементи — аргументи командного рядка *-s* і /.

Виконання програм на мовах *C* та *C++* починається з функції *main*(). Функція *main()* дістає доступ до списку аргументів за допомогою своїх параметрів. Традиційно ця функція визначається таким чином:

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

Параметр *argc* містить кількість переданих програмі параметрів, включаючи ім'я програми. Параметр *argv* — це масив покажчиків на рядки, що містять параметри. Розмір масиву рівний *argc*.

Через *argv*[0] адресується ім'я програми, *argv*[1] вказує на аргумент програми і так далі до *argv*[*argc*-1].

Якщо програма не передбачає прийом аргументів, то *argc* і *argv* можуть бути опущені. Робота з аргументами командного рядка зводиться до перегляду параметрів *argc* і *argv*.

Наведемо приклад програми, що виводить ім'я програми, кількість переданих параметрів і значення цих параметрів.

```
#include  <stdio.h>

int main(int argc, char*  argv[])
{
  printf("The name of this program is '%s'.\n", argv[0]);
```

```
  printf("This program was invoked with %d arguments.\n",argc-
1);

  /*  Чи є хоч  один  аргумент?   */
  if (argc  >  1)
  {
    int   i ;
    printf ("The arguments are:\n"); . •
    for (i = 1; i < argc; ++i)
      printf ("  %s\n", argv[i]);
  }

  return 0;
}
```

Правила формування і способи розбору командних рядків

Першим завданням будь-якої програми, зазвичай, є інтерпретація опцій і аргументів командного рядка.

В загальному випадку командний рядок складається з:

- імені програми;
- опцій;
- аргументів опцій;
- операндів команди.

Розрізняють традиційні (короткі) опції POSIX і довгі опції GNU. Перші складаються зі знаку «мінус» і одного символу, другі – з двох знаків «мінус», після яких слідує ім’я опції. Сучасні програми зазвичай підтримують обидва типи опцій.

Стандарт POSIX описує ряд угод, яких дотримуються програми, що відповідають цьому стандарту. Тому розробники службових програм повинні куруватися наступними правилами:

1. Ім'я утиліти складається не менше ніж з двох і не більше ніж з дев'яти малих латинських букв та/або цифр.
2. Ім'я опції – це один буквено-цифровий символ. Опціям передує знак мінус. Після одного мінуса можуть розташовуватися декілька опцій без аргументів.
3. Опції відокремлені від своїх аргументів.
4. У опцій немає необов'язкових аргументів.
5. Якщо у опції декілька аргументів, вони представляються одним словом і один від одного комами або екранованими пропусками.

6. Всі опції розташовуються в командному рядку перед операндами.
7. Спеціальний елемент командного рядка --, який не є ні опцією, ні операндом, позначає кінець опцій. Всі подальші слова трактуються як операнди, навіть якщо вони починаються із знаку мінус.
8. Порядок різних опцій в командному рядку не має значення. Якщо повторюється одна опція з аргументами, останні повинні інтерпретуватися в порядку, вказаному в командному рядку.
9. Порядок інтерпретації операндів може залежати від утиліти.
10. Якщо операнд задає файл тільки для читання або запису, то знак мінус на його місці використовується для позначення стандартного введення (або стандартного виведення, якщо з контексту ясно, що специфікується вихідний файл).

Багато існуючих утиліт не слідують всім цим угодам. Головною причиною є те, що багато таких програм було створено до систематизації цих угод.

Синтаксичний аналіз аргументів досить складне завдання. Тому в системі присутні функції автоматичного розбору опцій. Для розбору аргументів у відповідності зі стандартом POSIX використовується функція *getopt*() та асоційовані з нею зовнішні змінні.

```
#include <unistd.h> /*POSIX*/
int getopt(
     int argc,           //кількість аргументив
 char *const argv[], //аргументи програми
 const char *optstring // припустимі опції
);
// повертає наступну опцію в випадку успіху,
// ? або : в випадку помилки, -1 по закінченню розбору

extern char *optarg;
extern int optind, opterr, optopt;
```

Аргументи *argc* і *argv* зазвичай передаються безпосередньо з функції *main*(). Аргумент *optstring* є рядком символів опцій. Якщо за деякою буквою в рядку слідує «двокрапка», ця опція потребує аргументу.

Для використання *getopt*() потрібно викликати в циклі з передумовою до тих пір, доки вона не поверне –1. Кожен раз, зустрівши символ опції, функція *getopt*() повертає ім'я чергової опції з числа перелічених в рядку *optstring* (якщо таке вдалося виділити). За наявності у опції аргументу покажчик на нього поміщається в змінну *optarg* з відповідним збільшенням значення *optind*.

Якщо у опції аргумент відсутній, а на першому місці в *optstring* задано двокрапку, воно і служить результатом. Якщо зустрілося ім'я опції, не представлене в *optstring*, або у опції немає аргументу, а на першому місці в *optstring* задано не двокрапку, то результатом стане знак питання.

У будь-якій з перелічених вище помилкових ситуацій в змінну *optopt* поміщається ім'я «проблемної» опції. Крім того, в стандартний протокол видається діагностичне повідомлення. Для відміни видачі слід привласнити змінній *opterr* нульове значення.

Функція *getopt*() передбачає обробку тільки традиційних опцій командного рядка і не передбачає обробку опцій GNU. Для цього випадку існує функція *getopt_long*().

Наведемо приклад використання функції *getopt*().

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
   int c;
   char* s[2] = {"Hello","Bye"};
   char nm[] = "World";

   struct globalArgs_t {
      int hellobye;   /* опція -g (-b) */
      char *name;    /* опція -n */
      int errflag;
   } globalArgs = {0, nm, 0};

   while ((c=getopt (argc,argv,":gbn:")) != -1)
      switch (c)
      {
          case 'g' : globalArgs.hellobye = 0; break;
          case 'b' : globalArgs.hellobye = 1; break;
          case 'n' : globalArgs.name = optarg; break;
          case '?' : globalArgs.errflag =1;
                     printf("Wrong option %c\n", optopt);
                     break;
          case ':' : globalArgs.errflag = 1;
                     printf("Option %c is missed\n", optopt);
                     break;
      }

   if (globalArgs.errflag == 1)
      return EXIT_FAILURE;
   printf("%s, %s", s[globalArgs.hellobye],globalArgs.name);

   if (optind < argc)
```

```
      printf("\nOther arguments:\n");
   else
   {
      for ( ; optind < argc; optind++)
          printf("%s ", argv[optind]);
       printf("\n");
   }

   return EXIT_SUCCESS;
}
```

Змінні оточення

ОС UNIX надає кожній програмі середовище виконання або, кажучи іншими словами, оточення програми. Під середовищем мається на увазі сукупність пар змінна-значення. Ці змінні називаються змінними оточення. Імена змінних оточення і їх значення є рядками. За існуючою угодою змінні оточення записуються прописними буквами.

Наведемо приклади змінних оточення:

- *USER* — містить реєстраційне ім'я поточного користувача;
- *HOME* — містить шлях до початкового каталогу поточного користувача;
- *MAILBOX* — розташування поштової скриньки,
- *PS1* — підказка першого рівня, говорить користувачу, що він може вводити нову команду;
- *PS2* — підказка другого рівня, говорить користувачу, що він введення команди не закінчив;
- *PATH* — містить розділений двокрапками список каталогів, які операційна система переглядає в пошуку викликаної програми.

Стандарт POSIX.1 для того, щоб дістати доступ до списку змінних оточення, в оголошенні функції *main* визначає ще один аргумент *envp*:

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

Стандарт ANSI C визначає лише два перші аргументи. Тому рекомендується здійснювати доступ до змінних оточення через глобальну змінну *environ*:

```
extern char **environ;
```

Наведемо приклад програми, яка виводить всі змінні оточення програми.

```
#include  <stdio.h>
extern char **environ;

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

```
{
  char** var;
  for (var = environ; *var != NULL; ++var)
       printf ("%s\n" *var);
  return 0;
}
```

Для здобуття і установки конкретних значень змінних оточення використовуються функції *getenv*() і *putenv*().

```
#include <stdlib.h>
     char *getenv(
     const char *var //ім'я змінної
);
// повертає значення змінної або NULL, якщо така не знайдена

int putenv(
     const char *string //рядок у вигляді «ім'я=значення»
);
// повертає 0 у випадку успіху,
// ненульове значення у випадку помилки
```

Для установки і скидання значень змінних оточення призначені функції *setenv*() і *unsetenv*() відповідно.

```
#include <stdlib.h>
int setenv(
     const char *var,// ім’я змінної
     const char *val,// значення
     int flag // флаг дозволу створення нової змінної
);
// повертає 0 у випадку успіху, -1 у випадку помилки

int unsetenv(
          const char *var //видаляєма змінна
)
//повертає 0 у випадку успіху, -1 у випадку помилки
```

Зазвичай при запуску програма отримує копію середовища своєї батьківської програми (інтерпретатору команд, якщо вона була запущена користувачем). Таким чином, програми, запущені з командного рядка, можуть досліджувати середовище інтерпретатора команд.

Коди завершення програми

Коли програма завершує роботу, вона повідомляє операційну систему про свій стан, посилаючи їй код завершення, який є 16-розрядним цілим числом. За існуючою угодою нульовий код свідчить про успішне завершення, а ненульовою

вказує на наявність помилки. Деякі програми повертають різні ненульові коди, позначаючи різні ситуації.

У більшості інтерпретаторів команд код завершення останньої виконаної програми міститься в спеціальній змінній $?.

Програма, написана на мові *C* або *C++* вказує код повернення в операторові *return* у функції *main* або виконує виклик функції *exit*(). Процес може бути завершений і по незалежних від нього обставинах, наприклад унаслідок отримання сигналу. В цьому випадку функція *exit*() буде викликана ядром від імені процесу.

Системний виклик *exit*() виглядає таким чином:

```
#include <unistd.h>
void exit(int status);
```

Аргумент *status* повертається батьківському процесу і є кодом повернення програми.

Наявність коду завершення дозволяє програмам взаємодіяти між собою.

Окрім передачі кодів повернення, функція *exit*() проводить ряд додаткових дій, зокрема виводить дані, що містяться в буферах, і закриває потоки введення-виведення.

Завдання може зареєструвати до 32 обробників виходу (*exit handler*), - функції, які викликаються після виклику *exit*(), але до остаточного завершення процесу. Викликаються ці обробники за принципом LIFO, причому лише при добровільному завершенні процесу.

Обробники реєструються за допомогою функції *atexit*().

Приклад:

```
#include <stdio.h>
#include <stdlib.h>
void handler1()
{
  printf("handler1\n");
}
void handler2()
{
  printf("handler2\n");
}
void handler3()
{
  printf("handler2\n");
}
int main()
{
  atexit(&handler1);
```

```
  atexit(&handler2);
  atexit(&handler3);
  return(0);
}
```

Завдання до лабораторної роботи

Загальні завдання:

1. Вивести список всіх змінних оточення процесу с їх значеннями.
2. Вивести значення змінної оточення, ім'я якої (в якості єдиного припустимого параметру) вказано в командному рядку. Вивід повідомлень про помилки організувати за допомогою обробників виходу. (Повідомлення повинні поміщатися в стандартний потік помилок *stderr*).

Індивідуальні завдання:

Написати програму, яка виводить інформацію про змінні оточення згідно з набором опцій, що визначають її поведінку.

Перелік можливих опцій

Обов'язкові опції для всіх варіантів:

–h – коротка довідка по версії програми у відповідності з варіантом;

–0 (мінус нуль) – інформація про автора (ФИО, група);

–p – виводить інформацію о тих змінних оточення, імена яких задані аргументами командного рядка (відміняє всі опції відбору змінних, якщо такі присутні).

Перелік опцій, що беруть участь в виборі варіанта

1. Опції керування виводом:

–s – виводить тільки імена змінних оточення процесу (без значень);

–v – виводить тільки значення змінних оточення процесу (без імен);

–l – виводить довжини значень змінних оточення процесу;

–a – виводить повну інформацію о змінних оточення процесу: ім'я, значення, загальна довжина (включаючи ім'я і значення), а також порядковий номер в переліку змінних оточення.

Примітка. Будь-яка з опцій *-s, -v ,-l* відміняє опцію *–a*, при цьому мається на увазі виведення інформації о тих змінних, що відповідають всім заданим

опціям. Якщо при виклику програми не вказана жодна з опцій цієї групи, то повинна бути виведена підказка (тобто, мається на увазі опція – *h* за замовчуванням).

2. Опції відбору змінних:

–x string виводить інформацію о тих змінних оточення, в іменах яких зустрічається рядок *string*;

–b string виводить інформацію о тих змінних оточення, імена яких починаються з рядка *string*;

–t string виводить інформацію о тих змінних оточення, імена яких закінчуються рядком *string*;

–c symbol виводить інформацію о тих змінних оточення, імена яких починаються с символу *symbol*;

–g number – виводить інформацію о тих змінних оточення, довжина імен яких дорівнює числу *number*;

–y number – виводить інформацію о тих змінних оточення, довжина імен яких більше числа *number*;

–z number – виводить інформацію о тих змінних оточення, довжина імен яких менше числа *number*;

–d – виводить інформацію о тих змінних оточення, імена яких включають рядкові (маленькі) букви;

–f – виводить інформацію о тих змінних оточення, імена яких включають цифри.

Якщо при виклику програми задається декілька опцій цієї групи, то мається на увазі відбір змінних, що відповідають усім заданим опціям.

3. Агрегатні опції:

–k – визначити і вивести на екран кількість відібраних змінних оточення;

–u – визначити і вивести на екран загальну довжину всіх рядків відібраних змінних оточення в байтах.

Вимоги до виконання завдання

Для вирішення завдання повинна бути написана програма з декількох модулів, примірний склад яких виглядає наступним чином:

1. Модуль, що містить головну програму;
2. Модуль, що містить функції для реалізації інформаційних опцій (-0, -*h*);

3. Модуль, що містить функції для реалізації опцій виведення на екран *(–s, –v –l –a*);
4. Модуль, що містить функції фільтрації змінних оточення (опції – *x string, –h string, –t string, –c symbol, –g number, –y, –z number, –d, – f*)
5. Модуль, що містить допоміжні функції, (Наприклад, функції виведення повідомлень про помилки).

Для кожного модуля необхідно створити заголовний файл. За допомогою директив умовної компіляції забезпечити одноразове включення файлів.

Наведемо структуру каталогу проекту:

```
lab2
├── bin
│       └── makefile
├── include
└── src
```

Варіанти завдань

1. *–s, –a, –x, –g, –d, –p, –k*
2. *–s, –v, –b, –y, –d, –p, –u*
3. *–l, –a, –t, –g, –d, –p, –u*
4. *–s, –v, –c, –g, –f, –p, –k*
5. *–s, –l, –b, –z, -f –p, –u*
6. *–v, –a, –t, –g, –d, –p, –k*
7. *–l, –a, –x, –y, –f, –p, –u*
8. *–v, –l, –c, –g, –d, –p, –k*
9. *–l, –a, –t, –g, –f, –p, –u*
10. *–s, –a, –b, –z, -f, –p, –k*
11. *–s, –l, –x, –y, -d, –p, –u*
12. *–v, –a, –c, –g, –d, –k, -p*

Порядок дій при виконанні завдання

1. Визначити структуру для зберігання результатів розбору, розмістити її в окремому заголовному файлі. Рекомендація: якщо опція не має аргумента, то додати в структуру поле типа *int* (0 – опція не задана; 1 – опція задана)
2. Визначити специфікації для всіх функцій, що викликаються з функціи *main*(), визначити, в ґких модулях вони будут розміщені.

3. Для всіх цих модулів створити заголовний та основний файли. В заголовний файл помістити прототипи функцій.
4. У файл модуля на цьому етапі помістити функції-заглушки. Розмістивсі файли в структурі каталогів, створити *make*-файл.
5. Написати головну програму.
6. Перевірити працездатість схеми.
7. Перейти до реалізації функцій.

Додаткове завдання

Розбір аргументові провести за допомогою функції *getopt_long*, додавши опції GNU на свій розсуд.

Контрольні питання

1. Яким чином в командному рядку задаються аргументи програми? Яким чином функція *main*() дістає доступ до списку аргументів?
2. Що таке змінні оточення? В якому форматі вони зберігаються?
3. Яким чином програма дістає доступ до змінних оточення?
4. Що таке обробник виходу? Як його зареєструвати?

ЛАБОРАТОРНА РОБОТА № 4. ВИКОРИСТАННЯ СИСТЕМНИХ ВИКЛИКІВ РОБОТИ З ФАЙЛАМИ

Мета роботи

Ознайомитися з файловою системою ОС UNIX і програмними засобами роботи з нею.

Вміст роботи

1. Ознайомитися з файловою системою ОС UNIX і системними викликами для роботи з файлами.
2. Ознайомитися із завданням до лабораторної роботи.
3. Вибрати набір системних викликів, що забезпечують рішення задачі.
4. Для вказаного варіанту скласти програму на мові *C*, що реалізовує завдання.
5. Виконати тестування складеної програми.
6. Захистити лабораторну роботу, відповівши на контрольні питання.

Теоретичні відомості

Інтерфейс між призначеною для користувача програмою і зовнішнім пристроєм (або між двома призначеними для користувача програмами) в ОС UNIX здійснюється в рамках єдиної структури даних - файлу ОС UNIX.

У UNIX існує декілька типів файлів, що розрізняються по функціональному призначенню і діям операційної системи при виконанні тих або інших операцій над ними.

Звичайний файл (*regular file*) є найбільш загальним типом файлів, що містить дані в деякому форматі. Будь-яка інтерпретація вмісту файлу повністю покладається на прикладну програму, що обробляє файл. Для операційної системи такі файли є просто послідовністю байтів.

За допомогою ***каталогів*** (*directory*) формується логічне дерево файлової системи. Каталог – це файл, що містить імена файлів, що знаходяться в ньому, а також покажчики на додаткову інформацію – метадані, що дозволяють операційній системі виробляти дії з цими файлами. Каталоги визначають положення файлу в дереві файлової системи. Будь–який процес, що має право на

читання каталогу, може прочитати його вміст, але лише ядро має право на запис даних каталогу.

Спеціальний файл пристрою забезпечує доступ до фізичних пристроїв. У UNIX розрізняють символьні (*character special device*) і блокові (*block special device*) файли пристроїв. Доступ до пристроїв здійснюється шляхом відкриття, читання і запису в спеціальний файл пристрою.

Символьні файли пристроїв використовуються для обміну даними з пристроєм без буферізації. Блокові файли пристроїв дозволяють виробляти обмін даними у вигляді пакетів фіксованої довжини – блоків.

Кожен файл пристрою характеризується старшим і молодшим номерами пристрою. Старший номер пристрою ідентифікує драйвер пристрою і інколи вказує, з якою платою периферійного пристрою слід взаємодіяти. Молодший номер ідентифікує конкретний пристрій.

Іменований канал (*FIFO* файл використовується для зв'язку між процесами за принципом черги.) Іменовані канали вперше з'явилися в *UNIX System* V, але більшість сучасних систем підтримують цей механізм.

Каталог містить імена файлів і покажчики на їх метадані. Така архітектура дозволяє одному файлу мати декілька імен у файловій системі. Імена жорстко пов'язані з метаданими і, відповідно, з даними файлу, тоді як сам файл існує незалежно від того, як його називають у файловій системі.

Стандарт POSIX вимагає реалізувати підтримку двох типів посилань – жорстких і символічних.

Жорстким посиланням вважається елемент каталогу, що вказує безпосередньо на деякий індексний дескриптор. Жорсткі посилання дуже ефективні, але у них існують певні обмеження, оскільки вони можуть створюватися лише в межах однієї фізичної файлової системи. Коли створюється такий зв'язок, файл повинен вже існувати.

Символічне посилання (*symbolic link*) – це спеціальний файл, який містить дорогу до іншого файлу. Вказівка на те, що даний елемент каталогу є символічним посиланням, знаходиться в індексному дескрипторі. Тому звичайні команди доступу до файлу замість здобуття даних з фізичного файлу, беруть їх з файлу, ім'я якого приведене в посиланні. Цей шлях може вказувати на що завгодно: це може бути каталог, він може навіть знаходитися в іншій фізичній файловій системі, більш того, вказаного файлу може і зовсім не бути.

Сокети (*socket*) дозволяють представити у вигляді файлу в логічній файловій системі мережеве з'єднання. Сокети можна застосовувати і для обміну інформацією між процесами на одній і тій же машині.

Кожен файл в ОС UNIX містить набір прав доступу, по якому визначається, як користувач взаємодіє з даним файлом. Цей набір зберігається в індексному дескрипторі даного файлу у вигляді цілого значення, з якого зазвичай використовується 12 бітів. Причому кожен біт використовується як перемикач, вирішуючи (значення 1) або забороняючи (значення 0) той або інший доступ.

Три перших біта встановлюють різні види поведінки при виконанні. Дев'ять бітів, що залишилися, діляться на три групи по три біта, визначаючи права доступу для власника, групи і останніх користувачів. Кожна група задає права на читання, запис і виконання.

Біт читання для всіх типів файлів має одне і те ж значення: він дозволяє читати вміст файлу.

Біт запису також має одне і те ж значення: він дозволяє редагувати цей файл. Для каталогу – це можливість міняти його вміст, тобто створювати і видаляти файли.

Якщо для деякого файлу встановлений біт виконання, то файл може виконуватися як команда. В разі установки цього біта для каталогу, цей каталог можна зробити поточним.

Встановлений біт зміни ідентифікатора користувача *SUID* означає, що доступний користувачеві на виконання файл буде виконуватися з правами (з ефективним ідентифікатором) власника, а не користувача, що викликав файл (як це зазвичай відбувається).

Встановлений біт зміни ідентифікатора групи *SGID* означає, що доступний користувачеві на виконання файл буде виконуватися з правами (з ефективним ідентифікатором) групи–власника, а не користувача, що викликав файл (як це зазвичай відбувається).

Якщо біт *SGID* встановлений для файлу, не доступного для виконання, він означає обов'язкове блокування, тобто незмінність прав доступу на читання і запис доки файл відкритий певною програмою.

Встановлений клейкий біт (*sticky*) для звичайних файлів раніше (за часів *PDP*–11) означав необхідність зберегти образ програми (тобто код і дані) в пам'яті після виконання (для прискорення повторного завантаження). Зараз при установці звичайним користувачем він скидається.

Установка клейкого біта для каталогу означає, що файл в цьому каталозі може бути видалений або перейменований лише в наступних випадках:

- користувачем–власником файлу;
- користувачем–власником каталогу;
- якщо файл доступний користувачеві на запис, користувачем *root*.

Прикладом може служити каталог */tmp*, який відкритий на запис для всіх користувачів, але, в якому небажано видаляти чужі тимчасові файли.

Стандартом *POSIX*1988 замість восьмеричних чисел для опису прав доступу були рекомендовані до вживання спеціальні ідентифікатори. Принцип їх іменування слідує шаблону *S_Ipwww*, де *p* визначає режим доступу (*r*, *w* або *x*), а *www* – кому видається право на цей режим доступу (*USR*, *GRP* або *OTH*).

Наприклад, для попереднього файлу замість восьмеричного числа 755 можна записати:

```
S_IRUSR|S_IWUSR|S_IXUSR|S_IRGRP|S_IXGRP|S_IROTH|S_IXOTH
```

Існують окремі ідентифікатори для *USR*, *GRP* і *OTH*, які описують повні права доступу. Іменування цих ідентифікаторів слідує формі *S_IRWXw*. Тут символ *w* визначає, кому видається повне право доступу до файлу (від англ., «*whom*» – «кому») — *U, G* або *O*. Таким чином, попередній приклад може бути записаний в наступному вигляді:

```
S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH
```

Використання ідентифікаторів дає розробникам ОС свободу вибору порядку дотримання бітів, що описують права доступу до файлу.

Системний виклик *open*() відкриває існуючий файл (звичайний, спеціальний або іменований канал) або створює новий, але в цьому випадку може бути створено лише звичайний файл.

```
#include <sys/stat.h>
#include <fcntl.h>
     int open(
          const char *path,   //шляхове ім'я файлу
          int flags,          //прапори
          mode_t perms        //права доступу
     );
//повертає дескриптор файлу або –1 у випадку помилки
// код помилки в змінній errno
```

Виклик *open*() привласнює файловому дескриптору найменший з доступних номерів, але, як правило, це число не важливе.

Якщо файл з вказаним ім'ям не існує, виклик *open*() створить його, за умови, що аргумент *flags* містить прапор *O_CREAT*. Прапор *O_CREAT* зазвичай доповнюється прапором *O_WRONLY* або *O_RDWR*. Крім того, в такому разі необхідно визначити права доступу до файлу, наприклад:

```
fd = open("/home/user/newfile", O_RDWR | O_CREAT, PERM_FILE);
```

Остаточний набір прав доступу до файлу, що створюється, формується як результат логічного множення аргументу *perms*, переданого в системний виклик, на логічне доповнення ***маски прав доступу*** *(маски створення файлу)*. Маска

зазвичай встановлюється під час реєстрації користувача в системі (командою *umask*) або системним викликом *umask*().

Знов створеному файлу призначаються ідентифікатор користувача–власника і ідентифікатор групи–власника, які скорочено ми далі будемо називати «власник» і «група». Правила призначення власника і групи виглядають таким чином.

- Як власник файлу призначається ефективний ідентифікатор користувача процесу.
- Як група встановлюється або ідентифікатор групи каталогу, в якому створюється файл, або ефективний ідентифікатор групи процесу.

В програми немає можливості вибрати, яка з двох груп буде привласнена файлу, але вона може взнати це за допомогою системного виклику *stat*(). Аби примусово призначити файлу потрібну групу, можна скористатися системним викликом *chown*().

Наведемо всі прапори системного виклику *open*(), які визначені в SUS3.

O_RDONLY	Відкрити лише для читання.
O_WRONLY	Відкрити лише для запису.
O_RDWR	Відкрити для читання і для запису.
O_APPEND	Відкрити для доповнення в кінець файлу.
O_CREAT	Створити новий, якщо не існує.
O_DSYNC	Встановити режим синхронного введення–виведення.
O_EXCL	Не відкривати, якщо існує.
O_NOCTTY	Не робити пристрій керуючим терміналом.
O_NONBLOCK	Режим роботи без блокування з іменованими каналами і спеціальними файлами.
O_RSYNC	Встановити режим синхронного введення–виведення.
O_SYNC	Встановити режим синхронного введення–виведення.
O_TRUNC	Видалити вміст файлу — усікти його розмір до нуля

Системний виклик *write* () виконує запис у файловий дескриптор.

```
#include <unistd.h>
ssize_t write(
   int fd,          //дескриптор
   const void *buf, //адреса буфера з даними
   size_t nbytes    //кількість байтів для запису
);
// повертає кількість записаних байтів,
// -1 у випадку помилки
// код помилки в змінній errno
```

Системний виклик *write*() записує *nbytes* байт з буфера *buf* у відкритий файл, який представлений дескриптором *fd*. Запис починається з поточної позиції у файлі, а після її закінчення поточна позиція зміщується на число записаних байт. У програму, що викликає, повертається число байт, яке було записано або –1, якщо виникла помилка. Якщо файл був відкритий з прапором *O_APPEND*, то безпосередньо перед записом поточна позиція автоматично буде переміщатися в кінець файлу.

Системний виклик *read* () виконує читання з файлового дескриптору.

```
#include <unistd.h>
ssize_t read(
   int fd,          //дескриптор
   const void *buf, //адреса буфера для даних
   size_t nbytes    //кількість байтів для читання
);
// повертає кількість прочитаних байтів,
// -1 у випадку помилки
// код помилки в змінній errno
```

Системний виклик *read*() є протилежністю виклику *write*(). Він прочитує *nbytes* байт з файлу, представленого дескриптором *fd*, і розміщує їх в буфері *buf*. Читання починається з поточної позиції у файлі після закінчення і поточна позиція зміщується на кількість прочитаних байт. У програму, що викликає, повертається число прочитаних байт, 0 (після досягнення кінця файлу), або – 1, як ознака помилки.

Існує декілька ситуацій, коли кількість фактично прочитаних байтів менша за запитану:

- при читанні із звичайного файлу, коли кінець файлу зустрівся до того, як було прочитано необхідну кількість байт;
- при читанні з термінального пристрою. Зазвичай за одне звернення зчитується один рядок;
- при читанні даних з мережі. Проміжна буферізація в мережі може стати причиною того, що буде отримане менша кількість байт, ніж було запитано;
- при читанні з іменованих або неіменованих каналів. Якщо в каналі міститься менше байт, ніж було запитано, функція *read*() поверне лише те, що їй буде доступне;
- при читанні з пристрою, орієнтованого на доступ до окремих записів. Прикладом такого пристрою є накопичувач на магнітній стрічці, який може повернути лише одну запис за одне звернення;

- при перериванні операції читання сигналом в той момент, коли частина даних вже була прочитана.

Існує обмеження на кількість файлів, які процес може мати відкритими одночасно. Відповідно до цього будь-яка програма, що збирається працювати з багатьма файлами, має бути підготовлена до повторного використання дескрипторів файлів.

Системний виклик *close* () закриває дескриптор файлу.

```
#include <unistd.h>
     int close(
          int fd;              //дескриптор
     )
//повертає 0 у випадку успіху, -1 у випадку помилки
//(код помилки – в змінній errno)
```

З будь-яким відкритим файлом пов'язано таке поняття, як ***поточна позиція файлу***. Як правило, це не невід'ємне ціле число, яким виражається кількість байт від початку файлу. Зазвичай операції читання і запису починають виконуватися з поточної позиції файлу і збільшують її значення на кількість байт, яке було прочитане або записане, тобто введення і виведення здійснюється послідовно. Але при необхідності файл може читатися або записуватися в будь-якому довільному порядку.

Системний виклик *lseek* () використовується для зміни поточної позиції у файлі. Він не виконує жодних операцій введення–виводу і не віддає команд контролеру диска. Нове значення поточної позиції буде використано найближчою операцією читання або запису.

```
#include <unistd.h>
     off_t lseek(
          int fd,              //дескриптор
          off_t pos,
          int whence
     );
//повертає нову позицію у файлі, -1 у випадку помилки
//(код помилки – в змінній errno)
```

Аргумент *whence* може приймати одне з наступних значень:

SEEK_SET Аргумент ***pos*** містить зсув від початку файлу (абсолютна позиція у файлі).

SEEK_CUR Аргумент ***pos*** містить зсув від поточної позиції у файлі. Може бути позитивним числом, нулем і негативним

	числом. Вказавши в аргументі *pos* значення 0 — ми отримаємо поточну позицію у файлі.
SEEK_END	Аргумент *pos* містить зсув від кінця файлу. Може бути позитивним числом, нулем і негативним числом. Вказавши в аргументі *pos* значення 0 — ми встановимо поточну позицію в кінець файлу

Результатом роботи виклику може бути будь–яке не невід'ємне число, що навіть перевищує розмір файлу. Якщо нова поточна позиція виявилася за межами файлу, то найближча операція запису вставить «недостатній» шматок в кінець файлу і заповнить його байтами із значенням 0.

Варіанти завдань

1. Написати програму, що копіює байти з одного файлу в іншій у зворотному порядку. Імена файлів задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах.
2. Написати програму, що додає вміст одного файлу в кінець іншого. Імена файлів задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах.
3. Написати програму, що додає вміст одного файлу в кінець іншого у зворотному порядку. Імена файлів задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах.
4. Написати програму, що підраховує кількість рядків у файлі, ім'я якого задається у вигляді аргументу командного рядка. Перевірити, чи не виникають помилки при системних викликах.
5. Написати програму, що дописує в кінець файлу його вміст у зворотному порядку. Ім'я файлу задавати у вигляді аргументу командного рядка. Перевірити, чи не виникають помилки при системних викликах.
6. Написати програму порівняння двох файлів, що буде друкувати перший з рядків, що розрізняються, і позицію символу, у якому вони розрізняються. Імена файлів задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах.
7. Написати програму, що дописує вміст файлу в його кінець. Ім'я файлу

задавати у вигляді аргументу командного рядка. Перевірити, чи не виникають помилки при системних викликах.

8. Написати програму, що копіює вміст одного файлу в інший, видаляючи всі входження даного символу. Імена файлів і видаляє символ, що, задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах.
9. Написати програму, що копіює вміст одного файлу в інший, заміняючи при цьому всі багаторазові входження пробілів знаком табуляції. Імена файлів задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах.
10. Написати програму, що змінює порядок проходження байтів у файлі на протилежний. Ім'я файлу задавати у вигляді аргументу командного рядка. Перевірити, чи не виникають помилки при системних викликах.

Контрольні питання

1. Що таке файловий дескриптор? Для чого він використовується?
2. Які файлові дескриптори вважаються стандартними?
3. Чим відрізняється системний виклик *open*() від системного виклику *creat*()?
4. Чи має смисл комбінація прапорів *O_WRONLY* | *O_EXCL* | *O_TRUNC* при відкритті файлу? Чому?
5. Чи можна в програмі декілька разів відкрити один і той же файл?
6. Що станеться, якщо маску створення файлів задати рівною 777 (у вісімковій системі)? Перевірте результати за допомогою команди *umask.*
7. Як змінюється поточна позиція в результаті запису в файл (читання з файлу)?
8. Що служить ознакою кінця файлу при читанні?

ЛАБОРАТОРНА РОБОТА № 5. ПРОГРАМУВАННЯ ОБХОДУ ФАЙЛОВОГО ДЕРЕВА ОПЕРАЦІЙНОЇ СИСТЕМИ

Мета роботи

Оволодіти програмними засобами ОС UNIX для роботи з каталогами.

Вміст роботи

1. Ознайомитись з функціями та структурами для роботи з каталогами.
2. Випробувати різні методики для створення закладок на каталоги.
3. Ознайомитися із завданням до лабораторної роботи.
4. Вибрати набір системних викликів, що забезпечують рішення задачі.
5. Для вказаного варіанту скласти програму на мові С, що реалізовує завдання.
6. Налагодити і протестувати складену програму.
7. Захистити лабораторну роботу, відповівши на контрольні питання.

Теоретичні відомості

Фактичний формат файлів каталогів залежить від реалізації UNIX і архітектура файлової системи. Це означає, що будь–яка програма, яка виконує пряме читання даних з файлу каталогу, потрапляє в залежність від конкретної реалізації. Аби спростити положення справ, був розроблений набір функцій для роботи з каталогами, який став частиною стандарту POSIX.1. (Багато реалізацій не допускають читання вмісту файлів каталогів за допомогою функції *read*(), тим вони перешкоджають залежності застосувань від особливостей, властивих конкретній реалізації.)

Робота з каталогами, по суті, нічим не відрізняється від роботи з будь–яким іншим файлом. Перед початком роботи з ним його слід відкрити зверненням до стандартної функції *opendir*(), яка створює в програмі дескриптор каталогу, що використовується як посилання на відкритий каталог при виконанні необхідних операцій:

```
#include <dirent.h>
DIR *opendir(
     const char *path //ім'я каталогу
```

```
    );
// повертає покажчик на DIR у випадку успіху,
// -1 у випадку помилки
```

Функція *opendir*() служить для відкриття потоку інформації для каталогу з ім'ям *name*. Тип даних *DIR* є деякою структурою даних, що описує такий потік. Функція *opendir*() готує ґрунт для функціонування інших функцій, що виконують операції над каталогом, і позиціонує потік на першому записі каталогу.

При успішному завершенні функція повертає покажчик на відкритий потік каталогу, який надалі передаватимеся як параметр всім іншим функціям, що працюють з цим каталогом. При невдалому завершенні повертається значення *NULL*.

Після того, як потік каталогу створений за допомогою виклику *opendir*(), програма може починати читати записи з каталогу. Для цього використовується функція *readdir*():

```
#include <dirent.h>
     struct dirent *readdir(
          DIR *dirp // покажчик на DIR
     );
// повертає покажчик на структуру у випадку успіху,
// -1 у випадку помилки
```

Параметр *dirp* представляє покажчик на структуру, яка описує потік каталогу, що повернений функцією *opendir*().

В разі успіху виклик *readdir*() повертає черговий запис в каталозі. Запис каталогу представляється структурою *dirent*, яка містить наступні елементи:

```
struct dirent {
  ino_t  d_ino;       // Номер індексного дескриптора
  off_t  d_off;       // відстань до наступного елементу
  unsigned short d_reclen;    // довжина запису
  unsigned char d_type;    // тип файлу; не підтримується
                           //більшістю реалізацій
  char    d_name[256]; /* Ім'я файлу
};
```

Поля цього запису варіюються від однієї файлової системи до іншої. У стандарті POSIX обов'язковим є лише поле *char d_name*[] невизначеної довжини, що не перевищує значення *NAME_MAX*+1. Інші поля не обов'язкові. Поле *d_name* містить символьне ім'я файлу, яке завершується символом кінця рядка. Стандарт не визначає розмір *d_name*, але гарантує, що число байтів, передуючих нульовому символу, буде менше, ніж число, що зберігається в змінній

_PC_NAME_MAX, визначеною в заголовному файлі <*unistd.h*>. Зверніть увагу, що нульове значення змінної *d_ino* позначає порожній запис в каталозі.

Дані, що повертаються функцією *readdir*(), переписуються при черговому виклику цієї функції для того ж самого потоку каталогу.

При успішному завершенні функція повертає покажчик на структуру, що містить черговий запис каталогу. При невдалому завершенні або досягши кінця каталогу повертається значення *NULL*.

Відкритий в програмі каталог закривається функцією *closedir*() з єдиним параметром — дескриптором каталогу:

```
#include <dirent.h>
int closedir(
     DIR *dirp // покажчик на DIR,
);
//повертає 0 у випадку успіху  -1 у випадку помилки
```

Інколи після читання частини вмісту каталогу виникає необхідність знову повернутися до його початку. Функцією *rewinddir*( ) поточна позиція в каталозі встановлюється на початок, що дозволяє здійснювати повторне читання май файлів каталогу, не закриваючи його. Єдиним параметром цієї функції є дескриптор відкритого каталогу.

```
#include <dirent.h>
void rewinddir(
     DIR *dirp // покажчик на DIR
);
```

При читанні вмісту каталогу необхідно дотримуватися такої послідовності дій.

1. Викличите функцію *opendir*(), передавши їй шляхове ім'я необхідного каталогу.

2. Послідовно викликайте функцію *readdir*(), передаючи їй дескриптор, отриманий від функції *opendir*(). Всякий раз функція *readdir*() повертатиме покажчик на структуру типу *struct dirent*, що містить інформацію про наступний елемент каталогу. Після досягнення кінця каталогу буде набуто значення *NULL*.

3. Викличте функцію *closedir*(), передавши їй наявний дескриптор, щоб завершити сеанс роботи з каталогом.

Для кожного процесу визначений поточний робочий каталог. Відносно цього каталогу обчислюються всі відносні шляхи. Коли користувач входить в систему, поточним робочим каталогом зазвичай стає каталог, вказаний в шостому полі запису з файлу */etc/passwd*, – домашній каталог користувача.

Поточний робочий каталог – це атрибут процесу, домашній каталог – атрибут користувача.

Для зміни поточного каталогу передбачено два системних виклики: *chdir*() і *fchdir*().

Аргумент системного виклику *chdir*() може бути і абсолютним, і відносним маршрутом до каталогу.

```
#include <unistd.h>
int chdir(
     const char *path // ім'я каталогу
);
 //повертає 0 у випадку успіху, –1 у випадку помилки
//(код помилки – в змінній errno)
```

Системний виклик *chdir*() завершиться невдачею і поверне значення –1, якщо аргумент *path* не є коректним ім'ям каталогу або якщо процес не має доступу на виконання (проходження) для всіх каталогів в імені.

Системний виклик *fchdir*() приймає як аргумент дескриптор, відкритий для каталогу.

```
#include <unistd.h>
     int fchdir(
          int fd  // дескриптор файла
     );
 //повертає 0 у випадку успіху, –1 у випадку помилки
//(код помилки – в змінній errno)
```

Інколи необхідно виконати операцію над ієрархією каталогів, почавши від стартового каталогу, і обійти всі лежачі нижче файли і підкаталоги.

При реалізації обходу каталогів корисно створення закладок на каталоги. Порівняєте наступні дві методики:

1. Отримати повне ім'я поточного каталогу. 2. Зробити що–небудь, що змінить поточний каталог. 3. Викликати функцію *chdir*(), використовуючи ім'я каталогу, отримане на першому кроці.	1. Відкрити поточний каталог викликом *open*(). 2. Зробити що–небудь, що змінить поточний каталог. 3. Викликати функцію *fchdir*(), використовуючи дескриптор, що був отриманий на першому кроці.

Методика, яка приведена в лівому списку, декілька поступається тій, що приведена справа, тому що:

- є ряд складнощів із здобуттям імені поточного каталогу.

– це досить трудомісткий процес.

В деяких випадках, коли вам необхідно повернутися лише на один рівень вище, можна використовувати такий підхід:

```
chdir("..")
```

Варіанти завдань

1. Написати програму, що видаляє всі висячі посилання в заданому каталозі й всіх його підкаталогах. Ім'я каталогу задавати у вигляді аргументу командного рядка.
2. Написати програму, що підраховує кількість звичайних файлів, що належать заданому користувачу у заданому каталозі та всіх його підкаталогів». Ім'я користувача та каталогу задавати у вигляді аргументів командного рядка.
3. Написати програму, що підраховує сумарний обсяг звичайних файлів у заданому каталозі й всіх його підкаталогах. Ім'я каталогу задавати у вигляді аргументу командного рядка.
4. Написати програму, що підраховує сумарну кількість файлів кожного типу в заданому каталозі й всіх його підкаталогах. Ім'я каталогу задавати у вигляді аргументу командного рядка.
5. Написати програму, що видаляє всі звичайні файли нульової довжини в заданому каталозі й всіх його підкаталогах. Ім'я каталогу задавати у вигляді аргументу командного рядка.
6. Написати програму, що виводить уміст заданого каталогу й всіх його підкаталогів. Ім'я каталогу задавати у вигляді аргументу командного рядка.
7. Написати програму, що видаляє із заданого каталогу й всіх його підкаталогів файли із суфіксом «.o». Ім'я каталогу задавати у вигляді аргументу командного рядка.
8. Написати програму, що видаляє заданий не порожній каталог. Ім'я каталогу задавати у вигляді аргументу командного рядка.
9. Написати програму, що підраховує сумарну кількість символічних посилань у заданому каталозі й всіх його підкаталогах. Ім'я каталогу задавати у вигляді аргументу командного рядка.
10. Написати програму, що видаляє всі звичайні файли, імена яких не починаються із крапки, у заданому каталозі й всіх його підкаталогах. Ім'я каталогу задавати у вигляді аргументу командного рядка.

11. Написати програму, що видаляє із заданого каталогу й всіх його підкаталогів файли з ім'ям «core». Ім'я каталогу задавати у вигляді аргументу командного рядка.

Контрольні питання

1. Яка функція використовується для відкриття каталогу?
2. За допомогою якої структури можна отримати запис каталогу?
3. Як змінити поточний каталог?
4. Як отримати ім’я поточного каталогу?
5. Що таке закладка?
6. Які складнощі виникають при виконанні операцій над ієрархією каталогів?

ЛАБОРАТОРНА РОБОТА № 6. ОБРОБКА ПЕРЕРИВАНЬ ТА СИГНАЛІВ

Мета роботи

Вивчити програмні засоби створення процесів, а також прості способи обміну даними між процесами. Вивчити механізм сигналів ОС UNIX, що дозволяє процесам реагувати на різні події.

Вміст роботи

1. Вивчити правила використання системних викликів *fork*(), wait(), *exit*().
2. Вивчити засоби динамічного запуску програм в ОС UNIX (системні виклики execl(), execv(),...).
3. Вивчити засоби роботи з сигналами в ОС UNIX.
4. Ознайомитися із завданням до лабораторної роботи.
5. Вибрати набір системних викликів, що забезпечують рішення задачі.
6. Для вказаного варіанту скласти програму, що реалізовує завдання.
7. Відлагодити і протестувати складену програму.
8. Захистити лабораторну роботу, відповівши на контрольні питання.

Теоретичні відомості

Зазвичай процеси створюються за допомогою функції *fork*(). Дочірній процес, створений за допомогою *fork*() є копією батьківського процесу за тим винятком, що має власний ідентифікатор.

```
#include <unistd.h>
pid_t fork(void);
// повертає ідентифікатор дочірнього процесу або 0
//у випадку успіху та –1 у випадку помилки
//(код помилки – в змінній errno)
```

Породжений, або дочірній процес є точною копією процесу, що виконав системний виклик *fork().* Зокрема, цей виклик дочірній процес успадковує такі атрибути батька, як:

- ідентифікатори користувача й групи,
- змінні оточення,
- диспозицію сигналів та їхніх оброблювачів,
- обмеження, що накладають на процес,

- поточний і кореневий каталог,
- маску створення файлів,
- всі файлові дескриптори, включаючи файлові покажчики,
- керуючий термінал.

Деякі атрибути дочірнього процесу відрізняються від атрибутів батьківського процесу:

- дочірній процес має власний ідентифікатор;
- дочірній процес одержує власні копії відкритих файлових дескрипторів батьківського процесу;
- процесорний час дочірнього процесу у момент створення встановлюється рівним нулю;
- дочірній процес не успадковує захвати файлів батьківського процесу;
- дочірній процес не успадковує аварійних сигналів, установленнях батьківським процесом.

Після створення дочірнього процесу обидва процеси продовжують виконуватися в нормальному режимі. Знов створений дочірній процес виконує ту ж саму програму, що і батьківський процес, починаючи з тієї точки, де повернула управління функція *fork*().

Функція *fork*() викликається один раз, а управління повертає двічі – в дочірньому процесі вона повертає 0, тоді як в батьківському – ідентифікатор створеного дочірнього процесу. Остання обставина пояснюється тим, що в процесу може бути багато нащадків, а система не передбачає засобів для отримання ідентифікаторів дочірніх процесів. У дочірньому процесі функція *fork*() повертає 0, оскільки дочірній процес має лише одного батька і завжди може отримати його ідентифікатор за допомогою функції *getppid*().

Значення, яке повернув системний виклик *fork*(), може використовуватися для того, щоб визначити, виконується програма в батьківському або дочірньому процесі.

Загальна схема організації різної роботи процесу–дитини і процесу–батька виглядає так:

```
pid = fork();
if(pid == -1){
    ...
    /* помилка */
    ...
} else if (pid == 0){
    ...
    /* дитина */
    ...
```

```
} else {
    ...
    /* батько */
    ...
}
```

Виконання однієї і тієї ж програми декількома процесами не часто є корисним. Проте, дочірній процес може виконувати іншу програму, якщо він створений системним викликом *exec*.

Системний виклик *exec* повторно ініціалізує процес, підміняючи його вказаною програмою — програма змінюється, а процес залишається. Системний виклик *fork*(), навпаки, запускає новий процес, який є точною копією того, що існує, простим копіюванням сегментів з кодом і даними.

Насправді не існує одного системного виклику *exec*. Під цим ім'ям мається на увазі ціле сімейство з шести системних викликів, імена яких в загальному вигляді можна записати як *execAB*. *A* — це один з символів, «*l*» або «*v*», вони визначають, як вхідні аргументи передаються виклику — у вигляді списку (від англ. *list*) або у вигляді масиву (від англ. *vector*). *B* (може бути відсутнім) — це або «*p*», яка вказує, що пошук файлу програми повинен виконуватися за допомогою змінної середовища *PATH*, або «*e*» – такому виклику буле передане специфічне середовище оточення (помітимо, що немає системного виклику *exec*, який поєднував би в собі характерні особливості «e» і «*p*»). Таким чином, ми отримуємо шість різних системних викликів: *execl*(), *execv*(), *execlp*(), *execvp*(), *execle*() і *execve*().

Ці функції можуть використовуватися для того, щоб виконувати в процесі яку–небудь програму вже після того, як він був створений. Всі функції відрізняються використанням параметрів, проте насправді вони виконують одне і те ж. Прототипи функцій визначені в заголовному файлі <*unistd.h*>.

У більшості реалізацій UNIX лише одна з цих шести функцій, *execve*(), є системним викликом. Останні п'ять - звичайні бібліотечні функції, які кінець кінцем звертаються до цього системного виклику.

Функція *execl*() запускає програму, вхідні аргументи передаються у вигляді списку.

```
#include <unistd.h>
int execl(
  const char *path, //повний шлях до програми
  const char *arg0, //перший аргумент (arg[0]-ім'я
                    //файлу)
  const char *arg1, //другий аргумент(якщо необхідно)
  ...,          //інші аргументи(якщо необхідні)
  NULL          //пустий вказівник,завершуючий список
```

```
);
// повертає –1 у випадку помилки
//(код помилки – в змінній errno)
```

Аргумент *path* повинен містити повне ім'я виконуваного файлу з програмою. Сегмент коду процесу буде «затертий» кодом нової програми, сегмент даних також буде затертий сегментом даних нової програми, а стек буде переініціалізовано. Виконання нової програми почнеться із самого початку (буде викликана функція *main*()).

В разі успіху повернення з *execl*() не передбачене, тому що точка повернення буде загублена. В разі помилки *execl*() поверне –1, не має сенсу перевіряти це значення, оскільки ніякого іншого отримати не можна. Найчастіші причини неможливості запуску нової програму, це або відсутність файлу з програмою, або відсутність права на її запуск.

Всі інші аргументи виклику збираються в масив покажчиків на рядки, а останнім завжди повинен стояти порожній покажчик, який визначає кінець списку вхідних аргументів. Перший аргумент, відповідно до угод – це ім'я файлу програми. Нова програма дістає доступ до цього списку через вже знайомі нам аргументи функції *main*() – *argc* і *argv*. Середовище оточення також передається новій програмі і доступна їй через покажчик *environ* або за допомогою функції *getenv*().

Оскільки процес продовжує існувати і сегмент з системними даними практично не змінюється, майже всі атрибути процесу також залишаються незмінними. Перелічимо основні атрибути, що змінюються.

- якщо процес призначав свої обробники сигналів, то всі вони скидаються у вхідний стан, оскільки функції–обробники після запуску нової програми стануть недоступні;
- якщо в новій програмі встановлені біти *SUID* або *SGID*, то ефективні ідентифікатори користувача і групи встановлюються заново відповідно до ідентифікаторів власника і групи файлу програми. Немає жодного способу повернути колишні ефективні ідентифікатори, якщо вони відрізняються від реальних;
- реєстрація всіх функцій, яка була виконана за допомогою *atexit*(), відміняється, оскільки код цих функцій буде затертий;
- сегменти спільної пам'яті від'єднуються, оскільки точки з'єднання будуть загублені;
- іменовані семафори POSIX закриваються. Семафори System V залишаються без змін;

- доля відкритих файлів залежить від значення прапора *FD_CLOEXEC* (закрити-при-виклику-*exec*). Якщо цей прапор встановлений, дескриптор закривається. Інакше дескриптор залишається відкритим.

Функція *execv*()запускає програму, вхідні аргументи передаються у вигляді масиву. Це абсолютно необхідно, коли число аргументів у момент написання програми заздалегідь невідоме.

```
#include <unistd.h>
int execv(
   const char *path, //повний шлях до файлу з програмою
   char *const argv[] //масив аргументів
     );
// повертає –1 у випадку помилки
//(код помилки – в змінній errno)
```

Функція виконує файл, заданий рядком *path* як нове зображення процесу. Аргумент *argv* є масивом рядків, які використовуються як аргумент *argv* функції *main*(). Останнім елементом цього масиву має бути нульовий покажчик. Перший елемент масиву – ім'я файлу програми. Середовище нового процесу береться із змінної *environ* поточного процесу.

Зазвичай середовище оточення процесу передається дочірнім процесам без змін, але в деяких випадках виникає необхідність створити особливе середовище оточення для дочірнього процесу. Приклад такого випадку - програма *login*, що ініціалізує новий сеанс роботи користувача. Для вирішення зазначеної проблеми використовуються функції *execve*() і *execle*.

Функція *execve*() запускає програму, аргументи передаються у вигляді масиву, також передається середовище оточення.

```
#include <unistd.h>
     int execve(
     const char *path, //повний шлях до файлу з  програмою
char *const argv[] //масив аргументів
char *const envv[] //массив сформованного середовища
);
// повертає –1 у випадку помилки
//(код помилки – в змінній errno)
```

Ідентична *execv*() за винятком того, що процесу передаються змінні середовища в аргументі *envv*.

Функція *execle*() запускає програму, аргументи якої передаються у вигляді списку, також передається середовище оточення.

```
#include <unistd.h>
int execle(
  const char *path, // повний шлях до файлу з програмою
  const char *arg0, // перший аргумент (ім'я файлу)
  const char *arg1, // другий аргумент(якщо необхідно)
  ...,              // інші аргументи(якщо необхідні)
  NULL             // пустий вказівник,завершуючий
                  // список  аргументів
  char *const env[]  //масив сформованого середовища
     );
// повертає -1 у випадку помилки
//(код помилки - в змінній errno)
```

Ідентична попередній функції за винятком того, що аргументи програми передаються індивідуально.

Функція *execlp*() запускає програму, аргументи передаються у вигляді списку, пошук файлу ведеться з використанням змінної оточення *PATH*.

```
#include <unistd.h>
     int execlp(
  const char *file, // ім'я файлу з програмою
  const char *arg0, //перший аргумент (ім'я файлу)
  const char *arg1, //другий аргумент(якщо необхідно)
  ...,  //інші аргументи(якщо необхідні)
  NULL  //пустий вказівник,завершуючий список
);
// повертає -1 у випадку помилки
//(код помилки - в змінній errno)
```

Функція *execvp*() запускає програму, аргументи передаються у вигляді масиву, пошук файлу ведеться з використанням змінної оточення *PATH*.

```
#include <unistd.h>
int execvp(
  const char *file, // ім'я файлу з програмою
  char *const argv[] //масив аргументів
);
// повертає -1 у випадку помилки
//(код помилки - в змінній errno)
```

Якщо аргумент *file* у викликах *execlp*() або *execvp*() не містить символ «/», то виконується послідовна підстановка імен каталогів із змінної *PATH* і проводиться пошук файлу з правом на виконання. Якщо такий файл знайдений і він є виконуваною програмою (перевіряється сигнатура файлу), то вона запускається на виконання. Якщо немає, то робиться припущення про те, що

файл є сценарієм і викликається командний інтерпретатор, якому передається ім’я файлу з програмою в першому аргументі.

Змінна оточення PATH містить список каталогів, розділених двокрапками. Наприклад, рядок оточення у форматі name=value

```
PATH=/bin:/usr/bin:/usr/local/bin/:.
```

визначає чотири каталоги, в яких відбуватимеся пошук виконуваних файлів. Останнім вказаний поточний каталог. (Порожній префікс також означає поточний каталог.) По причинах, пов'язаних з безпекою системи, поточний каталог в змінну оточення PATH, як правило, не включають.

За видалення завершених дочірніх процесів відповідає батьківський процес. Для цього використовуються системні виклики сімейства *wait.*

Системні виклики *wait*() та *waitpid*() чекають, доки дочірній процес не змінить свій стан (призупинення, відновлення або завершення). Ці функції можуть:

- заблокувати процес, якщо всі його дочірні процеси продовжують роботу;
- відразу ж повернути управління з кодом завершення дочірнього процесу, якщо він вже закінчив роботу і чекає, поки батьківський процес обробить код завершення;
- відразу ж повернути управління з ознакою помилки, якщо в процесу, що викликав, немає жодного дочірнього процесу.

Почнемо обговорення з системного виклику *waitpid*().

```
#include <sys/wait.h>
pid_t   waitpid(
     pid_t pid, //ідентифікатор процесу або групи
     int *statusp,  //вказівник на статус або NULL
     int options   //прапори
);
//у випадку успіху повертає ідентифікатор процесу або 0
//та -1 у випадку помилки (код помилки – в змінній errno)
```

Аргумент *pid* може набувати наступних значень:

>0	Чекати зміни стану дочірнього процесу з вказаним ідентифікатором.
-1	Чекати зміни стану будь-якого дочірнього процесу.
0	Чекати зміни стану будь-якого дочірнього процесу, що належить до тієї ж групи процесів, що і що викликає.
< -1	Чекати зміни стану будь-якого дочірнього процесу, що належить до групи процесів з ідентифікатором -*pid*.

На виході з *waitpid*() процес отримує ідентифікатор процесу-нащадка, чий ідентифікатор збігся з аргументом *pid*. Нуль повертається лише у тому випадку, коли був встановлений прапор *WNOHANG*.

Допускається чекати зміни стану лише прямих нащадків, породжених системним викликом *fork*(). Очікування процесів «онуків» не припускається, навіть якщо їх батьки до моменту виклику вже завершили свою роботу.

Як правило, процеси зацікавлені в здобутті інформації про стан своїх нащадків — інакше процеси-нащадки після закінчення перетворюються на «зомбі» і перебувають у такому вигляді, поки не завершить роботу батьківський процес. Оскільки, багато процесів можуть виконуватися досить тривалий час (до декількох місяців), така «неувага» до дочірніх процесів може привести до переповнення системних таблиць. Якщо очікування нащадків неможливе, з тих або інших причин, процес для запобігання «зомбуванню» може використовувати сигнали.

Системний виклик *waitpid*() може повернути звіт про зміну стану дочірнього процесу лише один раз (такий дочірній процес називається очікуваним). Іншими словами: очікуваний нащадок перестає бути очікуваним, якщо звіт про зміну його стану вже отриманий. Тобто, якщо був отриманий стан нащадку і раптом виявилося, що це не той нащадок, якого чекали, то немає жодного способу повернути його в систему, щоб інший виклик *waitpid*() зміг його отримати.

Аргумент *options* може містити один або більш за прапори, об'єднаних операцією порозрядної диз’юнкції :

WCONTINUED	Повідомляти про відновлення роботи нащадком
WNOHANG	Не чекати зміни стану нащадка. Якщо воно ще не змінилося — повертати значення 0.
WUNTRACED	Повідомляти про призупинення роботи нащадка.

Якщо до моменту виклику *waitpid*() є очікуваний дочірній процес, відповідний заданому аргументу *pid*, управління в програму повертається негайно. Якщо нащадок знайдений, але ще не змінив свій стан, системний виклик *waitpid*() блокується до появи відповідного нащадка. Якщо нащадок не був знайдений, програмі повертається значення -1 і код помилки *ECHILD*. Це може статися тому, що аргумент *pid* заданий неправильно або процес-нащадок перестав бути очікуваним, тобто його стан вже було отриманий.

Якщо як аргумент *statusp* переданий непорожній покажчик, то за заданою адресою записується код стану нащадка. Він є комбінацією аргументу системного виклику *_exit*() або *exit()* (якщо йдеться про завершення нащадку) і числа, що описує причину завершення або призупинення.

Макроси, що вживаються для аналізу комбінації:

WIFEXITED(status)	*true*, якщо нащадок завершив роботу звичайним способом (зверненням до *_exit*() або *exit*()).
WEXITSTATUS(status)	Якщо *WIFEXITED* повернув *true*, молодші 8 бітом є аргумент виклику *_exit*() або *exit*().
WIFSIGNALED(status)	*true*, якщо нащадок завершився аварійно (по сигналу).
WTERMSIG(status)	Повертає номер сигналу, що викликав аварійне завершення нащадка, за умови, що *WIFSIGNALED* повернув *true*.
WIFSTOPPED(status)	*true*, якщо нащадок припинений. Можливо лише в разі установки прапора *WUNTRACED*).
WSTOPSIG(status)	Номер сигналу, який викликав призупинення нащадка, за умови, що *WIFSTOPPED* повернув true.
WIFCONTINUED(status)	true, якщо виконання нащадка було відновлене. Можливо лише в разі установки прапора *WCONTINUED*.
WCOREDUMP(status)	*true*, якщо створений файл з дампом пам'яті. Цей файл може використовуватися при пошуку причин, що викликали «звалювання» процесу (макрос нестандартний, але він присутній в більшості систем).

Наведемо декілька прикладів використання системного виклику *waitpid*():

1. Чекати завершення нащадка *pid* і отримати код завершення:

waitpid(pid &status, 0);

2. Чекати завершення будь-якого з нащадків без здобуття коду завершення:

```
pid = waitpid(-1, NULL, 0);
```

3. Отримати від будь-якого з нащадків по груповому ідентифікатору *pgid* повідомлення про завершення або про призупинення і отримати код стану. Не чекати якщо нащадок ще не змінив стан.

```
pid = waitpid(-pgid &status, WNOHANG | WUNTRACED);
```

Другий представник групи системних викликів *wait*() є спрощеним варіантом *waitpid*() і еквівалентний виклику останнього з аргументом *pid*, рівним -1, і з аргументом *options* рівним нулю:

```
#include <sys/wait.h>
pid_t   wait(
          int *statusp  //вказівник на статус або NULL
);
//повертає ідентифікатор процесу
// або -1 у випадку помилки
//(код помилки – в змінній errno)
```

Системний виклик *wait*() досить рідко використовується, оскільки чекає завершення будь-якого нащадка. Річ у тому, що коли деяка функція створює дочірній процес і намагається його почекати, вона може випадково «діждатися» завершення абсолютно іншого нащадка. Для таких випадків набагато краще користуватися викликом *waitpid*(), оскільки він дозволяє чекати конкретний процес або члена групи процесів.

Сигнали – це програмні переривання. Більшість серйозних застосувань мають справу з сигналами. Сигнали надають можливість обробки асинхронних подій – наприклад, коли користувач вводить символ переривання, аби зупинити програму, або коли одна з програм в конвеєрі аварійно завершується.

Сигнал дає можливість процесу реагувати на подію, джерелом якої може бути операційна система або інше завдання. Сигнали викликають переривання завдання і виконання заздалегідь передбачених дій. Сигнали можуть вироблятися синхронно (як результат роботи самого процесу), або асинхронно (направлені процесу іншим процесом). Синхронні сигнали найчастіше приходять від системи переривань процесора і свідчать про дії процесу, що блокуються апаратурою, наприклад, ділення на нуль, помилка адресації, порушення захисту пам'яті тощо.

Прикладом асинхронного сигналу є сигнал з терміналу. Для цього користувач може натискувати комбінацію клавіш *Ctrl+C*, внаслідок чого ОС UNIX виробляє сигнал *SIGINT* і направляє його активному процесу. Сигнал може поступити у будь–який момент виконання процесу, тобто він є асинхронним,

вимагаючи від процесу негайного завершення роботи. В даному випадку реакцією на сигнал є безумовне завершення процесу.

Сигнали генеруються ядром і забезпечують виклик певної процедури при настанні деякої події. Основні причини відправки сигналу:

- особливі ситуації. (ділення на нуль);
- термінальні переривання. (<*Del*>, <*Ctrl+C*>, <*Ctrl+Z*>);
- інші процеси. Як засіб взаємодії між процесами за допомогою виклику *kill*();
- управління завданнями;
- квоти. Перевищення квоти ресурсів;
- повідомлення. Наприклад, про готовність пристрою;
- алярми. Пов'язано з роботою таймерів.

В разі появи сигналу ядро може провести одну з трьох дій. Вовни називаються ***диспозиціями сигналу***, або діями, пов'язаними з сигналом.

- **ігнорувати сигнал**. Ця дія можлива для більшості сигналів, окрім сигналів *SIGKILL* і *SIGSTOP*. Ці два сигнали не можуть бути проігноровані тому, що ядру і суперкористувачеві необхідна можливість завершити або зупинити будь–який процес. Якщо проігнорувати деякі з сигналів, що виникають в результаті апаратних помилок (таких як ділення на 0 або спроба звернення до неіснуючої пам’яті), поведінка процесу може стати не передбачуваною;
- **перехопити сигнал.** Для цього треба повідомити ядру адресу функції, яка буде викликатися кожний раз при виявленні сигналу. У цій функції можна передбачити дії з обробки ситуації, яка породила сигнал. Наприклад, якщо процес створює тимчасові файли, то має сенс написати функцію обробки сигналу *SIGTERM* (сигнал завершення, що посилається командою *kill* за умовчанням), яка видалятиме тимчасові файли. Сигнали *SIGKILL* і *SIGSTOP* не можуть бути перехоплені;
- **застосувати дію за умовчанням**. Кожному сигналу поставлена в відповідність деяка дія за умовчанням. Для більшості сигналів ця дія полягає в завершенні процесу.

Реакція на сигнал (або дія з сигналу) відноситься до всього процесу, навіть якщо цей процес виконується в декількох потоках, хоча, кожен потік може мати свою маску сигналів.

Сигнали забезпечують логічний зв'язок між процесами, а також між процесами і користувачами (терміналами).

Сигнали з'явилися вже в ранніх версіях UNIX, але їх реалізація була недостатньо надійною. Сигнал міг бути втрачений, виникали труднощі з

відключенням (блокуванням) сигналів на час виконання критичних ділянок коди. В даний час стандарт POSIX.1 визначає інтерфейс надійних сигналів. Кожен сигнал має унікальне символьне ім'я і відповідний йому номер.

У стандарті SUS2002 визначено 28 різних сигналів, але більшість реалізацій доповнюють цей список своїми сигналами. Крім того, існують додаткові сигнали, які є частиною розширень реального часу POSIX. Перелік базових сигналів наведений у додатку Б.

Сигнали виявлення помилок, що мають природне походження, є результатом помилок в програмі. Для сигналів *SIGBUS, SIGSEGV, SIGFPE* і *SIGILL* точна причина помилки стандартами не обмовляється, але, як правило, всі вони виявляються на апаратному рівні. Крім того, ці чотири сигнали підкоряються визначеним правилам, в разі їх природного походження:

- якщо для цих сигналів, за допомогою *sigaction*(), встановлена дія *SIG_IGN*, то реакція застосування на них — не визначена (залежить від системи);
- реакція застосування, в разі нормального повернення з функції–обробника сигналу — не визначена (залежить від системи);
- результат блокування сигналу не визначений.

Для представлення множини сигналів необхідний спеціальний тип даних - ***набір сигналів***. Набір сигналів є набором бітів, в якому кожному сигналу відводиться окремий біт. Відмітимо, кількість різних сигналів може перевищувати кількість біт в цілочисловому типі, тому в більшості випадків не можна використовувати типу *int* для представлення набору сигналів. Стандарт *POSIX.1* визначає для цих цілей спеціальний тип *sigset_t*, з яким працюють наступні функції:

- *sigemptyset*() ініціалізує порожній набір сигналів

```
#include <signal.h>
int sigemptyset(
     sigset_t *set //набір сигналів
     );
// повертає 0 у випадку успіху або –1 у випадку помилки
```

- *sigfillset*() ініціалізує повний набір сигналів

```
#include <signal.h>
int sigfillset(
     sigset_t *set //набір сигналів
     );
// повертає 0 у випадку успіху або –1 у випадку помилки
```

- *sigaddset*() додає сигнал до набору

```
#include <signal.h>
int sigaddset(
```

```
      sigset_t *set, //набір сигналів
      int signum //номер сигналу

);
// повертає 0 у випадку успіху або –1 у випадку помилки
```

– *sigdelset*()видаляє сигнал з набору

```
#include <signal.h>
int sigdelset(
      sigset_t *set, //набір сигналів
      int signum //номер сигналу
);
// повертає 0 у випадку успіху або –1 у випадку помилки
```

Робота з маскою сигналів типу *sigset_t* починається з виклику функції *sigemptyset*() або *sigfillset*(), а потім за допомогою *sigaddset*() або *sigdelset*() додаються потрібні або віддаляються непотрібні сигнали. Для перевірки наявності сигналу в наборі використовується функція *sigismember*().

```
#include <signal.h>
int sigismember(
      const sigset_t *set //набір сигналів
      int signum //номер сигналу
);
// повертає 1 якщо сигнал додано до набору, 0 – якщо не
// додано до набору, –1 – у випадку помилки
```

Реакція програми на сигнали задається за допомогою системного виклику *sigaction*().Він викликається для кожного сигналу, дію якого необхідно змінити:

```
#include <signal.h>
int sigaction(
          int signum //номер сигналу
          const struct sigaction *act //нова дія
          struct sigaction *oact //стара дія
);
// повертає 0 у випадку успіху або –1 у випадку помилки
```

У аргументі *act* передається покажчик на структуру, яка визначає реакцію на сигнал для всього процесу. Якщо аргумент *oact* не є порожнім покажчиком, за заданою адресою записується опис колишньої реакції процесу на сигнал. Дляе отримання опис дії сигналу без зміни діспозиції треба передати в аргументі *act* значення *NULL*.

Для завдання цих аргументів використовується спеціальна структура *struct sigaction*.

```
struct sigaction
```

```
{
   void (*sa_handler)();
   void (sa_sigaction)(int, siginfo_t *, void *);
   sigset_t sa_mask;
    int sa_flags;
}
```

У структурі *sigaction* поле *sa_handler* визначає власне реакцію на сигнал і може мати одне з наступних значень:

SIG_DFL Реакція за замовуванням, яка залежить від вигляду сигналу.

SIG_IGN Процес ігнорує сигнал, доставка сигналу не впливає на його роботу. Побічний ефект: коли для сигналу *SIGCHLD* визначена реакція *SIG_IGN*, це рівносильне установці прапора *SA_NOCLDWAIT*. Реакція *SIG_DFL* не має такого ефекту, хоча вона і полягає в тому, щоб ігнорувати сигнал.

Функція Покажчик на функцію–обробник. В цьому випадку говорять: «сигнал буде перехоплений».

Функції–обробники сигналів виглядають приблизно так

```
static void fcn(int signum){
(void)write(STDOUT_FILENO, "Отриманий сигнал \n", 11);
_exit(EXIT_FAILURE);
}
```

У момент доставки сигналу викликається функція–обробник, якою як аргумент передається номер сигналу (наприклад *SIGINT* або *SIGUSR*1). Можна передбачити окремі функції для кожного з сигналів, можна всі сигнали обробляти однією функцією або вибрати щось середнє між цими двома крайнощами.

Якщо системою підтримуються сигнали реального часу, можна встановити прапор *SA_SIGINFO* і передати покажчик обробник в полі *sa_sigaction*. В цьому випадку обробник отримує значно більше відомостей про сигнал. Деякі реалізації використовують для зберігання покажчика *sa_handler* і *sa_sigaction* одну область пам'яті, тому слід записувати адресу обробника лише в один з них.

Нижче наводиться список прапорів для поля *sa_flags*. Звернете увагу: перші два застосовуються лише для сигналу SIGCHLD.

SA_NOCLDSTOP Не посилати сигнал при зупинці і відновленні дочірнього процесу.

SA_NOCLDWAIT Не перетворювати дочірній процес на «зомбі». Явна установка реакції *SIG_IGN* на сигнал SIGCHLD дає той же ефект.

SA_NODEFER	Не додавати сигнал до маски при виклику обробника, якщо він явно не вказаний в полі *sa_mask*. Цей прапор залишено виключно для збереження сумісності із застарілою функцією *signal*.
SA_ONSTACK	Доставляти сигнал на альтернативному стеку сигналів, якщо такий був оголошений зверненням до функції *sigaltstack*().
SA_RESETHAND	Встановити реакцію на сигнал в значення *SIG_DFL* і на вході у функцію–обробник скидати прапор *SA_SIGINFO*. Ігнорується для сигналів *SIGILL* і *SIGTRAP*. Додатково виконуються дії, властиві сигналу *SA_NODEFER*. Існує для сумісності з функцією *signal*.
SA_RESTART	Не переривати виконання системних викликів.
SA_SIGINFO	Покажчик на функцію–обробник слід брати з поля *sa_sigaction*, а не з *sa_handler*.

Приклад. Програма виводить число кожні три секунди, а при здобутті сигналу переривання (*SIGINT*), виводить повідомлення і завершується:

```
static void fcn(int signum)
{
  (void)write(STDOUT_FILENO, "Отриманий сигнал\n", 15);
  _exit(2);
}

int main(void)
{
  int i;
  struct sigaction act;

  memset(&act, 0, sizeof(act)); act.sa_handler = fcn;
  sigaction(SIGINT &act, NULL);
  for (i = 1; ; i++)
  {
    sleep(3);
    printf("%d\n", i);
  }
  exit(0);
}
```

Зверненням до системного виклику *sigaction*() проводиться установка функції–обробника сигналу *SIGINT*. Після запуску програми, коли на екрані з'явилося число 2, натискуватимемо клавіші *Ctrl+C*. Це привело до того, що виконання програми буде перервано і управління перейде функції *fcn*, яка виведе

текст повідомлення і завершить роботу програми, звернувшись до системного виклику *_exit*().

В результаті, на екран було виведено наступне:

```
1 2 Отриманий сигнал
```

Якби функція–обробник не була встановлена, натиснення комбінації *Ctrl+C* привело б до негайного завершення роботи процесу, тому що в цьому полягає реакція за умовчанням будь–якого процесу на сигнал *SIGINT*.

Варіанти завдань

1. Напишіть програму, яка при одержанні сигналу SIGUSR1 породжує новий процес і виводить його ідентифікатор, а по завершенні породженого процесу виводить повідомлення про цю подію. Породжений процес повинен бути припинений на випадкову кількість секунд, після чого завершений.
2. Напишіть програму, яка при одержанні сигналу SIGUSR1 реєструє оброблювач сигналу SIGINT, що скасовує завершення програми по цьому сигналі, породжує новий процес, а при одержанні сигналу SIGUSR2 скасовує дія цього оброблювача.
3. Напишіть програму, яка після одержання сигналу SIGUSR1 ігнорує сигнали SIGUSR1 й SIGINT, а при одержанні сигналу SIGUSR2 відновлює обробку цих сигналів за замовчуванням.
4. Напишіть програму, яка при одержанні сигналу SIGUSR1 породжує новий процес, що повинен містити нескінченний цикл, а при одержанні сигналу SIGUSR2 припиняє роботу цього процесу. Забезпечте неможливість одночасної роботи двох породжених процесів.
5. Напишіть програму, яка при одержанні сигналу SIGUSR1 породжує два нових процеси, після чого одержання сигналу SIGUSR1 повинне приводити до закінчення одного з них, а одержання сигналу SIGUSR2 припиняти роботу другого й відновлювати обробку сигналу SIGUSR1. Породжені процеси повинні очікувати одержання сигналу.
6. Напишіть програму, яка після одержання сигналу SIGUSR1 ігнорує сигнали SIGUSR1 й SIGINT, а при одержанні сигналу SIGUSR2 відновлює обробку цих сигналів за замовчуванням і видає повідомлення про кількість отриманих за цей час сигналів SIGUSR1 й SIGINT.

7. Напишіть програму, яка при одержанні сигналу SIGUSR1 виводить повідомлення про кількість оброблених сигналів SIGINT, при одержанні сигналу SIGINT ігнорує сигнал SIGUSR1, а при одержанні сигналу SIGUSR2 ігнорує сигнал SIGINT і відновлює обробку сигналу SIGUSR1. Якщо сигнал SIGINT ігнорується, то він не вважається обробленим.
8. Напишіть програму, яка при одержанні сигналу SIGUSR1 породжує новий процес і передає йому сигнал SIGSLEEP, а при одержанні сигналу SIGUSR2 будить цей процес. Породжений процес повинен містити системний виклик PAUSE. Забезпечте неможливість одночасної роботи двох породжених процесів.
9. Напишіть програму, яка при одержанні сигналу SIGUSR1 породжує два нових процеси, після чого одержання сигналу SIGUSR1 повинне приводити до закінчення одного з них, а одержання сигналу SIGUSR2 припиняти роботу другого. Породжені процеси повинні містити нескінченний цикл. Процес, що породжує, після завершення роботи нащадків закінчує свою роботу.

Контрольні питання

1. Яким чином може бути породжений новий процес?
2. Якщо процес-предок відкриває файл, а потім породжує процес-нащадок, а той, у свою чергу, змінює положення покажчика читання-запису файлу, то чи зміниться положення покажчика читання-запису файлу процесу-батька?
3. Що станеться, якщо процес-нащадок завершиться раніше, ніж процес-предок здійснить системний виклик wait()?
4. Як система повідомляє батьківський процес про завершення дочірнього процесу?
5. Які макроси використовуються для аналізу статусу нащадка?
6. Для чого використовуються сигнали в ОС UNIX?
7. Які види сигналів існують в ОС UNIX?

ЛАБОРАТОРНА РОБОТА № 7. СТВОРЕННЯ ПРОЦЕСА-ДЕМОНУ

Мета роботи

Здобути навички роботи з іменованими каналами ОС UNIX за допомогою написання програми-демона.

Вміст роботи

1. Вивчити засоби роботи з каналами в ОС UNIX.
2. Ознайомитися із завданням до лабораторної роботи.
3. Вибрати набір системних викликів, що забезпечують рішення задачі.
4. Для вказаного варіанту скласти програму на мові *C*, що реалізовує завдання.
5. Виконати тестування складеної програму.
6. Захистити лабораторну роботу, відповівши на контрольні питання.

Теоретичні відомості

Якщо два або декілька процесів спільно виконують одне і те ж завдання, то вони неминуче повинні використовувати спільні дані. Хоча сигнали і можуть бути корисними для синхронізації процесів або для обробки виняткових ситуацій та помилок, вони абсолютно не підходять для передачі даних від одного процесу до іншого. Один з можливих способів вирішення цієї проблеми полягає в спільному використанні файлів, оскільки ніщо не заважає декільком процесам одночасно виконувати операції читання або запису для одного і того ж файлу. Проте спільний доступ до файлів може виявитися неефективним і потребує спеціальних запобіжних засобів для уникнення конфліктів.

Для вирішення цих проблем система UNIX забезпечує конструкцію, яка називається ***каналом*** *(pipe*). Програмний канал служить для встановлення однобічного зв'язку, що сполучає один процес з іншим.

Ми будемо використовувати ***іменовані канали*** (*named pipes*), або канали *FIFO*.

Канали поводяться з даними в порядку «перший увійшов – першим вийшов» (*first–in first–out*, або скорочено *FIFO*). Іншими словами, дані, які

поміщаються в канал першими, першими і прочитуються на іншому кінці каналу і видаляються з каналу.

Канали працюють за однаковими правилами:

- якщо прочитана менша кількість байт, чим перебуває в каналі, то залишок зберігається для наступного прочитання;
- якщо треба прочитати більше, ніж перебуває в каналі, повертається доступна кількість байт, подальшими діями керує читаючий процес;
- якщо канал порожній і жоден процес не записує в нього, то буде отриманий 0 байт;
- якщо в канал записуються дані, кількість яких менше ємності каналу, то порції даних, отриманих від різних процесів не перемішуються;
- якщо в канал треба записати більшу кількість байт, чим він може вмістити, то запис блокується до звільнення необхідного місця.

Канали застосовуються в двох випадках:

- командними оболонками для передачі даних від одного конвеєра команд іншому без створення тимчасових файлів для зберігання проміжних даних.
- для організації взаємодій типу клієнт–сервер.

На відміну від звичайних файлів, створення іменованого каналу неможливо виконати за допомогою системного виклику *open*(), для цього повинен використовуватися окремий виклик:

```
#include   <sys/stat.h>
int mkfifo(
const char *path,  //ім'я каналу
mode_t perms  // права доступу
)
// повертає 0 у випадку успіху або -1 у випадку помилки
```

Аргумент *perms* використовується аналогічно третьому аргументу виклику *open*() — він визначає права доступу до каналу.

При відкритті іменованого каналу наявність прапора *O_NONBLOCK* робить наступний вплив:

- у звичайній ситуації (прапор *O_NONBLOCK* не вказаний) операція відкриття каналу лише для читання буде заблокована до тих пір, поки інший процес не відкриє канал для запису. Аналогічно, операція відкриття лише для запису буде заблокована, поки інший процес не відкриє канал для читання;
- якщо прапор *O_NONBLOCK* вказаний, при спробі відкрити канал лише для читання функція *open*() відразу ж поверне управління. Але при спробі

відкрити канал лише для запису функція *open*() поверне значення -1 і код помилки *ENXIO* в змінній *errno*, якщо канал не був відкритий іншим процесом для читання.

При спробі запису в *FIFO*, який не був відкритий для читання, процес отримає сигнал *SIGPIPE*. Коли що останній пише в FIFO процес закриє канал, що читає процес набуде ознаки кінця файлу.

Досить часто запис даних в канал FIFO виконується з декількох процесів. Це означає, що необхідно потурбуватися про атомарність операції запису, щоб уникнути змішування даних, які поступають від різних процесів. Максимальний об'єм даних, який може бути атомарно записаний в канал *FIFO*, визначається, як і для неіменованих каналів, константою *PIPE_BUF*.

Важливою сферою застосування іменованих каналів – передача даних між клієнтом і сервером.

Демони – це довго живучі процеси. Зазвичай вони запускаються під час завантаження системи і завершують роботу разом з нею. Оскільки демони пов’язані з жодним терміналом, говорять, що вони працюють у фоновому режимі. У системі UNIX демони вирішують безліч повсякденних завдань.

Найчастіше процеси–демони використовуються як серверні процеси.

Взагалі, під сервером мається на увазі деякий процес, який чекає запитів на надання певних послуг клієнтам. Взаємодія може бути як однобічною, коли клієнт посилає повідомлення сервера, але нічого від нього не отримує, або двобічною, коли клієнт посилає запит серверу, а сервер повертає клієнтові відповідь.

Демони не розраховані на те, щоб отримувати яку–небудь інформацію від користувача. Власну інформацію вони передають іншим програмам або записують в журнали системних подій.

При програмуванні демонів, щоб уникнути небажаних взаємодій, слід дотримуватися певних правил:

1. Передусім потрібно викликати функцію *umask*(), щоб встановити маску режиму створення файлів в значення 0. Поточна маска режиму створення файлів може маскувати деякі біти прав доступу. Наприклад, якщо демон створює файли з правом на читання і на запис для групи, маска режиму створення файлу, яка виключає будь–який з цих бітів, перешкодила б цьому.

2. Викликати функцію *fork*() і завершити батьківський процес. Тим самим, ми гарантуємо, що дочірній процес не буде лідером групи, а це необхідна умова для виконання наступного шагу.

3. Створити новий сеанс за допомогою функції *setsid*(). При цьому процес стає лідером нової сесії, лідером нової групи процесів позбавляється зв'язку з керуючим терміналом.

Для систем, заснованих на *System V*, деякі фахівці рекомендують в цьому місці ще раз викликати функцію *fork*() і завершити батьківський процес, щоб другий нащадок продовжував працю демоном. Такий прийом гарантує, що демон не буде лідером сесії. Це перешкоджає отриманню нової термінальної лінії. Як варіант, для досягнення ті їж мети при будь–якому відкритті термінального пристрою слід вказувати прапор *O_NOCTTY*.

4. Зробити кореневий каталог поточним робочим каталогом. Поточний робочий каталог, успадкований від батьківського процесу, може знаходитися на змонтованій файловій системі. У цій ситуації файлову систему неможливе буде відмонтувати.

Деякі демони можуть встановлювати власний поточний робочий каталог, в якому вони проводять всі необхідні дії. Наприклад, демони друку як поточний робочий каталог часто вибирають буферний каталог, куди поміщаються завдання для друку.

5. Закрити всі непотрібні файлові дескриптори. Це запобігає утриманню у відкритому стані деяких дескрипторів, успадкованих від батьківського процесу. За допомогою функції *getrlimit*() можна визначити максимально можливий номер дескриптора і закрити всі дескриптори аж до цього номера.

Деякі демони відкривають файлові дескриптори з номерами 0,1 і 2 на пристрої */dev/null* – таким чином, будь–які бібліотечні функції, що намагаються читати із стандартного пристрою введення або писати на стандартний пристрій виведення або повідомлень про помилки, не матимуть ніякого впливу. Оскільки демон не пов'язаний ні з одним термінальним пристроєм, він не зможе взаємодіяти з користувачем в інтерактивному режимі.

Деякі демони реалізовані таким чином, що допускають одночасну роботові лише одній своїй копії. Причиною такої поведінки може служити, наприклад, вимога монопольного володіння яким–небудь ресурсом.

Одним з основних механізмів, що забезпечують обмеження кількості одночасно працюючих копій демона, є блокування файлів і записів. Якщо кожен з демонів створить файл і спробує встановити для цього файлу блокування для запису, то система дозволить встановити лише одну таке блокування. Всі подальші спроби встановити блокування для запису терпітимуть невдачу, повідомляючи тим самим останнім копіям демона про те, що демон вже запущений.

Блокування файлів і записів є зручним механізмом взаємного виключення. Якщо демон встановить для цілого файлу блокування для запису, вона буде автоматично знята після закінчення демона. Це спрощує процедуру відновлення після помилок, оскільки знімає необхідність видалення блокування, що залишилося від попередньої копії демона.

Варіанти завдань

У даному завданні потрібно організувати взаємодію незалежних процесів за допомогою іменованого каналу. Процес-клієнт зчитує із клавіатури ім'я файлу, відправляє його процесу-серверу через неіменований канал. Процес сервер виконує запит, обумовлений варіантом, і передає результат запиту через інший канал процесу-клієнтові. Якщо запит виконати не вдалося, процесу клієнтові передається повідомлення про помилку. Після цього процес-сервер очікує запит від іншого клієнта. Процес клієнт відправляє результат запиту в стандартний файл виводу.

Формат запиту й спосіб іменування каналів визначити самостійно.

1. Процес клієнт запитує вміст файлу.
2. Процес клієнт запитує вміст файлу із заміною порядку проходження байтів у файлі на протилежний.
3. Процес клієнт запитує кількість рядків у файлі.
4. Процес клієнт запитує вміст файлу із заміною всіх багаторазових входжень пробілів знаком табуляції.
5. Процес клієнт запитує, чи існує заданий файл.
6. Процес клієнт запитує, чи є заданий файл каталогом.
7. Процес клієнт запитує останній байт файлу.
8. Процес клієнт просить додати право на читання файлу всім категоріям користувачів.
9. Процес клієнт запитує, чи є заданий файл символічним посиланням.
10. Процес клієнт запитує вміст файлу з перекладом всіх латинських букв у верхній регістр
11. Процес клієнт запитує тимчасові характеристики файлу
12. Процес клієнт запитує довжину файлу і його тип.

Контрольні питання

1. Відкриваючи іменований канал для читання, процес припиняється доти, поки ще один процес не відкриє канал для запису. Як Ви думаєте, чому?
2. Що розуміється під гарантією атомарності операції запису?
3. Який максимальний розмір програмного каналу?
4. Перелічить всі демони у вашій системі і вкажіть їх функціональне призначення.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

Основна література

1. Князєва Н.О., Жуковецька С.Л., Трубіна Н.Ф. Системне програмування – Одеса: ВМВ, 2013 – 272с.
2. Галісєєв Г.В. Системне програмування. – Київ: Університет «Україна», 2019. – 113 с.

Допоміжна література

3. Richard Antony. System Programming. 1st Edition. Morgan Kaufman. ISBN 9780128007297, 2015. – 548 p.
4. М. Митчелл, Д.Оулдем, А.Саммюэл. Программирование для LINUX. Профессиональный подход. Пер. с англ., – М.: Издательский дом «Вильямс», 2002
5. К. Хэвиленд, Д. Грэй, Б. Салама. Системное программирование в UNIX. – М., ДМК Пресс, 2000
6. Mecklenburg Robert. Managing Projects with GNU Make, 3rd Edition – O'Reilly Media, 2004, Pages: 302
7. Michael Kerrisk. The Linux Programming Interface. A Linux and UNIX System Programming Handbook [. – No Starch Press, October 2010, Pages: 1552
8. Соколова Н.О., Вовк С.М., Єгоров А.О., Синхронізація потоків в операційних системах: навч. посіб – Дніпропетровськ: "Ліра", 2015. – 96с.

Електронні інформаційні ресурси

9. UNIX / LINUX Tutorial – Режим доступу: https://www.tutorialspoint.com/unix/index.htm
10. Using the GNU Compiler Collection (GCC) – Режим доступу: http://www.physics.ohio-state.edu/doco/gnu/gcc/gcc.html
11. Book Online: Introduction to Systems Programming: a Hands-on Approach by Gustavo A. Junipero Rodriguez-Rivera and Justin Ennen – Режим доступу: https://www.cs.purdue.edu/homes/grr/SystemsProgrammingBook
12. GCC online documentation – Режим доступу: https://gcc.gnu.org/onlinedocs/

ДОДАТОК А

ТЕКСТИ ПРИКЛАДІВ ДО ЛАБОРАТОРНОЙ РОБОТИ №1

example1a.c:

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

example1b.cpp:

```
#include <iostream>
using namespace std;
int main()
{
   cout<<"Hello, World!\n";
   return 0;
}
```

example2.c:

```
#include <stdio.h>
#define A 2
#define B 4
#define square_sum(a,b) a*a + b*b

void dummy()
{
    printf("Just a dummy\n");
}

static inline greeting()
{
    printf("Hi!\n");
}

int main(int argc, char *argv[])
{
#ifdef HI
    greeting();
#endif
    printf("%d\n", square_sum (A,B));
    return 0;
}
```

example3_main.c:

```
#include <stdio.h>

int print_with_timestamp(char *msg);
int main (int argc, char **argv)
 {
    print_with_timestamp("Hello, world!");
    return 0;
 }
```

example3_func.c:

```
#include <time.h>
#include <stdio.h>
#define TIME_STR_LEN 60
int print_with_timestamp(char *msg)
{
    char buf [TIME_STR_LEN];
    time_t  rawtime;
    struct tm *timeinfo;
    time(&rawtime);
    timeinfo = localtime(&rawtime);
    if (strftime(buf, TIME_STR_LEN, "%c", timeinfo) == 0)
    {
        return -1;
    }
    printf("%s: \"%s\"\n", buf, msg);
    return 0;
}
```

ДОДАТОК Б

ФУНКЦІЇ, РЕКОМЕНДОВАНІ ДЛЯ ВИКОРИСТАННЯ В ЛАБОРАТОРНІЙ РОБОТІ №3

defs.h

```
#if !defined(_DEFS_H)
#define _DEFS_H
#define MAX_STRING_SIZE 1000
#endif
```

Заголовні файли *print.h, func.h* створити самостійно. Обов'язково використовувати стражі включення.

main.c

```
#include "func.h"
#include "print.h"
#include "defs.h"
#include <stdio.h>

int main ()
{
  char str[MAX_STRING_SIZE];

  // читання рядка зі словами (роздільники - пробіл,
  // знак табуляції, крапка и кома)
  fgets(str, MAX_STRING_SIZE, stdin);
  char ** words;
  words = split(str," \t,.");
  //
  printwords(words);
  return 0;

}
```

func.c

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>


char **split(char * string, char *delimiters)
{
  char * pch;
  char** result;

  pch = strtok (string,delimiters);
  long unsigned size= sizeof(char *);
```

```
  result=malloc(size);


  int n=0;
  while (pch != NULL)
  {
    result[n]=pch;

    pch = strtok (NULL, delimiters);
    n++; size+=sizeof(char *);
    result = realloc(result, size);
  }
  result[n] = NULL;
  return result;

}
```

print.c

```
#include <stdio.h>
void printwords(char** words)
{
  int i=0;
  while(words[i]!=NULL)
     printf("--- %s\n", words[i++]);
}
```

ДОДАТОК В
СИГНАЛИ

Сигнали SUS розділено на групи. Для сигналів, що мають виключно штучне походження, цей факт відмічений особливо. Будь–який природний сигнал може бути породжений штучно.У списку символи в дужках описують дію за замовчуванням, прийняту для сигналу.

Розшифровка значень символів в дужках:

I – сигнал ігнорується (від англ. «*Ignore*»)

T – наводить до завершення процесу (від англ. «*Terminate*»)

A – те ж саме, що і T, але при цьому виконуються додаткові дії, визначувані системою, наприклад, створення файлу з дампом пам'яті процесу (від англ. «*Abort*»).

S – наводить до призупинення роботи (від англ. «*Stop*»)

C – продовження після зупинки (від англ. «*Continue*»)

Виявлені помилки

SIGBUS – апаратна помилка, залежна від реалізації, часто спроба доступу до невизначеної частини пам'яті. (*A*)

SIGFPE – помилка арифметичної операції. (*A*)

SIGILL – некоректна машинна команда. (*A*)

SIGPIPE – запис в канал, з якого ніхто не читає, або спроба виконати запис в сокет типу *SOCK_STREAM*, коли з'єднання вже розірване. (*T*)

SIGSEGV – неприпустиме звернення до сегменту пам'яті. (*A*)

SIGSYS – некоректне звернення до системного виклику. (*A*)

SIGXCPU – вичерпаний ліміт процесорного часу (перевищено м'яке обмеження). (*A*)

SIGXFSZ – перевищено м'яке обмеження на розмір файлу. (*A*)

Сигнали, що генеруються користувачем або застосуванням

SIGABRT – звернення до системного виклику *abort*(). (*A*)

SIGHUP – виявлений обрив зв'язку з терміналом (передається лідеру сеансу) або завершення термінального процесу (передається всім процесам з групи процесів переднього плану). (T)

SIGINT – переривання роботи, генерується драйвером терміналу при введенні символу переривання (зазвичай, *Ctrl+C*), посилається всім процесам з групи процесів переднього плану (T)

SIGKILL – “ліквідувати”, може мати лише штучне походження, дає можливість системному адміністратору знищити будь-який процес. (T)

SIGQUIT – завершити, генерується драйвером терміналу при введенні символу завершення (часто, *Ctrl+*\). (*A*)

SIGTERM – завершити, може мати лише штучне походження, посилається командою *kill* за умовчанням (T)

*SIGUSR*1 – сигнал користувача № 1, має лише штучне походження (T)

*SIGUSR*2 – сигнал користувача №2, має лише штучне походження (T)

Управління завданнями

SIGCHLD – дочірній процес завершив або припинив роботу (I)

SIGCONT – продовжити виконання призупиненого процесу, для активного процесу ігнорується. (C)

SIGSTOP – припинити роботу, може мати лише штучне походження (C)

SIGTSTP – сигнал з терміналу, припинити роботу, генерується драйвером терміналу при введенні символу призупинення (часто, *Ctrl+Z*) (S)

SIGTTIN – спроба фонового процесу виконати операцію читання з керуючого терміналу. (S)

SIGTTOU – спроба виконати операцію запису в керуючий термінал, якщо така можливість не дозволена процесом. (S)

Події таймера

SIGALRM – витік час таймера**,** встановлений функцією *alarm*() або функцією *setitimer*() (T)

SIGVTALRM – витік час віртуального таймера, встановлений функцією *setitimer*() (T)

SIGPROF – витік час профілюючого таймера, встановлений функцією *setitimer*() (T)

Інші події

SIGPOLL – сталася очікувана подія на опитуваному пристрої (T)

SIGTRAP – апаратна помилка, залежна від реалізації, часто використовується для передачі управління відладчикам після досягнення точки останову. (*A*)

SIGURG – сталася екстрена подія або доступні позачергові дані в сокеті (I)

Навчальне видання

СИСТЕМНЕ ПРОГРАМУВАННЯ

Методичні вказівки
до виконання лабораторних робіт
здобувачами першого (бакалаврського) рівня вищої
освіти, спеціальності 123 – Комп'ютерна інженерія

Укладачі
Трубіна Наталія Федорівна
Лісіцина Ірина Миколаївна

В авторській редакції

Підписано до друку 30.10.2023 р. Формат 60x84/16.
Папір офсетний. Гарнітура Times. Цифровий друк.
Ум. друк. арк. 5,00. Наклад 30. Зам. № 1123-0903.
Віддруковано з готового оригінал-макета.

Видавництво та друк: ОЛДІ+
65101, Україна, м. Одеса, вул. Інглезі, 6/1
Свідоцтво ДК № 7642 від 29.07.2022 р.

Тел.: +38 (098) 559-45-45,
+38 (095) 559-45-45, +38 (093) 559-45-45
Для листування: 65101, Україна, м. Одеса, вул. Інглезі, 6/1
E-mail: office@oldiplus.ua