МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторного практикуму з курсу
«Операційні системи та засоби комп'ютерної безпеки»

для студентів спеціальності
122 «Комп’ютерні науки»

Затверджено
редакційно-видавничою
радою університету,
протокол № 1 від 16.02.2023 р.

Харків
НТУ «ХПІ»
2023

Методичні вказівки до лабораторного практикуму з курсу «Операційні системи та засоби комп'ютерної безпеки» для студентів спеціальності 122 «Комп’ютерні науки» / уклад.: Багмут І.О., Мєтєльов В. О., Місюра С. Ю., Охотська О.В. – Харків: НТУ «ХПІ». – 2023. – 192 с.

Укладачі: І.О. Багмут, В. О. Мєтєльов, С.Ю. Місюра, О.В. Охотська

Рецензент доц. Г.Е. Заволодько

Кафедра комп’ютерного моделювання процесів та систем
Кафедра математичного моделювання та інтелектуальних обчислень в інженерії
Кафедра геометричного моделювання та комп'ютерної графіки

Зміст

Лабораторна робота 1

Консоль, віртуальні термінали та оболонка

Теоретичні відомості

Коли створювалася система UNIX, персональних комп'ютерів не було, і користувачі працювали на великих комп’ютерах (мейнфреймах) через безліч послідовних інтерфейсів для підключення віддалених терміналів.

Термінал – це пристрій, який призначений для взаємодії користувача з комп'ютером та складається з монітора та клавіатури. До персонального комп'ютера, як правило, не підключені віддалені термінали, але є клавіатура та монітор, які виконують роль терміналу користувача (тільки до його складу додалася миша).

Мейнфрейми мали особливий термінал, який призначався для системного адміністратора і називався консоллю. Консоль зазвичай під'єднувалася до комп'ютера не за послідовним інтерфейсом, а через окремі роз'єми (іноді як пристрій виведення до її складу замість монітора входив друкуючий пристрій).

Оскільки в UNIX-системах зазвичай дотримуються традиції, клавіатура та монітор персонального комп'ютера поводяться так само, як і раніше консоль. Перевага такого рішення полягає в тому, що всі старі програми, які створювалися для адміністраторів UNIX, без проблем працюють і на новому типі системної консолі.

Linux дозволяє підключати до комп'ютера як справжні віддалені термінали, так і забезпечує можливість роботи з віртуальними терміналами.

Оболонка, або просто shell – це програма, яка здійснює все спілкування з користувачем. Саме оболонка сприймає всі команди, які вводять користувач з клавіатури, і організує виконання цих команд. Тому оболонку можна назвати ще командним процесором (звичніший термін для користувача DOS). Коли кажуть, наприклад, «система виводить

запрошення», то мається на увазі, що запрошення виводить саме оболонка, очікуючи введення користувачем чергової команди. Щоразу, коли черговий користувач входить у систему, команда **login** запускає йому командний процесор – оболонку. Тобто оболонку можна розглядати як посередника між терміналом та операційною системою.

У користувача root запрошення закінчується символом #, а у решти користувачів – символом $.

Групи команд:

I. Перегляд інформації про систему, перегляд інформації про користувачів, що працюють у системі, отримання можливості виконувати команди з правами адміністратора, завершення роботи з терміналом, вихід із системи, команда **echo.**

II. Виведення поточного часу та дати.

III. Робота з файлами та каталогами.

IV. Перенаправлення стандартного потоку виводу**.**

Отримання довідки за командою.

Більшість розглянутих нижче команд підтримують ключ – **help**. Виклик команди із цим ключем виводить довідкову інформацію про команду.

I. Перегляд інформації про систему, перегляд інформації про користувачів, що працюють у системі, отримання можливості виконувати команди з правами адміністратора, завершення роботи з терміналом, вихід із системи, команда **echo**

1. uname [ключ]

Відображає деякі відомості про операційну систему, встановлену на комп'ютері. Ключі:

-o – назва операційної системи

-v – ім'я версії та дату компіляції ядра

-s – показує назву ядра системи (використовується за умовчанням, тобто коли команда викликається без ключів)

-r – ім'я релізу ядра системи

-m - тип обладнання (i386, i686, x86_64, Alpha)

Приклад 1.1.

delux@ubuntu :~$ uname Linux

Працюючи з правами простого користувача оболонка, зазвичай, відображає запрошення (на введення команд) з допомогою знака долара $.

delux@ubuntu :~$ uname -o GNU/Linux

delux@ubuntu :~$ uname -v

#33-Ubuntu SMP Sun Sep 19 20:32:27 UTC 2010

delux@ubuntu :~$ uname-sr (два ключі -s та -r об'єднані в один -sr)

Linux 2.6.35-22-generic

delux@ubuntu :~$ uname -m

x86_64 (Система команд x86, 64-бітна архітектура)

2. arch

Виводить машинну архітектуру

Приклад 1.2.

delux@ubuntu :~$ arch x86_64

3. whoami

Виводить ім'я користувача (що працює в оболонці).

Приклад 1.3. delux@ubuntu :~$ whoami delux

4. who

Виводить інформацію про користувачів, що увійшли до системи.

Приклади 1.4.

delux@ubuntu :~$ who

deluxpts/02011-09-16 13:42 (:1.0) (Емулятор терміналу)

deluxtty82011-09-16 12:28 (:1) (оболонка, що використовується при вході в ОС)

user2tty102011-09-16 13:11 (:2) (оболонка, що використовується при вході в ОС)

У першому стовпці виводяться імена (login) користувачів, у другому – пристрої з яких увійшов користувач до операційної системи (термінали, емулятори терміналів), у третьому – дата і час входу користувача в систему, у четвертому (необов'язковому) – показані імена систем з яких увійшов користувач**.**

Як видно, в системі є два користувача – delux і user2.

5. sudo -i

Дозволяє працювати з правами адміністратора або користувача root (суперкористувача). Також для цієї мети можна використовувати команду su, проте в Ubuntu отримати права адміністратора за допомогою її не вдасться.

Працюючи з root-правами оболонка, зазвичай, відображає запрошення з допомогою знака решітки #.

Приклад 1.5.

delux@ubuntu :~$ sudo -i

[sudo] password fordelux: (пароль під час введення не відображається)

root@ubuntu :~#

6. exit

Дозволяє завершити поточний сеанс роботи з оболонкою.

Приклад 1.6.

root@ubuntu :~# exit вихід

delux@ubuntu :~$

У цьому випадку завершено сеанс роботи на правах адміністратора. При повторному виклику команди exit буде виконано вихід із терміналу (програма емулятора терміналу завершить свою роботу).

7. shutdown

Дозволяє вимкнути/перезавантажити систему. Для виконання команди потрібні права адміністратора.

Формат команди **shutdown [OPTION] ЧАС [ПОВІДОМЛЕННЯ]. OPTION:**

-h – повна зупинка системи (комп'ютер буде вимкнено), використовується за замовчуванням;

-r – перезавантажити систему.

ЧАС: час (у хвилинах) через який відбудеться вимкнення/перезавантаження системи ПОВІДОМЛЕННЯ: частина текстового повідомлення, яке побачать користувачі у себе в терміналі при виклику даної команди.

Приклади 1.7.

root@ubuntu :~# shutdown 60 go home!

Широкомовне повідомлення від **delux@ubuntu (/dev/pts/0) о 14:04.**

Система є **going down for maintenance в 60 хвилин! go home!**

8. halt

Дозволяє вимкнути систему, аналогічна команді shutdown -h 0. Для виконання команди потрібні права адміністратора.

Приклад 1.8. root@ubuntu :~# halt

9. echo[TEXT]

Відображає введений текст.

Приклад 1.9. delux@ubuntu :~$ echo hello hello

II. Виведення поточного часу та дати

1. date

Виводить поточний час і дату (при виклику з ключем -s дозволяє встановити нові дату та час).

Приклад 1.10. delux@ubuntu :~$ date

Пт. сент. 16 13:52:43 MSD 2011

2. cal

Виводить календар на місяць.

Приклад 1.11.

delux@ubuntu :~$ cal

Вересень 2011

Пн Вт Ср Чт Пт Сб Нд 1 2 3 4

5 6 7 8 9 10 11

12 13 14 15 16 17 18

19 20 21 22 23 24 25

26 27 28 29 30

III. Робота з файлами та каталогами

1. pwd

Виводить назву робочого (поточного) каталогу.

Приклад 1.12. delux@ubuntu :~$ pwd

/home/delux

2. cd [ім'я_каталогу]

Дозволяє встановити новий робочий каталог. Якщо ім'я каталогу не вказано, в якості робочого встановлюється домашній каталог користувача. Домашні каталоги всіх користувачів системи знаходяться у каталозі /home.

Команда **cd** дозволяє перейти до батьківського каталогу по відношенню до робочого. Слід зазначити. що замість імені домашнього каталогу можна використовувати символ ~.

Приклади 1.13.

delux@ubuntu :~$ cd /home

delux@ubuntu :/home$pwd (у запрошенні вказано новий робочий каталог /home)

/home delux@ubuntu :/home$ cd

delux@ubuntu :~$pwd (у запрошенні вказано новий робочий каталог ~)

/home/delux

3. ls [ім'я_каталогу] [ключ]

Виводить вміст каталогу. Якщо каталог не вказано, за промовчанням відображає вміст робочого каталогу.

При використанні ключа -F після імен каталогів ставиться символ /, щоб можна було відрізнити каталоги файлів.

Приклади1.14.

delux@ubuntu :~/ossp$ ls -F

INTUIT/ Lection_OSSP_1-4.doc Practice_OSSP_1.odt

4. mkdir [ключ] ім'я_каталогу

Дозволяє створити новий каталог. Якщо вказано базове ім'я каталогу (не повне), за промовчанням новий каталог створюється в робочому каталозі. Якщо каталог вже існує, за промовчанням буде виведено повідомлення про помилку.

При використанні ключа -p команда не буде видавати повідомлення про помилку, якщо каталог вже існує.

Приклади 1.15.

delux@ubuntu :~/ossp$ mkdir 1

delux@ubuntu :~/ossp$ ls -F

1/ INTUIT/ Lection_OSSP_1-4.doc Practice_OSSP_1.odt delux@ubuntu :~/ossp$ mkdir 1

mkdir: неможливо створити каталог `1': Файл існує delux@ubuntu :~/ossp$ mkdir -p1(не видає повідомлення про помилку)

5. rmdir [ключ] ім'я_каталогу

Дозволяє видалити каталог, якщо він порожній. Якщо вказано базове ім'я каталогу (не повне), то за умовчанням мається на увазі, новий каталог перебуває у робочому каталозі.

Для видалення непустого каталогу (який містить файли) потрібно використовувати команду rm -r.

Приклади.1.16

delux@ubuntu :~/ossp$ rmdir 1

delux@ubuntu :~/ossp$ ls -F

INTUIT/ Lection_OSSP_1-4.doc Practice_OSSP_1.odt

6. cp

Дозволяє копіювати копіювання файлів. Або створює копію одного файлу, або копіює кілька файлів до каталогу.

Використання: **cp [ключ]** файл_джерело файл_призначення

або: **cp [ключ]** список_файлів_джерел каталог_призначення

Приклади 1.17.

delux@ubuntu :~/ossp$ ls*.txt (Відображення всіх файлів з розширенням txt у робочому каталозі)

1.txt

delux@ubuntu :~/ossp$ cp 1.txt 2.txt

delux@ubuntu :~/ossp$ ls *.txt

1.txt 2.txt

delux@ubuntu :~/ossp$ cp 1.txt 2.txt~ (копіювання 2-х файлів у домашній каталог)

delux@ubuntu :~/ossp$ cd

delux@ubuntu :~$ ls *.txt 1.txt 2.txt

7. mv

Дозволяє перейменовувати або переміщувати файли чи каталоги.

Використання: mv [ключ] існуючий_файл нове_ім'я_файлу або: **mv [ключ]** список_існуючих_файлів каталог або: mv [ключ] існуючий_каталог новий_каталог

Перший формат використовується для перейменування/перейменування одного файлу. Другий формат використовується для переміщення кількох файлів. Третій формат використовується для перейменування/перейменування каталогу.

Приклади 1.18.

delux@ubuntu :~/ossp$ mv 2.txt 3.txt

delux@ubuntu :~/ossp$ ls *.txt

1.txt 3.txt

delux@ubuntu :~/ossp$ mv 3.txt ~/2.txt

delux@ubuntu :~/ossp$ ls*.txt (Новий файл 2.txt знаходиться в домашньому каталозі)

1.txt

delux@ubuntu :~/ossp$ mv 1.txt 2.txt ~

delux@ubuntu :~/ossp$ ls *.txt

ls: неможливо отримати доступ до *.txt: Немає такого файлу або каталогу

delux@ubuntu :~/ossp$ cd

delux@ubuntu :~$ ls*.txt (Тепер файли 1.txt 2.txt знаходяться в домашньому каталозі)

1.txt 2.txt

delux@ubuntu :~/ossp$ mv dir1dir2 (перейменовує каталог dir1 в dir2)

delux@ubuntu :~/ossp$ mv dir2~/dir3 (перейменовує та переміщає каталог dir2 у каталог

~/dir3)

8. rm

Дозволяє видалити файли (каталог із файлами).

Використання: ср [ключ] список_файлів або: **ср** -r каталог

Перший формат використовується для видалення одного або декількох файлів, другий формат – для видалення каталогу з усім вмістом.

Приклади 1.19.

delux@ubuntu :~/ossp$ ls*.txt (Відображення всіх файлів з розширенням txt у робочому каталозі)

delux@ubuntu :~/ossp$ rm 1.txt 2.txt(Видалення 2-х файлів у робочому каталозі)

delux@ubuntu :~/ossp$ rm -r dir1(Видалення непустого каталогу dir1 в робочому каталозі)

9. touch имя_файла

Дозволяє створити новий (порожній) файл.

Приклади 1.2

delux@ubuntu :~/ossp$ touch 1.txt

10. cat ім'я_файлу

Одне з призначень команди – виведення вмісту (текстового) файлу стандартний потік виведення. За умовчанням як стандартний поток виводу використовується консоль.

Приклади 1.21.

delux@ubuntu :~/ossp$ cat 1.txt 1 2

3 4

5 6

7 8

9 10

11. tac ім'я_файлу

Одне із призначень команди – виведення вмісту (текстового) файлу у зворотному порядку прямування рядків у стандартний потік виведення. За умовчанням як стандартний поток виводу використовується консоль.

Приклади1.22.

delux@ubuntu :~/ossp$ tac 1.txt 9 10

7 8

5 6

3 4

1 2

12. head -число ім'я_файлу

Виводить перший n-рядок (параметр число) текстового файлу (стандартний потік виведення). За умовчанням як стандартний поток виводу використовується консоль.

Приклади 1.22

delux@ubuntu :~/ossp$ head -2 1.txt 1 2

3 4

13. tail - число ім'я_файлу

Виводить останній n-рядок (параметр число) текстового файлу (стандартний потік виведення). За умовчанням як стандартний поток виводу використовується консоль.

Приклади 1.23.

delux@ubuntu :~/ossp$ tail -2 1.txt 7 8

9 10

14. less имя_файла

Здійснює поекранне виведення текстових файлів у стандартний потік виведення. За умовчанням як стандартний поток виводу використовується консоль.

IV. Перенаправлення стандартного потоку виводу

За замовчуванням як стандартний вивід **stdout** використовується екран монітора. Команди **>,** >> дозволяють перенаправити стандартний потік виведення у файл:

1) > – перенаправляє стандартний потік у файл (інший потік). У цьому якщо файл існує, він перезаписується, якщо немає – створюється**.**

2) >> **–** перенаправляє стандартний потік у файл. При цьому, якщо файл існує, то інформація додається в кінець, якщо не існує – файл створюється.

Приклади 2.23.

delux@ ubuntu:~/ossp$ cal > 1.txt

delux@ubuntu :~/ossp$ cat 1.txt

Вересень 2011

Пн Вт Ср Чт Пт Сб Нд 1 2 3 4

5 6 7 8 9 10 11

12 13 14 15 16 17 18

19 20 21 22 23 24 25

26 27 28 29 30

delux@ubuntu :~/ossp$ echo 12345 >> 1.txt delux@ubuntu :~/ossp$ cat 1.txt

Вересень 2011

Пн Вт Ср Чт Пт Сб Нд 1 2 3 4

5 6 7 8 9 10 11

12 13 14 15 16 17 18

19 20 21 22 23 24 25

26 27 28 29 30

12345

Завдання до лабораторної роботи 1

Необхідно запустити емулятор терміналу та

1) У домашньому каталозі створити каталог.
2) В каталозі ~ (псевдонім домашнього каталогу) створити каталог lab1.
3) В каталозі ~/lab1 створити порожній файл 1.txt (можна використовувати інше ім'я та розширення).
4) Вивести у створений файл наступну інформацію:
    - назву та версію ядра операційної системи;
    - тип обладнання (архітектуру процесора);
    - інформацію про користувачів, які увійшли до системи;
    - поточний час та дату;
    - календар на поточний місяць;
    - назву робочого каталогу;
    - вміст домашнього каталогу.
5) Вивести на консоль вміст створеного файлу за допомогою команд:
    - cat;
    - tac;
    - less.
6) Вивести на консоль перші 5 рядків створеного файлу.
7) Вивести на консоль останні 5 рядків створеного файлу.

Лабораторна робота 2

Теоретичні відомості

Групи команд:

I. Перенаправлення стандартного потоку введення. Використання каналів. Конвеєри команд. Використання команди **cat** для конкатенації файлів. Команди **split, sort.**

II. Команди пошуку файлів та інформації у файлах – **find, locate, updatedb, grep.**

III. Права доступу до файлів та каталогів, команда **chmod.**

IV. Архіватори, команди **zip та unzip, gzip та gunzip, bzip та bzip2.** Пакувальник **tar.**

V. Встановлення та видалення програмного забезпечення, команди **dpkg, apt-get (apt).**

I Перенаправлення стандартного потоку введення. Використання каналів. Конвеєри команд. Використання команди **cat** для конкатенації файлів. Команди **split, sort**

1. Символ <.

За умовчанням як стандартний поток введення **stdin** використовується потік інформації, що вводиться з клавіатури. Символ < дозволяє перенаправляти вміст файлу на стандартний потік введення.

Насправді перенаправлення стандартного потоку введення використовується рідше ніж перенаправлення стандартного потоку виведення. Це пов'язано з тим, що багато системних команд можуть отримувати вхідну інформацію як з файлу, так і зі стандартного потоку

введення, тому використання і невикористання символу < у зв'язці з іншою командою можуть дати один і той же результат.

Символ < використовують у зв'язці з прикладними програмами, щоб не вводити інформацію з клавіатури.

Приклади 2.1

При використанні команди **cat** без аргументів (імені файлу), як вхідний поток, за замовчуванням, буде використовуватися потік **stdin**, тому інформація, що вводиться з клавіатури після натискання клавіші <Enter> буде відображатися на екрані.

delux@ubuntu :~/ossp$ cat12345(введення)

12345(відображення)

123(введення)

123(відображення), <Ctrl> + D – кінець введення (кінець файлу)

delux@ubuntu :~/ossp$

За допомогою символу < можна перенаправити вміст файлу на стандартний потік введення команди **cat**.

delux@ubuntu :~/ossp$ cat < 1.txt 1 2 3 4 5

Однак аналогічний результат можна отримати без використання символу <.

delux@ubuntu:~/osp$ cat<1.txt 1 2 3 4 5

2. Канал (конвеєр), символ |

Канал задається символом , що міститься між двома командами. Канал з'єднує стандартний потік виведення **stdout** першої команди зі стандартним потоком введення **stdin** іншої команди. Тобто, канал організує передачу вихідних даних першої команди на вхід іншої команди.

Приклади 2.2

delux@ubuntu :~/ossp$ cat1.txt (Відображення файлу без сортування) 1 10

7 11

2 12

5 13

0 14

delux@ubuntu :~/ossp$ cat 1.txt |sort (Відображення з сортуванням по рядках) 0 14

1 10

2 12

5 13

7 11

delux@ubuntu :~/ossp$ who |lpr (виведення списку користувачів системи на принтер)

delux@ubuntu :~/ossp$ who | sort |lpr (аналогічно, плюс сортування на ім'я)

3 Команда **sort** [ключ] имя_файла.

Виводить відсортований рядками текстовий файл на стандартний висновок. Рядки сортуються в лексикографічному пасмо.

За умовчанням використовується сортування за зростанням. Ключ -r дозволяє сортувати дані рядки зі спадання.

Приклади 2.3

delux@ubuntu :~/ossp$ sort 1.txt 0 14

1 10

2 12

5 13

7 11

delux@ubuntu :~/ossp$ sort -r 1.txt 7 11

5 13

2 12

1 10

0 14 1

4. Використання команди **cat** для конкатенації файлів**.**

Команда **cat** об'єднує кілька файлів і результат об'єднання надсилає стандартний висновок.

Приклади 2.4

delux@ubuntu :~/ossp$ cat 1.txt 1

2

delux@ubuntu :~/ossp$ cat 2.txt 3

4

delux@ubuntu :~/ossp$ cat 1.txt 2.txt 1

2

3

4

delux@ubuntu :~/ossp$ cat 1.txt 2.txt > 3.txt

delux@ubuntu :~/ossp$ cat 3.txt

1

2

3

4

5. Команда **split [ключ] [ім'я_файлу [префікс]].**

Команда **split** дозволяє розбити файл на частини. Якщо файл не заданий або заданий як – команда читає стандартне введення. Вихідний файл у своїй не знищується.

Як аргументи їй треба вказати ім'я вихідного файлу та префікс імен вихідних файлів. Імена вихідних файлів будуть складатися з цього префікса і двох додаткових літер 'aa', 'ab', 'ac' і т. д. (без пробілів та точок між префіксом та літерами). Якщо префікс імен файлів не заданий, то за замовчуванням використовується 'x', так що вихідні файли будуть називатися 'xaa', 'xab' і т.д. Інше число рядків можна встановити за допомогою ключа -l, за яким необхідно вказати необхідну кількість рядків. Порядкова розбивка зазвичай використовується для текстових файлів.

Ключ –b дозволяє виконувати побайтове розбиття файлу. Слідом за -b має стояти число, а за ним – буква k (що показує, що розмір вихідного файлу вказаний у Кб) або m (означає розмір заданий у Мб) і т. д. – g, t, p.

Приклади 2.5

delux@ubuntu :~/ossp$ split -l1 3.txt
delux@ubuntu :~/ossp$ cat xaa
1

delux@ubuntu :~/ossp$ cat xab 2
delux@ubuntu :~/ossp$ cat xac 3

delux@ubuntu :~/ossp$ cat xad 4

delux@ubuntu :~/ossp$ cat xaa xab xac xad > 4.txt

delux@ubuntu :~/ossp$ cat 4.txt

1

2

3

4

delux@ubuntu :~/ossp$ split -b30m book.djvu book_ (Розбиваємо файл на частини по 30 Мб)

delux@ubuntu :~/ossp$ ls book_* book_aa book_ab book_ac

II. Команди пошуку файлів та інформації у файлах – **find, locate**, **updatedb, grep**

1. Команда **find [список_каталогів] [ключ] [вираз]**

Виконує пошук файлів (каталогів) у списку_каталогів, що відповідають заданим критеріям (вираз). За замовчуванням пошук виконується у робочому каталозі. Також, за промовчанням, імена знайдених файлів виводяться у стандартний потік виводу (на екран). У виразі зазвичай задається ім'я або частина імені шуканого файлу.

Ключ -name дозволяє шукати файли на ім'я, за замовчуванням - без ключів, пошук виконується на ім'я. При використанні ім'я бажано поміщати в подвійні лапки.

Ключ -size дозволяє виконувати пошук за розміром файлів:

find -size 10M (пошук файлів розміром строго рівним 10 Мб)

find -size 10K -o -size -10K (пошук файлів розміром 10 Кб і менше, -o (OR))

find -size +1G -a -size -5G (пошук файлів розміром більшим за 1 Гб і меншим за 5 Гб, -a(AND))

Приклад 2.7.

delux@ubuntu :~$ find *1* 1.txt

Practice_OSSP_1.doc

delux@ubuntu :~/ossp$ find -name "*1*"

./Lection_OSSP_1-4.doc

./Practice_OSSP_1.odt

./1.txt

delux@ubuntu :~/ossp$ find -size +1M -a -size -100M

./book_ab

./book_aa

./book.djvu

./book_ac

2.Команда **locate [ключ] имя_файла**

Виконує пошук файлів (каталогів) на ім'я (частини імені) у системній базі даних імен файлів та каталогів, яка щодня оновлюється. За замовчуванням під час пошуку враховується регістр символів.

Ключ -i дозволяє не враховувати регістр символів.

Команда updatedb дозволяє оновити системну базу даних імен файлів перед пошуком (потрібні права адміністратора). Оновлення може виконуватися досить довго у разі наявності досить великої кількості файлів та малої продуктивності комп’ютера.

Приклади 2.8.

delux@ubuntu :~$ locate Practice

/home/delux/ossp/Practice_OSSP_1.odt

/home/delux/ossp/Practice_OSSP_2.odt

/home/delux/Робочий стіл/Посилання на Practice_OSSP_1.doc

3.Команда **grep [ключ] шаблон [список_файл**ів]

Виконує рядковий пошук даних в одному або декількох текстових файлах, використовуючи шаблон, який може бути простим рядком. Команда отримує введення з файлу (список_файлів) або зі стандартного введення.

Ключ-с дозволяє вивести лише кількість рядків, що містять відповідність шаблону в кожному файлі.

В результаті виконання команда виводить на стандартний висновок фрагменти файлів, в яких присутні рядки, що цікавлять.

Приклади 2.9

delux@ubuntu :~/asd$ grep List lab1.txt

Додати до класу List функцію пошуку значення максимального елемента серед елементів, що зберігаються у списку**.**

III.Права доступу до файлів та каталогів, команда **chmod**

Для кожного об'єкта у файловій системі Linux існує набір прав доступу, що визначає взаємодію користувача з цим об'єктом. Такими об'єктами можуть бути файли, каталоги, а також спеціальні файли (наприклад, пристрої) — тобто будь-який об'єкт файлової системи. Так, у кожного об'єкта в Linux є власник, то права доступу застосовуються щодо власника файлу.

Вони складаються з набору 3 груп по три атрибути:

читання (r), запис (w), виконання (x) для власника (творця файлу) - user (u); читання, запис, виконання групи власника — **group (g)**;

читання, запис, виконання всім інших — o**ther (o**). Такі права можна надати коротким записом:

ugo

----

rwxrwxrwx – дозволено читання, запис та виконання для всіх

rwxr-xr-x – запис дозволено лише для власника файлу, а читання та виконання для всіх.

rw-rw-r-- – запис дозволено для власника файлу та групи власника файлу, а читання – для всіх. Такий розподіл прав дозволяє гнучко управляти ресурсами, доступними користувачам.

Права доступу поширюються і каталоги. Вони означають:

r – якщо встановлено право читання з каталогу, можна побачити його вміст командою ls .

w – якщо встановлено право запису до каталогу, то користувач може створювати та видаляти файли у каталозі.

x – якщо встановлено право виконання на каталог, то користувач має право перейти до такого каталогу, наприклад, командою cd.

Слід зазначити, що системний адміністратор має доступ до всіх файлів та каталогів.

1. Команда **ls** -l ім'я_файлу_або_каталогу

Дозволяє переглянути права доступу для певного файлу або каталогу.

Приклади 2.9.

delux@ubuntu :~/ossp$ ls -l 1.txt

-rw-r--r-- 1 delux delux 4 2011-09-23 11:48 1.txt

В даному випадку тільки власник має право на запис (зміна) та читання файлу, решта користувачів може тільки читати файл. Одиниця означає, що на даний файл існує одне посилання, далі йде ім'я власника та ім'я групи власника файлу, далі розмір в байтах, дата і час створення файлу та ім'я файлу.

2. Команда **chmod.**

Команда **chmod** (Change MODe – змінити режим) – змінює права доступу до файлу. Для використання цієї команди також необхідно мати права власника файлу або права root.

Синтаксис команди

chmod права_доступу ім'я_файлу, де:

права_доступу – права доступу, що встановлюються на файл. имя_файла – ім'я файлу, у якого змінюються права доступу.

Використання команди chmod представлено на рис. 1.

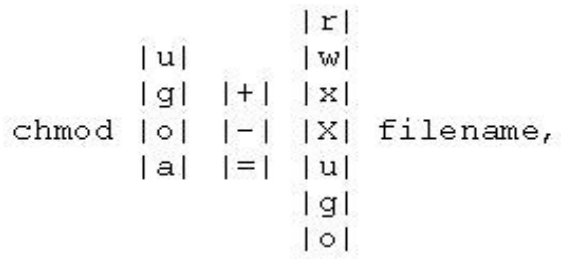

Рис. 1

Де:

u – встановлення прав для користувача; g – встановлення прав для групи;

o – встановлення прав для інших користувачів; a – встановлення прав всім груп **(u, g і o).**

+,-,= – додати, видалити, встановити дозвіл відповідно.

r – право читання;

w – право запису;

x – право читання виконання;

X – право виконання якщо є таке право ще в якійсь із груп доступу;

u – такі ж права як у власника;

g – такі ж права, як у групи;

o – такі ж права, як у інших користувачів. filename – Ім'я файлу, у якого змінюються права**.**

Приклади. **2.10**

Дозволимо всім користувачам змінювати файл 1.txt.

delux@ubuntu :~/ossp$ chmod a+w 1.txt

delux@ubuntu :~/ossp$ ls -l 1.txt

-rw-rw-rw- 1 delux delux 4 2011-09-21 19:48 1.txt

delux@ubuntu :~$ ./Qt_SDK_Lin64_offline_v1_1_3_en.run

bash: ./Qt_SDK_Lin64_offline_v1_1_3_en.run: Відмовлено у доступі

При запуску інсталятора системи розробки програм з використанням бібліотеки Qt отримали помилку «відмовлено у доступі». Це з тим, що з запуску установника необхідно встановити йому права виконання.

delux@ubuntu :~$ ls -l Qt_SDK_Lin64_offline_v1_1_3_en.run

-rw 1 delux delux 1287828028 2011-09-08 23:52 Qt_SDK_Lin64_offline_v1_1_3_en.run

Додамо право доступу на виконання для власника файлу та спробуємо запустити інсталятор ще раз.

delux@ubuntu :~$ chmod u+x Qt_SDK_Lin64_offline_v1_1_3_en.run

delux@ubuntu :~$ ls -l Qt_SDK_Lin64_offline_v1_1_3_en.run

-rwx------ 1 delux delux 1290846804 2011-09-09 00:34 Qt_SDK_Lin64_offline_v1_1_3_en.run

delux@ubuntu :~$ ./Qt_SDK_Lin64_offline_v1_1

IV. Архіватори, команди **bzip** та **bunzip2, gzip** та **gunzip.** Пакувальник **tar**

Архіватори дозволяють виконувати компресію (стиснення) та декомпресію файлів. Пакувальник дозволяє набір файлів запакувати в один файл (тому) в загальному випадку без стиснення.

1. Команди **bzip2, bunzip2 bzip2 [ключ] [список_файлів]**
bunzip2 [ключ] [список_файлів]

Команда bzip2 здійснює стиснення файлів, команда bunzip2 відновлює файли, стиснуті утилітою bzip2. Список_файлів - це список, що складається з одного або декількох файлів (які не є каталогами), які потрібно стиснути або відновити. Якщо список_файлів відсутні команди працюють зі стандартним введенням та виведенням.

За замовчуванням команди видаляють вихідні файли після виконання своєї роботи - bzip2 видаляє файл, що стискається, bunzip2 видаляє стислий файл.

Ключ -k дозволяє зберегти вихідні файли.

Приклади 2.11.

delux@ubuntu :~$ ls 1* 1.txt

delux@ubuntu :~$ bzip2 -k 1.txt
delux@ubuntu :~$ ls 1*
txt 1.txt.bz2

delux@ubuntu :~$ bzip2 1.txt 2.txt (архівування 2-х файлів)

delux@ubuntu :~$ ls *.bz2 1.txt.bz2 2.txt.bz2

delux@ubuntu :~$ bunzip2 1.txt.bz2 2.txt.bz2 (Розархівування 2-х файлів)

2 Команди **gzip, gunzip**

gzip [ключ] [список_файлів] gunzip [ключ] [список_файлів]

Команда **gzip** здійснює стиснення файлів, команда **gunzip** відновлює файли, стиснуті утилітою **gzip**. Список_файлів – це список, що складається з одного або декількох файлів (загалом не є каталогами), які потрібно стиснути або відновити. Якщо список_файлів відсутні команди працюють зі стандартним введенням та виведенням.

Команди видаляють вихідні файли після виконання своєї роботи.

Ключ -r списку_файлів дозволяє стискати файли в каталозі (проте сам каталог не буде стиснутий), тобто як можна вказати каталог.

Приклади 2.12.

delux@ubuntu :~$ gzip 1.txt 2.txt (архівування 2-х файлів)
delux@ubuntu :~$ ls *.gz 1.txt.gz 2.txt.gz

delux@ubuntu :~$ gunzip 1.txt.gz 2.txt.gz (Розархівування 2-х файлів)

delux@ubuntu :~$ gzip -r asd (архівування файлів у каталозі asd)
delux@ubuntu :~/asd$ ls *.gz

AlgStrDan_Lab1.DOC.gz AlgStrDan_Lab4.DOC.gz

AlgStrDan_Lab7.DOC.gz AlgStrDan_Lab2.DOC.gz

AlgStrDan_Lab5.DOC.gz lab1.txt.gz AlgStrDan.

delux@ubuntu :~/asd$ cd

delux@ubuntu :~$ gunzip -r asd (Розархівування файлів у каталозі asd)

Команда tar

tar [ключи] [ім'я_архіву] [список_файлів_і_або_каталгів]

Команда tar здійснює упаковку/розпакування зазначених файлів та/або каталогів в один файл (архів). За замовчуванням стиснення файлів не виконується.

Ключ -cf використовується при упаковці даних, і призначений для встановлення імені файлу архіву і свідчить про те, що вхідні дані для упаковки знаходяться у файлах, а не в стандартному потоці введення.

Ключ -xf використовується при розпакуванні даних, призначений для встановлення імені файлу архіву.

Приклади 2.13.

delux@ubuntu :~$ tar -cf asd.tar asd (упаковка каталогу asd у файл asd.tar)

delux@ubuntu :~$ tar -xf asd.tar (Розпакування файлу asd.tar в каталог asd)

Ключ -zcf дозволяє одночасно упаковувати файли та стискати їх за допомогою утиліти gzip. Ключ -zxf дозволяє одночасно розпаковувати файли та розтискати їх за допомогою утиліти gunzip. Як правило, файл отриманий в результаті робіт утиліт tar і gzip закінчується символами .tar.gz або .tgz або .tar.gzip.

Приклади 2.14.

delux@ubuntu :~$ tar -zcf asd.tar.gz asd (упаковка та компресія каталогу)

delux@ubuntu :~$ tar -zxf asd.tar.gz (декомпресія та розпакування каталогу)

Ключ -jcf дозволяє одночасно упаковувати файли та стискати їх за допомогою утиліти bzip2. Ключ -jxf дозволяє одночасно розпаковувати файли та розтискати їх за допомогою утиліти bunzip2. Як правило, файл отриманий в результаті робіт утиліт tar і bzip2 закінчується символами .tar.bz2 або .tbz2 або .tar.bzip2.

Приклади 2.15.

delux@ubuntu :~$ tar -jcf asd.tar.bz2 asd (упаковка та компресія каталогу)

delux@ubuntu :~$ tar -jxf asd.tar.bz2 (декомпресія та розпакування каталогу)

V. Встановлення та видалення програмного забезпечення, команди **dpkg, apt-get (apt)**

У системі Linux програмне забезпечення можна встановити за допомогою двох основних способів:

1) шляхом інсталяції інсталяційного пакета програми (програм);
2) компіляцією програмного забезпечення із вихідних джерел.

Останнім часом найчастіше використовується перший спосіб. Розглянемо його.

Для встановлення пакетів в ОС Linux типу Debian (до цього типу належить Ubuntu) використовуються команди (утиліти) **dpkg, apt-get.**

Утиліта dpkg виконує базову «чорнову» роботу з встановлення та видалення програмного забезпечення.

Утиліта apt-get (apt) є більш зручною для оновлення пакетів, встановлення взаємопов'язаних пакетів і функціонує як оболонка навколо утиліти dpkg.

Для встановлення програмного забезпечення необхідно мати права адміністратора. У Ubuntu перед командами **dpkg, apt-get** для цього можна використовувати команду **sudo**.

Слід зазначити, що у разі використання утиліти **dpkg**, при видаленні пакетів вони задаються власними іменами, а чи не іменами файлів, з яких вони встановлені. Наприклад, при установці файлового менеджера Midnight Commander (аналог Far, Total Commander) може використовуватися ім'я файлу пакета mc_4.7.0.6-1_amd64.deb, а при видаленні пакета необхідно використовувати його назву - mc.

При використанні програми apt-get в обох випадках використовуються назви пакетів, а не імена файлів файлів пакетів.

За промовчанням програми встановлюються в каталог /usr/bin/.

1. Команда **dpkg [ключ] [пакет] Ключ -i** дозволяє встановити пакет. **Ключ -r** дозволяє вилучити пакет.

Ключ –purge дозволяє видалити пакет разом із налаштуваннями.

Приклади 2.14.

delux@ubuntu :~/Завантаження$ sudo dpkg -i mc_4.7.0.6-1_amd64.deb (Установка mc)

mc (виклик mc)

delux@ubuntu :~/Завантаження$ sudo dpkg --purgemc (Видалення mc)

Команда apt-get [ключ] [параметр] [пакет]

Параметр install дозволяє встановити пакет. Якщо для встановлення заданого пакета потрібно встановити інші пакети (тобто пакети є взаємопов'язаними), команда їх (знайде та) встановить самостійно.

Параметр remove дозволяє видалити пакет.

Параметр upgrade оновлює раніше встановлені пакети до нових версій.

Ключ –purge дозволяє видалити пакет разом із налаштуваннями.

Приклади.

delux@ubuntu :~/Завантаження$ sudo apt-get installmc (Установка mc)

delux@ubuntu :~/Завантаження$ sudo apt-get removemc (Видалення mc)

Завдання до лабораторної роботи 2

Необхідно запустити емулятор терміналу та:

1) В каталозі ossp2022 ~ створити каталог lab2.

2) В каталозі lab2 створити файл 1.txt (можна використовувати інше ім'я та розширення) що складається з 9 рядків (у першому рядку – цифра 1, у другому – 2, тощо).

3) За допомогою команди split розбити цей файл на три частини (три файли).

4) За допомогою команди cat об'єднати нові три файли в один з назвою 2.txt і вивести вміст на екран.

5) Вивести на екран список файлів, розмір яких менше 100 Кб.

6) Вивести на екран рядки файлу 1.txt, що містять цифру 1

7) За допомогою команди ls –l вивести інформацію про права доступу до файлу 1.txt.

8) За допомогою команди chmod заборонити запис до файлу 1.txt та його виконання для всіх користувачів. Вивести інформацію про права доступу до файлу 1.txt.

9) Упакувати та стиснути каталог ossp в файл asd.tar.gz. Виконати зворотну операцію.

10) Встановити файловий менеджер Midnight Commander (mc).

Лабораторна робота 3

Теоретичні відомості

Групи команд:

I. Перегляд інформації про процеси. Управління процесами. Команди **ps, top, bg (символ &), fg, nice, kill**, комбінації клавіш <Ctrl>+<Z>, <Ctrl>+<C>

1. Команда ps [ключ] [список процесів].

Утиліта s виводить інформацію про стан процесів, запущених на локальній системі. Список процесів - це розділений пробілами список PID-номерів. Якщо використовувати список процесів, ps виводить лише звіт про процеси, які вказані у списку.

Ключі:

-A, A – (all - все), виведення звіту про всі процеси (аналог ключа -e);

-e, e - (everything), виведення звіту про всі процеси (аналог ключа -a);

-f, f - (Full - повний), виведення лістингу, що містить більше стовпців з інформацією;

-l, l - (long - довгий), виведення довгого лістингу, що показує більше інформації про кожний процес;

-u – висновок інформації про процеси, запущені користувачами, чиї імена або UID-номери перераховані в аргументі імена_користувачів з використанням двокрапки як роздільник. Без ключів ps виводить стан всіх активних процесів, керованих терміналом.

Інформація про процеси виводиться по стовпцям, які мають такі заголовки:

PID – ідентифікаційний номер процесу;

TTY - назва терміналу керуючого процесом (у якому було запущено процес);

TIME – час витрачений на запуск (завантаження) процесу (годинник, хвилина, секунда);

CMD або COMMAND - Командний рядок, з якого був викликаний процес (може усікатися);

%CPU або C – відсоток, використовуваний процесом від часу центрального процесора (приблизно);

%MEM – відсоток використовуваної процесом оперативної пам'яті (RAM);

F – прапори, пов'язані із процесом;

NI – значення nice процесу;

PPID – ідентифікаційний номер батьківського процесу;

PRI - пріоритет процесу;

RSS – кількість блоків пам'яті використовуваної процесом;

STIME або START - дата старту процесу;

STAT або S – стан процесу:

< - високопріоритетний;

D – призупинено і не піддається перериванню;

N – низькопріоритетний;

R – доступний для виконання (перебуває у черзі готових на запуск процесів);

S – призупинено;

T – або зупинений або трасований;

X – непрацюючий;

Z – зомбі-процес (процес у яких відсутній батьківський процес). USER або UID – ім'я користувача, який володіє процесом (що запустив процес). WCHAN – функція ядра що змушує процес перебуває у режимі очікування.

Приклади 3.1.

Виведемо список процесів керованих терміналом (запущених у терміналі).

delux@ubuntu :~$ ps

PID TTY TIME CMD

2538 pts/0 00:00:00bash (сама оболонка, що обробляє команди)

2557pts/000:00:00 ps (команда ps)

Виведемо перелік всіх процесів системи.

delux@ubuntu :~$ ps -e PIDTTYTIME CMD1?00:00:00 init

2?00:00:00 kthreadd

3?00:00:00 ksoftirqd/0

…

2538pts/000:00:00 bash

2617pts/000:00:00 ps

Виведемо повнішу інформацію про процеси керованих терміналом.

delux@ubuntu :~$ ps -f

UIDPID PPID C STIME TTY TIME CMD

delux2538 2534 0 15:23 pts/0 00:00:00 bash

delux2737 2538 0 15:37 pts/0 00:00:00 ps -f

delux@ubuntu :~$ ps f

PIDTTYSTAT TIME COMMAND

2782pts/0Ss0:00bash 2902pts/0R+0:00_ ps f

Виведемо найповнішу інформацію про процеси керованих терміналом.

delux@ubuntu :~$ ps -l

FS UID PID PPID C PRI NIADDR SZWCHAN TTYTIME CMD

0 S 1000 2538 2534 4 80 0 - 5322 wait pts/0 00:00:00 bash

0 R 1000 2756 2538 0 80 0 - 2033 - pts/0 00:00:00 ps

delux@ubuntu :~$ ps l

F UID PID PPID PRI NI VSZ RSS WCHAN STAT TTY

TIME COMMAND

01000 2782 2778 20 0 21288 3932 wait Sspts/0 0:00bash

01000 2899 2782 20 0 8132 760 - R+pts/0 0:00ps l

Виведемо інформацію про процеси користувача delux.

delux@ubuntu :~$ ps -u delux

PID TTY TIME CMD

1581 ?00:00:00 gnome-keyring-d

1600 ?00:00:00 gnome-session

1630 ?00:00:00 ssh-agent

1633 ?00:00:00 dbus-launch

2. Команда **top** [ключ] [ім'я_].

Виводить інформацію про стан системи, включаючи відомості про поточні процеси. Інформація оновлюється за кілька (3) секунд.

Ключі:

-i – ігнорування непрацюючих (простоюючих) та зомбі-процесів.

-n – визначає кількість процесів, відомості про які потрібно вивести.

У перших кількох рядках утиліта виводить загальні відомості про стан системи. Виведення будь-якого з цих рядків можна увімкнути/вимкнути за допомогою командних клавіш.

У першому рядку виводиться поточний час, кількість часу, що минув з моменту останнього завантаження системи, кількість користувачів, що ввійшли в систему, і середні показники завантаженості за останню хвилину, 5 хвилин і 15 хвилин (включається/вимикається клавішею <l>). У другому рядку показується кількість запущених процесів (вмикається та вимикається клавішею <t>).

Останні три рядки виводять відомості про використання центрального процесора (також включається та вимикається клавішою <t>), пам'яті (включається та вимикається клавішою <m>) та свопованого простору (також включається та вимикається клавішею <m>).

Вся решта виведеної інформації відноситься до відомостей про окремі процеси, перерахованих у порядку зменшення часу використання центрального процесора (тобто спочатку виводяться відомості про процес, що використовує центральний процесор інтенсивніше). За замовчуванням утиліта top виводить інформацію про таку кількість процесів, що міститься на екрані.

Назви та значення більшої частини полів, що виводяться для кожного процесу аналогічні значенням полів, що виводяться утилітою ps. Інші поля мають такі назви та значення: PR – пріоритет процесу;

VIRT - кількість кілобайт віртуальної пам'яті, що використовується процесом;

RES – кількість кілобайт фізичної (несвопірованої) пам'яті використовуваної процесом; SHR – кількість кілобайт спільно використовуваної пам'яті, використовуваної процесом; TIME+ – загальна кількість часу центрального процесора, використаного процесом. За допомогою кнопки <q> можна завершити роботу утиліти.

Приклад 3.2.

delux@ubuntu :~$ top

top - 16:29:32 2:02, 2 користувачів, load average: 0.14, 0.04, 0.01

Tasks: 170 total, 1 running, 169 sleeping, 0 stopped, 0 zombie

Cpu(s): 0.2%us, 0.2%sy, 0.0%ni, 99.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st Mem: 4056208k total, 2458560k used, 1596848

Swap: 261116ktotal,0k used, 261116k free, 1054472k cached

PIDUSERPR NI VIRT RES SHRS%CPU %MEMTIME+ COMMAND

1070 root 200164m 20m 9.9mS0 0.5 0:50.04 Xorg

1863 delux 200965m 96m 39mS0 2.4 0:29.05 rhythmbox

...

delux@ubuntu :~$ top -i

top - 16:38:53 2:11, 2 користувачів, load average: 0.03, 0.06, 0.00

Tasks: 170 total, 2 running, 168 sleeping, 0 stopped, 0 zombie

Cpu(s): 1.1%us, 0.7%sy, 0.0%ni, 98.2%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st Mem: 4056208k, 2461584k, 1598k64

Swap: 261116ktotal,0k used, 261116k free, 1055580k cached

PIDUSERPR NI VIRT RES SHRS%CPU %MEMTIME+ COMMAND 3004delux20 019276 1356 956R00.00:00.07 top

3. Символ &.

Символ & дозволяє запустити процес у фоновому режимі.

Приклади.

Створимо кілька сценаріїв loop1, loop2, loop3 оболонки bash які реалізує нескінченний цикл, і запустимо їх як програму у фоновому режимі.

delux@ubuntu :~$ echo "while true; do true; done" > loop1

delux@ubuntu :~$ chmod u+x loop1

delux@ubuntu :~$ cp loop1 loop 2

delux@ubuntu :~$ cp loop1 loop2

delux@ubuntu :~$ cp loop1 loop3

delux@ubuntu :~$ ./loop1& [1] 3203

delux@ubuntu :~$ ./loop2& [2] 3204

delux@ubuntu :~$ ./loop3& [3] 3205

Кожному запущеному у фоні сценарію було надано номер у квадратних дужках — номер завдання та номер процесу: 3203-3205.

Приклад 3.3.

delux@ubuntu :~$ ps -f

UID PID PPID C STIME TTY TIME CMD

delux 3182 3178 0 16:59 pts/0 00:00:00 bash

delux 3203 3182 51 16:59 pts/0 00:02:22 bash

delux 3204 3182 59 16:59 pts/0 00:02:42 bash

delux 3205 3182 88 16:59 pts/0 00:03:56 bash

delux 3226 3182 0 17:04 pts/0 00:00:00 ps -f

4. Команда **fg** [ключ] [номер_задания].

Дозволяє перевести фонове завдання (команду) у активний режим.

Замість імені команди можна використовувати символ відсотка (%), за яким необхідно вказати номер завдання.

Перекладемо завдання №1 з фонового в активний режим і завершимо його комбінацією клавіш

<Ctrl>+<C>.

Приклад 3.4.

XXXXXXXXX

delux@ubuntu :~$ fg %1 ./loop1 ^ C

delux@ubuntu :~$ ps

PIDTTYTIME CMD

3182pts/000:00:00 bash

3204pts/000:04:36 bash

3205pts/000:07:08 bash

3249pts/000:00:00 ps

5. Команда **bg [номер_задания**].

При натисканні клавіш <Ctrl>+<Z> завдання, яке виконується в активному режимі, негайно припиняється і виводиться повідомлення, до якого вмикається слово Stopped (Зупинено).

Команда **bg** відновлює виконання зупиненого завдання у фоновому режимі.

Приклад 3.5.

Перекладемо завдання №2 із фонового в активний режим. Припинимо його, натиснувши <Ctrl>+<Z>, та відновимо його роботу у фоновому режимі за допомогою команди bg.

delux@ubuntu :~$ fg %2 ./loop2 ^Z [2]+ Зупинено ./loop2

delux@ubuntu :~$ ps l 3204

F UID PID PPID PRINIVSZ RSS WCHAN STATTTYTIME COMMAND

1 1000 3204 318220021304 2824signalTpts/017:37 bash

delux@ubuntu :~$ bg 2

[2]+ ./loop2 & delux@ubuntu :~$ ps l 3204

F UID PID PPID PRINIVSZ RSS WCHAN STATTTY TIME COMMAND1 1000 3204 318220021304 2824-Rpts/017:59 bash

6. Команда **nice [ключ]** командний рядок.

Дозволяє змінити пріоритет команди (процесу) або повідомляє пріоритет оболонки (терміналу). Звичайний користувач може лише знизити пріоритет команди, для підвищення пріоритету необхідно мати права адміністратора.

Командна строка – це командний рядок, який потрібно виконати з іншим рівнем пріоритету. Без ключів та аргументів nice виводить рівень пріоритету оболонки, що запустила nice.

Ключ -n інкремент – позитивний інкремент знижує поточний пріоритет, а негативний інкремент його підвищує на значення інкременту. Інкремент має діапазон від -20 (найвищий) до 19 (найнижчий). При вказівці

інкременту, в результаті якого пріоритет може вийти за межі допустимого діапазону, пріоритет встановлюється за його межею.

Без ключів команда Nice знижує рівень пріоритету до 10.

Приклад 3.6

Запустимо ще раз сценарії loop1 та loop2, але для loop1 знизимо пріоритет. Виведемо показники процесів.

delux@ubuntu :~$ nice -n 19 ./loop1& [1] 3519

delux@ubuntu :~$ ./loop2& [2] 3520

delux@ubuntu :~$ ps l

F UIDPIDPPID PRINIVSZRSS WCHAN STATTTYTIME COMMAND

0 1000 3500 3496 20 0 21304 3944 wait Ss pts/1 0:00 bash

0 1000 3519 3500 39 19 4148 576 - RN pts/1 0:20 /bin/sh ./l

1 1000 3520 3500 20 0 21304 2820 - R pts/1 2:36 bash

0 1000 3535 3500 20 0 8132 764 - R+ pts/1 0:00 ps l

7. Команда **renice -n інкремент** список процесів

Дозволяє змінити пріоритет запущених процесів, для підвищення пріоритету необхідно мати права адміністратора.

Список_процесів – це список PID-номерів процесів (через пропуск), пріоритет яких має бути змінено.

Ключ -n інкремент – Позитивний інкремент знижує поточний пріоритет, а негативний інкремент його підвищує на значення інкременту.

Ключ -u (перед списком_процесів) – дозволяє трактувати список_процесів як список_користувачів, у яких потрібно змінити пріоритет процесів.

Приклад 3.7.

Змінимо пріоритети процесів. Тепер loop1 матиме високий пріоритет, а loop2 низький.

delux@ubuntu :~$ sudo renice -n -20 3519 [sudo] password for delux:

3519: старий пріоритет 19, новий пріоритет -20

delux@ubuntu :~$ renice -n 19 3520

3520: старий пріоритет 0, новий пріоритет 19

delux@ubuntu :~$ ps l

F UID PID PPID PRI NI VSZ RSS WCHAN STAT TTY TIME COMMAND

0 1000 35003496 200 21304 3944 wait Ss pts/1 0:00bash

0 1000 35193500 0-20 4148 576 - R< pts/1 13:56 /bin/sh ./l

1 1000 35203500 3919 21304 2820 - RN pts/1 11:00 bash

0 1000 36203500 200 8132 764 - R+ pts/1 0:00ps l

Як видно, тепер процес з PID = 3519 (loop1) має більший пріоритет і займає більше процесорного часу.

8. Команда **kill [ключ] [ім'я_].**

Виклик:

1) kill [ім'я_сигналу_або_номер_сигналу] PID-список;

2) kill -l [назва_або_номер_сигналу].

Команда **kill** посилає сигнал одному чи кільком процесам. Зазвичай цей сигнал завершує процеси. Для коректного виконання команди необхідно, щоб процеси, яким відправляються сигнали, належали користувачеві, який використовує команду **kill**. Природно, користувач з правами адміністратора може завершити будь-який процес.

PID-список – це список PID-номерів процесів, яким надсилається сигнал.

Команда kill ім’я_сигналу PID-список дозволяє відправити сигнал, заданий на ім'я, списку процесів.

Команда kill номер_сигналу PID-список дозволяє надіслати сигнал, заданий за номером, списком процесів.

Команда kill -l дозволяє вивести повний перелік номерів та імен сигналів. Команда kill -l имя_сигнала виводить номер відповідного сигналу.

Команда kill -l номер_сигналу виводить ім'я відповідного сигналу.

Замість PID-номера процесу можна вказати номер завдання як %n, де n — номер. Найчастіше використовувані сигнали такі:

Іменомір призначення

SIGKILL або 9 завершення процесу, що не може бути проігнорована процесом

KILL SIGTERM або 15 завершення процесу, може бути проігнорована процесом

TERM SIGTSTP або 20 натискання клавіші переривання терміналу (<Ctrl>+<Z>) TSTP

Приклади 3.8.

delux@ubuntu :~$ ./loop1& [1] 3788

delux@ubuntu :~$ ./loop2& [2] 3789

delux@ubuntu :~$ ./loop3& [3] 3790

delux@ubuntu :~$ ps

PIDTTYTIME CMD

3767pts/100:00:00 bash

3788pts/100:00:16 bash

3789pts/100:00:12 bash

3790pts/100:00:11 bash

3791pts/100:00:00 ps

delux@ubuntu :~$ kill 9 3788

bash: kill: (9) - Операція не дозволяється

delux@ubuntu :~$ ps

PIDTTYTIME CMD

3767pts/100:00:00 bash

3789pts/100:00:18 bash

3790pts/100:00:17 bash

3794pts/100:00:00 ps

[1]Завершено./loop1

delux@ubuntu :~$ kill 15 3789

bash: kill: (15) - Операція не дозволяється

delux@ubuntu :~$ ps

PIDTTYTIME CMD

3767pts/100:00:00 bash

3790pts/100:00:58 bash

3798pts/100:00:00 ps

[2] -Завершено./loop2

Завдання лабораторної роботи 3

1. Виведіть на екран вміст вашого домашнього каталогу.

2. Створіть два піддерева з одного та з двох каталогів. (Каталог в каталозі, в якому знаходиться 2 каталоги)

3. Зробіть поточним останній каталог меншого дерева, будь-який каталог з двох.

4. Визначте повне маршрутне ім'я. (Повна адреса каталогу)

5. Змініть останній каталог на підкаталог більшого дерева, в якому вже знаходиться 2 каталоги.

6. Визначте повне маршрутне ім'я.

7. Створіть 3-4 текстові файли.

8. Створіть два нові паралельні підкаталоги.

9. В один підкаталог скопіюйте наявні текстові файли поточного каталогу зі зміною імен, а в інший – перемістіть. Проаналізуйте як змінився зміст всіх каталогів.

10. Запишіть в один із текстових файлів Hello world!.

11. Перемістіть текстовий файл, до якого додали текст, до домашнього каталогу та перегляньте його зміст.

12. Створіть посилання з іншим ім'ям в одному з підкаталогів на один із текстових файлів іншого підкаталогу. Команда ln

13. Створіть ще одне посилання з іншим ім'ям.

14. Створіть посилання на кілька файлів, що належать одному з каталогів, в іншому каталозі одним командним рядком.

15. Видаліть всі створені вами каталоги та файли. Отримайте підтвердження виконання команд із вмісту домашнього каталогу.

Лабораторна робота 4

Теоретичні відомості

Групи команд:

I. Планування завдань, команди **at, atc, atrm, crontab**.

II. Отримання інформації про команди, команди **man, whatis, apropos, which**.

I. Планування завдань, команди at, atc, atrm, crontab

1. Команда **at [ключ] [ім'я_файлу] час [дата].**

Виконують команди (завдання) у вказаний час.

За замовчуванням отримують команди, які потрібно виконати за стандартного введення або з файлу.

Ключ -f – дозволяє у файлі **ім'я_файлу** задавати команди, які потрібно виконати.

Завдання – це група команд, що виконуються за один виклик **at**.

Час – цей час доби, коли at запускає завдання. За замовчуванням час задається у 24-годинному форматі – **гг:хх**, можливі й інші формати.

Дата – це **день тижня** або **день місяця**, коли потрібно виконати завдання.

Якщо дата не вказана, **at** виконує завдання цього ж дня, якщо вказаний час більше поточного. Якщо час менший за поточний, **at** виконує завдання наступного дня.

День тижня вказується його повною назвою (англійською) або абревіатурою з трьох літер, також можна використовувати слова today (сьогодні) та tomorrow (завтра).

Для зазначення **дня місяця** використовується назва місяця та число, за місяцем та числом можна зазначити рік.

Приклад 4.1.

Створимо файл task із завданням. Завдання містить команди запуску текстового редактора та табличного процесора. Час запуску завдання – 15:00.

delux@ubuntu:~$ cat task

/usr/bin/oowriter # запуск текстового редактора

/usr/bin/oocalc # запуск табличного процесора

delux@ubuntu:~$ at -f task 15:00

warning: commands will be executed using /bin/sh job 17 at Fri Oct 14 15:00:00 2011

Призначимо на виконання о 15.10 нове завдання. Завдання містить команду створення файлу з поточною датою та календарем на поточний місяць, і команду виведення даного файлу на друк. Команди завдання введемо з консолі.

delux@ubuntu:~$ at 15:10

warning: commands will be executed using /bin/sh

at> date > 1.txt; cal >> 1.txt; lpr 1.txt

at> <Ctrl>+<D> <EOT> (треба натиснути <Ctrl>+<D> -кінець введення (файлу))

job 18 at Fri Oct 14 15:10:00 2011

Призначимо виконання завдання у файлі task наступної п'ятниці о 12:00.

delux@ubuntu:~$ at -f task 12:00 Fri

warning: commands will be executed using /bin/sh job 19 at Fri Oct 21 12:00:00 2011

Призначимо виконання завдання у файлі task 20 жовтня о 12:00.

delux@ubuntu:~$ at -f task 12:00 Oct 20

warning: commands will be executed using/bin/sh job 20 at Thu Oct 20 12:00:00 2011

2. Команда **atq.**

Виводить список завдань, поставлених у чергу.

Приклад 4.2.

Виведемо список завдань у черзі.

delux@ubuntu:~$ atq

20Thu Oct 20 12:00:00 2011 a delux

19Fri Oct 21 12:00:00 2011 a delux

3. Команда **atrm** список_завдань**.**

Скасовує завдання, які очікують на виконання, за допомогою команди **at.**

Список_завдань – це список із одного або декількох номерів завдань, розділених пробілом.

Приклад 4.3.

Видалимо всі завдання з черги завдань.

delux@ubuntu:~$ atrm 19 20

delux@ubuntu:~$ atq

delux@ubuntu:~$ (у черзі завдань немає - висновок порожній)

4. Команда **crontab, демон cron.**

Виклик:

1) **crontab [-u ім’я_користувача] ім'я_файлу.**
2) **crontab [-u ім’я_користувача] ключ.**

Дозволяє запускати завдання (команди) із заданою періодичністю, на відміну від команди **at**, яка дозволяє запустити завдання один раз у призначені дату та час.

Сама команда **crontab** безпосередньо не запускає завдання, а призначена для обслуговування так званих **crontab-файлів**, в яких міститься список команд, час і дата, коли їх треба запускати. Самі завдання виконує системна утиліта (демон) **cron**. Демон **cron** автоматично запускається після завантаження системи, і далі щохвилини переглядає crontab-файли у відповідних каталогах і запускає завдання, якщо такі є.

Crontab-файли користувача зберігаються в каталозі **var/spool/cron** або **var/spool/cron/crontabs**. Кожен файл має ім'я користувача, якому він належить. Для системного адміністратора існують спеціальні каталоги, де знаходяться crontab-файли з іменами команд, які необхідно запускати кожну годину, день, тиждень і місяць.

Перший формат команди (crontab [-u ім'я_користувача] ім'я_файлу) використовується для копіювання вмісту файлу **ім'я_файлу** (що містить crontab-команди) в **crontab-файл** користувача, який запустив цю команду або того користувача, чиє ім'я вказано в команді. Якщо у користувача ще немає crontab-файлу, він буде створений, інакше перезаписаний. Тобто команда задає crontab-файл користувача. Якщо **ім'я_файлу** не задано, crontab читає команди зі стандартного введення.

Другий варіант (crontab [-u ім'я_користувача] ключ) використовується для редагування (ключ -e), виведення на екран (-l) і видалення (r) crontab-файлу.

Формат crontab-файлу такий:

хвилини години дні_місяця місяць дні_тижня команда

хвилини – числа від 0 до 59, або *

години – числа від 0 до 23, або *

дні_місяця – числа від 1 до 31, або *

місяць – числа від 1 до 12, або *

дні_тижня – числа від 0 до 7, причому 0 або 7 – неділя, або *

Символ "*" є груповим, тобто означає будь-яке можливе значення - кожну хвилину, кожну годину і т.д.

Наприклад:

0 10 * * * /home/student/bin/script #запуск о 10:00 щоденно

15 * * * 1 /home/student/bin/script2 #в 15 хвилин кожної години в понеділок

Приклад 4.3.

Подивимося список завдань у crontab-файлі поточного користувача (delux).

delux@ubuntu:~$ crontab -l

no crontab for delux(користувач delux ще не створив crontab-файл**)**

Створимо crontab-файл, у якому помістимо команду друкування на принтері календаря на поточний місяць першого числа кожного місяця о 10.00.

delux@ubuntu:~$ echo "00 10 1 * * cal | lpr" > cb.txt

delux@ubuntu:~$ cat cb.txt

00 10 1 * * cal | lpr delux@ubuntu:~$ crontab cb.txt

Подивимося список завдань у crontab-файлі поточного користувача.

delux@ubuntu:~$ crontab -l 00 10 1 * * cal | lpr

Видалимо crontab-файл поточного користувача.

delux@ubuntu:~$ crontab -r

delux@ubuntu:~$ crontab -l

no crontab for delux

II. Отримання інформації про команди, команди **man, whatis, apropos**

1. Команда **man [ключ] [розділ_посібника] тема**

Надає документацію з Linux.

Розділ_посібника – це один з розділів посібника з Linux, в якому буде виконано пошук інформації по **темі**, що цікавить. У загальному випадку команда man виводить інформацію з будь-якої теми, що стосується Linux, наприклад, про системні команди, системні виклики (функції на мові С), прикладні програми і т.д.

Посібник з системи Linux розбитий на десять розділів, і в кожному розділі описуються відповідні інструментальні засоби:

1) Команди користувача
2) Системні виклики
3) Підпрограми
4) Пристрої
5) Формати файлів
6) Ігри
7) Різне
8) Адміністрування системи
9) Ядро
10) Нове

Якщо розділ_посібника не вказано, man відобразить найпершу знайдену інформацію про команду в одному з посібників.

Ключ -a – дозволяє вивести інформацію з усіх розділів за вказаною командою.

Прокручувати довідку з теми можна за допомогою клавіші ПРОБІЛ, для припинення роботи команди необхідно натиснути клавішу <Q>.

Приклад 4.4.

Виведемо довідку за командою time з першого розділу.

delux@ubuntu:~$ man 1 time TIME(1)

NAME

time - run programs and summarize system resource usage

SYNOPSIS

time [-apqvV] [-f FORMAT] [-o FILE]

…

Виведемо довідку за командою time з другого розділу.

delux@ubuntu:~$ man 2 time

TIME(2)

NAME

time - get time in seconds

SYNOPSIS

#include <time.h>

time_t time(time_t *t);

…

2. Команда **whatis** команда

Виводить коротку довідку з посібника man за командою, що цікавить.

Приклад 4.5.

Виведемо коротку довідку за командою time.

delux@ubuntu:~$ whatis time

time (7)- overview of time and timers time (2)- get time in seconds

time (1)- run programs and summarize system resource usage

У круглих дужках вказані номери man-розділів.

3. Команда **apropos** ключове_слово

Шукає **ключове_слово** в короткому рядку опису команд на всіх man-сторінках і відображає ті з них, які містять відповідне слово.

Команда **apropos** використовується, коли невідоме ім'я команди, необхідної для виконання конкретного завдання, однак є інформація про те, які дії вона виконує. Тоді за ключовими словами можна знайти («згадати») назву самої команди.

Приклад 4.6.

Припустимо, ми забули, яка команда виводить інформацію про поточні процеси. Для пошуку цієї команди, як ключове слово, виберемо **processes** (процеси).

delux@ubuntu:~$ apropos processes

aa-unconfined (8)- output a list of processes with tcp or udp ports that do not have AppArmor profiles loaded

cpuset (7)- confine processes to processor and memory node subsets Dpkg::Compression::Process (3) - run compression/decompression processes

faked (1)- daemon that remembers fake ownership/permissions of files manipulated by fakeroot processes.

faked-sysv (1)- daemon that remembers fake ownership/permissions of files manipulated by fakeroot processes.

faked-tcp (1)- daemon that remembers fake ownership/permissions of files manipulated by fakeroot processes.

fuser (1)- identify processes using files or sockets gnome-system-monitor (1) - view and control processes killall (1) - kill processes by name

killall5 (8) - send a signal to all processes.

peekfd (1)- peek at file descriptors of running processes

pgrep (1)- look up or signal processes based on name and other attributes pkill (1)- look up or signal processes based on name and other attributes

ps (1)- report a snapshot of the current processes.

pstree (1)- display a tree of processes pstree.x11 (1)- display a tree of processes renice (1)- alter priority of running processes

unconfined (8)- output a list of processes with tcp or udp ports that do not have AppArmor profiles loaded

Завдання лабораторної роботи 4

Необхідно запустити емулятор терміналу та:

1) В каталозі ~/ossp створити каталог lab3.

2) В каталозі lab3 створити два виконувані сценарії prog1 і prog2, що реалізують нескінченний цикл.

3) Запустити сценарій prog1 з підвищеним на 10 пріоритетом, а сценарій prog2 зі зниженим на 10 пріоритетом у фоновому режимі.

4) Вивести інформацію про запущені процеси.

5) Змінити пріоритети між процесами prog1 та prog2. І вивести інформацію про запущені процеси.

6) Завершити процеси prog1 та prog2 за допомогою команди kill.

7) Запустити сценарій prog1 у звичайному (не фоновому) режимі.

8) Зупинити виконання prog1.

9) Продовжити виконання prog1 у фоновому режимі.

10) Перевести prog1 в активний режим і надалі завершити його роботу.

Лабораторна робота 5

Складання програм та бібліотек під ОС Linux з використанням GCC

Теоретичні відомості

GNU Compiler Collection (GCC) — набір компіляторів для різних мов програмування (C, C++, Objective-C, Java, Fortran), розроблений у рамках проекту GNU (проект з розробки вільного програмного забезпечення).

Як компілятор мови С в Linux зазвичай служить програма gcc (GNU C Compiler) з пакета компіляторів GCC. Існують версії різних реалізацій Unix (та інших систем ОС), які дозволяють генерувати код для процесорів різної архітектури.

Процес створення програми (бібліотеки) складається із двох етапів:

1) Компіляція файлів з вихідним кодом мовою С, С++ в об'єктні файли («напівфабрикати» – проміжна ланка між текстовими файлами з вихідним кодом програми і готовою програмою).

2) Компонування (link – лінківка) об'єктних файлів у виконуваний файл (програму) або бібліотеку.

Часто під компіляцією розуміють розглянуті вище два етапи як ціле.

Програму gcc можна використовувати як компіляції програм в об'єктні модулі (з розширенням ".o") так компонування отриманих модулів в єдину виконувану програму. Компілятор здатний аналізувати імена та розширення файлів, що передаються йому як аргументи, і визначати, які дії необхідно виконати.

Основні типи файлів з якими працює gcc:

1) Файл із розширенням ".c" містить код мовою С.

2) Файл з розширенням ".o" являє собою об'єктний файл (об'єктний модуль).

3) Файл із розширенням ".a" є статичною бібліотекою (у Microsoft

Windows такі файли мають розширення «.lib»).

4) Файл з розширенням ".so" є бібліотекою, що динамічно підключається (у Microsoft Windows такі файли мають розширення «.dll»).

5) Файл з розширенням ".out" є виконуваним (тобто програмою) отриманим в результаті компіляції, якщо не було задано ім'я результуючого файлу.

Ядро та всі системні виклики Linux написані мовою С (без використання об'єктно-орієнтованого підходу) із вставками на мові Асемблера. Тому розглянемо далі компіляцію програм мовою С.

Загальний формат виклику компілятора gcc:

gcc [опції] список_імен_файлів

Основні опції компілятора:

1) -c – виконувати лише компіляцію вихідних файлів без компонування об'єктних файлів**;**

2) **-o** – для компонування одного або кількох об'єктних файлів, отриманих з вихідного коду в єдиний файл, що виконується **(бібліотеку),** якщо об'єктні файли відсутні, вони будуть отримані з вихідних;

3) **-l, -L** – для підключення бібліотек бібліотеки повинні бути перераховані після вихідних або об'єктних файлів, що містять дзвінки до відповідних функцій;

4) -g – помістити в об'єктний або виконуваний файл налагоджувальну інформацію для налагоджувача gdb, опція повинна бути вказана і для компіляції, і для компонування**;**

5) **-pg –** помістити в об'єктний або виконуваний файл інструкції профілювання для генерації інформації, яка використовується утилітою **gprof.** Опція повинна бути вказана і для компіляції, і компонування. Профілювання – це процес вимірювання тривалості виконання окремих

ділянок програми. При вказівці -pg отримана виконувана програма при запуску генерує файл статистики. Програма gprof на основі цього файлу створює розшифрування, що вказує час, витрачений на виконання кожної функції;

6) **-Wall –** виведення повідомлень про всі попередження або помилки, що виникають під час трансляції програми;

7) **-O1 –** встановлює оптимізацію рівня 1, компілятор намагається зменшити розмір коду та час виконання;

8) **-O2 –** встановлює оптимізацію рівня 2, компілятор виконує майже всі оптимізації, що підтримуються, які не включають зменшення часу виконання за рахунок збільшення довжини коду. Компілятор не виконує розкручування циклів або встановлення функцій;

9) **-O3** – Встановлює оптимізацію рівня 3, тобто включає всі можливі оптимізації.

Створимо текстовий файл hello.c (наприклад, у редакторі gedit) з вихідним кодом програми, що виводить рядок «Hello World!».

"hello.c" #include <stdio.h> int main (void)

{

printf ("Hello World!\n"); return 0;

}

Створення виконуваного файлу

Щоб відкомпілювати програму, необхідно в командному рядку викликати gcc, вказавши в якості аргументу ім'я файлу з вихідним кодом:

delux@ubuntu :~/hello$ gcc hello.c

delux@ubuntu :~/hello$ ls

a.out hello.c

Після виклику gcc з робочим каталогом з'явився файл **a.out** що є програмою. Викликаємо програму:

delux@ubuntu :~/hello$ ./a.out Hello World!

Виконувані файли програм зазвичай розташовуються в каталогах, імена яких через символ ":" перераховані в системній змінній PATH. Переглянути значення цієї змінної можна так:

delux@ubuntu :~/hello$ echo $PATH

/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games

Якщо файл, що виконується, знаходиться в одному з таких каталогів, то для його виклику шлях до нього можна не вказувати. Інакше потрібно буде вказати шлях до програми (у нашому прикладі це поточний каталог "./").

Для того щоб явно вказати ім'я файлу необхідно запустити gcc з ключем -o:

delux@ubuntu :~/hello$ gcc -o hello hello.c

delux@ubuntu :~/hello$ ls

hello hello.c

delux@ubuntu :~/hello$ ./hello Hello World!

У випадку, якщо у вихідному коді є помилки gcc видасть повідомлення про них.

Наприклад, закоментуємо у файлі hello.c заголовок функції main() і спробуємо знову відкомпілювати програму:

delux@ubuntu :~/hello$ gcc -o hello hello.c

hello.c:4:error: expected identifier or '(' before '{' token tf'

Як видно, gcc повідомляє про помилку на четвертому рядку.

Компонування та створення об'єктних файлів

Часто вихідний код програми для зручності поділяється на кілька частин, які компілюються окремо в об'єктні файли, а потім з'єднуються в один файл або бібліотеку. Такий підхід, зокрема, використовується коли вихідний код програми має достатній обсяг або окремі частини програми написані різними мовами (наприклад, C і Асемблері). Об'єктні файли містять об'єктний код (машинний код) і мають розширення ".o".

Для об'єднання об'єктних файлів використовується компонувальник (лінковник). Процес об'єднання називають компонуванням або лінковкою. У Linux для компонування об'єктних файлів використовується програма GNU ld. Компілятор gcc автоматично викликає компонувальник ld з потрібними параметрами для створення файлу або бібліотеки, що виконується. Можна також самостійно викликати ld для компонування об'єктних файлів.

Щоб відмовитися від автоматичного компонування та створити об'єктний файл необхідно викликати gcc з ключем -c, наприклад:

delux@ubuntu :~/hello$ gcc -c hello.c

delux@ubuntu :~/hello$ ls

hello.c hello.o

У разі виклику gcc з ключем -c, ім'я вихідного файлу виходить заміною розширення .c на .o.

Далі перетворюємо об'єктний файл на виконуваний:

delux@ubuntu :~/hello$ gcc -o hello hello.c

delux@ubuntu :~/hello$ ls

hello hello.c hello.o

Як видно, тепер у робочому каталозі знаходиться три файли: 1) з вихідним кодом, 2) об'єктний файл і 3) файл, що виконується. Якщо ж не відмовлятися від автоматичного компонування, після створення виконуваного файлу, «проміжний» об'єктний файл буде автоматично видалено.

Компіляція та компонування багатофайлових проектів

Процес складання багатофайлового проекту реалізується за таким алгоритмом:

1) Створюються вихідні файли (файли з вихідним кодом) з **розширенням ".c".**

2) Створюються заголовні файли **з розширенням ".h".** Заголовні файли мають таке призначення: вони встановлюють угоди щодо використання спільних ідентифікаторів (імен змінних, функцій тощо) у різних файлах програми. Наприклад, якщо функція *f()* реалізована у файлі **func.c,** а викликається у **файлі ac,** то обидва файли необхідно включити директивою **#include** заголовковий файл (наприклад, #include "funcs.h"), що містить оголошення (прототип) функції *f ().*

3) Кожен вихідний файл компілюється окремо з **ключем -c**. Внаслідок цього з'являється набір об'єктних файлів.

4) Отримані файли поєднуються компонувальником в одну програму (бібліотеку). Слід зазначити, що файли заголовків ніколи окремо не компілюються.

Так як на стадії препроцесування (перед компіляцією) всі директиви **#include** замінюються на вміст зазначених у них файлів.

Приклад 5.1. Створимо програму, яка приймає як аргумент рядок, переводить у ній усі символи у верхній регістр і виводить результуючий рядок на екран. Вихідний код програми складатиметься з трьох файлів:

1) main.c - основний файл програми, що містить функцію **main();**

2) printup.c - містить прототип функції ***printup(),*** що переводить символи у верхній регістр з наступним виведенням їх на екран**;**

3) printup.h - містить реалізацію функції ***printup*****().**

"main.c"

```
#include <string.h>
#include <stdio.h>
#include "printup.h"

int main(int argc, char ** argv)
{
if (argc < 2) {
fprintf(stderr,"Wrong arguments\n"); return 1;
}
printup (argv [1]); return 0;
}
```

"printup.h"

```
void printup (const char * str)
```

"printup.c"

```
#include <stdio.h>
#include <string.h>
#include "printup.h"

void printup (const char * str)
{
int i;

for (i = 0; i < strlen(str); i++)
printf("%c", toupper(str[i])); printf("\n");
}
```

Зберемо проект. Спочатку відкомпілюємо вихідні файли:

delux@ubuntu :~/printup$ gcc -c printup.c

delux@ubuntu :~/printup$ gcc -c main.c

В результаті з'являться файли об'єкта printup.o main.o, які скомпонуємо у виконуваний файл printup:

delux@ubuntu :~/printup$ gcc -o printup printup.o main.o

Запустимо програму без аргументів:

delux@ubuntu :~/printup$ ./printup

Wrong arguments

Тепер з аргументами:

delux@ubuntu :~/printup$ ./printup

HeLlOw

HELLOW

Утиліта make

Утиліта GNU make, надалі просто make, використовується для автоматизації процесом збирання багатофайлових програмних проектів. Утиліта make працює з особливими файлами – make-файлами, в яких міститься вся інформація про складання проекту.

Автоматичне складання проекту здійснюється за наступним алгоритмом:

1) Підготовляються вихідні та заголовні файли.

2) Підготовляються make-файли, що містять відомості про проект. В принципі навіть для великих проектів достатньо одного make-файлу**. Make**-файл може називатися як завгодно, проте зазвичай вибирають одне із трьох стандартних імен **(Makefile, makefile або GNUmakefile),** які розпізнаються автоскладачем **make** автоматично. Якщо **make**-файл знаходиться в робочому каталозі, при запуску утиліти make вказувати **make**-файл не потрібно. Якщо в

проекті використовується нестандартне ім'я **make**-файлу, його потрібно вказати після опції -f при виклику **make.**

3) Викликається **утиліта make, яка збирає проект на підставі даних, одержаних з make-файлу.**

Розробники **GNU make** рекомендують використовувати для make-файлів ім'я Makefile. Для правильного складання make-файлу необхідно визначити основну мету складання проекту. Потім слід виявити проміжні цілі, якщо такі є. Наприклад, розглянемо компіляцію програми printup. Основною метою є формування файлу **printup**. Щоб його отримати, потрібні файли **printup.o** та **main.o**. Це проміжні цілі. У **make**-файлах за кожну мету відповідає своє цільове зв'язування.

Після визначення цілей необхідно виявити залежності. У нашому прикладі основна мета (файл **printup**) може бути досягнута лише за наявності файлів **printup.o** та **main.o**. А файл pr**intup.**o може бути отриманий лише за наявності вихідного файлу printup.c та заголовного файлу **printup.h**. Аналогічно файл **main.o** може бути отриманий тільки за наявності файлів **main.c** і **printup.h.**

У **make**-файлах, як правило, присутні такі конструкції:

1) Коментарі. Допустимі однорядкові коментарі, які починаються символом # (решітка) і закінчуються символом нового рядка**.**

2) Макроси. Макроси в **make-файлах** служать для підстановки. Макроси багато чому схожі з макросами препроцесора мови С**.**

3) Цільові зв'язки. Ці елементи є найважливішими в **make-**файлів. За допомогою цільових зв'язок задаються залежності між різними частинами програми, а також визначаються дії, які виконуватимуться при складанні програми. У будь-якому make-файлі має бути хоча б одна цільова зв'язка.

У **make**-файлі обов'язкові лише цільові зв'язки. Кожна цільова зв'язка складається з наступних компонентів:

1) Ім'я цілі. Якщо метою є файл, то зазначається його ім'я. Після імені мети слідує двокрапка**.**

2) Список залежностей. Тут просто перераховуються через пропуск імена файлів або імена проміжних цілей. Якщо мета ні від чого не залежить, цей список буде порожнім**.**

3) Інструкції. Це команди, які мають виконуватися задля досягнення мети. Наприклад, у цільовій зв'язці **printup.o** інструкцією буде команда компіляції файлу **printup.c**. Кожна інструкція пишеться на новому рядку і починається із символу табуляції.

Загальний формат цільової зв'язки наступний:

ціль: файли від яких вона залежить

<tab>команда

Загальний формат виклику утиліти:

make [опції] [ім'я make-файлу] [ім'я мети]

Приклад 5.2.

Створимо make-файл для автоскладання проекту printup.

"Makefile"

```
# Makefile for printup
printup: printup.o main.o
gcc -o printup printup.o main.o

printup.o: printup.c printup.h
gcc -c printup.c

main.o: main.c
gcc -c main.c

clean:
rm -f *.o
rm -f printup
```

Розглянемо цей **Makefile.**

Перший рядок **Makefile** – це коментар. Потім слідують цільові зв'язки. Перша зв'язка відповідає за створення файлу printup і формується таким чином:

1) Спочатку записується ім'я мети **(printup).**

2) Після двокрапки перераховуються залежності **(printup.o і main.o).**

3) На наступному рядку після знаку табуляції пишеться правило (команда) для отримання файлу **printup.**

Аналогічним чином оформляються такі дві зв'язки.

Остання зв'язка (c**lean**) призначена для видалення файлів, створених під час попереднього збирання. У разі список залежностей є порожнім. Це означає, що ця зв'язка не вимагає наявності будь-яких файлів і не передбачає виконання проміжних цілей. Далі йдуть команди видалення виконуваного та об'єктних файлів. Для виконання такої мети її необхідно явно вказувати після утиліти **make**.

Виконаємо автоскладання проекту.

delux@ubuntu :~/printup$ make

gcc -c printup.c

gcc -c main.c

gcc -o printup printup.o main.o

Виконаємо видалення виконуваного та об'єктних файлів. Для цього після утиліти make вкажемо ім'я цілі.

delux@ubuntu :~/printup$ make clean

rm -f *.o

rm -f printup

Використання макросів у make-файлах.

Макроси (перевизначені константи) використовуються для параметризації процесу збирання. Для оголошення макросів використовується наступний шаблон:

NAME=VALUE

Тут **NAME** – це ім'я макросу, **VALUE** – його значення.

Ім'я макросу не повинне починатися з цифри. Значення може містити будь-які символи, включаючи пробіли. Ознака закінчення константи - символ нового рядка. Тобто будь-які символи, що стоять між знаком "рівно" і символом перенесення рядка, будуть значення константи.

У make-файлі макрос, значення якого складається з одного символу, використовується так:

$NAME

а значення якого має більше одного символу, використовується наступним чином:

$(NAME)

Приклад 5.3. Замінимо у попередньому make-файлі ім'я компілятора макросом CC. У цьому випадку для використання іншого компілятора достатньо одного разу змінити значення макросу CC. Також додамо макроси для імені кінцевої програми та операції видалення файлів.

"Makefile"

```
# Makefile for printup
CC=gcc
CLEAN=rm -f
PROGRAM_NAME=printup

$(PROGRAM_NAME): printup.o main.o
$(CC) -o $(PROGRAM_NAME) printup.o main.o
```

printup.o: printup.c

$(CC) -c printup.c

main.o: main.c

$(CC) -c main.c

clean:

$(CLEAN) *.o

$(CLEAN) $(PROGRAM_NAME)

Виконаємо автоскладання проекту.

delux@ubuntu :~/printup$ make

gcc-c -o printup.o printup.c

gcc -c main.c

gcc -o printup printup.o main.o

Використання спеціалізованих макросів утиліти make.

Утиліта make підтримує ряд спеціалізованих констант, серед яких найчастіше використовуються наступні дві:

1) **$@ -** містить ім'я поточної мети**;**
2) **$^ -** містить список залежностей у поточному зв'язуванні**.**

Приклад 5.4. Перепишемо попередній make-файл з використанням спеціалізованих констант.

"Makefile"

 # Makefile for printup

CC=gcc

CLEAN=rm -f

PROGRAM_NAME=printup

$(PROGRAM_NAME): printup.o main.o

$(CC) -o $@ $^

printup.o: printup.c

$(CC) -c $^

main.o: main.c

$(CC) -c $^

clean:

$(CLEAN) *.o

$(CLEAN) $(PROGRAM_NAME)

Виконаємо автоскладання проекту.

delux@ubuntu :~/printup$ make

make: `printup' не потребує оновлення.

Ми отримали повідомлення, що свідчить про те, що вихідні і результуючі файли проекту не були змінені, тому повторне складання не потрібно. Видалимо виконуваний та об'єктний файли і спробуємо знову.

delux@ubuntu :~/printup$ make clean

rm -f *.o

rm -f printup

delux@ubuntu :~/printup$ make

gcc-c -o printup.o printup.c

gcc -c main.c

gcc -o printup printup.o main.o

delux@ubuntu :~/printup$ in.o

Приклад 5.5. Додамо до попереднього make-файлу опцію -Wall для компілятора gcc і макроси для об'єктних та вихідних файлів, щоб зробити make -файл більш універсальним.

"Makefile"

```
# Makefile for printup
CC=gcc
CCFLAGS=-Wall
CLEAN=rm -f
PROGRAM_NAME=printup
OBJECT_FILES=*.o
SOURCE_FILES=printup.c main.c

$(PROGRAM_NAME): $(OBJECT_FILES)
$(CC) $(CCFLAGS) -o $@ $^

$(OBJECT_FILES): $(SOURCE_FILES)
$(CC) $(CCFLAGS) -c $^

clean:
$(CLEAN) *.o $(PROGRAM_NAME)
```

Виконаємо автоскладання проекту.

```
delux@ubuntu :~/printup$ make
gcc -Wall -c printup.c main.c
printup.c: In function 'printup':
printup.c:10: warning: implicit declaration of function 'toupper'
gcc -Wall -o printup *.o
```

Передача та обробка параметрів командного рядка у програмі

Параметри командного рядка передаються у програму написану мовою С через аргументи функції main, яка може мати синтаксис:

int main (int argc, char *argv[]) { ... }, де

argc- задає число параметрів командного рядка (вважаючи саме ім'я програми, тобто. якщо програма запущена без параметрів, argc дорівнюватиме 1, якщо з одним параметром, то 2 і т. д.);

argv- це символьний масив, у якому містяться аргументи командного рядка (argv[0] - (Зазвичай) ім'я самої програми, argv [l] - перший аргумент і т.д.).

Наприклад, під час запуску

./lab5 file1 file2

argc дорівнюватиме 3, а **argv** - масиву { "./lab1", "file1 ", "file2"}.

Обробка параметрів командного рядка в Linux

У випадку, коли аргументи командного рядка є, наприклад, іменами файлів, їхня обробка не є складною. Значно трудомісткішою вона стає тоді, коли програма запускається з ключами (тобто використовується дефіс) або іменованими аргументами, (наприклад, як компілятор **gcc -g** -про ім'я-виконуваного файлу **(file.c),** оскільки потрібно окремо відстежити, чи задані ті чи інші опції (можливо з параметрами) і т.д.

Для полегшення обробки у UNIX системах існує набір спеціальних функцій. Найчастіше використовується функція **getopt(),** яка виробляє синтаксичний розбір параметрів командного рядка. Її синтаксис такий:

int getopt (int argc, char *argv[], char *options);

Перші два аргументи – це **argc** і **argv**, отримані в **main().** Третій параметр задає список допустимих (односимвольних, тобто символ) опцій

командного рядка. Якщо опція вимагає після себе параметр (як -o для завдання файлу, що виконується при компонуванні за допомогою gcc), то після відповідної літери ставиться ':'.

Для використання цієї функції необхідно підключити заголовок **<getopt.h>.**

Наприклад, **getopt (argc, argv, "lf:h");** показує, що опція f вимагає після себе завдання параметра (вільного аргументу) і програма може викликатися так:

1) **lab5 -l**
2) **lab5 -h**
3) **1ab5 -lh**
4) **1ab5 -f text**
5) **і т.д.**

Функція **getopt()** за один раз повертає лише одну (чергову) опцію (у вигляді символу, наприклад, 'l' при виклику lab5 -l), тому її потрібно викликати у циклі. Цикл продовжується доти, доки функція не поверне -1 (EOF), що означає те, що всі опції оброблені.

Якщо якась опція потребує вільного аргументу (як -f), то його значення повертається до визначеної системної зовнішньої рядкової змінної **optarg**.

Зовнішня системна цілочисленна змінна **optind** зберігає індекс першого вільного аргументу в масиві argv. При цьому слід враховувати, що **getop**t() переміщає всі вільні аргументи в кінець argv. Таким чином, вони знаходяться між **optind** і **argc-**1. Якщо **optind >= argc,** вільні аргументи відсутні.

Приклад 5.6. Створимо програму, яка виводить у файл передані їй вільні аргументи. Опція -o буде використовуватися для виведення вільних аргументів у файл, ім'я якого вказується в першому вільному аргументі, тобто наступного безпосередньо за опцією. Опція -h виводитиме коротку довідку про роботу з програмою.

```
#include <stdio.h>
#include <getopt.h>

int main (int argc, char** argv)
{
FILE *outfile=NULL; char * filename = NULL; int i, opt;
char help_str[] =
"Usage: mygetopt[OPTIONS] ARGUMENTS ...\n"
"-h- Print help and exit \n"
"-o <outfile>-  Write output to file\n"; while ((opt = getopt (argc, argv,
"ho:")) != -1)
{
switch (opt)
{
case 'h':
printf("%s", help_str); return 0;

case 'o':
filename = optarg; break;

default:
printf("Unknown error\n"); return 1;
}
}

if (optind >= argc)
{
printf("No argument(s) found\n%s", help_str); return 1;
}
```

```
if (filename! = NULL)
{
outfile = fopen (filename, "w");
if (outfile == NULL)
{
printf("Cannot open output file (%s)\n", filename); return 1;
}
}

for (i = optind; i < argc; i++)
fprintf(outfile, "%s\n", argv[i]);
fclose (outfile); return 0;
}
```

Відкомпілюємо та запустимо програму.

```
delux@ubuntu :~/mygetopt$ gcc -o mygetopt mygetopt.c
delux@ubuntu :~/mygetopt$ ./mygetopt
No argument(s) found
Usage: mygetopt[OPTIONS] ARGUMENTS ...
-h-  Print help and exit
-o<outfile>- Write output to file
delux@ubuntu :~/mygetopt$ ./mygetopt -h
Usage: mygetopt [OPTIONS] ARGUMENTS ...
-h-  Print help and exit
-o<outfile>- Write output to file
delux@ubuntu :~/mygetopt$ ./mygetopt -o 1 2 3 4 5
delux@ubuntu :~/mygetopt$ ls
1 mygetopt mygetopt.c
delux@ubuntu:~/mygetopt$ cat 1 2 3 4 5
```

Завдання лабораторної роботи 5

Необхідно розробити консольну програму, призначену для сортування послідовності цілих чисел за допомогою бульбашкового сортування.

Напрямок сортування (зростання/зменшення) повинен задаватися у вигляді опції під час виклику програми. Вхідна послідовність чисел задається як вільні аргументи через пропуск при виклику програми. Відсортована послідовність має виводитись на екран.

Проект програми має складатися із трьох файлів:

1) вихідний файл із функцією main;

2) заголовний файл із прототипом функції сортування;

3) вихідний файл із реалізацією функції сортування.

Для складання проекту необхідно створити make-файл та скористатися утилітою make.

Лабораторна робота 6
Бібліотеки, що статично та динамічно підключаються

Теоретичні відомості

Бібліотека є програмним модулем, що складається з набору з'єднаних певним чином об'єктних файлів. Бібліотека зберігається на диску як файл зі спеціальним розширенням і може містити як функції, так і дані. Бібліотеки призначені головним чином для розробки бібліотек функцій, класів, які можуть використовуватись різними додатками. Це дозволяє зменшити витрати на розробку програмного забезпечення, той самий програмний код може використовуватися різними розробниками.

Бібліотеки поділяються на дві категорії:

1) **статичні (архіви);**
2) **що динамічно підключаються, або просто "динамічні" (розділені).**

Статичні бібліотеки (Static libraries) створюються програмою ar. Файли статичних бібліотек мають розширення ".a" (архів – архів). Статичні бібліотеки підключаються до програми (нової бібліотеці, що створюється) на етапі її компонування. При підключенні код статичних бібліотек розміщується безпосередньо у програмі, тому розмір програми збільшується відповідно на розмір бібліотеки.

Бібліотеки, що динамічно підключаються (розділяються) створюються компонувальником ld при виклику компілятора gcc з опцією -shared і мають розширення ".so" (shared objects – об'єкти, що розділяються). При підключенні динамічних бібліотек їх код безпосередньо до програми не додається, а вставляються "посилання" на динамічні бібліотеки. Під час запуску програми та/або в процесі її роботи за цими посиланнями підвантажуються відповідні динамічні бібліотеки.

У порівнянні зі статичними, динамічні бібліотеки дозволяють зменшити обсяг використовуваної фізичної (оперативної) пам'яті при одночасної роботі кількох додатків, які використовують ту саму бібліотеку. Це досягається завдяки механізму проектування динамічної бібліотеки у віртуальну пам'ять процесів, т.к. у цьому випадку всі додатки поділяють один і той же екземпляр коду бібліотеки, що виконується, завантажений у фізичну пам'ять.

При цьому важливо зазначити таке. Виконувані файли програм та файли динамічних бібліотек розбиті на кілька розділів, кожен з яких зберігає інформацію певного типу. Один із цих розділів містить лише виконуваний код програми або бібліотеки. При завантаженні динамічної бібліотеки цей розділ зберігається в оперативній пам'яті в одному примірнику та відображатиметься в адресному просторі всіх процесів. Той розділ бібліотеки, який містить дані (змінні), буде зберігатися для кожного процесу в окремому екземплярі.

Таким чином, всі процеси спільно поділяють виконуваний код (функції) динамічної бібліотеки, але кожен процес має свій набір змінних з цієї бібліотеки.

Підключення бібліотек

Підключення бібліотеки до програми реалізується у два етапи:

1) На етапі розробки у вихідні файли програми підключається відповідний заголовний файл, що містить оголошення (прототипи) функцій та змінних бібліотеки**.**

2) На етапі компонування програми при виклику компілятора **gcc** з використанням опції -o вказуються ім'я (імена) бібліотеки. Безпосередньо перед ім'ям бібліотеки (можна без пробілу, що розділяє) розташовується опція **-l (library).**

Імена файлів бібліотек зазвичай починаються з префікса **lib**. Ім'я бібліотеки виходить з імені файлу відкиданням префікса **lib** та розширення.

Наприклад, математична бібліотека мови має ім'я **m**. Отже файл зі статичною реалізацією цієї бібліотеки має ім'я **libm.a**, а файл з динамічною версією бібліотеки **- libm.so**.

За наявності динамічної та статичної версій бібліотеки компонувальник за замовчуванням підключає динамічну версію. Для підключення статичної бібліотеки слід використовувати ключове слово **-static**.

Приклад 6.1. Створимо програму, яка обчислює рівень числа. Саме число та його ступінь задаються аргументами командного рядка, з якими викликається програма.

"main.c"

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

int main (int argc, char** argv)
{
if (argc < 3) {
printf ("Too few arguments\n"); return 1;
}

printf ("%f\n", pow (atof (argv [1]), atof (argv [2]))); return 0;
}
```

Для використання функції **fprintf()** підключений заголовний файл stdio.h, для використання функції **atof(),** що перетворює рядок в речове число - заголовний файл **stdlib.h**, для використання функції *pow()*, що обчислює ступінь числа - файл заголовка **math.h.**

Виконаємо складання (компіляцію та компонування) програми. Спробуємо зібрати програму без підключення бібліотек.

delux@ubuntu :~/lib/power$ gcc -o power main.c

/tmp/cc3rtevS.o: In function `main':

main.c:(.text+0x71): undefined reference to `pow' collect2: ld returned 1 exit status

В результаті отримали повідомлення про помилку компіляції:

`pow' - невідоме посилання на `pow'. Тобто для використання функції pow() необхідно під час складання підключити математичну бібліотеку m.

Спочатку підключимо динамічну версію математичної бібліотеки та перевіримо працездатність програми.

delux@ubuntu :~/lib/power$ gcc -o power main.c -lm

delux@ubuntu :~/lib/power$ ./power 2 8

256.000000

Як видно, збірка програми пройшла успішно. При цьому, при складанні підключення стандартної бібліотеки мови С (ім'я бібліотеки — c, назва файлу бібліотеки libc.so або glib.so тощо) для функцій fprintf() і atof() не знадобилося, тому що дана бібліотека автоматично (неявно) підключається до проекту під час складання.

Перевіримо розмір файлу програми, що виконується.

delux@ubuntu :~/lib/power$ ls -l power

-rwxr-xr-x 1 delux delux 8581 2011-10-28 10:49 power (8581 байт)

Тепер підключимо статичну версію математичної бібліотеки, перевіримо працездатність програми та дізнаємося її розмір.

delux@ubuntu :~/lib/power$ gcc -static -o power main.c -lm

delux@ubuntu :~/lib/power$ ./power 2 8

256.000000

delux@ubuntu :~/lib/power$ ls -l power

-rwxr-xr-x 1 delux delux 838996 2011-10-28 11:00 power (838 996 байт)

Як видно, розмір файлу, отриманого при складанні зі статичною бібліотекою в 100 разів більше ніж при складанні з динамічною бібліотекою. Це пов'язано з тим, що статична бібліотека повністю впроваджується у виконуваний файл, але в динамічну вставляється лише посилання.

При підключенні бібліотеки, за умовчанням передбачається, що вона знаходиться в одному зі спеціальних каталогів: **/lib, /usr/lib, usr/local/lib** та ін. Якщо потрібно підключити бібліотеку, що знаходиться в іншому каталозі (наприклад, каталозі проекту), то при Компонування слід використовувати опцію -L, після якої (можна без пробілу, що розділяє) вказується потрібний каталог.

Приклад 6.2.

Для підключення бібліотеки з ім'ям mylib, що зберігається у файлі libmylib.so в каталозі /usr/project1, необхідно виконати таку команду:

gcc -o myprog myprog.o -L/usr/project1 -lmylib

Приклад6.3.

Для підключення бібліотеки з ім'ям mylib, що зберігається у файлі libmylib.so, у робочому каталозі (./) необхідно виконати таку команду:

gcc-o myprog myprog.o-L. -lmylib

Створення статичної бібліотеки

Статична бібліотека є архівом, створюваним архіватором ar. Даний архіватор підтримує безліч опцій, основні з яких такі (використовуються без дефісів):

1) r - створює архів**;**

2) v — включає режим виведення докладних повідомлень про створення бібліотеки**.**

Синтаксис виклику архіватора ar наступний:

ar [опції] ім'я_бібліотеки [об'єктні файли]

Приклад 6.4. Перепишемо код програми, що обчислює ступінь числа таким чином, щоб перевірка кількості аргументів командного рядка виконувалася в окремій функції check(). Цю функцію розмістимо у статичній бібліотеці mycheck, яку підключимо до програми.

"check.h"

```
#ifndef CHECK_H
#define CHECK_H

#include <stdio.h>
int check(int argc); #endif
```

"check.c"

```
#include "check.h" int check(int argc)
{
if (argc < 3)
{
printf ("Too few arguments\n"); return 1;
}
else return 0;
}
```

Створимо статичну бібліотеку mycheck:

delux@ubuntu :~/lib/power$ gcc -c check.c (Створюємо об'єктний файл check.o)

delux@ubuntu :~/lib/power$ ar rv libmycheck.a check.o (Створюємо бібліотеку)

ar: creating libmycheck.a - check.o

"main.c"

```
#include <stdlib.h>
#include <math.h>
#include "check.h"

int main (int argc, char** argv)
{
int c = check (argc); if (c! = 0)
return c;
else { }
}

printf ("%f\n", pow (atof (argv [1]), atof (argv [2]))); return 0;
```

Зберемо та запустимо програму.

```
delux@ubuntu :~/lib/power$ gcc -c main.c
delux@ubuntu :~/lib/power$ gcc -o power main.o -L. -lmycheck -lm
delux@ubuntu :~/lib/power$ ./power 5 3
125.000000
```

Створення бібліотеки, що динамічно підключається (поділяється)

Динамічні бібліотеки створюються за допомогою виклику компілятора gcc наступного шаблону:

gcc -shared -o ім'я_бібліотеки об'єктний_файл1 [об'єктний_файл2 …]

наприклад,

gcc -shared -o mylib mylib1.o mylib2.o mylib3.o

Компілятор gcc, у свою чергу, викликає компонувальник ld і передає йому параметри для створення бібліотеки. Тобто насправді динамічну бібліотеку створює компонувальник ld.

При створенні та використанні динамічних бібліотек слід враховувати наступні два важливі моменти:

1) У процесі компонування бібліотеки, що спільно використовується, повинні брати участь тільки об'єктні файли, що містять позиційно-незалежний код **- Position Independent Code (PIC).** Код такого типу може підвантажуватись до програми в момент її запуску та в процесі її роботи. Щоб отримати файл з об'єктним позиційно-незалежним кодом, необхідно вихідний файл компілювати з опцією **fPIC.**

2) Опція -L вказана при компонуванні дозволяє доповнити список каталогів, в яких буде здійснюватися пошук бібліотек**.**

У момент запуску програми для пошуку динамічних бібліотек переглядаються каталоги, перелічені у файлі /etc/ld.so.conf, каталоги /usr/lib та /usr/local/lib, а також каталоги перераховані (через символ двокрапки) у змінному оточенні **LD_LIBRARY_PATH.** Якщо бібліотека в цих каталогах не буде знайдена, програма не запуститься або програма працювати не коректно.

Використання змінної **LD_LIBRARY_PATH** не рекомендується, тому що інші працюючі процеси можуть змінювати її значення «під себе». Подивитися значення змінної в командному рядку **LD_LIBRARY_PATH** можна так:

echo $LD_LIBRARY_PATH

Тому в процесі компонування програми окремо можуть вказуватися каталоги де розміщуватимуться динамічні бібліотеки. Також під час інсталяції програми можна додати відповідний запис з каталогом розташування бібліотеки у файл /etc/ld.so.conf. Слід зазначити, що занадто

довгий список каталогів у цьому файлі може негативно вплинути на швидкість завантаження операційної системи. Це пов'язано з тим, що багато ОС Linux відразу при завантаженні читають файл /etc/ld.so.conf і створюють кеш динамічних бібліотек (щоб надалі не витрачати час на пошук та завантаження бібліотек).

Для того щоб компонувальнику ld передати інформацію про місцезнаходження бібліотек, що використовуються програмою, необхідно викликати компілятор gcc з опціями -Wl («л» а не «1») і -rpath наступним чином:

gcc -o prog prog1.o [prog2.o…] [-L...] -llib -Wl,-rpath,path

де,

prog - ім'я програми;

prog1.o, prog2.o ... - Імена об'єктних файлів;

-L – опція вказує де на даний момент при складанні знаходиться бібліотека (на комп'ютері розробника);

-l – опція підключення бібліотеки; lib - ім'я бібліотеки;

-Wl – повідомляє gcc про необхідність передати компонувальнику ld деяку опцію (у даному разі це -rpath);

-rpath – задає каталог (вже не на комп'ютері розробника а користувача), де буде розміщуватися динамічна бібліотека.

path – каталог де розташовуватиметься динамічна бібліотека. Причому між -Wl, -rpath і path не повинні стояти символи пропуску.

Наприклад,

gcc -o prog main.o -L. -lmylib -Wl,-rpath,/home/delux/prog

тут,

prog - ім'я програми;

main.o - ім'я об'єктного файлу;

. – на даний момент бібліотека знаходиться у робочому каталозі;

mylib – ім'я бібліотеки;

/home/delux/prog – каталог де розташовуватиметься динамічна бібліотека.

Приклад 6.5.

Розв'яжемо приклад 5.2, тільки функцію **check()** розмістимо в динамічній бібліотеці **mycheck**, яку підключимо до програми. Як каталог розміщення бібліотеки на комп'ютері користувача вкажемо /home/delux/prog. Для складання бібліотеки та програми створимо **Makefile** та скористаємося утилітою **make**.

"Makefile"

```
prog: main.o libmycheck.so
gcc-o prog main.o-L. -lmycheck -Wl,-rpath,/home/delux/prog -lm

main.o: main.c
gcc -c $^

libmycheck.so: mycheck.o
gcc -shared -o libmycheck.so $^

mycheck.o: mycheck.c
gcc -fPIC -c $^

clean:
rm -f prog libmycheck.so *.o
```

Зберемо програму та бібліотеку.

```
delux@ubuntu :~/lib/prog$ make gcc -c main.c
gcc -fPIC -c check.c
gcc -shared -o libmycheck.so check.o
```

gcc-o prog main.o-L. -lmycheck -Wl,-rpath,/home/delux/prog -lm

Запустимопрограму.

delux@ubuntu :~/lib/prog$ ./prog 3 5

./prog: error while loading shared libraries: libmycheck.so: Open shared object file: No such file or directory

Отримали повідомлення про помилку, що свідчить про неможливість завантажити динамічну бібліотеку **libmycheck.so**. Виправимо помилку, скопіювавши бібліотеку з каталогу проекту до каталогу вказаного при складанні - **/home/delux/prog.**

Запустимо програму знову.

delux@ubuntu :~/lib/prog$ ./prog 3 5

243.000000

Однак найкращим варіантом при складанні програми, яка використовує «власну» динамічну бібліотеку, є не вказівка каталогу де повинна розміщуватися дана бібліотека, а розміщення даної бібліотеки в каталозі /usr/lib або /usr/local/lib при інсталяції програми.

Приклад 6.6.

Розв'яжемо наступне завдання. Необхідно створити динамічну бібліотеку, яка містить одну функцію та одну цілочисленну змінну з початковим значенням 0. При викликі функції вона збільшуватиме значення бібліотечної змінної на одиницю. Також необхідно зібрати програму, в якій викликатиметься ця функція та виводити значення бібліотечної змінної на екран. Далі запустити кілька копій програми та подивитися, як змінюватиметься значення бібліотечної змінної.

Для складання бібліотеки та програми створимо **Makefile** та скористаємося утилітою make.

"Makefile"

```
prog: main.o libinc.so
gcc-o prog main.o-L. -linc -Wl,-rpath,.

main.o: main.c
gcc -c $^

libinc.so: inc.o
gcc -shared -o libinc.so $^

inc.o: inc.c
gcc -fPIC -c $^

clean:
rm -f prog libinc.so *.o "main.c"

#include <stdio.h>
#include <stdlib.h>
#include "inc.h"

void menu()
{
printf("Press:\n"); printf("<1><Enter> - inc()\t");
printf("<2><Enter> - exit program\n");
}

int main(int argc, char ** argv)
{
extern int value; // Оголошуємо посилання на зовнішню (бібліотечну) змінну із файлу inc.c
```

```
char ch;

menu(); while(1)
{
ch = (char) getchar();
getchar(); // зчитуємо код "непотрібного" символу <Enter> switch (ch)
{
case '1':
inc();
printf("value = %d\n", value); break;

case '2':
exit(0);
}
menu();
}
return 0;
}

"inc.h"

#ifndef INC_H #define INC_H

void inc(); #endif
"inc.c" #include "inc.h" int value = 0;
void inc()
{
value++;
}
```

Зберемо програму та бібліотеку.

delux@ubuntu :~/lib/inc$ make gcc -c main.c

gcc -fPIC -c inc.c

gcc -shared -o libinc.so inc.o

gcc-o prog main.o-L. -linc -Wl,-rpath,.

Запустимо програму.

delux@ubuntu :~/lib/inc$./prog

Press:

<1><Enter> -inc()<2><Enter> - exit program 1

value = 1 Press:

<1><Enter> -inc()<2><Enter> - exit program 1

value = 2 Press:

<1><Enter> -inc()<2><Enter> - exit program

Функція *inc()* була викликана 2 рази, тому значення бібліотечної змінної value = 2. Призупинимо (<Ctrl>+<Z>) виконання програми prog, подивимося номер відповідного їй процесу.

^Z

[1]+Зупинено ./prog

delux@ubuntu :~/lib/inc$ ps PIDTTYTIME CMD

1845pts/000:00:00 bash

4418pts/000:00:00 prog(4418)

4420pts/000:00:00 ps

Далі запустимо ще одну копію програми prog.

delux@ubuntu :~/lib/inc$./prog

Press:

<1><Enter> -inc()<2><Enter> - exit program 1

value = 1 Press:

<1><Enter> -inc()<2><Enter> - exit program

Як видно значення бібліотечної змінної після її збільшення на одиницю дорівнювало 1 а не 3 (попереднє значення дорівнює 2 було встановлено іншою копією програми prog). Це пов'язано з тим, що хоча обидві копії програми prog використовують одну й ту ж динамічну бібліотеку, копія бібліотечної змінної value створюється для кожної програми окремо. Коли бібліотечна функція inc() зберігається в оперативній пам'яті в одному примірнику для всіх програм, що звертаються до динамічної бібліотеки.

Припинимо виконання другої копії програми prog, подивимося номер відповідного процесу.

^Z

[1]+Зупинено ./prog

delux@ubuntu :~/lib/inc$ ps PIDTTYTIME CMD

1845pts/000:00:00 bash

4418pts/000:00:00 prog

4426pts/000:00:00 prog(4426)

4521pts/000:00:00 ps

Продовжимо виконання першої копії програми prog у фоновому режимі та одразу ж переведемо її в активний режим.

delux@ubuntu :~/lib/inc$ bg 1; fg %1

[1]+./prog &

./prog1

Тепер значення бібліотечної змінної після її збільшення на одиницю дорівнювало 3.

Завершимо обидва процеси.

delux@ubuntu :~/lib/inc$ kill -9 4418 4426

Взаємодія бібліотек

Під час створення, динамічні бібліотеки можуть компонуватися з іншими динамічними або статичними бібліотеками. Визначити, які динамічні бібліотеки необхідні для працездатності програми або іншої бібліотеки,

можна за допомогою утиліти ldd. Слід зазначити, що статичні бібліотеки включаються до файлу програми (або бібліотеки), що виконується, на етапі складання програми, тому в списку залежностей вони фігурувати не будуть.

Приклад 6.7.

Визначимо, які бібліотеки використовуються під час роботи програми prog з прикладу 5.4.

delux@ubuntu :~/lib/inc2$ ldd prog

linux-vdso.so.1 => (0x00007fffa19ff000)

libinc.so => ./libinc.so (0x00007f8e45172000)

libc.so.6 => /lib/libc.so.6 (0x00007f8e44dd5000)

/lib64/ld-linux-x86-64.so.2 (0x00007f8e45376000)

Видно, що для забезпечення працездатності програми необхідні 4 динамічні бібліотеки: 1) linux-vdso.so.1, 2) libinc.so, 3) libc.so.6, 4) /lib64/ld-linux-x86-64.so .2.

Завдання лабораторної роботи 6

Необхідно модернізувати програму (лабораторна робота № 5) наступним чином.

1. Перетворення вільних аргументів в масив цілих чисел необхідно реалізувати у вигляді функції з наступними параметрами: 1) ім'я масиву вільних аргументів, 2) кількість елементів масиву, 3) ім’я масиву цілих чисел. Дану функцію необхідно розмістити в статичній бібліотеці.

2. Функцію сортування послідовності цілих чисел необхідно помістити в динамічну (роздільну) бібліотеку. Параметри функції: 1) ім'я масиву, 2) кількість елементів масиву, 3) напрямок сортування - зростання/спадання).

При збірці програми необхідно підключити зазначені бібліотеки. Для збірки проекту рекомендується створити make-файл та скористатися утилітою make.

Лабораторна робота 7

Багатозадачність у Linux

Теоретичні відомості

Багатозадачність в Linux заснована на процесах, які утворюють ієрархію, яка називається деревом процесів, коренем якого є процес init (предок всіх процесів).

Процес – це деяка сукупність набору виконуваних команд, асоційованих з ним ресурсів (виділена для виконання пам'ять або адресний простір, стеки, файли та пристрої введення-виводу, що використовуються, і т. д.) і поточного моменту його виконання (значення регістрів, програмного лічильника, стан стека та значення змінних), що знаходиться під керуванням операційної системи.

Процес, що створив цей, називається **батьківським**.

Кожен процес має цілу низку **атрибутів**. Найважливішими серед них є такі:

1) ідентифікатор процесу – **process identifier (PID)** - позитивне ціле число, що однозначно ідентифікує процес протягом часу його життя;

2) ідентифікатор батьківського процесу – **process parent identifier (PPID);**

3) ідентифікатор користувача, що створив процес – **user identifier (UID)**;

4) ідентифікатор групи користувача – **group identifier (GID).**

Процеси можуть створюватися та завершуватися. Час життя процесу – це період від створення до повернення його ідентифікатора операційній системі.

Після того, як процес створений, він вважається активним. Процес може перейти в неактивний стан, і тоді деякі його ресурси (але не ідентифікатор) можуть бути повернені системі.

Завершення процесу може бути **нормальним** чи **аварійним**. Нормальне завершення відбувається, зокрема, під час повернення з функції main(). Аварійне завершення може мати місце при посиланні процесу сигналу **KILL**. Після завершення процесу системі повертаються його ідентифікатор, який далі може використовуватися при створенні нових процесів.

У разі завершення батьківського процесу, дочірній процес як новий батько отримує процес init, який за умовчанням усиновлює всі осиротілі процеси.

Зомбі-процес – це процес, що завершився (проте займає ресурси системи), код завершення якого ще не переданий батьківському процесу. Після передачі коду батьківському процесу або після завершення батьківського процесу, зомбі-процес звільняє ресурси системи.

Створення нового процесу за допомогою функції system() та отримання інформації про процес

Функція system() представляє зручний спосіб виконання команд усередині програми:

#include<stdlib.h>

int system(const char * cmdstring);

Параметр cmdstring являє собою рядок команду, що зберігає (наприклад, виклик програми з аргументами) яку необхідно виконати.

Функція повертає статус завершення команди (статус завершення процесу буде розглянуто пізніше) або -1 у разі помилки.

Отримати інформацію про поточний процес (атрибути) можна за допомогою таких функцій:

#include <unistd.h> pid_t getpid (void); pid_t getppid (void); pid_t getpgrp (void); uid_t getuid (void);

Поточний процес – це процес, «у якому» ці функції викликаються. Функція *getpid()* повертає ідентифікатор процесу **(PID)**.

Функція *getppid()* повертає ідентифікатор батьківського процесу (PPID). Функція *getpgrp()* повертає ідентифікатор групи процесу (GID).

Функція *getuid()* повертає ідентифікатор користувача (UID) від імені якого виконується процес.

Типи даних **pid_t** і **uid_t** є цілими числовими типами, оголошеними в заголовному файлі <sys/types.h>.

Приклад 7.1. Розробимо консольну програму, яка виконує команду ls. (виведення вмісту поточного каталогу), а також атрибути процесу, що виводить на консоль.

"main.c"

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

int main (void)
{
system("ls."); // новий дочірній процес
printf ("PID: %d\n", getpid());
printf ("PPID: %d\n", getppid());
printf ("UID: %d\n", getuid());
printf ("GID: %d\n", getgid()); return 0;
}
```

Викликаємо програму.

delux@ubuntu :~/QtSDK/system$ ./system

main.cpp system system.pro system.pro.user

PID: 2696

PPID: 2665

UID: 1000

GID: 1000

Створення нового процесу за допомогою функції fork(), призупинення процесу

Функція *fork()* породжує новий процес шляхом «клонування». Тобто новий процес є точною копією свого батька і виконує ту саму програму. Функція має наступний прототип:

#include <unistd.h> pid_t fork (void);

Новий (породжений) процес є точною копією процесу, що викликав fork() (батьківського), за винятком, зокрема, наступних моментів:

1) Породжений процес має свої (відмінні від батьківського) **PID** та **PPID.**

2) Породжений процес має власну точну копію даних (змінних) батьківського процесу. Далі батьківський і дочірній процес опрацьовуватимуть ці дані незалежно один від одного. Наприклад, якщо батьківський процес має змінну **n=5, то** після створення дочірнього процесу він також матиме змінну n зі значенням 5. Далі, якщо в дочірньому процесі змінити значення змінної n на 10, то в батьківському ця змінна все одно залишиться рівною 5 (і навпаки).

3) Породжений процес має власну копію файлових дескрипторів, що посилаються на ті ж описи відкритих файлів, що й відповідні дескриптори батьківського процесу.

У разі успішного завершення функція *fork()* повертає породжений процес 0, а батьківський процес – ідентифікатор породженого процесу**.** Після цього обидва процеси починають незалежно виконувати інструкції,

розташовані за зверненням до *fork().* При невдачі батьківського процесу повертається -1 новий процес не створюється.

Оскільки значення, що повертаються функцією *fork(),* різні для обох копій, батьківський і породжений процеси можуть далі виконуватися по-різному. Наприклад, процес-предок перетворюється на стан очікування завершення процесу-нащадка чи продовжує виконання паралельно із нею.

Призупинення процесу (головного потоку в процесі) реалізується за допомогою функції *sleep():*

#include <unistd.h>

int sleep(unsigned int seconds)

Параметр seconds задає інтервал часу припинення за секунди. Функція повертає 0 або кількість секунд, що залишилися до призупинення у разі дострокового переривання призупинення сигналу.

Приклад 7.2. Розробимо консольну програму, що виводить на консоль атрибути батьківського та дочірнього процесів, а також значення цілісної змінної n. Після виведення даних кожен процес «засинає» (зупиняється) на 10 секунд.

"main.c"

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

// Вивід атрибутів процесу
void print_atr()
{
printf ("PID: %d\t", getpid()); printf ("PPID: %d\t", getppid());
printf ("UID: %d\t", getuid()); printf ("GID: %d\t\n", getgid());
```

```
}

int main (void)
{
int n = 5;

pid_t result = fork(); // створення дочірнього процесу
if (result == -1)
{
fprintf (stderr, "Помилка створення дочірнього процесу\n"); return
1;
}

if (result == 0)
{
n += 10;
printf ("Дочірній процес, n = % d\n", n); print_atr();
sleep(10);
}
else
{
n += 5;
printf ("Батьківський процес, n = % d\n", n); print_atr();
sleep(10);
}

return 0;
}
```

Викликаємо програму у фоновому режимі.

delux@ubuntu :~$ ./fork& [1] 3497

Батьківський процес, n = 10

PID:3497 PPID: 2665 UID: 1000 GID: 1000

Дочірній процес, n = 15

PID:3498PPID: 3497UID: 1000GID: 1000

delux@ubuntu :~$ ps

PIDTTYTIME CMD

2665pts/000:00:00 bash

3497pts/000:00:00 fork батьківський процес

3498pts/000:00:00 fork дочірній процес

3500pts/000:00:00 ps

delux@ubuntu :~$ ps

PIDTTYTIME CMD

2665pts/000:00:00 bash

3521pts/000:00:00 ps

[1]+Готово.

Як видно, операції над змінною n у батьківському процесі не впливають на значення копії цієї змінної у дочірньому процесі (і навпаки).

Заміна поточного образу процесу

Код, що виконується всередині процесу, називається образом процесу (process image). Сімейство функцій exec() дозволяє замінити поточний образ процесу на новий. Новий образ створюється на основі файлу (файлу програми), що виконується, званого файлом образу процесу. Іншими словами, ці функції завантажують у процес іншу програму і передають їй безповоротне управління.

Функції сімейства exec() мають такі прототипи (розглянемо лише частина їх):

#include <unistd.h>

int execl (const char * path, const char * arg0, ... / *, (char *) 0 * /);

int execv (const char * path, char * const argv []);

int execlp (const char *file, const char *arg0,... /*, (char *) 0 */);

int execvp (const char * file, char * const argv []);

Аргумент path вказує на абсолютний чи відносний шлях до файлу з новим чином процесу. Наприклад, для запуску утиліти date за допомогою цієї функції, необхідно передати абсолютний шлях до файлу: «/usr/bin/date».

Аргумент file має аналогічний зміст, проте, якщо він заданий як просте ім'я, то провадиться пошук у каталогах, заданих змінної оточення PATH. Наприклад, для запуску утиліти date необхідно передати просте ім'я файлу, що виконується: «date».

Аргументи arg0, ... є вказівниками на символьні рядки, що складають список аргументів нового образу процесу. Останнім у списку розташовується порожній покажчик NULL, а аргумент arg0 повинен вказувати на ім'я файлу-образу (файлу, що виконується).

Аргумент argv має той самий сенс і призначення, що і змінна argv функції main(), тобто є масивом аргументів програми. При цьому останнім у списку розташовується порожній покажчик NULL.

Наступні атрибути (і навіть ряд інших) процесу залишаються незмінними:

1) **ідентифікатор процесу;**
2) **ідентифікатор батьківського процесу;**
3) **ідентифікатор групи процесів;**
4) **поточний та кореневий каталоги.**

Приклад 7.3. Розробимо консольну програму, що виводить інформацію про систему за допомогою утиліти (системної програми) uname.

"main.c"

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

```
int main (void)
{
char * uname_args[] = {"uname", "-a", NULL};
// перший виклик
int code = execv("/bin/uname", uname_args);
if (code == -1)
{
fprintf (stderr, "Помилка виклику execv\n"); return 1;
}
//другий виклик - недосяжний код
printf ("Цей текст не буде виведено\n");
code = execl("/bin/uname", "uname", "-a", NULL);
if (code == -1)
{
fprintf (stderr, "Помилка виклику execl\n"); return 1;
}
```

Викликаємо програму.

delux@ubuntu :~$ ./exec1

Linux ubuntu 2.6.35-31-generic #62-Ubuntu SMP Tue Nov 8 14:20:11 UTC 2011 x86_64 GNU/Linux

Як видно, повторне виведення інформації про систему не було виконано (виклик execl()), тому що (перший) виклик функції execv() повністю без повернення передав управління програмі uname.

Приклад 7.4. Модернізуємо код попереднього прикладу таким чином, щоб виклик функції execl() став досяжним.

"main.c"

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

int main (void)
{
char * uname_args[] = {"uname", "-a", NULL}; int code;

pid_t result = fork(); // створення дочірнього процесу
if (result == -1)
{
fprintf (stderr, "Помилка створення дочірнього процесу\n"); return
1;
}

if (result == 0) // Дочірній процес
{
printf("Дочірній процес\n");
code = execv("/bin/uname", uname_args);
if (code == -1)
{
fprintf (stderr, "Помилка виклику execv\n"); return 1;
}
}
else //Батьківський процес
{
printf ("Батьківський процес\n");
code = execl("/bin/uname", "uname", "-a", NULL);
if (code == -1)
{
fprintf (stderr, "Помилка виклику execl\n"); return 1;
```

```
}
}
return 0;
}
```

Викликаємо програму.

delux@ubuntu :~$ ./exec2 Батьківський процес Дочірній процес

Linux ubuntu 2.6.35-31-generic #62-Ubuntu SMP Tue Nov 8 14:20:11 UTC 2011 x86_64 GNU/Linux

Linux ubuntu 2.6.35-31-generic #62-Ubuntu SMP Tue Nov 8 14:20:11 UTC 2011 x86_64 GNU/Linux

Реалізація очікування батьківських процесів-нащадків та отримання інформації про статус їх завершення

Батьківський процес може реалізувати очікування завершення процесу-нащадка та отримати інформацію про статус його завершення за допомогою функцій сімейства wait():

```
#include <sys/wait.h>
pid_t wait (int * exit_status);
pid_t waitpid (pid_t pid, int * exit_status, int options);
```

Функція *wait()* блокує батьківський процес, доки не завершиться будь-який з його нащадків. Параметр exit_status вказує на змінну, в яку буде передано статус завершення процесу нащадка.

Статус завершення процесу-нащадка — це ціле число, що містить код повернення і іншу інформацію про те, як завершився процес. Для отримання цієї інформації у файлі <sys/wait.h> визначено ряд макросів:

1) **WIFEXITED()** буде ненульовим у разі нормального завершення породженого процесу;

2) **WEXITSTATUS()** повертає код повернення у разі нормального

завершення процесу.

3) WIFSIGNALED() повертає ненульове значення, якщо процес був завершений шляхом отримання сигналу**.**

Функція *wait()* повертає ідентифікатор процесу, що завершився.

Якщо інформація про статус завершення доступна до виклику wait(), батьківський процес не припиняється, повернення з *wait()* відбувається негайно.

Функція *waitpid()* аналогічна *wait(),* проте дозволяє батьківському процесу очікувати завершення конкретного процесу-нащадка. Аргумент *exit_status* має той самий зміст як і функції *wait().* Аргумент options задається як побітове АБО наступних прапорів, визначених у заголовному файлі **<sys/wait.h>.** До таких прапорів належить таке:

WNOHANG - функція *waitpid()* не призупиняє батьківського процесу, якщо процес нащадок ще працює.

Функція *waitpid()* еквівалентна *wait(),* якщо аргумент pid дорівнює **(pid_t) (-1),** а аргумент options має нульове значення.

Приклад 8.5. Розробимо консольну програму, у якій батьківський процес обробляє статус завершення нащадка щодо отримання сигналу. У нащадку викликається системна програма sleep (аналогічна функції sleep()) з інтервалом 30 секунд.

"main.c"

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

int main (void)
{
```

```
pid_t result, childpid; int exit_status;

result = fork(); // створення дочірнього процесу
if (result == -1)
{
fprintf (stderr, "Помилка створення дочірнього процесу\n"); return 1;
}
//Дочірній процес
if (result == 0)
{
execlp ("sleep", "sleep", "30", NULL);
fprintf (stderr, "Помилка виклику execlp\n"); return 1;
}
//Батьківський процес
childpid = wait(&exit_status);
if (WIFEXITED(exit_status))
{
printf ("Процес з PID=%d завершив свою роботу з кодом =%d\n", childpid, WEXITSTATUS (exit_status));
}
if (WIFSIGNALED(exit_status))
{
printf ("Процес з PID=%d завершив свою роботу по сигналу\n", childpid);
}
return 0;
}
```

Викликаємо програму.

delux@ubuntu :~$ ./wait1

Процес PID=5124 завершив свою роботу з кодом =0

Викликаємо програму у фоновому режимі та завершимо дочірній процес командою kill.

delux@ubuntu :~$ ./wait1 & [1] 5207

delux@ubuntu :~$ ps

PIDTTYTIME CMD

2665pts/000:00:00 bash

5207pts/000:00:00 wait1

5208pts/000:00:00 sleep

5209pts/000:00:00 ps

delux@ubuntu :~$ kill 5208

delux@ubuntu :~$ Процес з PID=5208 завершив свою роботу за сигналом [1]+Готово./wait1

Приклад 7.5. Розробимо наступну консольну програму.

За допомогою системного виклику *fork()* породжується новий дочірній процес, у якому виконується цикл щомиті виведення точок на екран. Після цього циклу в дочірньому процесі викликається функція *exit(),* яка його завершує. Код завершення дочірнього процесу визначається рівним 5 і обраний довільно.

У цей час у батьківському процесі кожні 2 секунди на консоль виводиться символ ***** і викликається функція *waitpid()* з першим аргументом, що дорівнює -1, що означає очікування завершення будь-якого процесу. У третьому аргументі функції передається прапор **WNOHANG**, завдяки якому батьківський процес не блокується, якщо дочірній ще не завершився, а продовжує виводити символ ***** і чекати завершення нащадка.

"main.c"

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

```
#include <wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#define DOT_COUNT 10
int main (void)
{
pid_t result, childpid; int i, exit_status;

result = fork(); // створення дочірнього процесу
if (result == -1)
{
fprintf (stderr, "Помилка створення дочірнього процесу\n"); return
1;
}
//Дочірній процес
if (result == 0)
{
for (i = 0; i < DOT_COUNT; i++)
{
fprintf(stderr,"."); sleep (1);
}
exit (5);
}
// Батьківський процес
while (1) // Нескінченний цикл
{
if ((childpid = waitpid (-1, exit_status, WNOHANG)) == 0)
fprintf (stderr, "*");
else
```

```
{
fprintf (stderr, "(вихід)\n"); break;
}
sleep (2);
}
if (WIFEXITED(exit_status))
{
printf ("Процес з PID=%d завершив свою роботу з кодом =%d\n", childpid, WEXITSTATUS (exit_status));
}
if (WIFSIGNALED(exit_status))
{
printf ("Процес з PID=%d завершив свою роботу по сигналу\n", childpid);
}
return 0;
}
```

Викликаємо програму.

```
delux@ubuntu :~$ ./wait2
*..*..*..*..*..*(вихід)
Процес PID=5428 завершив свою роботу з кодом =5
```

Викликаємо програму у фоновому режимі, перенаправивши потік помилок у файл 1.txt та завершимо дочірній процес командою kill.

```
delux@ubuntu :~$ ./wait2 2> 1.txt & [1] 5483
delux@ubuntu :~$ ps
PIDTTYTIME  CMD
2665pts/000:00:00  bash
5483pts/000:00:00  wait2
5484pts/000:00:00  wait2
5485pts/000:00:00  ps
```

delux@ubuntu :~$ kill 5484

Процес з PID=5484 завершив свою роботу за сигналом

[1] +Готово

delux@ubuntu :~$ ./wait2 2> 1.txt

delux@ubuntu :~$ cat 1.txt

***..*..*..*..*..(вихід)**

Потоки (нитки виконання)

Потоки (threads) дозволяють у межах однієї програми виконувати одночасно кілька дій, використовуючи загальні дані процесу. Механізми роботи з потоками реалізовані в Linux бібліотеці **pthread** (заголовний файл **<pthread.h>).**

Будь-який процес має щонайменше один потік. Кожен потік містить таку інформацію:

1) ідентифікатор потоку, що дозволяє ідентифікувати потік усередині процесу;

2) набір значень у регістрах процесора;

3) стек;

4) та інші дані специфічні для потоку.

Усі компоненти процесу, включаючи виконуваний код програми, глобальні змінні та динамічну пам'ять, файлові дескриптори можуть спільно використовувати різні потоки цього процесу.

Потоки реалізуються в Linux так:

1) Створюється функція, що називається потоковою функцією.

2) За допомогою функції *pthread_create()* створюється потік, в якому починає паралельно решті програми виконувати потокову функцію.

3) Викликаюча сторона продовжує виконувати свою роботу, не чекаючи завершення потокової функції.

Створення потоку

Створити новий потік усередині поточного потоку можна за допомогою функції:

#include <pthread.h>

int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);

Новий потік буде виконувати функцію start_routine із прототипом:

void * start_routine (void *);

передаючи їй як аргумент параметр arg. Якщо потрібно передати більше одного параметра, вони поєднуються в структуру, і функції передається адресу цієї структури. Значення, що повертається функцією start_routine, не повинно вказувати на динамічний об'єкт цього потоку.

Параметр attr служить завдання різних атрибутів створюваного потоку, наприклад розмір стека потоку в байтах. Далі будемо вважати їх заданими за умовчанням, підставляючи як аргумент значення NULL.

При вдалому завершенні функція повертає значення 0 і поміщає ідентифікатор нової нитки виконання за адресою, який вказує параметр thread типу **pthread_t**. Тип даних **pthread_t** призначений для зберігання ідентифікатора потоку і є цілим типом, оголошеним в заголовному файлі **<sys/types.h>.**

У разі помилки повертається позитивне значення (а не негативне, як у більшості системних викликів та функцій), яке визначає код помилки, описаний у файлі **errno.h.** Значення системної змінної errno у своїй не встановлюється.

Завершення потоку

Потоки завершуються при поверненні з потокової функції, завершення процесу в рамках якого вони виконуються або за допомогою спеціальної функції *pthread_exit()*, яка викликається в самому потоці:

#include <pthread.h>

void pthread_exit (void * status);

Функція ніколи не повертається в потік, що викликав її. Об'єкт, на який вказує параметр status, зберігає інформацію про завершення потоку, і може бути використаний в іншому потоці, наприклад, в потік, що породив завершився. Тому він не повинен вказувати на динамічний об'єкт потоку, що завершився.

Блокування потоку

Блокувати (зупинити) роботу потоку можна за допомогою функції:

#include <pthread.h>

int pthread_join (pthread_t thread, void **status_addr);

Функція *pthread_join()* блокує роботу потоку виконання, що викликав її, до завершення потоку з ідентифікатором ***thread***. Після розблокування в покажчик, розташований за адресою **status_addr**, заноситься адреса, яка повернула потік, що завершився, або при виході з асоційованої з ним функції, або при виконанні функції pthread_exit(). Якщо немає значення, що повернув потік, як цей параметр можна використовувати значення NULL.

Функція повертає значення 0 у разі успішного завершення. У разі помилки повертається позитивне значення (а не негативне, як у більшості системних викликів та функцій), яке визначає код помилки, описаний у файлі <errno.h>. Значення системної змінної errno у своїй не встановлюється.

Приклад 7.6

.Розробимо наступну консольну програму.

В рамках одного процесу створюються два додаткові потоки, один з яких у циклі виводить на екран цифру 1, а другий цифру 2. Кількість ітерацій циклу передається в потокову функцію як аргумент.

"main.c"

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

void * thread_func1 (void * arg)
{
int i;
int n = * (int *) arg;
for (i = 0; i < n; i++)
{
fprintf(stderr, "1"); sleep(1);
}
return NULL;
}

void * thread_func2(void * arg)
{
int i;
int n = * (int *) arg;
for (i = 0; i < n; i++)
{
fprintf(stderr, "2"); sleep(1);
}
return NULL;
}

int main (void)
{
```

```
pthread_t thread1, thread2; int n1 = 10, n2 = 15;

// створюємо потік 1
if (pthread_create (&thread1, NULL, &thread_func1, &n1) != 0)
{
fprintf (stderr, "Помилка (thread1)\n"); return 1;
}
// створюємо потік 2
if (pthread_create (&thread2, NULL, &thread_func2, &n2) != 0)
{
fprintf (stderr, "Помилка (thread2)\n"); return 1;
}
// Дочекаємося завершення потоку 1
if (pthread_join (thread1, NULL)! = 0)
{
fprintf (stderr, "Помилка pthread_join() (thread1)\n"); return 1;
}
// Дочекаємося завершення потоку 2
if (pthread_join (thread2, NULL)! = 0)
{
fprintf (stderr, "Помилка pthread_join() (thread2)\n"); return 1;
}
fprintf (stderr,"Hello World\n"); return 0;
}
```

Викликаємо програму.

```
delux@ubuntu :~$ ./thread1
1 2 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 2 2 2 2 Hello World
```

Отримання ідентифікатора потоку та порівняння ідентифікаторів потоку

Отримати ідентифікатор поточного потоку можна за допомогою функції *pthread_self():*

#include <pthread.h> pthread_tpthread_self(void);

pthread_self() повертає ідентифікатор поточного потоку.

Для порівняння ідентифікаторів двох потоків одного процесу рекомендується використовувати:

#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);

Якщо параметри t1 і t2 вказують на той самий потік, функція поверне не нульове значення, інакше - функція поверне 0.

Надіслати запит в одному потоці для примусового завершення іншого потоку в рамках одного процесу можна за допомогою функції:

#include <pthread.h>

int pthread_cancel(pthread_t thread);

Параметр thread є ідентифікатором потоку, який потрібно завершити. Функція повертає 0 у разі успіху або код помилки у разі невдачі.

Приклад 7.7.

Розробимо наступну консольну програму.

В рамках одного процесу створюється один додатковий потік, який у нескінченному циклі виводить на екран цифру 1. В основному потоці програми цей додатковий потік примусово завершується після 5 секунд роботи.

"main.c"

#include <stdio.h> #include <pthread.h> #include <unistd.h>

```
void * thread_func (void * arg)
{
while(1)
{
fprintf (stderr, "1");
sleep (1);
}
return NULL;
}

int main (void)
{
pthread_t thread;
// створюємо додатковий потік
if (pthread_create (&thread, NULL, &thread_func, NULL) != 0)
{
fprintf (stderr, "Помилка (thread)\n"); return 1;
}
sleep (5);
// викликаємо функцію примусового завершення додаткового потоку
pthread_cancel(thread);
// очікуємо примусового завершення додаткового потоку
// Дочекаємося завершення потоку 1
if (pthread_join (thread, NULL)! = 0)
{
fprintf (stderr, "Помилка pthread_join() (thread)\n"); return 1;
}
fprintf (stderr,"Hello World\n"); return 0;
}
```

Викликаємо програму.

delux@ubuntu :~$ ./thread2

11111Hello World

Приклад 7.8.

Розробимо наступну консольну програму. Випадково генерується масив дійсних чисел. Далі у ньому шукається максимальний елемент, який виводиться на консоль. Програма має працювати як у однопотоковому так і двопотоковому режимі. Кількість елементів масиву та режим програми задаються у програмі з консолі. Програма повинна виводити на консоль час, витрачений на пошук максимального елемента.

"main.c"

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>

struct thread_arg
{
double * first; int n;
double max;
};

void * thread_max (void * arg)
{
int i;
double * a = ((thread_arg *) arg) -> first;
int n = ((thread_arg *) arg) -> n;
```

```
double max = a[0];
for (i = 0; i < n; i++)
{
if (max < a[i])
max = a[i];
}
((thread_arg *) arg) -> max = max; return NULL;
}

int main (void)
{
pthread_t thread1, thread2; struct thread_arg arg1, arg2; int i, mode,
n;
float sec;
double max, max1, max2;
struct timespec t1, t2;

printf("Задайте режим роботи програми: 1 - однопотоковий, 2 -
двопотоковий\n"); scanf("%d", &mode);
printf("Задайте кількість елементів масиву:"); scanf("%d", &n);
double * a = (double*) calloc(n,sizeof(double));
for (i = 0; i < n; i++)
a[i] = (double)rand();

if (mode == 1) //один потік
{
clock_gettime(CLOCK_REALTIME,&t1);
arg1.first = a;
arg1.n = n;
thread_max(&arg1); max = arg1.max;
```

```
clock_gettime(CLOCK_REALTIME, &t2);
sec = (double) ((t2.tv_sec - t1.tv_sec) * 1.0e9 + (t2.tv_nsec - t1.tv_nsec)) /
1.0e9;
}
else //два потоки
{
clock_gettime(CLOCK_REALTIME, &t1);
// Створюємо потік 1
arg1.first = a; arg1.n = n/2;
if (pthread_create (&thread1, NULL, &thread_max, &arg1) != 0)
{
fprintf (stderr, "Помилка (thread1)\n"); return 1;
}
// Створюємо потік 2
arg2.first = a + n/2; arg2.n = n - n/2;
if (pthread_create (&thread2, NULL, &thread_max, &arg2) != 0)
{
fprintf (stderr, "Помилка (thread2)\n"); return 1;
}
// Дочекаємося завершення потоку 1
if (pthread_join (thread1, NULL)! = 0)
{
fprintf (stderr, "Помилка pthread_join() (thread1)\n"); return 1;
}
// Дочекаємося завершення потоку 2
if (pthread_join (thread2, NULL)! = 0)
{
fprintf (stderr, "Помилка pthread_join() (thread2)\n"); return 1;
}
max = arg1.max > arg2.max? arg1.max: arg2.max;
```

```
clock_gettime(CLOCK_REALTIME, &t2);
sec = (double) ((t2.tv_sec - t1.tv_sec) * 1.0e9 + (t2.tv_nsec - t1.tv_nsec)) / 1.0e9;
}
free(a);
printf ("Max = %lf\n Витрачений час: %lf сек\n", max, sec); return 0;
}
```

Викликаємо програму кілька разів.

```
delux@ubuntu :~$ ./threadmin
Встановіть режим роботи програми: 1 - однопотоковий, 2 - двопотоковий 1
Вкажіть кількість елементів масиву: 100000000 Max = 2147483611.000000
Витрачений час: 0.096966 сек

delux@ubuntu :~$ ./threadmin
Встановіть режим роботи програми: 1 - однопотоковий, 2 - двопотоковий 1
Вкажіть кількість елементів масиву: 100000000 Max = 2147483611.000000
Витрачений час: 0.097142 сек

delux@ubuntu :~$ ./threadmin
Встановіть режим роботи програми: 1 - однопотоковий, 2 - двопотоковий 2
Вкажіть кількість елементів масиву: 100000000 Max = 2147483611.000000
Витрачений час: 0.052659 сек
```

delux@ubuntu :~$ ./threadmin

Встановіть режим роботи програми: 1 - однопотоковий, 2 - двопотоковий 2

Вкажіть кількість елементів масиву: 100000000 Max = 2147483611.000000

Витрачений час: 0.050252 сек

Як бачимо, у двопотоковому режимі, програма вирішує завдання майже 2 разу швидше, ніж у однопотоковому режимі.

Завдання лабораторної роботи 7

Необхідно розробити консольну програму, призначену для виведення інформації про користувачів та файли. Ім'я користувача та ім'я файлу задаються як аргументи командного рядка. Інформація повинна виводитись лише про користувача або лише про файл. Для того щоб визначити, яку інформацію хоче отримати користувач, необхідно аналізувати ключ з якої запускається програма, наприклад:

1) ./prog -u user1 // буде виведено інформацію про користувача

2) ./prog -f 1.txt // буде виведено інформацію про файл

Інформація, яка повинна бути виведена, відповідає всім полям структури passwd для користувача і наступним полям структури stat для файлу:

1) nlink_t st_nlink; // Число жорстких посилань на файл

2) uid_t st_uid; // Ідентифікатор власника файлу

3) gid_t st_gid; // Ідентифікатор групи, що володіє

4) off_t st_size; // Для звичайних файлів та символьних посилань – розмір у байтах.

5) а також ім'я власника файлу.

Лабораторна робота 8

Отримання інформації про користувачів системи

Теоретичні відомості

Операційна система Linux підтримує базу даних користувачів, у якій про кожного з них зберігається щонайменше наступна інформація:

1) ім'я користувача;

2) числовий ідентифікатор користувача;

3) числовий ідентифікатор групи користувача;

4) домашній (початковий робочий) каталог;

5) початкова програма користувача.

Кожен зареєстрований користувач ОС має ім'я, яке він вказує на цілі ідентифікації при вході в систему. Після проведення ідентифікації з ним асоціюються (невід'ємні) числові ідентифікатори користувача та початкової групи. Числовий ідентифікатор 0 характеризує суперкористувача (root). На відміну від імен, ОС оперує ними у всіх випадках, крім початкової ідентифікації. Потім може запускатися початкова програма користувача (наприклад, командна оболонка) із зазначеним початковим робочим каталогом. Поля початкового робочого каталогу та початкової програми користувача можуть бути порожніми. Зазвичай у системі визначена передбачувана початкова програма, якою часто використовується /bin/bash.

Користувачі об'єднуються у групи, кожен користувач є членом хоча б однієї з існуючих груп у системі. Для груп також існує база даних, її записи містять принаймні наступні поля кожної групи:

1) ім'я групи;

2) числовий ідентифікатор групи;

3) список користувачів цієї групи.

У базі даних користувачів вказується ідентифікатор початкової групи, до неї (групу) користувач потрапляє відразу після входу до системи.

Вхідне ім'я поточного користувача (який запустив програму) можна дізнатися за допомогою функції *getlogin():*

#include <unistd.h>

char * getlogin (void);

Над базою даних користувачів визначено операції пошуку за ідентифікатором або ім'ям користувача, що реалізуються відповідно функціями getpwuid() і getpwnam():

#include <pwd.h>

struct passwd *getpwuid (uid_t uid);

struct passwd *getpwnam (const char *name);

Дані функції повертають інформацію про користувача в структурі типу passwd.

За стандартом структура passwd повинна містити принаймні наступні поля, що відповідають описаним вище обов'язковим елементам бази даних користувачів:

struct passwd

{

char *pw_name;// Ім'я користувача */

uid_t pw_uid;//Числовий ідентифікатор користувача */

gid_t pw_gid;// Числовий ідентифікатор початкової групи */

char *pw_dir;// Початковий робочий каталог */

char *pw_shell;// Початкова програма користувача */

};

Типи uid_t і gid_t визначаються у заготівельному файлі <sys/types.h>.

Приклад 8.1. Створимо програму userinfo, що виводить інформацію про поточного користувача та суперкористувача root з ідентифікатором 0.

"main.c"

#include <unistd.h>

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
// Друк елемента бази даних користувачів
static void print_pwent (const struct passwd * pw)
{
printf ("Ім'я користувача: %s\n", pw->pw_name);
printf ("Ідентифікатор користувача: %d\n", pw->pw_uid);
printf ("Ідентифікатор групи: %d\n", pw->pw_gid);
printf ("Початковий каталог: %s\n", pw->pw_dir);
printf ("Початкова програма: %s\n", pw->pw_shell);
}
int main (void)
{
char *lgnm; // Ім'я поточного користувача
struct passwd * pw; // Дані про поточного користувача
// Пошук та друк інформації про поточного користувача
if ((lgnm = getlogin ()) == NULL || (pw = getpwnam (lgnm)) == NULL)
{
fprintf (stderr, "\nНе вдалося знайти інформацію про поточного користувача\n");
return 1;
}
printf ("\nІнформація про поточного користувача\n");
print_pwent (pw);
// Те ж саме для користувача root
if ((pw = getpwuid ((uid_t) 0)) == NULL)
{
fprintf (stderr, "\nНе вдалося знайти інформацію про користувача root\n");
```

```
return 1;
}
printf ("\nІнформація про користувача root\n");3

print_pwent (pw);
return 0;
}
```

Викликаємо програму.

```
delux@ubuntu :~/user/bin/Debug$ ./userinfo

Інформація про поточного користувача
Ім'я користувача: delux
Ідентифікатор користувача: 1000
Ідентифікатор групи: 1000
Початковий каталог: /home/delux
Початкова програма: /bin/bash
Інформація про користувача root

Ім'я користувача: root
Ідентифікатор користувача: 0
Ідентифікатор групи: 0
Початковий каталог: /root
Початкова програма: /bin/bash
```

Аналогічні функції є для вилучення інформації про групу за її ідентифікатором або ім'ям -getgrgid() і getgrnam() з бази даних груп:

```
#include <grp.h>
struct group *getgrgid (gid_t gid);
```

struct group *getgrnam (const char *name);

Функції повертають інформацію про групу у структурі типу group.

Структура group містить такі поля:

```
struct group
{
char *gr_name;  // Ім'я групи
gid_t gr_gid;       // Числовий ідентифікатор групи
char **gr_mem;// Покажчик на масив символьних покажчиків на імена
     // користувачів,
    // яким дозволено ставати членами цієї групи
};
```

Приклад 8.2. Створимо програму, що виводить список користувачів, включених до групи із заданим ідентифікатором.

"main.c"

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

// Друк списку користувачів, включених до групи із заданим ідентифікатором
int print_gr_mem (const gid_t gid)

{
struct group *grp;// Дані про групу
char **c_gr_mem;// Поточний покажчик на ім'я члена групи
```

```
char *c_gr_mem_name;// Поточне ім'я члена групи
if ((grp = getgrgid (gid)) == NULL)
{
fprintf (stderr, "\nНе вдалося знайти інформацію про групу з ідентифікатором %d\n", gid);
return 1;
}
printf ("\Користувачі, включені до групи з ідентифікатором %d:\n", gid);
while ((c_gr_mem_name = *c_gr_mem++) != NULL)
{
printf("%s\n", c_gr_mem_name);
}
return 0;
}
int main (void)
{
return print_gr_mem (1000);
}
```

Викликаємо програму.

```
delux@ubuntu :~/groupinfo/bin/Debug$ ./groupinfo
Користувачі, включені до групи з ідентифікатором 1000:
Delux
```

Файли та файлова система

Поняття файлу охоплює все, що може містити, споживати та/або постачати інформацію. У Linux розрізняють такі типи файлів:

1) звичайний файл;

2) каталог;

3) канал;

4) спеціальний файл;

5) символьне посилання;

6) сокет.

Звичайний файл є послідовністю байт з можливістю випадкового доступу і без будь-якої додаткової структури, накладеної операційною системою.

Каталог складається з елементів (посилань), що асоціюють імена із файлами. Посилання вказують на файли, які зберігає каталог.

Канал можна уявляти як транспортера, з одного боку якого перебуває постачальник (процес, що пише у канал), з другого - споживач (процес, що читає з каналу). Дані читаються у тому порядку, у якому здійснювалася запис, тобто з погляду структур даних канал - це чергу FIFO.

Спеціальні файли відповідають апаратним компонентам комп'ютера-пристроям. Розрізняють символьні пристрої (не підтримують довільний доступ до даних): стример, модем, термінал та блокові пристрої (підтримують довільний доступ до даних): жорсткий диск, CD-ROM та ін.

Символьне посилання – це файл, що зберігає ланцюжок символів. Коли подібний файл є компонентом маршрутного (шляхового) імені іншого файлу (див. далі), ланцюжок символів, що зберігається, впливає на результуючий маршрут. Зазвичай посилання розкривається (прозорим для додатків чином), тобто замість її імені підставляється вміст (ланцюжок символів).

Сокет – це точка міжпроцесних (мережевих) комунікацій.

Файлова система

Файли разом зі службовою інформацією, що зберігається в об'єктах, які називаються індексними дескрипторами (описувачами) файлів, об'єднуються в ієрархічну структуру (направлений граф), яку називають файловою

системою. Усі некінцеві вершини графа (тобто вершини, звідки виходить хоча б одна дуга) є каталогами; всі кінцеві мають інші типи.

Для кожного процесу визначено кореневий каталог з ім'ям /-вершина графа, з якої здійснюється доступ до інших файлів даної файлової системи.

У межах файлової системи кожен файл має унікальний ідентифікатор (порядковий номер - він номер дескриптора файлу (див. далі)).

Імена, що асоціюються з файлами (точніше, з їх порядковими номерами), називаються простими. З формальної точки зору вони можуть включати будь-які символи, крім слеша (/), проте для підвищення мобільності додатків стандарт рекомендує обмежитися латинськими літерами, цифрами, а також символами ., _, - (точка, підкреслення, мінус), причому мінус не повинен бути першим символом. В операційній системі Linux зберігається різницю між великими і малими літерами.

При використанні в іменах файлів деяких інших символів часто виникають проблеми. Наприклад. Символи національних алфавітів створюють труднощі із відображенням імен. Символи, які мають особливе значення для командного інтерпретатора, можуть викликати несподівані та небажані модифікації імен. Символ : (двокрапка) у імені каталогу здатний порушити нормальну інтерпретацію значення змінної оточення PATH. Мінус як перший символ надає імені файлу вигляд комбінації опцій, порушуючи процес розбору аргументів утиліт, та ін.

У кожному каталозі присутні імена. (точка) та .. (за стандартом читається як точка-точка), які трактуються як посилання на поточний та вищерозташований каталоги. Для кореневого каталогу ім'я .. може також посилатися на корінь.

Під маршрутним ім'ям розуміється ланцюжок символів, що ідентифікує файл і складається з кількох (у тому числі з нуля) простих імен файлів, розділених символами / і так званими компонентами маршруту. Маршрутні імена, що починаються символом /, називаються абсолютними, вони задають

маршрут від кореня файлової системи. Інші імена називаються відносними, маршрут у них заданий щодо поточного каталогу.

Маршрутним префіксом називається маршрутне ім'я, яке посилається на каталог.

Приклад 8.3

/home/delux/link/1.txt- Абсолютне маршрутне ім'я файлу

link/1.txt-відносне маршрутне ім'я файлу

1.txt - просте ім'я файлу

Жорсткі посилання. Індексні дескриптори. Символьні (м'які) посилання.

Жорсткі посилання.

Кожен файл являє собою область даних на жорсткому диску комп'ютера або іншому носії інформації, яку можна знайти за заданим ім'ям. У файловій системі Linux вміст файлу пов'язується з його ім'ям за допомогою жорстких посилань.

Створення файлу за допомогою будь-якої програми означає, що буде створено жорстке посилання – ім'я файлу, і відкрито нову область даних на диску. Причому кількість посилань на ту саму область даних (файл) не обмежена, тобто у файлу може бути кілька імен.

Користувач Linux може додати файлу ще одне ім'я (створити ще одне жорстке посилання на файл) за допомогою утиліти ln (від англ. "link" – "з'єднувати, зв'язувати"):

ln <file> <link>

де

параметр **file** – це ім'я файлу, на який потрібно створити посилання,

параметр link – ім.'я нового посилання.

За замовчуванням посилання буде створено в поточному каталозі.

Приклад. Створимо в робочому каталозі текстовий файл 1.txt та одне жорстке посилання на нього.

delux@ubuntu :~/link$ echo Hello! > 1.txt

delux@ubuntu :~/link$ ls -l

разом 4

-rw-r--r--1 delux delux 7 2011-11-04 10:47 1.txt (одне жорстке посилання - «перша» одиниця)

delux@ubuntu :~/link$ ln 1.txt 2.txt (Створюємо жорстке посилання)

delux@ubuntu :~/link$ ls -l

разом 8

-rw-r--r--2 delux delux 7 2011-11-04 10:47 1.txt (Дві жорсткі посилання - «перша» двійка)

-rw-r--r--2 delux delux 7 2011-11-04 10:47 2.txt (Дві жорсткі посилання - «перша» двійка)

delux@ubuntu :~/link$ cat 1.txt

Hello!

delux@ubuntu :~/link$ cat 2.txt

Hello!

Як видно, у файлів 1.txt і 2.txt збігаються і розмір ("7"), час створення та інші параметри. тобто. тепер **"/home/delux/link/1.txt"** та **"/home/delux/link/2.txt"** – це два імені одного і того ж файлу.

Доступ до одного і того ж файлу за допомогою декількох імен (жорстких посилань) може знадобитися в таких випадках:

1) Одна й та програма відома під кількома іменами.

2) Доступ користувачів до деяких каталогів у системі може бути обмежений з міркувань безпеки. Однак якщо все ж таки потрібно організувати доступ користувачів до файлу, який знаходиться в такому каталозі, можна створити жорстке посилання на цей файл в іншому каталозі.

3) Сучасні файлові системи навіть на домашніх персональних комп'ютерах можуть налічувати до кількох десятків тисяч файлів та тисячі каталогів. Зазвичай у таких файлових систем складна багаторівнева ієрархічна організація – в результаті шляхи до багатьох файлів стають дуже довгими. Щоб організувати більш зручний доступ до файлу, який знаходиться дуже «глибоко» в ієрархії каталогів, можна також використовувати жорстке посилання в більш доступному каталозі.

4) Повне ім'я деяких програм може бути дуже довгим (наприклад, **«codeblocks-10.05-1-debian-dbg-amd64.tar.bz2»),** до таких програм зручніше звертатися за допомогою скороченого імені (жорсткого посилання) – «cb-10.05».

Індексні дескриптори.

Завдяки жорстким посиланням файл може мати кілька імен, відомо, що вся істотна інформація про файл у файловій системі прив'язана не до імені. У файлових системах Linux вся інформація, необхідна роботи з файлом, зберігається в індексному дескрипторі. Для кожного файлу існує індексний дескриптор: як для звичайних файлів, але й каталогів тощо. **Кожному файлу відповідає один індексний дескриптор.**

Індексний дескриптор - це опис файлу, в якому міститься:

1) тип файлу (звичайний файл, каталог тощо);

2) права доступу до файлу;

3) інформація про те, кому належить файл;

4) позначки про час створення, модифікацію, останній доступ до файлу;

5) розмір файлу;

6) покажчики на фізичні блоки на диску, що належать цьому файлу - у цих блоках зберігається «вміст» файлу.

Всі індексні дескриптори пронумеровані, тому номер індексного дескриптора – це унікальний ідентифікатор файлу у файловій системі - на відміну від імені файлу (жорсткого посилання на нього), яких може бути

кілька. Дізнатися номер індексного дескриптора будь-якого файлу можна за допомогою тієї ж утиліти ls з ключем -i.

Приклад 8.4 Дізнаємося номери індексних дескрипторів текстового файлу 1.txt і жорсткого посилання на нього 2.txt.

delux@ubuntu :~/link$ ls -i

2884540 1.txt 2884540 2.txt

Як видно, ці номери збігаються ("2884540"), тобто цим двом іменам відповідає один індексний дескриптор, тобто один і той же файл.

Всі операції з файловою системою – створення, видалення та переміщення файлів – проводяться насправді над індексними дескрипторами, а імена потрібні тільки для того, щоб користувач міг легко орієнтуватися у файловій системі. (Було б дуже незручно запам'ятовувати багатозначний номер кожного потрібного файлу чи каталогу.) Більше того, ім'я (або імена) файлу в його індексному дескрипторі не вказано. У файловій системі імена файлів зберігаються в каталогах: кожен каталог є список імен файлів і номерів їх індексних дескрипторів. Жорстке посилання (ім'я файлу, що зберігається в каталозі) можна представляти як каталожну картку, де вказаний номер індексного дескриптора – ідентифікатор файлу.

Таким чином, жорстке посилання (hard link) – це запис виду:

ім'я файлу+номер індексного дескриптора у каталозі.

Жорсткі посилання в Linux – основний спосіб звернутися до файлу на ім'я.

Символьні посилання.

У жорстких посилань є два суттєві обмеження:

1) Жорстке посилання може вказувати тільки на файл, але не на каталог, тому що в іншому випадку у файловій системі можуть виникнути цикли-нескінченні шляхи.

2) Жорстке посилання не може вказувати на файл в іншій файловій системі. Наприклад, на жорсткому диску неможливо створити жорстке посилання на файл, розташований на лазерному диску. Щоб уникнути цих обмежень, було розроблено символьні посилання.

Символьне посилання – це просто файл, в якому міститься ім'я (шлях) іншого файлу. Символьні посилання, як і жорсткі, надають можливість звертатися до одного й того ж файлу за різними іменами. Крім того, символьні посилання можуть вказувати і каталог, чого не дозволяють жорсткі посилання. Символьні посилання називаються так тому, що містять **символи** – шлях до файлу або каталогу.

Символьне посилання (symbolic link, файл-посилання) – це файл особливого типу, в якому міститься шлях до іншого файлу. Якщо на шляху до файлу зустрічається символьне посилання, система виконує підстановку: вихідний шлях замінюється тим, що міститься в посиланні.

Символьне посилання можна створити за допомогою команди ln з ключем "-s" (скорочення від «symbolic»):

ln -s <file> <slink>

де

параметр **file** – це ім'я файлу, на який потрібно створити посилання,

параметр **slink** – ім'я нового символьного посилання.

За промовчуванням посилання буде створено в поточному каталозі.

Приклад 8.5 Створимо в домашньому каталозі символьне посилання на текстовий файл 1.txt, і виведемо інформацію про це посилання.

delux@ubuntu :~$ ln -s ~/link/1.txt slink-1.txt

delux@ubuntu :~$ ls -il slink-1.txt

2492388 lrwxrwxrwx 1 delux delux 23 2011-11-04 11:24 slink-1.txt -> /home/delux/link/1.txt

Стрілочка (" -> ") вказує, куди спрямоване посилання. Крім того, номер індексного дескриптора (**2492388**), розмір і час створення файлу slink-1.txt відрізняються від 1.txt, а також у другому полі (кількість жорстких посилань на файл) slink-1.txt вказано "1". Всі ці ознаки свідчать, що slink-1.txt і 1.txt - це різні файли. Однак, якщо виконати команду cat **slink-1.txt**, то на екран буде виведено вміст файлу 1.txt:

delux@ubuntu :~$ cat slink-1.txt

Hello!

Символьне посилання може містити ім'я неіснуючого файлу. У цьому випадку посилання буде існувати, але не буде «працювати»: наприклад, якщо спробувати вивести вміст такого «битого» посилання за допомогою команди cat, буде видано повідомлення про помилку:

delux@ubuntu :~$ rm link/1.txt link/2.txt (видаляємо жорсткі посилання)

delux@ubuntu :~$ cat slink-1.txt

cat: slink-1.txt: Немає такого файлу чи каталогу

Дізнатися, куди вказує символьне посилання, можна за допомогою утиліти realpath:

realpath <slink>

де

параметр **slink** – ім'я символьного посилання.

Приклад.

delux@ubuntu :~$ realpath slink-1.txt

/home/delux/link/1.txt9

Отримання інформації про файли, каталоги та файлову систему

З кожним файлом асоційована принаймні наступна службова інформація:

1) режим - об'єкт, що містить біти режиму та тип файлу;

2) числовий ідентифікатор власника-користувача;

3) числовий ідентифікатор групи, що володіє.

Біти режиму включають біти режиму доступу, а також ряд інших бітів, про які можна дізнатися в довідковому посібнику.

Біти режиму доступу визначають, чи має процес, що діє від імені деякого користувача, право на відповідні операції з файлом.

По відношенню до конкретного файлу всі користувачі поділяються на три категорії:

1) власник файлу;

2) члени групи, що володіє;

3) інші користувачі.

Для кожної з них режим доступу визначає права на операції з файлом, а саме:

1) право на читання;

2) право на запис;

3) право на виконання (для каталогів - право на пошук).

Вказаних видів прав достатньо, щоб визначити допустимість будь-якої операції з файлами. Наприклад, для видалення файлу необхідно мати право на запис у відповідний каталог.

Виконання більшості операцій із файлами потребує їх відкриття. Відкритому файлу відповідає файловий дескриптор – невід'ємне ціле число, унікальне в межах процесу і використовуване для цілей щодо доступу до файлу. Дескриптор є посиланням на відкритий файл.

Щоб дізнатися абсолютне маршрутне ім'я робочого каталогу, додаток може скористатися функцією *getcwd():*

#include <unistd.h>

char * getcwd (char * buf, size_t size);

Дана функція поміщає абсолютне маршрутне ім'я робочого каталогу масив buf довжини size, який і повертається як результат (при помилці результат дорівнює NULL).

Приклад 8.6. Створимо програму, що виводить на екран абсолютне маршрутне ім'я робочого (поточного) каталогу.

"main.c"

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int main(void)
{
size_t size;
char *buf;
char *apath;
// З'ясуємо, яким має бути розмір буфера
// для абсолютного маршрутного імені робочого каталогу
size = (size_t) pathconf(".", _PC_PATH_MAX);
if ((buf = (char *) malloc (size)) == NULL)
{
fprintf (stderr, "\n Не вдалося виділити буфер розміру %d\n", size);
free(buf);
return 1;
}
if ((apath = getcwd (buf, size)) == NULL)
{
fprintf (stderr, "\n Не вдалося визначити абсолютне маршрутне ім'я робочого каталогу\n");
```

```
free(buf);
return 1;
}
printf ("\n Абсолютне маршрутне ім'я робочого каталогу: %s\n", apath);
free(buf);
return 0;
}
```

Викликаємо програму.

```
delux@ubuntu :~/cwd/bin/Debug$ ./cwd
Абсолютне маршрутне ім'я робочого каталогу: /home/delux/cwd/bin/Debug
```

У розглянутому прикладі використовується функція *pathconf(),* що повертає поточне значення одного з конфігураційних параметрів цільової системи - в даному випадку довжину маршрутного імені:

```
#include <unistd.h>
long pathconf (const char * pathname, int name);
```

Тут pathname - об'єкт (файл, каталог та ін), для якого необхідно отримати значення деякого параметра, name – ім'я даного параметра.

Інформацію про файли отримують за допомогою функцій (системних викликів) сімейства stat():

```
#include <sys/stat.h>
int stat (const char *restrict path, struct stat *restrict buf);
int fstat (int fildes, struct stat *buf);
int lstat (const char *restrict path, struct stat *restrict buf);
```

Ключове слово restrict означає, що оголошений покажчик вказує на блок пам'яті, на який не вказує ніякий інший покажчик. Така специфікація розширює оптимізаційні можливості компілятора.

Функція *stat()* надає інформацію про названий файл: аргумент path вказує на маршрутне ім'я файлу. Щоб отримати ці відомості, достатньо мати право на пошук усіх компонентів маршрутного префікса.

Функція *fstat()* повідомляє дані про відкритий файл, який задається дескриптором файлу fildes.

Функція *lstat()* еквівалентна *stat()* за одним винятком: якщо аргумент path задає символьне посилання, *lstat()* повертає інформацію про неї, а *stat*() - про файл, на який це посилання вказує.

У разі нормального завершення результат функцій сімейства *stat()* дорівнює 0.

Аргумент buf є покажчиком на структуру типу stat, в яку міститься інформація про файл. У ній містяться принаймні такі поля:

struct stat

{

dev_t st_dev; // Ідентифікатор пристрою, що містить файл

ino_t st_ino; // Порядковий номер файлу у файловій системі

mode_t st_mode; // Режим файлу

nlink_t st_nlink; // Число жорстких посилань на файл

uid_t st_uid; // Ідентифікатор власника файлу

gid_t st_gid; // Ідентифікатор групи, що володіє.

off_t st_size;// Для звичайних файлів і символьних посилань – розмір у байтах

// Для інших типів файлів значення цього поля неспеціфіковано

time_t st_atime;// Час останнього доступу

time_t st_mtime;// Час останньої зміни файлу

time_t st_ctime;// Час останньої зміни статусу файлу

};

Комбінація значень (st_dev, st_ino) має однозначно визначати файл у межах об'єднаної (зокрема мережевої) файлової системи. Статус файлу змінюється, коли модифікуються його атрибути (наприклад, режим), а не вміст.

У файлі <sys/stat.h> визначено як структура stat, а й константи, корисні до роботи з різними характеристиками файла. Наприклад, константа S_IFMT - виділяє тип файлу:

1) **S_IFDIR** - каталог;

2) **S_IFBLK** - спеціальний файл блокового пристрою;

3) **S_IFCHR** - спеціальний файл символьного пристрою;

4) **S_IFIFO** - канал;

5) **S_IFLNK** - символічне посилання;

6) **S_IFREG** - звичайний файл;

7) **S_IFSOCK** - сокет;

Приклад 8.7. Створимо програму, що виводить на екран інформацію про файли-аргументи командного рядка.

"main.c"

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

// Функція повертає односимвольне позначення типу файлу
// Для невідомого типу повертається 'u'
static char my_filetype (const mode_t mode)
{
switch (mode & S_IFMT)
{
case S_IFDIR:
```

```
return ('d');
case S_IFBLK:
return ('b');
case S_IFCHR:
return ('c');
case S_IFLNK:
return ('l');
case S_IFIFO:
return (p);
case S_IFREG:
return ('f');
case S_IFSOCK:
return ('s');
default:
return ('u');
}
}
int main (int argc, char *argv[])
{
struct stat buf;
int i;
for (i = 1; i < argc; i++)
{
if (stat(argv[i], &buf))
{
fprintf (stderr, "\nstat: не вдалося отримати інформацію про файл %s\n", argv [i]);
return (-1);
}
printf ("\stat-інформація про файл %s:\n", argv [i]);
```

```
printf ("Тип: % c \ n", my_filetype (buf.st_mode));
printf ("Розмір: %ld\n", buf.st_size);
if (lstat (argv [i], &buf))
{
fprintf (stderr, "\nlstat: не вдалося отримати інформацію про файл %s\n", argv [i]);
return (-1);
}
printf ("\lstat-інформація про файл %s:\n", argv [i]);
printf ("Тип: % c \ n", my_filetype (buf.st_mode));
printf ("Розмір: %ld\n", buf.st_size);
}
return 0;
}
```

Викликаємо програму.

```
delux@ubuntu :~/fileinfo/bin/Debug$ ./fileinfo ~/link/1.txt
stat-інформація про файл /home/delux/link/1.txt:
Тип: f
Розмір: 7
lstat-інформація про файл /home/delux/link/1.txt:13
Тип: f
Розмір: 7
delux@ubuntu :~/fileinfo/bin/Debug$ ./fileinfo ~/link/link-1.txt
stat-інформація про файл /home/delux/link/link-1.txt:
Тип: f
Розмір: 7
lstat-інформація про файл /home/delux/link/link-1.txt:
Тип: l
```

Розмір: 22

Для отримання інтегральної інформації про файлові системи, що містять задані файли, існують функції *fstatvfs()* та *statvfs():*

#include <sys/statvfs.h>

int fstatvfs (int fildes, struct statvfs *buf);

int statvfs (const char *restrict path, struct statvfs *restrict buf);

Пара функцій *statvfs()* і *fstatvfs()* за інтерфейсом аналогічна функцій *stat()* і *fstat(),* тільки у вихідну структуру (типу fstatvfs) міститься інформація не про файли, а про файлові системи, що містять ці файли. У структурі statvfs повинні бути такі поля:

struct statvfs

{

unsigned long f_bsize;// Розмір блоку файлової системи

fsblkcnt_t f_blocks;// Загальна кількість блоків у файловій системі

fsblkcnt_t f_bfree;// Загальна кількість вільних блоків

fsblkcnt_t f_bavail;// Число вільних блоків, доступних непривілейованим процесам

fsfilcnt_t f_files;// Загальна кількість дескрипторів файлів

fsfilcnt_t f_ffree;// Загальна кількість вільних дескрипторів файлів

fsfilcnt_t f_favail;// Число дескрипторів файлів, доступних непривілейованим процесам

unsigned long f_fsid; // Ідентифікатор файлової системи

unsigned long f_flag;// Бітова шкала прапорів

unsigned long f_namemax;// Максимальна довжина імені файлу

};

Серед прапорів, що входять до шкали f_flag, можна виділити **ST_RDONLY** - ознака того, що файлова система змонтована тільки на читання.

Приклад 8.8. Створимо програму, що виводить на екран інформацію про файлові системи, що містять задані в командному рядку файли-аргументи.

"main.c"

```
#include <sys/statvfs.h>
#include <stdio.h>

int main (int argc, char *argv[])14
{
struct statvfs buf;
int i;
for (i = 1; i < argc; i++)
{
if (statvfs (argv [i], &buf))
{
fprintf (stderr, "\nstatvfs: не вдалося отримати інформацію про файлову систему, що містить файл %s\n", argv [i]);
return (-1);
}
printf ("statvfs-інформація про файлову систему, що містить файл %s:\n", argv [i]);
printf ("Розмір блоку файлової системи: %ld\n", buf.f_bsize);
printf ("Загальна кількість блоків у файловій системі: %ld\n", buf.f_blocks);
printf ("Загальна кількість вільних блоків: %ld\n", buf.f_bfree);
```

printf ("Кількість вільних блоків, доступних непривілейованим процесам: %ld\n", buf.f_bavail);

printf ("Загальна кількість описувачів файлів: %ld\n", buf.f_files);

printf ("Загальна кількість вільних описувачів файлів: %ld\n", buf.f_ffree);

printf ("Кількість описувачів файлів, доступних непривілейованим процесам: %ld\n", buf.f_favail);

printf ("Ідентифікатор файлової системи: %ld\n", buf.f_fsid);

printf ("Файлова система змонтована лише на читання: %ld\n", buf.f_flag & ST_RDONLY);

printf ("Максимальна довжина імені файлу: %ld\n", buf.f_namemax);

}

return 0;

}

Викликаємо програму (файлова система ext4).

delux@ubuntu :~/fsinfo/bin/Debug$ ./fsinfo ~/link/1.txt

statvfs-інформація про файлову систему, що містить файл /home/delux/link/1.txt:

Розмір блоку файлової системи: 4096

Загальна кількість блоків у файловій системі: 60077186

Загальна кількість вільних блоків: 55054137

Число вільних блоків, доступних непривілейованим процесам: 52002387

Загальна кількість описувачів файлів: 15261696

Загальна кількість вільних описувачів файлів: 15082992

Число описувачів файлів, доступних непривілейованим процесам: 15082992

Ідентифікатор файлової системи: -75930068

Файлова система змонтована лише для читання: 0

Максимальна довжина імені файлу: 25515

Файлове введення-виведення

При програмуванні під ОС Linux на мові С використовують дві основні групи функцій, що обслуговують операції введення/виведення:

1) системні функції (нижнього рівня), що використовують цілі файлові дескриптори (заголовні файли <fcntl.h>, <unistd.h>);

2) функції вищого рівня зі стандартної бібліотеки мови С, що здійснюють буферизоване введення/виведення із застосуванням потоків (заголовний файл <stdio.h>).

Застосування потоків у порівнянні з системними викликами дозволяє додаткам отримати додатковий сервіс у вигляді керованої буферизації і форматованого введення/виведення, однак за рахунок деякого зниження швидкості, оскільки функції стандартної бібліотеки зрештою викликають системні функції файлового введення/виведення.

Потік – це об'єкт, який служить доступу до файлів як до впорядкованої послідовності символів.

Потік є структурою типу FILE, з якою асоційовано відповідний дескриптор відкритого файлу. Декілька дескрипторів та/або потоків можуть посилатися на один відкритий файл.

І файлові дескриптори, і потоки формуються внаслідок виконання функцій відкриття файлів, які мають передувати операціям введення/виведення. Є, однак, три зумовлені потоки, що відкриваються ще перед початком роботи програми:

1) стандартне введення (**stdin**);

2) стандартне виведення (**stdout)**;

3) стандартний потік помилок (стандартний протокол) (**stderr**).

Для звернення до них використовуються покажчики на визначені об'єкти типу FILE з іменами відповідно **stdin, stdout і stderr.**

Аналогічно **в системних функціях** як файловий дескриптор можна використовувати зумовлені файлові дескриптори **STDIN_FILENO** (стандартне введення), **STDOUT_FILENO** (стандартне виведення) і **STDERR_FILENO** (стандартний потік помилок).

При відкритті файлів вказується вид наступних операцій введення/виведення: читання, запис, оновлення (читання та запис), додавання (запис у кінці). Вигляд операцій має бути узгоджений із правами доступу до файлу; інакше відкриття закінчиться невдачею.

Якщо файл підтримує запити на позиціонування (звичайний файл, на відміну від файлів послідовних символьних пристроїв-терміналів), то після відкриття індикатор поточної позиції встановлюється на початок (на нульовий байт) за умови, що файл не відкривали для додавання; у цьому випадку індикатор вказуватиме на кінець файлу. Надалі індикатор поточної позиції зміщується під впливом операцій читання, запису та позиціонування, щоб спростити послідовне просування у файлі.

Потоки можуть бути наступного типу:

1) повністю буферизованими;

2) буферизованими рядково;

3) небуферизованими.

У першому випадку передача байт з файлу/файл здійснюється переважно блоками, коли буфер виявляється заповненим. При рядковій буферизації передача даних також здійснюється блоками, після досягнення символу перекладу рядка або заповнення буфера. За відсутності буферизації байти передаються по можливості без затримок.

Після завершення роботи з файлом його потрібно закрити. При цьому не тільки розривається зв'язок між файлами з одного боку та дескрипторами і потоками з іншого, але й забезпечується передача даних, що залишалися буферизованими.

Функції створення та відкриття файлів

Системна функція

#include <fcntl.h>

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

Функція ***creat()*** має два аргументи: маршрутне ім'я створюваного файлу **path** і встановлюваний режим доступу **mode** (ідентифікатори власника та групи, що володіє, успадковуються у поточного користувача).

Режим доступу задається за допомогою визначених прапорів (заголовний файл <sys/stat.h>), об'єднаних за допомогою логічної операції АБО (|):

1) **S_IRUSR** – доступно власнику (файлу) для читання (user-read);

2) **S_IWUSR** – дост пно власнику для запису (user-write);

3) **S_IXUSR** – доступно власнику для виконання (user-execute);

4) **S_IRGRP** – доступно групі (власника файлу) для читання (group-read);

5) **S_IWGRP** – доступно групі для запису (group-write);

6) **S_IXGRP** – доступно групі для виконання (group-execute);

7) **S_IROTH** – доступно іншим для читання (other-read);

8) **S_IWOTH** – доступно іншим для запису (other-write);

9) **S_IXOTH** – доступно іншим для виконання (other-execute).

Результатом функції служить файловий дескриптор (який представлено невід'ємним цілим числом), тобто функція *creat()* не тільки створює файл, але й відкриває його. Якщо файл, який намагаються створити за допомогою *creat(),* вже існує, він очищається (розмір стає 0), а режим доступу і власник не змінюються.

У разі невдачі результат *creat()* дорівнює -1, а зовнішньої системної змінної errno надається код помилки, що дозволяє визначити причину її (помилки) виникнення. Змінна **errno**, а також ідентифікатори для кодів помилок визначені в заголовку <errno.h>. Для формування системного повідомлення про помилку можна скористатися функцією perror():

#include <stdio.h>

void perror (const char * msg);

Ця функція виводить рядок повідомлення **msg**, двокрапка, пробіл і текст повідомлення про помилку, що відповідає змінній errno.

Системна функція

#include <fcntl.h>

int open (const char * path, int oflag, …/*, mode_t mode]*/);

Функція ***open()*** відкриває файл із заданим маршрутним ім'ям (перший аргумент **path**). Файловий дескриптор, що повертається в якості результату, є мінімальним з числа тих, що не використовуються в даний момент поточним процесом (при невдачі повертається -1). Другий аргумент, **oflag**, встановлює прапори статусу файлу та визначає допустимі види операцій введення/виведення. Його значення формується як побітове АБО перелічених нижче прапорів.

З перших трьох прапорів має бути заданий рівно один:

1) **O_RDONLY** – відкрити файл тільки для читання.

2) **O_WRONLY** – відкрити файл тільки на запис.

3**) O_RDWR** – відкрити файл на читання та запис.

Наступні прапори можуть комбінуватися довільним чином:

1) **O_APPEND** – перед кожним записом встановлювати індикатор поточної позиції кінець файлу.

2) **O_CREAT** – якщо файл існує, даний прапор береться до уваги лише за наявності описуваного далі прапора O_EXCL. Якщо немає файлу, він створюється від імені поточного користувача.

3) **O_EXCL** – якщо встановлені прапори O_CREAT та O_EXCL, а файл існує, виклик *open()* завершиться невдачею. Перевірка існування файлу та його створення являє собою атомарну дію стосовно спроб інших процесів виконати аналогічний запит.

4) **O_TRUNC** якщо файл існує, його розмір встановлюється 0.

Третій аргумент **mode** є необов'язковим (відповідно до стандарту ISO мови С позначається багатокрапкою «...») і задає режим доступу до файлу аналогічно функції **creat()**. Використовується лише при створенні нового файлу, проте передавати його у параметрі можна у будь-якому випадку.

Розглянута раніше функція *creat (path, mode*) за визначенням еквівалентна виклику:

open (path, O_WRONLY | O_CREAT | O_TRUNC, mode).

Функції закриття файлів

Системна функція

#include <unistd.h>

int close (int fildes);

Функція стандартної бібліотеки введення/виведення мови С

#include <stdio.h>

int fclose (FILE *stream);

Функція ***close()*** звільняє файловий дескриптор fildes, який стає доступним для подальшого використання під час відкриття файлів. Коли останній дескриптор, що посилається на відкритий файл, закривається, файл закривається. Функція close() повертає 0 у разі успішного завершення та -1 при невдачі.

Функції читання файлів

Системні функції

#include <unistd.h>

ssize_t read (int fd, void * buf, size_t nbyte);

ssize_t readlink (const char restrict link_name, char restrict buf, size_t buf_size);

Функція ***read()*** намагається прочитати **nbyte** байт із файлу, асоційованого з дескриптором файлу **fd**, і помістити їх у буфер **buf**. Для файлів, що допускають позиціонування, read() виконує читання, починаючи зі значення індикатора поточної позиції, асоційованого з дескриптором fd. Після завершення операції цей індикатор збільшується на кількість прочитаних байт. Для пристроїв, які не підтримують розташування (наприклад, термінал), значення згаданого індикатора не визначено, а читання виконується з поточної позиції пристрою.

При успішному завершенні ***read()*** повертає кількість байт, реально прочитаних та поміщених у буфер; це значення може виявитися меншим за значення аргументу nbyte, якщо до кінця файлу залишалося менше, ніж nbyte байт. Наприклад, якщо поточна позиція збігалася з кінцем файлу, результат дорівнюватиме 0. У разі помилки повертається -1.

Функція **readlink** читає вміст символьних посилань. Вона містить вміст посилання з ім'ям **link_name** в буфер **buf** довжини **buf_size** (якщо буфер малий, залишок вмісту відкидається). Результат дорівнює кількості поміщених у буфер байт або -1 у разі невдачі.

Приклад 8.9. Реалізуємо програму, яка копіює дані зі стандартного введення на стандартне виведення з використанням системних функцій введення-виведення.

"main.c"

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

int main(void)
{
int n;
```

```
size_t nbyte = 4096;
char buf [nbyte];
while((n = read (STDIN_FILENO, buf, nbyte)) > 0)
if (write (STDOUT_FILENO, buf, n)! = n)
perror("помилка запису");
if (n < 0)
perror ("помилка читання");
return 0;
}
```

Викликаємо програму.

```
delux@ubuntu :~/QtSDK/iofile$ ./iofile
1234
1234
235
235
delux@ubuntu :~/QtSDK/iofile$
```

Викличемо програму і перенаправимо стандартне введення у файл main.cpp.

```
delux@ubuntu :~/QtSDK/iofile$ ./iofile < main.cpp
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

int main(void)
{
int n;
size_t nbyte = 4096;
char buf [nbyte];
```

```
while((n = read (STDIN_FILENO, buf, nbyte)) > 0)
if (write (STDOUT_FILENO, buf, n)! = n)19
perror ("помилка запису");
if (n < 0)
perror ("помилка читання");

return 0;
}
```

Викличемо програму та перенаправимо стандартне виведення у файл 1.txt.

```
delux@ubuntu :~/QtSDK/iofile$ ./iofile >1.txt
123
456
789
delux@ubuntu :~/QtSDK/iofile$ cat 1.txt
123
456
789
```

Викличемо програму і перенаправимо стандартне введення у файл 1.txt, а стандартне виведення у файл 2.txt.

```
delux@ubuntu :~/QtSDK/iofile$ ./iofile < 1.txt > 2.txt
delux@ubuntu :~/QtSDK/iofile$ cat 2.txt
123
456
789
```

Приклад 8.10. Реалізуємо програму, яка копіює дані з файлу, заданого як аргумент командного рядка, на стандартний висновок з використанням системних функцій введення-виведення.

"main.c"

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

int main(int argc, char **argv)
{
int n;
size_t nbyte = 4096;
char buf [nbyte];
int fd;
if ((fd = open (argv[1], O_RDONLY)) == -1)
{
perror ("помилка відкриття файлу");
exit(errno);
}
while((n = read (fd, buf, nbyte)) > 0)
if (write (STDOUT_FILENO, buf, n)! = n)20

perror ("помилка запису");
if (n < 0)
perror ("помилка читання");
if ((close (fd)) == -1)
```

```
{
perror ("помилка закриття файлу");
exit(errno);
}
return 0;
}
```

Викликаємо програму.

```
delux@ubuntu :~/QtSDK/iofile$ ./iofile 1.txt
123
456
789
```

Викликаємо програму.

```
delux@ubuntu :~/QtSDK/iofile$ ./iofile
помилка відкриття файлу: Bad address
```

Викликаємо програму.

```
delux@ubuntu :~/QtSDK/iofile$ ./iofile 3.txt
помилка відкриття файлу: No such file or directory
```

Приклад 8.11. Реалізуємо програму, що копіює дані зі стандартного введення файлу, заданий як аргумент командного рядка, з використанням системних функцій введення-виведення.

"main.c"

```
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
```

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

int main(int argc, char **argv)
{
int n;
size_t nbyte = 4096;
charbuf [nbyte];
int fd;
if ((fd = open (argv[1], O_WRONLY | O_CREAT | O_TRUNC)) == -1)
{
perror ("помилка відкриття файлу");
exit(errno);
}
while((n = read (STDIN_FILENO, buf, nbyte)) > 0)
if (write (fd, buf, n)! = n)
perror ("помилка запису");
if (n < 0)
perror ("помилка читання");
if ((close (fd)) == -1)
{
perror ("помилка закриття файлу");
exit(errno);
}
return 0;
}
```

Викликаємо програму.

```
delux@ubuntu :~/QtSDK/iofile$ ./iofile 1.txt
1 2 3 4 5 6 7
```

delux@ubuntu :~/QtSDK/iofile$ cat 1.txt

1 2 3 4 5 6 7

Функції позиціонування файлів

Системна функція

#include <unistd.h>

off_t lseek (int fildes, off_t offset, int whence);

Функція **lseek()** встановлює індикатор поточної позиції в такий спосіб. Спочатку, залежно від значення третього аргументу, **whence**, вибирається точка відліку:

1) 0 -якщо це значення дорівнює **SEEK_SET**;

2) поточна позиція для **SEEK_CUR**;

3) розмір файлу для **SEEK_END**.

Потім до точки відліку додається зміщення **offset** (другий аргумент, може мати як позитивне, так і від'ємне значення). Індикатор поточної позиції можна перемістити за кінець файлу.

Результатом функції *lseek()* є нове значення індикатора поточної позиції, відраховане в байтах від початку файлу. У разі помилки повертається (off_t) (-1), а поточна позиція залишається незмінною.

Приклади.

lseek(fildes, (off_t) 0, SEEK_SET); //встановлення індикатора поточної позиції на початок файлу

lseek(fildes, (off_t) 0, SEEK_END);// встановлення індикатора поточної позиції в кінець файлу

lseek(fildes, inc, SEEK_CUR);// Збільшення індикатора поточної позиції на inc байт

Приклад 8.12. Реалізуємо програму, що ділить вихідний файл 1.txt на 2 приблизно рівні частини 1_1.txt та 1_2.txt з використанням системних функцій.

"main.c"

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

int main()
{
int n;
int ifd, ofd1, ofd2;
int fsize;
if ((ifd = open ("1.txt", O_RDONLY)) == -1)
{
perror ("помилка відкриття файлу 1.txt");
exit(errno);
}
if ((ofd1 = Open ("1_1.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR)) == -1)
{
perror ("помилка відкриття файлу 1_1.txt");
exit(errno);
}
if ((ofd2 = open ("1_2.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR)) == -1)
```

```
{
perror ("помилка відкриття файлу 1_2.txt");
exit(errno);
}
fsize = lseek (ifd, (off_t) 0, SEEK_END); // розмір вихідного файлу
size_t nbyte = fsize/2; // Розмір першого нового файлу (першої частини)
char buf [nbyte +1];
size_t nbyte2 = fsize -nbyte + 1; // Розмір другого нового файлу (другої частини)
// зчитуємо першу половину вихідного файлу та записуємо перший новий файл
lseek(ifd, (off_t) 0, SEEK_SET);
n = read (ifd, buf, nbyte);
if (n < 0)
perror ("помилка читання");
else if (write (ofd1, buf, n)! = n)
perror ("помилка запису");
// зчитуємо другу половину вихідного файлу та записуємо другий новий файл
n = read (ifd, buf, nbyte2); //
if (n < 0)
perror ("помилка читання");
else if (write (ofd2, buf, n)! = n)
perror ("помилка запису");
if ((close (ifd)) == -1 || (close (ofd1)) == -1 || (close (ofd2)) == -1)
{
perror ("помилка закриття файлу");
exit(errno);
}
```

```
return 0;
}
```

Викликаємо програму.

```
delux@ubuntu :~/QtSDK/iofile$ ./iofile
delux@ubuntu :~/QtSDK/iofile$ cat 1.txt
1 2 3
4 5 6
delux@ubuntu :~/QtSDK/iofile$ cat 1_1.txt
1 2 3
delux@ubuntu :~/QtSDK/iofile$ cat 1_2.txt
4 5 6
```

Завдання лабораторної роботи 8

Необхідно розробити консольну програму, яка дозволяє:

1) розбити вихідний файл на 4 файли, імена яких задаються користувачем;

2) об'єднати 4 файли, задані користувачем, в один результуючий файл.

Інтерфейс взаємодії користувача з програмою — на вибір розробника (наприклад, усі дані вводяться в командному рядку як аргументи програми).

Лабораторна робота 9

Зміна атрибутів файлів, перейменування (переміщення) файлів за допомогою системних викликів Linux (Unix)

Теоретичні відомості

Зміна власника та групи файлу

Для зміни власника та групи файлу існує утиліта:

1) chown [ключи] новий_власник [:нова_група] ім'я_файла_або_каталогу

та функції:

2) #include <unistd.h>
int chown (const char * path, uid_t owner, gid_t group);
int fchown (int fildes, uid_t owner, gid_t group);

При зверненні до утиліти **chown** власник і група задаються **іменами** або **числовими ідентифікаторами**. На початку проводиться пошук заданих аргументів як імен у базах даних користувачів та груп і далі витягуються звідти відповідні числові ідентифікатори; якщо пошук виявиться невдалим, аргументи розглядаються як ідентифікатори. Змінити власника може лише нинішній власник файлу або користувач із правами адміністратора. Ключі дозволяють по-різному трактувати символьні посилання та змінювати атрибути вкладених каталогів.

Функція ***chown()*** змінює власника файлу owner і/або групу власника файлу **group** заданих числовими ідентифікаторами для файлу, заданого аргументом path (ім'я файлу). Функція ***fchown()*** аналогічна *chown()*, тільки використовується для відкритих файлів, що задаються файловим дескриптором **fildes**.

Якщо за використанні функцій *chown()* і *fchown()* змінюється лише власник, то аргумент group задається рівним (gid_t) (-1); при зміні лише групи ідентифікатор власника слід задати як (uid_t) (-1).

Обидві функції повертають 0 у разі успіху та -1 у разі помилки.

Приклад 9.1. Змінимо власника файлу 1.txt за допомогою утиліти **chown**.

delux@ubuntu :~$ ls -l 1.txt

-rw-r--r--1 delux delux 10 2011-11-18 08:23 1.txt

delux@ubuntu :~$ sudo chown delux2 1.txt *

[sudo] password for delux:

delux@ubuntu :~$ ls -l 1.txt

-rw-r--r--1 delux2 delux 10 2011-11-18 08:23 1.txt

Приклад 9.2. Розробимо консольну програму, що дозволяє змінити власника файлу.

"main.c"

```
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <pwd.h>

// функція визначення імені користувача за його ідентифікатором
char* get_name(uid_t uid)
{
struct passwd*pswd;
```

```
// отримаємо дані про користувача щодо його ідентифікатора
if ((pswd = getpwuid(uid)) == NULL)
{
perror ("помилка визначення імені власника файлу");
exit(errno);
}
return pswd->pw_name;
}
int main(int argc, char *argv[])
{
const int len = 256;
char fname[len];
char owner [len];
struct stat st;
struct passwd*pswd;

printf("Введіть ім'я файлу:");
scanf("%s", fname);
// отримаємо інформацію про файл
if ((stat(fname, &st))! = 0)
{
perror ("помилка отримання інформації про файл");
exit(errno);
}
// визначимо ім'я власника файлу за його ідентифікатором
printf("Власник: %s\n", get_name(st.st_uid));
// Змінимо ім'я власника файлу
printf("Введіть ім'я нового власника файлу: ");
scanf("%s", owner);
if ((pswd = getpwnam(owner)) == NULL)
```

```
{
perror ("помилка отримання даних про користувача");
exit(errno);
}
if ((chown(fname, pswd->pw_uid, gid_t(-1))) != 0)
{
perror ("помилка зміни власника файлу");
exit(errno);
}
// отримаємо інформацію про файл
if ((stat(fname, &st))! = 0)
{
perror ("помилка отримання інформації про файл");
exit(errno);
}

// визначимо ім'я власника файлу за його ідентифікатором
printf("Новий власник: %s\n", get_name(st.st_uid));
return 0;
}
```

Викликаємо програму (з правами адміністратора).

```
delux@ubuntu :~$ sudo ./fowner
Введіть назву файлу: 1.txt
Власник: delux2
Введіть ім'я нового власника: delux
Новий власник: delux
```

Зміна режиму (прав) доступу до файлу

Змінити режим доступу до файлу можна за допомогою утиліти chmod (була розглянута раніше) та наведених нижче функцій:

#include <sys/stat.h>

int chmod (const char * path, mode_t mode);

int fchmod (int fildes, mode_t mode);

Функція ***chmod()*** визначає новий режим доступу (аргумент mode) до файлу path (ім'я файлу). Функція ***fchmod*()** аналогічна ***chmod()***, тільки використовується для відкритих файлів, що задаються файловим дескриптором **fildes**.

Режим доступу **mode** задається за допомогою визначених прапорів (заголовний файл <sys/stat.h>), об'єднаних за допомогою логічної операції АБО (|):

1) **S_IRUSR** – доступно власнику (файлу) для читання (user-read);

2) **S_IWUSR** – доступно власнику для запису (user-write);

3) **S_IXUSR** – доступно власнику для виконання (user-execute);

4) **S_IRGRP** – доступно групі (власнику файлу) для читання (group-read);

5) **S_IWGRP** – доступно групі для запису (group-write);

6) **S_IXGRP** – доступно групі для виконання (group-execute);

7) **S_IROTH** – доступно іншим для читання (other-read);

8) **S_IWOTH** – доступно іншим для запису (other-write);

9) **S_IXOTH** – доступно іншим для виконання (other-execute).

Обидві функції повертають 0 у разі успіху та -1 у разі помилки.

Приклад 9.3. Розробимо консольну програму, що дозволяє змінити режим доступу до файлу для інших користувачів. Ім'я файлу має задаватися в командному рядку як аргумент програми. Програма повинна послідовно

запитувати користувача щодо підтвердження установки кожного біта прав доступу.

"main.c"

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>

// функція обробляє рядок, введений користувачем,
// якщо введено рядок "так", то до поточного режиму
// додається відповідний біт прав доступу
void add_mode(mode_t * mode, const char * answer, mode_t bit)
{
if (strcmp(answer, "так") == 0)
{
*mode |= bit;
printf ("встановлено\n");
}
else
printf ("скинуто n");
}
I nt main (int argc, char *argv[])
{
const int len = 256;
char answer[len];
mode_t mode = 0000;
```

```
struct stat st;
if (argc < 2)
{
fprintf (stderr, "Програма має один аргумент s\n");
return 1;
}
printf ("доступно іншим для читання:");
scanf ("%s", answer);
add_mode (&mode, answer, S_IROTH);
printf ("доступно іншим для запису:");
scanf ("%s", answer);
add_mode (&mode, answer, S_IWOTH);
printf ("доступно іншим для виконання:");
scanf ("%s", answer);
add_mode (&mode, answer, S_IXOTH);
// отримаємо інформацію про файл
if ((stat(argv[1], &st)) != 0)
{
perror ("помилка отримання інформації про файл");
exit(errno);
}
// скидаємо всі біти режиму доступу для інших користувачів
st.st_mode | = S_IROTH | S_IWOTH | S_IXOTH;
st.st_mode ^= S_IROTH | S_IWOTH | S_IXOTH;
// встановлюємо задані користувачем біти режиму доступу
if (chmod (argv [1], st.st_mode | mode) == -1)
{
perror ("chmod() error");
exit(errno);
}
```

return 0;

}

Викликаємо програму (з правами адміністратора).

delux@ubuntu :~$ ls -l 1.txt

-rw-r--r--1 delux delux 10 2011-11-18 10:02 1.txt

delux@ubuntu :~$ sudo ./fmode 1.txt

доступно іншим для читання: так

встановлено

доступно іншим для запису: так

встановлено

доступно іншим для виконання: ні

скинуто

delux@ubuntu :~$ ls -l 1.txt

-rw-r --rw-1 delux delux 10 2011-11-18 10:02 1.txt

Зміна календарного часу останнього доступу та останньої модифікації файлу

Під календарним часом розуміють час та дату.

Для зміни часу останнього доступу та останньої модифікації файлу існує утиліта (яка без ключів просто створює новий файл):

touch [-a | -m] [-r еталонний_файл | -t час] [-c] файл

Ключі:

-a – наказує модифікувати час останнього доступу;

-m – наказує модифікувати час останньої зміни;

-r – час успадковується у еталонного_файлу;

-t – час задається явно у форматі [[CC]YY]MMDDhhmm[.SS], де пари символів вказують, відповідно, на дві старші цифри року, дві молодші цифри

року, місяць у році, номер дня в місяці, годину в дні, хвилину в годині та секунду в хвилині. Як бачимо, рік вказувати не обов'язково.

Правила формування старших цифр року, якщо їх опущено, такі. Коли молодші цифри лежать у діапазоні від 69 до 99, мається на увазі 19; в іншому випадку - 20.

Приклад 9.4. Змінимо за допомогою утиліти touch час останньої модифікації файлу 1.txt.

delux@ubuntu :~$ ls -l 1.txt

-rw-r --rw-1 delux delux 10 2011-11-18 10:02 1.txt6

delux@ubuntu :~$ touch -mt 11181012 1.txt

delux@ubuntu :~$ ls -l 1.txt

-rw-r --rw-1 delux delux 10 2011-11-18 10:12 1.txt

Змінити час останнього доступу та останньої модифікації файлу можна також за допомогою функції:

```
#include <utime.h>
int utime(const char *pathname, const struct utimbuf *times);
```

Функція повертає 0 у разі успіху та -1 у разі помилки, і використовує наступну структуру:

```
#include <utime.h>
struct utimbuf
{
time_t actime; // час останнього доступу
time_t modtime; // час останньої зміни
};
```

де **time_t** – тип даних, призначений для створення змінних, що зберігають календарний час (дату і час), виражене кількістю секунд, що

пройшли з початку Епохи (часу і дати створення ОС UNIX): 00:00:00 1 січня 1970 по Гринвічу (UTC, GMT).

Дія цієї функції та привілеї, необхідні для її виконання, залежать від того, чи є аргумент **times** порожнім покажчиком:

а) Якщо в аргументі **times** передається порожній покажчик (NULL), час останнього доступу до файлу та час останньої зміни файлу встановлюються рівними поточному часу. Для цього процес повинен мати право запису у файл або мати ідентифікатор користувача власника файлу.

б) Якщо в аргументі **times** передається не порожній покажчик, час останнього доступу до файлу і час останньої зміни файлу беруться зі структури, на яку вказує аргумент **times**. Для цього процес повинен мати привілеї суперкористувача або мати ідентифікатор користувача власника файлу.

Слід зазначити, що час останнього доступу та останньої модифікації файлу зберігаються у структурі типу **stat**:

```
struct stat
{
...
time_t st_atime;// Час останнього доступу
time_t st_mtime;// Час останньої зміни файлу
};
```

значення якої для конкретного файлу можна встановити за допомогою функції stat() (див. практику 9-10). У деяких реалізаціях UNIX параметри даної структури st_atime і st_mtime можуть мати тип **timespec**, тобто:

```
struct stat
{
...
timespec st_atime;// Час останнього доступу7
timespec st_mtime;// Час останньої зміни файлу
```

};

Тип часу в свою чергу також є структурою, що містить календарний час в секундах і наносекундах:

struct timespec

{

...

time_t tv_sec;// секунди

long st_mtime;// наносекунди

};

Тому, наприклад, для визначення часу останнього доступу з точністю до секунд може знадобитися виконати таку дію:

time_t atime = st->st_atim.tv_sec; // st-покажчик на структуру типу stat

замість такого:

time_t atime = st->st_atim;

Для представлення часу в програмах часто також використовується структура tm (#include<time.h>), що містить час і дату у розділеній на компоненти формі:

struct tm

{

int tm_sec; // Секунди від початку хвилини (0,59)

int tm_min; // хвилини від початку години (0,59)

int tm_hour; // години від півночі (0,23)

int tm_mday; // Число місяця (1,31)

int tm_mon; // місяці після січня (0,11)

int tm_year; // роки з 1900

int tm_wday; // Дні з неділі (0,6)

int tm_yday; // Дні з першого січня (0,365)

...

};

Для перетворення даних типу **time_t** в тип **struct tm** і назад, а також для отримання поточного часу та перетворення його на наочний строковий формат можна скористатися такими функціями:

1) **#include <time.h>**

time_t time(time_t *time);

Функція time() повертає поточний календарний час системи. Якщо в системі відлік часу не проводиться, повертається значення -1.

Функцію ***time***() можна викликати або з нульовим покажчиком (NULL), або з покажчиком змінної типу time_t. В останньому випадку цій змінній буде надано календарний час.

2) **#include <time.h>**

char * ctime (const time_t * time);

Повертає покажчик на рядок, наступного виду: **День місяць рік години:хвилини:секунди year\n\0**

Функції передається вказівник на календарний час. Буфер, що використовується ***ctime()*** для зберігання форматованого рядка виведення є «службовим» статично розподіленим масивом символів. Він перезаписується під час кожного виклику функції. Для збереження рядка необхідно скопіювати її в якусь іншу область пам'яті.

3) **#include <time.h>**

struct tm * localtime (const time_t * time);

Повертає покажчик на структуру типу tm, що містить час у розділеній на компоненти формі. Час представлений як місцевий. Вказівник time зазвичай одержують за допомогою функції *time()*. Пам'ять для структури, у якій localtime() зберігає розділений на компоненти час, виділяється статично. Тому ця структура перезаписується під час кожного виклику функції. Для

збереження змісту структури необхідно скопіювати її в якусь іншу область пам'яті.

Повертає NULL у разі помилки.

4) **#include <time.h>**

time_t mktime(struct tm *tp);

Перетворює місцевий час, заданий структурою *tp, під час, заданий типом time_t. Функція повертає календарний час або -1 у разі помилки.

Приклад 9.5. Розробимо консольну програму, яка відображатиме час останнього доступу та останньої модифікації файлу, а також дозволяє змінити ці параметри.

"main.c"

```
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <utime.h>

// функція виводить час останнього доступу та модифікації файлу
void print_time(struct stat *st)
{
const int len = 256;
char * stime = (char *) calloc (len, sizeof (char)); // рядок містить час
time_t atime = st->st_atim.tv_sec; // час останнього доступу
time_t mtime = st->st_mtim.tv_sec; // час модифікації
if ((stime = ctime(&atime)) == NULL)
```

```
{
perror ("помилка визначення часу останнього доступу");
exit(errno);
}
printf("Час останнього доступу: %s", stime);
if ((stime = ctime(&mtime)) == NULL)
{
perror ("помилка визначення часу модифікації");9

exit(errno);
}
printf("Час модифікації: %s", stime);
}
void get_time(int *year, int *mon, int *mday, int *hour, int *min, int
*sec)
{
printf("Введіть рік після 2000 р. (гг): ");
scanf("%d", year);
*year += 100;
printf("Введіть місяць: ");
scanf("%d", mon);
printf("Введіть день місяця:");
scanf("%d", mday);
printf("Введіть години: ");
scanf("%d", hour);
printf("Введіть хвилини:");
scanf("%d", min);
printf("Введіть секунди: ");
scanf("%d", sec);
}
```

```
int main(int argc, char *argv[])
{
const int len = 256;
char fname[len];
struct stat st;
struct tm t;
time_t time;
struct utimbuf tbuf;
printf("Введіть ім'я файлу:");
scanf("%s", fname);
// отримаємо інформацію про файл
if ((stat(fname, &st))! = 0)
{
perror("помилка отримання інформації про файл");
exit(errno);
}
print_time(&st);
// Змінимо час останнього доступу та модифікації файлу
printf("--Час останнього доступу--\n");
get_time(&t.tm_year, &t.tm_mon, &t.tm_mday, &t.tm_hour,
&t.tm_min, &t.tm_sec);
// Перетворимо час на формат time_t
tbuf.actime = mktime(&t);
printf("--Час модифікації--\n");
get_time(&t.tm_year, &t.tm_mon, &t.tm_mday, &t.tm_hour,
&t.tm_min, &t.tm_sec);
// Перетворимо час на формат time_t
tbuf.modtime = mktime(&t);
if ((utime(fname, &tbuf)) != 0)
{
```

```
perror ("помилка модифікації тимчасових характеристик файлу");
exit(errno);
}
// отримаємо інформацію про файл
if ((stat(fname, &st))! = 0)
{
perror("помилка отримання інформації про файл");
exit(errno);
}
print_time(&st);
return 0;
}
```

Викликаємо програму.

```
delux@ubuntu :~$ ./ftime
Введіть назву файлу: 1.txt
Час останнього доступу: Fri Nov 18 10:16:44 2011
Час модифікації: Fri Nov 18 10:16:43 2011
--Час останнього доступу--
Введіть рік після 2000р. (yy): 10
Введіть місяць: 0
Введіть день місяця: 1
Введіть години: 2
Введіть хвилини: 3
Введіть секунди: 4
--Час модифікації--
Введіть рік після 2000р. (yy): 11
Введіть місяць: 1
Введіть день місяця: 2
Введіть години: 3
```

Введіть хвилини: 4

Введіть секунди: 5

Час останнього доступу: Fri Jan 1 02:03:04 2010

Час модифікації: Wed Feb 2 03:04:05 2011

Видалення файлів

Видалити файли можна за допомогою таких функцій:

```
#include <unistd.h>
int unlink (const char * path);
#include <stdio.h>
int remove (const char * path);
```

Ім'я файлу, що видаляється, передається в параметрі path. У разі успіху функції повертають 0, якщо помилки повертають -1. Строго кажучи, ці функції видаляють не файли, а зменшують на одиницю кількість жорстких посилань на ці файли. Файл видаляється і зайнятий ним простір звільняється, тільки якщо інших посилань не залишається.

Приклад 9.6. Розробимо консольну програму, що видаляє файл, ім'я якого задається як аргумент програми

"main.c"

```
#include <stdio.h>
#include <unistd.h>
int main (int argc, char** argv)
{
if (argc < 2) {
fprintf (stderr, "Функція викликається з одним аргументом\n");
return 1;
```

```
}
if (unlink (argv[1]) == -1) {
fprintf (stderr, "Неможливо видалити файл %s\n", argv[1]);
return 1;
}
return 0;
}
```

Викликаємо програму.

```
delux@ubuntu :~$ ls -l 1.txt
-rw-------1 delux delux 10 2011-11-18 11:45 1.txt
delux@ubuntu :~$ ./rmfile 1.txt
delux@ubuntu :~$ ls -l 1.txt
ls: неможливо отримати доступ до 1.txt: Немає такого файлу чи каталогу
```

Перейменування/переміщення файлів

Перейменувати/перемістити файли (і каталоги) можна за допомогою функції:

```
#include <stdio.h>
int rename (const char *old_path, const char *new_path);
```

Параметр **old_path** визначає старе ім'я файлу, параметр **new_path** нове ім'я файлу. Якщо старе і нове імена файлу належать одному каталогу, файл перейменовується, інакше – переміщається. У разі успіху функція повертає 0, якщо помилки повертає -1.

Хоча функція оголошена в стандартному файлі stdio.h, Linux є системним викликом.

Приклад 9.7. Розробимо консольну програму, що перейменовує файл, старе та нове ім'я якого задається як аргументи програми.

"main.c"

```
#include <stdio.h>
int main (int argc, char** argv)
{
if (argc < 3) {
fprintf (stderr, "Функція викликається із двома аргументами\n");12
return 1;
}
if (rename (argv[1], argv[2]) == -1) {
fprintf (stderr, "Неможливо перейменувати файл %s на %s\n", argv[1], argv[2]);
return 1;
}
return 0;
}
```

Викликаємо програму.

```
delux@ubuntu :~$ ls -l 1.txt
-rw-r--r--1 delux delux 10 2011-11-21 05:55 1.txt
delux@ubuntu :~$ ./rnmfile 1.txt 2.txt
delux@ubuntu :~$ ls -l 1.txt
ls: неможливо отримати доступ до 1.txt: Немає такого файлу чи каталогу
delux@ubuntu :~$ ls -l 2.txt
-rw-r--r--1 delux delux 10 2011-11-21 05:55 2.txt
```

Створення, видалення, зміна робочого каталогу. Отримання списку файлів каталогу

Створити каталог можна за допомогою функції:

```
#include <sys/stat.h>
int mkdir (const char * path, mode_t mode);
```

Параметр **path** визначає ім'я каталогу, параметр **mode** – режим доступу до каталогу. Слід зазначити, що декларація про «виконання», у разі каталогу означає можливість переглядати його вміст (виконувати у ньому пошук даних). У разі успіху функція повертає 0, у разі помилки повертає -1.

Приклад 9.8. Розробимо консольну програму, що створює каталог (ім'я каталогу задається як аргумент програми) з правами доступу на читання, запис та виконання каталогу тільки для його власника.

"main.c"

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

int main (int argc, char** argv)
{
mode_t mode = S_IRUSR | S_IWUSR | S_IXUSR;
if (argc < 2) {
fprintf (stderr, "Функція викликається одним аргументом\n");
return 1;
}
if (mkdir (argv[1], mode) == -1) {13
fprintf (stderr, "Неможливо створити каталог %s\n", argv[1]);
return 0;
}
return 0;
}
```

Викликаємо програму.

```
delux@ubuntu :~$ ./mkdir newdir
delux@ubuntu :~$ ls -l | grep newdir
drwx------2 delux delux 4096 2011-11-18 12:23 newdir
```

Видалення каталогу

Видалити порожній каталог можна за допомогою функції:

```
#include <unistd.h>
int rmdir (const char * path);
```

Параметр **path** визначає ім'я каталогу. У разі успіху функція повертає 0, якщо помилки повертає -1.

Приклад 9.9. Розробимо консольну програму, що видаляє каталог, ім'я якого задається як аргумент програми.

"main.c"

```
#include <stdio.h>
#include <unistd.h>
int main (int argc, char** argv)
{
if (argc < 2) {
fprintf (stderr, "Функція викликається одним аргументом\n");
return 1;
}
if (rmdir (argv[1]) == -1) {
fprintf (stderr, "Неможливо видалити каталог %s\n", argv[1]);
return 1;
}
return 0;
}
```

Викликаємо програму.

delux@ubuntu :~$ ./rmdir newdir

delux@ubuntu :~$ ls -l | grep newdir

Зміна робочого (поточного) каталогу процесу

Змінити робочий каталог процесу можна за допомогою функції:

```
#include <unistd.h>
int chdir (const char * path);
```

Параметр **path** задає ім'я нового робочого каталогу. У разі успіху функція повертає 0, якщо помилки повертає -1.

Приклад 9.10. Розробимо наступну консольну програму. На початку програма виводить на екран значення робочого каталогу, змінює його каталог /etc, і далі виводить значення нового робочого каталогу.

"main.c"

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
void print_cwd()
{
const int len = 256;
char * cwd = getcwd (NULL, len);
if (cwd == NULL)
{
fprintf (stderr, "Неможливо визначити робочий каталог\n");
exit(1);
```

```
}
printf ("Робочий каталог: %s\n", cwd);
free (cwd);
}
int main (void)
{
print_cwd();
if (chdir ("/etc") == -1)
{
fprintf (stderr, "Неможливо змінити робочий каталог\n");
return 1;
}
print_cwd();
return 0;
}
```

Викликаємо програму.

```
delux@ubuntu :~$ ./chdir
Робочий каталог: /home/delux
Робочий каталог: / etc
delux@ubuntu :~$
```

Відкриття каталогу для читання його вмісту

Обробка каталогів, як і звичайних файлів, починається з їхнього відкриття. Відкрити каталог можна за допомогою функції (із стандартної бібліотеки С,

```
#include <dirent.h>):
#include <dirent.h>

DIR * opendir (const char * dirname);
```

Функція відкриває каталог заданий параметром **dirname** і повертає на нього покажчик типу **DIR** (аналогічний типу **FILE** для файлів). Далі за допомогою цього покажчика можна читати вміст каталогу. У разі помилки функція повертає пустий покажчик NULL.

Після відкриття поточним стає першим елементом каталогу. Якщо надалі потрібно знову позиціонуватися на перший елемент, можна скористатися функцією **rewinddir():**

#include <dirent.h>

void rewindir (DIR * dir);

Читання вмісту каталогу

Прочитати вміст каталогу можна за допомогою функції:

#include <dirent.h>

structdirent *readdir (DIR *dir);

Параметр **dir** є вказівником на відкритий каталог. Функція повертає покажчик структуру типу **dirent**, що представляє поточний елемент каталогу; після завершення функції поточним стає наступний елемент каталогу. При досягненні кінця каталогу та у разі помилки повертається порожній покажчик NULL. Тому, якщо додатку необхідно розрізняти обидві ситуації, воно має обнулити значення змінної **errno** перед викликом **readdir()**, а потім, якщо результат дорівнює NULL, проаналізувати це значення.

Структура dirent містить принаймні одне поле:

struct dirent

{

char d_name []; // Ім'я файлу (каталогу)

...

};

Таким чином, вміст каталогу можна прочитати викликаючи в циклі функцію readdir() (поки вона не поверне NULL) зчитуючи значення поля **d_name** структури **dirent**.

Закриття каталогу

Після завершення роботи з каталогом його слід закрити за допомогою функції:

```
#include <dirent.h>
int closedir (DIR * dir);
```

Параметр **dir** є вказівником на відкритий каталог. У разі успіху функція повертає 0, якщо помилки повертає -1.

Приклад 9.11. Розробимо консольну програму, що виводить вміст каталогу, заданого як аргумент програми.

"main.c"

```
#include <stdio.h>
#include <dirent.h>
int main (int argc, char** argv)
{
DIR*dir;
struct dirent*item;
if (argc < 2) {
fprintf (stderr, "Функція викликається з одним аргументом\n");
return 1;
}
dir = opendir (argv [1]);
if (dir == NULL) {
fprintf (stderr, "Неможливо відкрити каталог %s\n", argv[1]);
return 1;
```

```
}
printf ("Вміст каталогу:\n");
while ((item = readdir (dir))! = NULL)
printf ("%s\n", item->d_name);
closedir (dir);
return 0;
}
```

Викликаємо програму.

```
delux@ubuntu :~/!$ ./readdir .
Вміст каталогу:
readdir
.
..
Practice_OSSP_7-8.pdf
Practice_OSSP_9-10.pdf
Practice_OSSP_11-12.pdf
```

Завдання лабораторної роботи 9

Необхідно розробити консольну програму, що виводить вміст каталогу, заданого як аргумент програми. Про кожен елемент каталогу (файлу) має виводитись наступна інформація:

1) ім'я;

2 розмір;

3) календарний час останньої модифікації;

4) власник.

Список літератури

1. Чекалов О. П. Основи функціонування операційних систем. Практикум: навч. пос. Гриф МОН. — Суми : СумДУ, 2010. — 85 с.

2. Шеховцов В. А. Операційні системи: підручник; Гриф МОН — К. : Видавнича група BHV, 2008. — 576 с.

3. Бондаренко М. Ф., Качко О. Г. Операційні системи: навч. посіб. Гриф МОН. – Х. : Компанія СМІТ, 2008 с. — 432 с.

4. Авраменко В. С., Авраменко А. С. Основи операційних систем. Навчальний посібник. – Черкаси: ЧНУ імені. Богдана Хмельницького, 2018. – 524 с.

Навчальне видання

Методичні вказівки
до лабораторного практикуму
з курсу «Операційні системи та засоби комп'ютерної безпеки»
для студентів спеціальності 122 «Комп’ютерні науки»

Укладачі:
БАГМУТ Іван Олександрович
МЄТЄЛЬОВ Володимир Олександрович
МІСЮРА Сергій Юрійович
ОХОТСЬКА Олена Вадимівна

Відповідальний за випуск доц. Водка О.О.
Роботу рекомендував до друку доц. ТАТАРІНОВА О.А.

В авторській редакції

План 2023 р., поз. 303

Підп. до друку 16.02.2023 р.
Гарнітура Times New Roman.

Видавничий центр НТУ «ХПІ»,
вул. Кирпичова, 2, м. Харків, 61002
Свідоцтво суб’єкта видавничої справи ДК № 5478 від 21.08.2017 р.

Електронна версія