МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ПОЛІТЕХНІКА»

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторних робіт з дисципліни «Операційні системи»

Одеса: Одеська політехніка, 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ПОЛІТЕХНІКА»

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторних робіт з дисципліни
«Операційні системи»
для здобувачів першого (бакалаврського) рівня вищої освіти

за спеціальністю

121 – Інженерія програмного забезпечення

Затверджено
На засіданні кафедри ІПЗ
Протокол №1 від 30.08.2024р.

Одеса: Одеська політехніка, 2024

Методичні вказівки до лабораторних робіт з дисципліни «Операційні системи» для здобувачів спеціальності 121 – Інженерія програмного забезпечення, (освітньо-професійна програма: Інженерія програмного забезпечення – 2022)/ Укл.: *С.Л.Зіноватна, І. В. Єгоращенко.*–Одеса: Одеська політехніка, 2024.-54с.

Укладачі: **Зіноватна С.Л.** канд. техн. наук, доцент

Єгоращенко І.В., асистент

Методичні вказівки містять теоретичні положення та рекомендовані шаблони, які застосовуються під час лабораторних робіт з дисципліни «Операційні системи». Можуть також застосовуватись під час виконання самостійних робіт з дисципліни та підготовки кваліфікаційних робіт бакалаврів.

Зміст

Вступ

Метою лабораторних занять є одержання навичок по практичному використанні окремих технологій програмного керування операційними системами технологій для розв’язання задач з програмування.

У процесі виконання робіт здобувач повинен написати програми на мові програмування С. Кожна програма повинна виконати завдані функції: виконує обчислення з обробкою помилок та використанням ключем для розгалуження дій програми; обробка файлів, заданих у якості аргументів командного рядка; створює канали для передачі повідомлень; обробляє сигнали; створює чат.

Всі лабораторні роботи виконуються студентом за єдиним варіантом.

Лабораторна робота №1
Робота в CLI операційної системи Linux

Мета роботи: підготувати на домашньому комп'ютері середовище розроблювача для виконання подальших лабораторних робіт з дисципліни «Операційні системи». Вивчити на практиці технологію роботи віртуальних машин і процес установки операційної системи (ОС) Linux.

Теоретичні відомості

CLI - Command-LineInterface

Для його вивчення потрібно включити термінал *(Застосування > Стандартні > Термінал або Ctrl+Alt+T).*

Роботу ОС Linux можна представити у вигляді функціонування множини взаємозалежних процесів. При завантаженні системи спочатку запускається ядро (процес 0), який у свою чергу запускає командний інтерпретатор **shell** (процес 1). Взаємодія користувача із системою Linux відбувається в інтерактивному режимі за допомогою командної мови. Оболонка операційної системи shell інтерпретує команды, що вводяться, запускає відповідні програми (процеси), формує й виводить відповідні повідомлення.

Shell це інтерфейс, що забезпечує взаємодію між ядром і користувачем. Інтерфейс shell дуже простий. Звичайно він складається із запрошення, за яким користувач уводить команди й натискає клавішу Enter. Рядок, у якому набирається команда, називається командним рядком. Shell не тільки інтерпретує команди, але й створює середовище, яке можна конфігурувати й програмувати. В shell є своя мова програмування, що дозволяє писати програми, що містять досить складні послідовності команд Linux. Мова програмування shell має багато властивостей звичайної мови програмування, зокрема в неї передбачене використання циклів й умовних переходів. Кожному користувачеві системи Linux надається свій власний користувальницький інтерфейс shell. Користувачі можуть модифікувати свій shell відповідно до конкретних потреб. У цьому смислі shell користувача функціонує скоріше як операційне середовище, яким користувач може управляти за своїм розсудом.

Файлова структура: каталоги й файли

В операційній системі Linux всі файли організовані в каталоги, які, у свою чергу, ієрархічно з'єднані один з одним, створючи одну загальну файлову структуру. При звертанні до файлу необхідно вказувати не тільки його ім'я, але й місце, яке він займає в цій файловій структурі.

Каталог може містити файли й інші каталоги.

Кожен каталог може містити множину підкаталогів, але сам повинен бути нащадком тільки одного батьківського каталогу. Угорі файлової системи перебуває **кореневий каталог** (позначається символом «коса риса»), від якого відгалужуються інші каталоги.

У кореневому каталозі є кілька **системних каталогів**, які містять файли й програми, що відносяться до самої ОС Linux. Корнєвій каталог, крім того, містить каталог home, що може містити домашні каталоги всіх користувачів системи. **Домашній каталог** кожного користувача, у свою чергу, буде містити в собі каталоги, які користувач створює для своїх потреб. Кожний із цих каталогів теж може містити каталоги. Всі ці вкладені каталоги відгалужуються від домашнього каталогу користувача.

Одержати доступ до каталогу можна або по імені, або зробивши його каталогом за замовчуванням. Якщо при проведенні якої-небудь операції над файлами імена каталогів не вказуються, то використається каталог за замовчуванням, що називають робочим каталогом. У цьому смислі **робочий каталог** це каталог, у якому в цей момент працює користувач. При реєстрації в системі у якості робочого приймається домашній каталог, ім'я якого звичайно збігається з реєстраційним ім'ям користувача.

Робочий каталог можна замінити за допомогою команди cd. У процесі заміни робочого каталогу виконується перехід з одного каталогу в інший.

Шляхові імена

Ім'я, що дається каталогу або файлу при його створенні, не є повним. **Повним іменем** каталога є його шляхове ім'я. Ієрархічні зв'язки, що існують між каталогами, утворять шляхи, і ці шляхи можна використовувати для однозначної вказівки каталогу або файлу й звертання до нього. Можна сказати, що кожен каталог у файловій структурі має власний унікальний шлях. Фактичне ім'я, яким система позначає каталог, завжди починається з кореневого каталогу й складається з імен всіх каталогів, що ведуть до даного каталогу. Ці імена відокремлюються друг від друга символами "коса риса". Коса риса перед першим каталогом шляху позначає кореневий каталог (/). Шляхові імена можуть бути абсолютними й відносними. **Абсолютне шляхове ім'я** це повне ім'я файлу або каталогу, що починається символом кореневого каталогу. **Відносне шляхове ім'я** починається символом робочого каталогу і являє собою позначення шляху до файлу щодо робочого каталогу.

Команди

В DOS, як й в інших операційних системах, кількість команд обмежена й статична — користувач не може вводити власні команди. В Unix (отже, і Linux) поняття команди трохи інше. Тут **команда** це будь-який виконуваний файл, тобто файл, призначений для виконання, а не для зберігання даних або конфігураційних параметрів. Будь-який виконуваний файл, записаний у систему, стає її командою.

Засоби групування команд:

cmd1 arg ...; cmd2 arg ...; ... cmdN arg ... послідовне виконання команд;

cmd1 arg ...& cmd2 arg ...& ... cmdN arg ... асинхронне виконання команд;

cmd1 arg ... && cmd2 arg ... залежність наступної команди від попередньої таким чином, що наступна команда виконується, якщо попередня видала нульове значення;

cmd1 arg ... || cmd2 arg ... залежність наступної команди від попередньої таким чином, що наступна команда виконується, якщо попередня видала ненульове значення.

Керування каталогами

Каталоги створюються й видаляються відповідно командою mkdir і командою rmdir. У тім й іншому випадку можна використати шляхові імена каталогів. У наступному прикладі користувач спочатку створює каталог reports, а потім, використовуючи абсолютне шляхове ім'я каталог letters. (Тут і далі символ $ означає запрошення до уведення й не відноситься до команд).

```
$ mkdir reports
$ mkdir /home/chris/letters
```

Для видалення каталогу потрібно дати команду rmdir з ім'ям цього каталогу. У наведеному нижче прикладі користувач спершу видаляє командою rmdir каталог reports, а потім, указавши абсолютне шляхове ім'я, каталог letters.

```
$ rmdir reports
$ rmdir /home/chris/letters
```

За допомогою команди ls можна одержати список файлів і каталогів, що перебувають у робочому каталозі.

```
$ ls
```

Для того щоб імена файлів й імена каталогів розрізнялися між собою, цю команду потрібно дати з опцією -F. У цьому випадку після кожного імені каталогу в списку ставиться коса риса.

```
$ ls
weather reports letters
$ ls -F
weather/ reports/ letters/
```

У якості аргументу команда ls може використовувати ім'я або шляхове ім'я каталогу. Це дозволяє одержати список файлів будь-якого каталогу, не переходячи в нього. У наступному прикладі команда ls використає як аргумент ім'я каталогу reports. Потім вона виконується ще раз, але вже з абсолютним шляховим ім'ям цього каталогу.

```
$ ls reports
monday tuesday
$ ls /home/chris/reports
monday tuesday
```

Перехід з одного каталогу в інший здійснюється командою cd. Перехід у каталог робить його робочим. Файлові команди, наприклад ls, будуть маніпулювати файлами, що перебувають саме в робочому каталозі, якщо іншого не зазначено. Як аргумент команда cd використає ім'я каталогу, у який птрібно перейти.

```
$ cd имя_каталогу
```

Всі створювані каталоги будуть перебувати в робочому каталозі. Робочий каталог є для знову створеного каталогу батьківським. Для позначення батьківського каталогу можна користуватися двома крапками (..). Цей спеціальний символ позначає **шляхове ім'я батьківського каталогу**. Його допускається використовувати в команді cd для переходу назад у батьківський каталог, у такий спосіб знову роблячи цей каталог робочим. Якщо необхідно повернутися в початковий каталог, потрібно ввести команду cd без аргументу.

У кожному каталозі можна створювати інші каталоги, здійснюючи, по суті справи, вкладення одного каталогу в інший. Команда cd дозволяє переходити з одного каталогу в інший, однак, ніякого покажчика на те, у якому каталозі в цей момент перебуває користувач, немає. Для того щоб визначити, у який каталог виконано перехід, потрібно дати команду pwd, що повідомить абсолютне шляхове ім'я робочого каталогу, як показано в наступному прикладі. Шляхове ім'я складається з імен робочого каталогу dylan і каталогу, частиною якого він є, home. Імена каталогів розділені косою рисою. Корнєвій каталог позначений першою косою рисою.

```
$ pwd
/home/dylan
```

Кожен каталог обов'язково має **батьківський каталог** (за винятком, природно, кореневого каталогу). При створенні каталогу в ньому відразу ж робляться два записи. Один з них буде представлений крапкою (.), а другий двома крапками (..). Крапка позначає шляхове ім'я даного каталогу, а дві крапки шляхове ім'я його батьківського каталогу. Дві крапки, використовувані як аргумент команди, позначають батьківський каталог. Одна крапка позначає робочий каталог. Крапка використається для позначення робочого каталогу замість вказівки його шляхового імені. Наприклад, для копіювання файлу в робочий каталог зі збереженням імені файлу можна замість шляхового імені робочого каталогу поставити крапку. У цьому смислі крапка ще одне ім'я робочого каталогу. Символ .. часто використовується для позначення файлів батьківського каталогу. Використовуючи команду cd із символом .., можна повертатися з каталогу нижнього рівня, послідовно переходячи в батьківські каталоги по дереву каталогів. У багатьох випадках у команді допускається використання обох символів. Наприклад, якщо letters входить в каталог chris та є робочим каталогом і потрібно скопіювати в нього файл weather, то каталог chris можна позначити двома крапками, а каталог letters одною:

```
$ cp ../weather .
```

Як згадувалося вище, файли й каталоги можна позначати абсолютними й відносними шляховими іменами. В обох варіантів, однак, є свої недоліки. Абсолютне шляхове ім'я

придатне для позначення будь-якого файлу й каталогу, але такі імена, як правило, дуже довгі й складні, що утрудняє роботу з ними. Відносне шляхове ім'я коротше й простіше в роботі, але число файлів, які їм можна позначити, обмежено. Як правило, відносні шляхові імена потрібно використати при кожній можливості, а абсолютні тільки якщо буде потреба. У деяких shell передбачена можливість скорочення абсолютних шляхових імен. Відносні шляхові імена застосовують для позначення тільки файлів, що перебувають у підкаталогах робочого каталогу. Цих підкаталогів, вкладених один в інший, може бути як завгодно багато, але їхні шляхи повинні відгалужуватися від робочого каталогу. Припустимо, потрібно звернутися до каталогу, розташованому по дереву каталогів вище робочого або в іншій гілці, тоді необхідно використати абсолютне шляхове ім'я.

У системі Linux використовуються різні утиліти, серед яких редактори, програми-листоноші й керівництва. Ці утиліти являють собою окремі програми, що мають власні інтерфейси із власними наборами команд.

Прикладом такої утиліти є діалогове керівництво man, що дозволяє користувачеві отримати інформацію про будь-яку команду й програму ОС Linux. Для звертання до діалогового керівництва потрібно ввести команду man та ім'я команди, інформація про яку потрібна. Нижче наведений приклад, у якому користувач викликає з діалогового керівництва інформацію про команду ls,

```
$ man ls
```

Після натискання клавіші Enter користувач попадає в утиліту man, що видає першу сторінку документа про команду ls. В утиліті man використовується власний набір команд, для завдання яких, як правило, досить натискання однієї клавіші. Натискання клавіші пробілу або клавіші f виводить наступну сторінку. Натискання клавіші b повертає на попередню сторінку. Вийти із утиліти й повернутися в командний рядок можна натисканням клавіші q. Опис команд у керівництві складається з декількох частин. Найчастіше їх п'ять: синопсис, опис, опції, файли й перехресні посилання. **Синопсис** містить синтаксис команди із вказівкою її опцій й аргументів. В описі команди розповідається, для чого конкретно вона застосовується в системі. Потім перераховуються й пояснюються опції. У наступній частині перераховуються системні файли, які використає команда, а в списку перехресних посилань указуються родинні команди й пункти керівництва. Нижче наведений скорочений варіант сторінки керівництва, присвяченій команді ls.

```
LS(1L)
LS(1L)
NAME
ls, dir, vdir - list contents of directories
SYNOPSIS
 ls [-abcdfgiklmnpqrstuxABCFGLNQRSUXl] t-w cols] [-T cols]
 [-1 pattern] [-iall] [-idirectory] [-iinode] [-ikilobytes]
 [-ino-group] [-ihide-control-chars] [-ireverse] [-isize]
 [-iwidth=cols] [-isort^fnone,time,size,extension}]
DESCRIPTION
 This manual page documents the GNU version of Is. dir and
vdir are versions of Is with different default output for-
mats. These programs list each given file or directory
name. Directory contents are sorted alphabetically. For
 Is, files are by default listed in columns, sorted verti-
cally, if the standard output is a terminal; otherwise
they are listed one per line. For dir, files are by
default listed in columns, sorted vertically. For vdir,
files are by default listed in long format.
```

OPTIONS
-a, -iall
List all files in directories, including all files
that start with '.'.
-b, -iescape
Quote nongraphic characters in file names using
alphabetic and octal backslash sequences like those
used in C.
-c, -itime=ctime, -itime=status
Sort directory contents according to the files'
status change time instead of the modification
time. If the long listing format is being used,
print the status change time.instead of the modifi-
cation time.
-d, -idirectory
List directories like other files, rather than
listing their contents.
-f Do not sort directory contents; list them in what-
ever order they are stored on the disk. The same
as enabling -a and -U and disabling -1, -s, and -t.
-ifull-time
List times in full, rather than using the standard
FSF GNU FileUtilities 1

Утиліта man має кілька корисних особливостей, зокрема вона дозволяє проводити пошук. Ця функція активізується натисканням або клавіші /, або клавіші ?. Перший варіант передбачає пошук уперед, а другий пошук назад. Після натискання клавіші / у нижній частині екрана з'являється рядок, у яку потрібно ввести шукане слово. Потім потрібно нажати Enter. Пошук здійснюється за зразком, тому можна ввести частину слова або практично будь-який набір символів. Повторення пошуку здійснюється натисканням клавіші n. Повторно вводити зразок не потрібно.

Команди whatis й apropos забезпечують пошук у базі даних заголовків man-сторінок і видають всі знайдені заголовки з коротким описом кожного.

Команда whatis дозволяє шукати заголовки по цілих словах. Наприклад, якщо потрібно побачити всі пункти керівництва, у яких є окремо буква x, необхідно дати наступну команду:

```
$ whatis x
X (3) - a portable, network-transparent window system
X Consortium (3) - X Consortium information
X Standards (3) - X Consortium Standards
X security (3) - x display access control
(END)
$
```

За допомогою цих команд здійснюється посторінковий вивід результатів пошуку. Якщо виведені дані займають кілька сторінок, можна переміщатися по них за допомогою клавіш f й b. Допускається й виконання пошуку за зразком (клавішами / й ?). Для виходу призначена клавіша q.

Команда apropos виконує те ж завдання, що й команда whatis, але пошук виконується за зразком, а не по цілих словах. Команда apropos x видасть кілька сторінок з даними, у яких будуть перераховані всі пункти керівництва, що починаються з букви x, наприклад xwpe й xloadimage. У наступному прикладі видається перелік пунктів керівництва, які починаються з комбінації ls. Сюди входить команда ls й інші команди, наприклад lseek й lsearch.

```
$ apropos ls
ls, dir, vdir (1) - list contents of directories
lsattr (1) - list file attributes on a Linux second extended file system
lsearch (n) - see if a list contains a particular element
lseek (2) - reposition read/write file offset
lsort (n) - sort the elements of a list
lsattr (1) - list file attributes on a Linux second extended file system
lsearch (n) - See if a list contains a particular element
lseek (2) - reposition read/write file offset
lsort (n) - Sort the elements of a list
(END)
$
```

Команди cat й more виводять уміст файлу на екран.

```
$ cat mydata
```

Команда cat виводить на екран відразу весь текст файлу. Якщо файл має великий розмір, то текст дуже швидко миготить на екрані. Для усунення цього недоліку служить команда more, за допомогою якої текст на екран можна виводити порціями.

```
$ more mydata
```

Коли more викликає файл, відображається його перший фрагмент, що вміщається на екрані. Для відображення наступного фрагмента натискається клавіша f або клавіша пробілу. Для повернення до попереднього тексту використається клавіша b. Нажавши клавішу q, можна вийти з даної програми.

Якщо потрібно надрукувати файл, його треба відправити на принтер, підключений до системи. Це робиться за допомогою команди lpr. У наступному прикладі користувач дає команду друкувати файл mydata.

```
$ lpr mydata
```

Якщо потрібно одночасно надрукувати кілька файлів, варто вказати їхні імена в командному рядку. У наступному прикладі користувачеві необхідно друкувати файли mydata й preface:

```
$ lpr mydata preface
```

Завдання на друк ставляться в чергу й виконуються у фоновому режимі. Команда lpq дозволяє в будь-який момент перевірити хід виконання завдань на друк. За її допомогою на екран виводяться ім'я власника завдання (реєстраційне ім'я користувача, що послав це завдання), ідентифікатор завдання, його розмір у байтах й ім'я тимчасового файлу, у якому воно в цей момент перебуває. У даному прикладі власник chris, а ідентифікатор завдання *00015*:

```
$ lpq
Owner ID CharsFilename
Chris 00015 360 /usr/lpd/cfa00015
```

Для виявлення одного або декількох файлів певного типу можна провести пошук за допомогою команди find. Як аргументи в ній використовуються імена каталогів, за якими ідуть кілька опцій, що задають тип і критерії пошуку. Команда find дає можливість шукати файли за іменем, типом, власникоом й за часом останньої зміни.

```
$ find список каталогів -опція критерії
```

Для того, щоб створити копію файлу, потрібно вказати команді cp два імені файлу. Перше з них ім'я файлу, що копіюється та вже існує. Друге ім'я, яке потрібно присвоїти копії. Це буде новий файл, що містить копію всіх даних вхідного файлу. Команда cp має наступний синтаксис:

```
$ cp вхідний файл вихідний файл
```

У наступному прикладі користувач копіює файл proposal у новий файл, oldprop.

$ cp proposal oldprop

При копіюванні файлу за допомогою команди cp можна ненавмисно зруйнувати інший файл. При створенні копії за допомогою цієї команди спочатку створюється файл, а потім у нього копіюються дані. Якщо який-небудь файл уже має то ім'я, що зазначене для вихідного файлу, перший з них руйнується й створюється новий файл із цим ім'ям. У деякому смислі можна сказати, що файл-оригінал перезаписується новою копією.

Щоб виявити подібні випадки, краще користуватися командою cp із опцією -i. Така команда спочатку перевіряє, чи існує файл під зазначеним ім'ям. Якщо так, то програма запитає, чи хочете перезаписати цей файл. Якщо відповісти y, то існуючий файл буде зруйнований, і програма створить новий файл у якості його копії. У випадку іншої відповіді, він буде вважатися негативним і виконання команди cp буде перервано, а файл-оригінал збережен.

$ cp -i newprop proposal
Overwrite proposal? n
$

Для того щоб скопіювати файл із робочого каталогу в інший каталог, потрібно вказати ім'я цього каталогу команді cp як другий аргумент. Ім'я нової копії буде таким же, як в оригіналу, але перебувати вона буде в іншому каталозі. Файли в різних каталогах можуть мати однакові імена.

$ cp імена_файлів ім'я_каталогу

Команда cp може використовувати як аргументи імена багатьох файлів, задані у вигляді списку, тому можна одночасно копіювати в каталог кілька файлів.

При створенні списку імен файлів для команди cp або команди mv можна використовувати будь-які спеціальні символи. Нехай, наприклад, потрібно скопіювати в заданий каталог всі файли з вхідними текстами програм, написаними мовою C. Можна ввести спеціальний символ * з розширенням .c, позначаючи тим самим всі файли з з розширенням .c (тобто всі файли вхідних текстів C-програм) і формуючи їхній список. У наступному прикладі користувач копіює всі файли вхідних текстів програм з поточного каталогу в каталог sourcebks.

$ cp *.c sourcebks

Якщо потрібно скопіювати всі файли з одного каталогу в інший, можна за допомогою позначення *.* одержати список всіх файлів (які мають розширення або в імені яких є крапка). У наступному прикладі користувач копіює всі файли з каталогу props у каталог oldprop.

$ cp props/*.* oldprop

Допускається використовувати й інші спеціальні символи, наприклад ? і [ ]. У наведеному нижче прикладі користувач копіює файли вхідного коду й файли об'єктного коду (.c й.o) у каталог projbk.

$ cp *.[oc] projbk

При копіюванні файлу можна дати копії ім'я, відмінне від імені оригіналу. Для цього потрібно помістити нове ім'я файлу після косої риси, що випливає слідом за ім'ям каталогу.

$ cp ім'я_файлу ім'я_каталогу/нове_ім'я файлу

У наступному прикладі файл newprop копіюється в каталог props і копії привласнюється ім'я version1. Потім користувач переходить у каталог props й одержує список файлів. У ньому є тільки один файл, що називається version1.

$ cp newprop props/version1
$ cd props
$ ls version1

Якщо потрібно скопіювати файл із дочірнього каталогу, наприклад, з props, у батьківський каталог, потрібно вказати ім'я цього дочірнього каталогу. Перший аргумент команди порівн - ім'я файлу, що копіюється. Перед ним повинне через косу рису стояти ім'я дочірнього каталогу. Другий аргумент ім'я, яке файл буде мати в батьківському каталозі.

$ cp ім'я_дочірнього_каталогу/ім'я_файлу нове_ім'я_файлу

У наступному прикладі файл version1 копіюється з каталогу props у початковий каталог:

$ cp props/version1 version1

Якщо потрібно скопіювати файл із дочірнього каталогу в батьківський, потрібно якось указати на цей батьківський каталог, наприклад, двома крапками, які позначають шляхове ім'я батьківського каталогу:

$ cp ім'я файлу ..

$ cp ім'я_файлу .. /нове_ім'я_файлу

Наприклад, якщо props поточний робочий каталог і потрібно скопіювати файл version1 з props у його батьківський каталог (у цьому випадку в початковий каталог користувача), потрібно замість другого аргументу команди cp використати подвійну крапку.

$ cp version1 ..

Якщо потрібно дати копії файлу version1 нове ім'я, варто додати його до другого аргументу через косу рису:

$ cp version1 ../newversion

За допомогою команди mv можна або змінити ім'я файлу, або перемістити файл із одного каталогу в інший. Використовуючи mv для перейменування файлу, як другий аргумент потрібно вказати нове ім'я файлу. Перший аргумент поточне ім'я файлу.

$ mv поточне_ім'я_файла нове_ім'я_файлу

У наступному прикладі ім'я файлу proposal міняється на version1.

$ mv proposal version1

Як і при використанні команди cp, тут можна зробити помилку, видаливши потрібний файл. Перейменовуючи файл, користувач може вибрати ім'я, що вже носить інший файл, і цей файл буде вилучений. Команда mv теж має опцію -i, що спочатку перевіряє, чи існує файл із зазначеним ім'ям. Якщо так, програма запитає, чи хочете перезаписати його. У наступному прикладі файл із ім'ям version1 уже існує. Програма виявляє, що буде здійснена перезапис, і запитує, хочете ви це зробити чи ні.

```
$ ls
proposal version1
$ mv -i version1 proposal
Overwriteproposal? n
$
```

Файл можна перенести з одного каталогу в інший. Для цього потрібно як другий аргумент у команді mv поставити ім'я каталогу. У цьому випадку можна вважати, що команда mv не перейменовує файл, а просто переміщає його з одного каталогу в інший. Після переміщення файлу в нього залишиться те ім'я, що він носив у вхідному каталозі (якщо не вказати іншого).

$ mv ім'я_файла ім'я _каталогу

У наступному прикладі файл newprop переміщається з початкового каталогу в каталог props.

$ mv newprop props

Якщо при переміщенні файлу потрібно перейменувати його, варто вказати нове ім'я файлу після імені каталогу. Ім'я каталогу відокремлюється від нового імені файлу косою

рисою. У наступному прикладі файл newprop переміщається в каталог props й одержує ім'я version1.

$ mv newprops props/version1
$ cd props
$ ls version1

Указавши ім'я дочірнього каталогу перед ім'ям файлу, його можна перемістити із цього каталогу назад у батьківський.

$ mv props/version1 version1

Як і команда cp, команда mv дозволяє одночасно перемістити з одного каталогу в інший кілька файлів. Потрібно тільки ввести імена цих файлів у командному рядку. Останнім завжди повинне стояти ім'я нового каталогу. У наступному прикладі користувач переміщає файли wednesday й friday у каталог lastweek.

$ cp wednesday friday lastweek

При створенні списку імен файлів для команди mv можна використати будь-які спеціальні символи. У наступному прикладі користувач переміщає всі файли вихідних текстів програм з поточного каталогу в каталог newproj.

$ mv *.c newproj

Система Linux дозволяє копіювати й переміщати цілі каталоги. Як перший аргумент команди cp й mv можуть використовувати ім'я каталогу, дозволяючи копіювати й переміщати підкаталоги з одного каталогу в інший. Перший аргумент ім'я переміщуваного або копіюємого каталогу, а другий ім'я каталогу, у який він буде поміщений. При переміщенні й копіюванні каталогів діє та ж структура шляхових імен, що й при відповідних операціях з файлами. Підкаталоги можна так само легко, як і файли, копіювати з одного каталогу в інший. Для копіювання каталогу команду порівн необхідно використати з опцією -r (скорочення від recursive, тобто «рекурсивний»). Ця опція дає команді cp вказівку копіювати каталог разом з усіма його підкаталогами. Інакше кажучи, копіюється все піддерево каталогів, починаючи із зазначеного. У наступному прикладі каталог thankyou копіюється в каталог oldletters. Після завершення цієї операції починають рівноправно співіснувати два підкаталоги thankyou: один у каталозі letters, інший в oldletters.

$ cp -r letters/thankyou oldletters
$ ls -F letters
thankyou/
$ ls -F oldletters
thankyou/

Спеціальний символ ~

Тильдою можна позначити абсолютне шляхове ім'я початкового каталогу. У наведеному нижче прикладі користувач переходить у каталог reports, а потім копіює з нього файл monday у початковий каталог.

$ cd reports
$ cp monday ~

Для того щоб при копіюванні файлу в початковий каталог дати йому нове ім'я, нове ім'я необхідно поставити після пари символів ~/. У наступному прикладі файл monday копіюється в початковий каталог, і копія отримує ім'я today.

$ cp monday ~/today

В аргументах команди mv тильда використовується точно так само. Нижче показано, як файл monday перемещается з каталогу reports у початковий каталог.

$ mv monday ~

Якщо при переміщенні файлу з нижчестоячого каталогу в початковий змінюється його ім'я, то перед новим ім'ям файлу ставиться тильда з косою рисою, ~/. У наступному

прикладі користувач переходить у каталог reports, а потім переміщає файл monday у початковий каталог і дає йому ім'я today.

```
$ cd reports
$ mv monday ~/today
```

Тильду можна використовувати у всіх випадках, коли мова йде про шляхове ім'я початкового каталогу. У наведеному нижче прикладі команди mv й ls виконуються з тильдою.

```
$ mv weather ~/reports/monday
$ ls ~/reports
monday
$
```

Видалити файли можна за допомогою команди rm. У наступному прикладі користувач видаляє файл oldprop.

```
$ rm oldprop
```

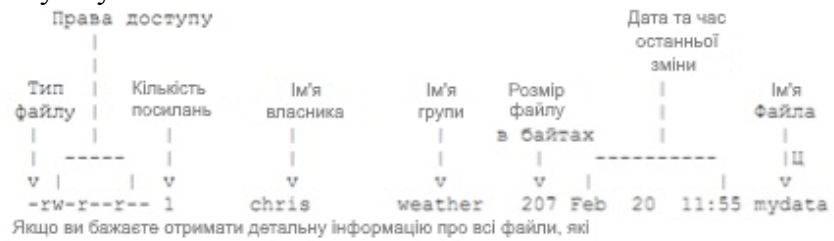
Команда rm може бути використана з будь-яким числом аргументів, що дозволяє одночасно видаляти кілька файлів. Імена цих файлів указуються в командному рядку після імені команди.

```
$ rm proposal version1 version2
```

Командою rm варто користуватися обережно, тому що скасувати її дію не можна. Якщо файл вилучений, відновити його не вдасться. Для того щоб уникнути таких помилок, можна використати команду rm з опцією -i, що ініціює видачу запиту на підтвердження видалення. Тепер перед видаленням кожного файлу система буде запитувати, чи дійсно користувач хоче видалити його. Якщо буде уведено у, файл буде вилучений. При будь-якій іншій відповіді файл не видаляється. У наступному прикладі за допомогою команди rm система одержує вказівку видалити файли proposal й oldprop, а потім запитує підтвердження по кожному з них. Користувач вирішує видалити oldprop, а proposal залишити.

```
$ rm -i proposal oldprop
Remove proposal? n
Remove oldprop? в
$
```

Команда ls -l дозволяє отримати докладну інформацію про файл. Спочатку вказуються права доступу, потім кількість посилань, ім'я власника файлу, ім'я групи, до якої він відноситься, розмір файлу в байтах, дата й час останньої зміни й, нарешті, ім'я файлу. Ім'я групи позначає групу, якій надається доступ по категорії «група». Наприклад, тип файлу mydata звичайний файл. У нього всього одне посилання; це говорить про те, що у файлу немає інших імен. Ім'я власника chris, воно збігається з реєстраційним ім'ям даного користувача. Ім'я групи weather. Імовірно, є й інші користувачі, що входять у цю групу. Розмір файлу 207 байт. Останній раз його коректували 20 лютого в 11:55. Ім'я файлу mydata.



Якщо ви бажаєте отримати детальну інформацію про всі файли, які знаходяться в даному каталозі, дайте команду ls -l без аргумента:

```
$ ls -l
-rw-r--r-- chris weather 207 Feb 20 11:55 mydata
-rw-rw-r-- chris weather 568 Feb 14 10:30 today
-rw-rw-r-- chris weather 308 Feb 17 12:40 monday
```

Знайомство з компілятором *gcc*

Засобами, традиційно використовуваними для створення програм для відкритих операційних систем, є інструменти розроблювача GNU. Одним із цих інструментів є компілятор gcc.

Перша програма, створена за допомогою gcc,за сформованою традицією буде просто виводити в консолі вітання «Hello world!».

Файли з вхідними кодами програм це звичайні текстові файли, і створювати їх можна за допомогою будь-якого текстового редактора. Крім текстових редакторів, існують спеціалізовані середовища розробки зі своїми убудованими редакторами. Так що можна прямо в одній програмі, не перемикаючись між вікнами, і редагувати код і давати консольні команди.

Необхідно створити окремий каталог для першого проекту. У ньому створити текстовий файл hello.c з наступним текстом:

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

Зайти в консолі в каталог проекту. Набрати команду

gcc hello.c

У каталозі з'явиться новий файл a.out. Це і є файл, що виконується. Запустити його, набравши в консолі:

./a.out

Повинен з'явитися текст:

Hello world!

Компілятор gcc за замовчуванням привласнює всім створеним файлам, що виконуються, ім'я a.out. Якщо потрібно назвати його по-іншому, треба до команди на компіляцію додати прапор -o й ім'я, яким потрібно його назвати. Команда може виглядати в такий спосіб:

gcc hello.c -o hello

У каталозі з'явиться файл, що виконується, з назвою hello, який можна запустити командою.

./hello

Прапор -o є одним із численних прапорів компілятора gcc. Щоб переглянути всі можливі прапори, можна скористатися довідковою системою man. Для цього необхідно набрати в командному рядку:

man gcc

Якщо при запуску програми з каталогу розробки набрати тільки назва файлу, що виконує, без вказівки перед назвою файлу крапки й слеша, операційна система буде шукати його в каталогах /usr/bin й /usr/local/bin. Каталоги /usr/bin й /usr/local/bin системні каталоги розміщення програм, що виконуються. Перший з них призначений для розміщення стабільних версій програм, як правило, що входять у дистрибутив Linux. Другий для програм, установлюваних самим користувачем. Така система потрібна,щоб відокремити їх друг від друга. За замовчуванням при зборці програми встановлюються в каталог /usr/local/bin. Украй небажано поміщати що-небудь зайве в /usr/bin або видаляти щось відтіля вручну, тому що це може привести до краху системи. Там повинні розміщатися програми, за стабільність яких відповідають розроблювачі дистрибутива.

Щоб запустити програму, що перебуває в іншому місці, треба прописати повний шлях до неї, наприклад, так:

/home/dima/projects/hello/hello

Інший варіант: прописати шлях щодо поточного каталогу. При цьому одна крапка означає поточний каталог, дві крапки батьківський. Наприклад, команда ./hello запускає програму hello, що перебуває в поточному каталозі, команда ../hello програму hello, що перебуває в батьківському каталозі, команда ./projects/hello/hello програму у вкладених каталогах, що перебувають усередині поточного.

Робота програми gcc включає три етапи: обробка препроцесором, компіляція й компонування.

Препроцесор включає в основний файл уміст всіх заголовних файлів, зазначених у директивах #include. У заголовних файлах звичайно перебувають оголошення функцій, використовуваних у програмі, але не визначених у тексті програми. Їхні визначення перебувають десь в іншому місці: або в інших файлах з вхідним кодом або в бінарних бібліотеках.

Друга стадія компіляція. Вона полягає в перетворенні тексту програми мовою С в набір машинних команд. Результат зберігається в об'єктному файлі. На машинах з різною архітектурою процесора двійкові файли виходять у різних форматах, і на одній машині неможливо запустити двійковий файл, зібраний на іншій машині (тільки, якщо в них однакова архітектура процесора й однакові операційні системи). Тому програми для UNIX-подібних систем поширюються у вигляді вхідних кодів: вони повинні бути доступні всім користувачам, незалежно від того, у кого який процесор й яка операційна система. Остання стадія компонування. Воно полягає у зв'язуванні всіх об'єктних файлів проекту в один, зв'язуванні викликів функцій з їхніми визначеннями, і приєднанням бібліотечних файлів, що містять функції, які викликаються, але не визначені в проекті. У результаті формується файл, що запускається. Якщо якась функція в програмі використовується, але компоновник не знайде місце, де ця функція визначена, він видасть повідомлення про помилку, і не створить файл, що виконується.

Завдання

1. Установити на домашньому комп'ютері віртуальну машину VirtualBox.

2. Установити у віртуальній машині операційну систему Linux (рекомендується Linux Mint https://linuxmint.com/download.php або Ubuntu).

3. Освоїти роботу з командами для роботи з файлами.

Переглянути вміст робочого каталогу.

Переглянути вміст батьківського каталогу, не переходячи в нього.

Перейти в батьківський каталог. Переглянути його вміст. Переглянути вміст початкового каталогу. Повернутися в початковий каталог.

Створити в домашньому каталозі наступну структуру підкаталогів (існуючі каталоги не видаляти!)

```
|-домашній каталог
   |-ВашеПрізвище
      |-1
        |-2
        |-3
      |-4
```

Скопіювати файл /etc/group у каталог 1, використовуючи абсолютні імена копіюємого файлу й каталогу призначення.

Скопіювати файл /etc/group у каталог 2, використовуючи абсолютне ім'я копіюємого файлу й відносне ім'я каталогу призначення.

Скопіювати файл /etc/group у каталог 3, використовуючи відносні імена копіюємого файлу й каталогу призначення.

Скопіювати файл /etc/group у каталог 4, використовуючи абсолютні імена копіюємого файлу й відносне ім'я каталогу призначення з використанням спеціального символу ~.

За допомогою однієї команди зайти в каталог 3.

Вивести перших й останні 13 рядків файлу /etc/group (команди head та tail).

Видалити файл group з каталогу 4 за допомогою однієї команди.

Перейти у свій домашній каталог. Видалити каталоги 1 й 4.

3. Виконати розробку програми HelloWorld на C в Linux, використовуючи стандартний текстовий редактор і компілятор командного рядка.

4. Відобразити виконані етапи в протоколі.

Контрольні питання

1. Опишіть, яким чином користувач взаємодіє з операційною системою Linux.
2. опишіть. які види каталогів можуть бути виділені у файловій системі.
3. Опишіть, яким чином визначається повне ім’я файлу.
4. Дайте загальне визначення команди в ОС Linux.
5. Назвіть, якою командою можна отримати доступ до діалогового керівництва.

Лабораторна робота №2
Обробка помилок у програмі.
Аргументи командного рядка й ключі

Мета роботи: навчитися звертатися в програмі до аргументів командного рядка й ключів; навчитися обробляти можливі помилки в програмі.

Теоретичні відомості

Аргументи командного рядка й ключі

Функція getopt, що оголошується в заголовку <stdlib.h> може бути використана для реалізації програм, які приймають UNIX-подібні ключі й аргументи командного рядка. Синтаксис виклику таких програм повинен бути наступним:

<ім'я_програми> [-<ключ> ...] [<аргумент> ...]

Всі ключі (або опції) повинні починатися із символу "-" і складатися з однієї букви, наприклад: *-o*. При цьому враховується й регістр. Кілька ключів можна поєднувати (так, ключі -a, -b можна дати як -ab або -ba). За ключем може іти тільки з ним зв'язаний необов'язковий аргумент (наприклад, -o a.out). Якщо два або кілька ключів об'єднані, то тільки останній з них може приймати такий аргумент. Наприклад, -o a.out -O може бути записане як -Oo a.out, але не як -oO a.out.

Якщо аргумент не пов'язаний із ключем, то після нього ключі ставити не можна. Таким чином, наступний виклик не вірний, оскільки перед ключем *-o* зазначений не пов'язаний з ним аргумент /usr/prog/test.c.

$ a.out -1 /usr/prog/test.c -o abc

Якщо при виклику програми зазначені правила дотримуються, то за допомогою функції getopt ключі й всі пов'язані з ними аргументи можуть бути отримані з командного рядка.

Функція *getopt* (і пов'язані з нею глобальні змінні opterr, optarg й optind оголошуються в заголовку <stdlib.h> (<unistd.h> в Unix):

```
extern int      optind, opterr;
extern char* optarg;
int getopt  (int argc, char* const* argv[], const char* optstr);
```

Першими двома аргументами функції getopt є змінні argc й argv функції main. Аргумент optstr містить список букв ключів, які припустимі для даної програми. Функція переглядає вектор argv і шукає ключі, які визначені в optstr. Після кожного виклику функція повертає букву ключа, зазначену в optstr і знайдену в argv. Якщо ключ визначений в optstr як <switch_letter>, то цей ключ, якщо він знайдений, повинен супроводжуватися аргументом, що може бути отриманий через глобальний покажчик argv.

Якщо ключ перебуває в argv, але не зазначений в optstr, то функція getopt направить повідомлення в стандартний потік помилок і поверне символ «?», Якщо ж користувач перед викликом getopt установлює глобальну змінну opterr у ненульове значення, то наступні неприпустимі ключі, знайдені в argv, функція буде ігнорувати.

Нарешті, якщо в argv більше немає ключів, то функція getopt повертає значення EOF і змінна optind установлюється так, щоб указувати на елемент argv, де зберігається перший неключовий аргумент командного рядка, Якщо optind збігається з argc, то в програмі не пов'язаних із ключем аргументів немає.

Наступна програма приймає ключі -a, -b й -o. Якщо зазначено ключ -o, то з ним повинне бути задане ім'я файлу:

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

```
#include  <unistd.h>
#include <string.h>
int main (int  argc,   char* argv[])
{
        int ch;
        while( (ch=getopt(argc,   argv,"o:ab"))!= EOF)
        switch(ch){
                case   'a':
                        printf("Key:%s\n","a");
                        break;
                case 'b':
                        printf("Key:%s\n","b");
                        break;
                case 'o':
                        printf("Key:%s %s\n","o", optarg);

                        break;.
                case   '?':       /*   getopt  видасть повідомлення про помилку */
                default:
                        break;       // неприпустимий ключ
        }
        /*  ключів більше немає. Переглянути інші неключові аргументи */
        for( ; optind<argc; optind++ )
                printf("non-switch argument:   :%s\n", argv[optirtd);
        return  0;
}
```

Результат виконання програми

```
 $ gcc test_getopt.C  -o test_getopt

$ test_getopt

$  test_getopt  -abo xyz   /etc/hosts
Key:a
Key:b
Key:o xyz
non_switch  argument:   /etc/hosts

 $   test_getopt -xay  -bz  /usr/lib/libc.a
test_getopt:   invalid  option -x
teat_getopt:   invalid  option - y
test_getopt:   invalid  option  -z
non_switch  argument:   /usr/lib/libc.a
```

Функція getopt супроводжується наступними обмеженнями:

− ключі повинні складатися тільки з однієї букви;

− ключі повинні мати або пов'язані з ними аргументи, або не мати їх взагалі;

− користувачі не мають права визначати ключі, які можуть іноді приймати, а іноді не приймати аргументи;

− функція не перевіряє тип даних ключів;

− користувачі не можуть вказувати взаємовиключні ключі.

Бібліотека <assert.h>

У заголовку <assert.h> оголошується макрокоманда, використовувана для перевірки деяких умов виконання процесу, які в нормальній ситуації завжди повинні бути правдиві. Якщо все-таки під час виконання процесу умова не виконується, то макрокоманда виводить повідомлення про помилку в стандартний потік помилок із вказівкою того рядка вхідного файлу, у якій порушується умова, що перевіряється. Після цього макрокоманда перериває процес.

Тяким чином, макрокоманда assert допомагає заощадити час, що йде на налагодження програми, перевіряючи умови типу «не повинне було відбутися».

Для того, щоб відключити перевірку, не обов'язково виключати її з коду або коментувати оголошення макросу, досить лише оголосити ще один макрос — NDEBUG у програмі перед #include <assert.h>:

```
#define NDEBUG
```

У наведеній нижче програмі демонструється використання макрокоманди *assert*.

```
#include  <fstream.h>
#include  <string.h>
#include  <assert.h>
int main(int argc, char* argv[])
{
        assert(argc>1);             // повинен бути хоча б 1 аргумент
        return   0;
}
```

Коли програма компілюється й виконується без аргументу, на консоль виводиться повідомлення:

```
 $  gcc  asaert.C  -o assert;   assert
Assertion  failed:   file   "assert.C",   line  5
```

Завдання

Написати програму, що реалізує примітивний калькулятор.

Програма приймає у якості аргументів операнди й знак операції.

За допомогою ключа визначається, виводити як результат програми повне вираження (a+b=c) або тільки результат обчислення (c).

Програма повинна обробляти помилки: 1) звичайними перевірками з видачею повідомлень за допомогою printf ; 2) з використанням бібліотеки <assert.h> .

Контрольні питання

6. Опишіть, як формуються аргументи командного рядка.
7. Опишіть змінні argc й argv функції main.
8. Назвіть варіанти обробки помилок в програмі.

Лабораторна робота №3-4
Взаємодія процесів. Потік вводу-виводу

Мета роботи: навчитися працювати з потоками вводу-виводу при взаємодії процесів.

Теоретичні відомості

Серед всіх категорій засобів комунікації процесів найбільш уживаними є канали зв'язку, що забезпечують досить безпечне й досить інформативну взаємодію процесів.

У випадку потокової моделі передачі даних по каналах зв'язку операції передачі/прийому інформації не цікавляться вмістом того, що передається або приймається. Вся інформація в каналі зв'язку розглядається як безперервний потік байт, що не володіє ніякою внутрішньою структурою.

Поняття про pipe

Найбільш простим способом для передачі інформації за допомогою потокової моделі між різними процесами або навіть усередині одного процесу в операційній системі UNIX є ***pipe*** (канал, труба, конвеєр).

Важлива відмінність *pipe*'а від файлу полягає в тім, що прочитана інформація негайно видаляється з нього й не може бути прочитана повторно.

Pipe можна уявити собі у вигляді труби обмеженої ємності, розташованої усередині адресного простору операційної системи, доступ до вхідного й вихідного отвору якої здійснюється за допомогою системних викликів. У дійсності *pipe* являє собою область пам'яті, недоступну користувальницьким процесам прямо, найчастіше організовану у вигляді кільцевого буфера (хоча існують й інші види організації). По буфері при операціях читання й записи переміщаються два покажчики, що відповідають вхідному й вихідному потокам. При цьому вихідний покажчик ніколи не може перегнати вхідний і навпаки. Для створення нового екземпляра такого кільцевого буфера усередині операційної системи використається системний виклик *pipe*().

```
#include <unistd.h>
int pipe(int *fd);
```

Параметр *fd* є покажчиком на масив із двох цілих змінних. При нормальному завершенні виклику в перший елемент масиву - *fd*[0] - буде занесений файловий дескриптор, що відповідає вихідному потоку даних pip'а й що дозволяє виконувати тільки операцію читання, а в другий елемент масиву - *fd*[1] - буде занесений файловий дескриптор, що відповідає вхідному потоку даних і дозволяє виконувати тільки операцію запису.

Значення, що повертаються

Системний виклик повертає значення 0 при нормальному завершенні й значення -1 при виникненні помилок.

У процесі роботи системний виклик організує виділення області пам'яті під буфер і покажчики й заносить інформацію, що відповідає вхідному й вихідному потокам даних, у два елементи таблиці відкритих файлів, зв'язуючи тим самим з кожним pipe'ом два файлових дескриптори. Для одного з них дозволена тільки операція читання з pipe'а, а для іншого - тільки операція запису в pipe. Для виконання цих операцій можна використовувати ті ж самі системні виклики *read*() і *write*() , що й при роботі з файлами. Природно, по закінченні використання вхідного або/і вихідного потоку даних, потрібно закрити відповідний потік за допомогою системного виклику *close*() для звільнення системних ресурсів. Необхідно відзначити, що, коли всі процеси, що використають pipe , закривають всі асоційовані з ним файлові дескриптори, операційна система ліквідує pipe . Таким чином, час існування pipe'а в системі не може перевищувати час життя процесів, що працюють із ним.

Приклад 1. Програма для pipe в одному процесі

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

int main(){
   int fd[2];
   size_t size;
   char string[] = "Hello, world!";
   char resstring[14];
   /* Спроба створити pipe */
   if(pipe(fd) < 0){
      /* Якщо створити pipe не вдалося, друкується про це повідомлення
      і припиняється робота */
      printf("Can\'t create pipe\n");
      exit(-1);
   }
   /* Спроба записати в pipe 14 байт із масиву, тобто весь
   рядок "Hello, world!" разом з ознакою кінця рядка */
   size = write(fd[1], string, 14);
   if(size != 14){
      /* Якщо записалася менша кількість байт, повідомлення про  помилку */
      printf("Can\'t write all string\n");
      exit(-1);
   }
   /* Спроба прочитати з pip'a 14 байт в інший масив, тобто весь
   записаний рядок */
   size = read(fd[0], resstring, 14);
   if(size < 0){
      /* Якщо прочитати не вдалося, повідомлення про помилку */
      printf("Can\'t read string\n");
      exit(-1);
   }
   /* Друк прочитаного рядка */
   printf("%s\n",resstring);
   /* Закриття вхідного потоку*/
   if(close(fd[0]) < 0){
      printf("Can\'t close input stream\n");
   }
   /* Закриття вихідного потоку */
   if(close(fd[1]) < 0){
      printf("Can\'t close output stream\n");
   }
   return 0;
}
```

Організація зв'язку через pipe між процесом-батьком і процесом-нащадком

С допомогою системного виклику fork()можна створити новий процес

```
#include <sys/types>
#include <unistd.h>
pid_t fork(void);
```

pid_t є примітивним типом даних, що визначає ідентифікатор процесу або групи процесів. При виклику *fork*() породжується новий процес (процес-нащадок), що майже ідентичний процесу-батькові, який породжує. Процес-нащадок успадковує наступні ознаки батька:

− сегменти коду, даних і стека програми;

− таблицю файлів, у якій перебувають стани прапорів дескрипторів файлу, що вказують, чи читається файл або пишеться. Крім того, у таблиці файлів міститься поточна позиція покажчика запису-читання;

− робочий і кореневий каталоги;

− реальний й ефективний номер користувача й номер групи;

− пріоритети процесу (адміністратор може змінити їх через nice);

− контрольний термінал;

− маску сигналів;

− обмеження по ресурсах;

− відомості про середовище виконання;

− поділювані сегменти пам'яті.

Нащадок не успадковує від батька наступних ознак:

− ідентифікатора процесу (PID, PPID);

− витраченого часу ЦП (воно обнуляється);

− сигналів процесу-батька, що вимагають відповіді;

− блокованих файлів (record locking).

При виклику *fork*() виникають два повністю ідентичних процеси. Весь код після *fork*() виконується двічі, як у процесі-нащадку, так й у процесі-батьку.

Процес-нащадок і процес-батько одержують різні коди повернення після виклику *fork*(). Процес-батько одержує ідентифікатор (PID) нащадка. Якщо це значення буде негативним, отже, при породженні процесу відбулася помилка. Процес-нащадок одержує як код повернення значення 0, якщо виклик *fork*() виявився успішним.

Таким чином, можна перевірити, чи був створений новий процес:

```
switch(ret=fork())
{
 case -1: /{*}при виклику fork() виникла помилка{*}/
 case 0 : /{*}це код нащадка{*}/
 default : /{*}це код батьківського процесу{*}/
}
```

Приклад 2. Породження процесу через fork().

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
main()
{
 pid_t pid;
 int rv;
 switch(pid=fork()) {
 case -1:
       perror("fork"); /* відбулася помилка */
       exit(1); /*вихід з батьківського процесу*/
 case 0:
```

```
        printf(" CHILD: Це процес-нащадок!\n");
        printf(" CHILD: Мій PID -i %d\n", getpid());
        printf(" CHILD: PID мого батька -i %d\n",
           getppid());
        printf(" CHILD: Уведіть мій код повернення
                  (якнайменше):");
        scanf(" %d");
        printf(" CHILD: Вихід!\n");
        exit(rv);
 default:
        printf("PARENT: Це процес-батько!\n");
        printf("PARENT: Мій PID -i %d\n", getpid());
        printf("PARENT: PID мого нащадка %d\n",pid);
        printf("PARENT: Я чекаю, поки нащадок
                    не викличе exit()...\n");
        wait(0);
        printf("PARENT: Код повернення нащадка:%d\n",
              WEXITSTATUS(rv));
        printf("PARENT: Вихід!\n");
  }
}
```

Коли нащадок викликає *exit*(), код повернення передається батькові, що очікує його, викликаючи *wait*(). WEXITSTATUS() являє собою макрос, що одержує фактичний код повернення нащадка з виклику *wait*().

Функція *wait*() чекає завершення першого із всіх можливих нащадків батьківського процесу. Іноді необхідно точно визначити, який з нащадків повинен завершитися. Для цього використається виклик *waitpid*() з відповідним PID нащадка як аргумент. Ще один момент, на який варто звернути увагу при аналізі приклада, це те, що й батько, і нащадок використають змінну *rv*. Це не означає, що змінна розділено між процесами. Кожен процес містить власні копії всіх змінних.

Приклад3. Створення декількох нащадків за допомогою fork().

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
  char pid{[}255{]};
  fork();
  fork();
  fork();
  sprintf(pid, "PID : %d\n",getpid());
  write(STDOUT_FILENO, pid, strlen(pid));
  exit(0);
}
```

У цьому випадку буде створено сім процесів-нащадків. Перший виклик *fork*() створює першого нащадка. Як зазначено вище, процес успадковує положення покажчика команд від батьківського процесу. Покажчик команд містить адреса наступного оператора програми. Це значить, що після першого виклику *fork*() покажчик команд і батька, і нащадка перебуває перед другим викликом *fork*().Після другого виклику *fork*() і батько, і перший нащадок роблять нащадків другого покоління - у результаті утвориться чотири процеси. Після третього виклику *fork*()кожен процес робить свого нащадка, збільшуючи загальне число процесів до восьми.

Так називані процеси-зомбі виникають, якщо нащадок завершився, а батьківський процес не викликав *wait*(). Для завершення процесів використовують або оператор повернення, або виклик функції *exit*() зі значенням, яке потрібно повернути операційній системі. Операційна система залишає процес зареєстрованим у своїй внутрішній таблиці даних, поки батьківський процес не одержить коду повернення нащадка, або не закінчиться сам. У випадку процесу-зомбі його код повернення не передається батькові, і запис про цей процес не видаляється з таблиці процесів операційної системи. При подальшій роботі й появі нових зомбі таблиця процесів може бути заповнена, що приведе до неможливості створення нових процесів.

Таблиця відкритих файлів успадковується процесом-дитиною при породженні нового процесу системним викликом *fork*() і входить до складу незмінної частини системного контексту процесу при системному виклику *exec*() (за винятком тих потоків даних, для файлових дескрипторів яких був спеціальними засобами виставлений ознака, що спонукує операційну систему закрити їх при виконанні *exec*()). Ця обставина дозволяє організувати передачу інформації через *pipe* між родинними процесами, що мають загального прабатька, що створив pipe .

Приклад4. Програма для організації односпрямованого зв'язку між родинними процесами через pipe

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

int main(){
   int fd[2], result;
   size_t size;
   char resstring[14];
   /* Спроба створити pipe */
   if(pipe(fd) < 0){
      /* Якщо створити pipe не вдалося, друкується про це повідомлення
      і припиняється робота */
      printf("Can\'t create pipe\n");
      exit(-1);
   }
   /* Породжується новий процес */
   result = fork();
   if(result<0){
      /* Якщо створити процес не вдалося, повідомляється про це й завершується робота */
      printf("Can\'t fork child\n");
      exit(-1);
   } else if (result > 0) {
      /* Робота в батьківському процесі, що буде
      передавати інформацію процесу-дитині. У цьому процесі
      вихідний потік даних не знадобиться, тому він закривається.*/
      close(fd[0]);
      /* Спроба записати в pipe 14 байт, тобто весь рядок
      "Hello, world!" разом з ознакою кінця рядка */
      size = write(fd[1], "Hello, world!", 14);
      if(size != 14){
         /* Якщо записалася менша кількість байт, повідомляється
         про помилку й завершується робота */
         printf("Can\'t write all string\n");
```

```
            exit(-1);
        }
        /* Закривається вхідний потік даних, на цьому  батько припиняє роботу */
        close(fd[1]);
        printf("Parent exit\n");
    } else {
        /* Робота в породженому процесі, що буде
        одержувати інформацію від процесу-батька. Він успадкував
        від батька таблицю відкритих файлів й, знаючи файлові
        дескриптори, що відповідають pipe, може його використати.
        У цьому процесі вхідний потік даних не  знадобиться, тому він закривається.*/
        close(fd[1]);
        /* Спроба прочитати з pip'a 14 байт у масив, тобто весь  записаний рядок */
        size = read(fd[0], resstring, 14);
        if(size < 0){
            /* Якщо прочитати не вдалося, повідомляється про помилку й
            завершується робота */
            printf("Can\'t read string\n");
            exit(-1);
        }
        /* Друкується прочитаний рядок */
        printf("%s\n",resstring);
        /* Закривається вхідний потік і завершується робота */
        close(fd[0]);
    }
    return 0;
}
```

Організація двонаправленого зв'язку між родинними процесами через *pipe*

Pipe служить для організації односпрямованого або симплексного зв'язку. Якби в Прикладі 2 була спроба організувати через *pipe* двосторонній зв'язок, коли процес-батько пише інформацію в *pipe*, припускаючи, що її одержить процес-дитина, а потім читає інформацію з *pipe*'а, припускаючи, що її записав породжений процес, то могла б виникнути ситуація, у якій процес-батько прочитав би власну інформацію, а процес-дитина не одержала б нічого. Для використання одного *pipe*'а у двох напрямках необхідні спеціальні засоби синхронізації процесів. Більш простий спосіб організації двонаправленого зв'язку між родинними процесами полягає у використанні двох *pipe*.

Особливості поводження викликів read() і write() для pipe'а

Системні виклики *read*() і *write*() мають певні особливості поводження при роботі з *pipe*'ом, пов'язані з його обмеженим розміром, затримками в передачі даних і можливістю блокування процесів, що обмінюються інформацією.

Необхідно бути уважним при написанні програм, що обмінюються великими обсягами інформації через *pipe*. Потрібно пам'ятати, що за один раз із *pipe*'а може прочитатися менше інформації, чим було запитано, і за один раз в *pipe* може записатися менше інформації, чим хотілося. Необхідно перевіряти значення, що повертають викликами!

Одна з особливостей поводження системного виклику *read*(), що блокується, пов'язана зі спробою читання з порожнього *pipe*'а. Якщо є процеси, у яких цей *pipe* відкритий для запису, то системний виклик блокується й чекає появи інформації. Якщо таких процесів ні, він поверне значення 0 без блокування процесу. Ця особливість приводить до необхідності закриття файлового дескриптора, асоційованого із вхідним

кінцем *pipe'*а, у процесі, що буде використати *pipe* для читання (*close* (*fd*[1]) у процесі-дитині в програмі із Приклад 2. Аналогічною особливістю поводження при відсутності процесів, у яких pipe відкритий для читання, володіє й системний виклик *write*(), із чим зв'язана необхідність закриття файлового дескриптора, асоційованого з вихідним кінцем *pipe*'а, у процесі, що буде використати *pipe* для запису (*close* (*fd*[0]) у процесі-батьку в Приклад2).

Поняття FIFO

Для організації потокової взаємодії будь-яких процесів в операційній системі UNIX застосовується засіб зв'язку, що отримав назву FIFO (First Input First Output) або іменований pipe. FIFO у всьому подібний *pipe*'у, за одним виключенням: дані про розташування FIFO в адресному просторі ядра і його стані процеси можуть одержувати не через родинні зв'язки, а через файлову систему. Для цього при створенні іменованого pipe'а на диску заводиться файл спеціального типу, звертаючись до якого процеси можуть одержати інформацію, що їх цікавить. Для створення FIFO використається системний виклик *mknod*() або існуюча в деяких версіях UNIX функція *mkfifo*().

При їхній роботі не відбувається дійсного виділення області адресного простору операційної системи під іменований *pipe*, а тільки заводиться файл-мітка, існування якої дозволяє здійснити реальну організацію FIFO у пам'яті при його відкритті за допомогою системного виклику *open*() .

Після відкриття іменований pipe поводиться точно так само, як і неіменований. Для подальшої роботи з ним застосовуються системні виклики *read*() , *write*() і *close*() . Час існування FIFO в адресному просторі ядра операційної системи, як й у випадку з *pipe*'ом, не може перевищувати час життя останнього з його процесів, що використовували. Коли всі процеси, що працюють із FIFO, закривають всі файлові дескриптори, асоційовані з ним, система звільняє ресурси, виділені під FIFO. Вся непрочитана інформація втрачається. У той же час файл-мітка залишається на диску й може використовуватися для нової реальної організації FIFO надалі.

Використання системного виклику mknod і функції mkfifo для створення FIFO

```
#include <sys/stat.h>
#include <unistd.h>
int mknod(char *path, int mode, int dev);
int mkfifo(char *path, int mode);
```

Параметр *dev* є несуттєвим для організації FIFO, і буде завжди задаватися рівним 0.

Параметр *path* є покажчиком на рядок, що містить повне або відносне ім'я файлу, що буде міткою FIFO на диску. Для успішного створення FIFO файлу з таким ім'ям перед викликом існувати не повинне.

Параметр *mode* установлює атрибути прав доступу різних категорій користувачів до FIFO. Цей параметр задається як результат побітової операції "або" значення S_IFIFO, що вказує, що системний виклик повинен створити FIFO, і деякої суми наступних восьмеричних значень:

- 0400 - дозволене читання для користувача, що створив FIFO;
- 0200 - дозволений запис для користувача, що створив FIFO;
- 0040 - дозволене читання для групи користувача, що створив FIFO;
- 0020 - дозволений запис для групи користувача, що створив FIFO;
- 0004 - дозволене читання для всіх інших користувачів;
- 0002 - дозволений запис для всіх інших користувачів.

При успішному створенні FIFO системний виклик повертає значення 0, при неуспішному - негативне значення.

Важливо розуміти, що файл типу FIFO не служить для розміщення на диску інформації, що записується в іменований *pipe* . Ця інформація розташовується усередині адресного простору операційної системи, а файл є тільки міткою, що створює передумови для її розміщення.

Системні виклики *read*() і *write*() при роботі з FIFO мають ті ж особливості поводження, що й при роботі з *pipe*'ом. Системний виклик *open*() при відкритті FIFO також поводиться трохи інакше, чим при відкритті інших типів файлів, що пов'язане з можливістю блокування виконуючих його процесів. Якщо FIFO відкривається тільки для читання, і прапор O_NDELAY не заданий, то процес, що здійснив системний виклик, блокується доти, поки який-небудь інший процес не відкриє FIFO на запис. Якщо прапор O_NDELAY заданий, то повертається значення файлового дескриптора, асоційованого з FIFO. Якщо FIFO відкривається тільки для запису, і прапор O_NDELAY не заданий, то процес, що здійснив системний виклик, блокується доти, поки який-небудь інший процес не відкриє FIFO на читання. Якщо прапор O_NDELAY заданий, то констатується виникнення помилки й повертається значення -1. Завдання прапора O_NDELAY у параметрах системного виклику open() приводить і до того, що процесу, що відкрив FIFO, забороняється блокування при виконанні наступних операцій читання із цього потоку даних і запису в нього.

Приклад 5. Програма с FIFO у родинних процесах

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

int main(){
   int fd, result;
   size_t size;
   char resstring[14];
   char name[]="aaa.fifo";
   /* Обнуляється маска створення файлів поточного процесу для того,
   щоб права доступу в створюваного FIFO точно відповідали
   параметру виклику mknod() */
   (void)umask(0);
   /* Спроба створити FIFO з ім'ям aaa.fifo у поточної  директорії */
   if(mknod(name, S_IFIFO | 0666, 0) < 0){
      /* Якщо створити FIFO не вдалося, друкується про цьому
      повідомлення й припиняється робота */
      printf("Can\'t create FIFO\n");
      exit(-1);
   }
   /* Породжується новий процес */
   if((result = fork()) < 0){
      /* Якщо створити процес не вдалося, повідомляється про цьому й  завершується
      робота */
      printf("Can\'t fork child\n");
      exit(-1);
   } else if (result > 0) {
      /* Робота  в батьківському процесі, що буде передавати інформацію
```

```
      процесу-дитині. У цьому процесі    відкривається FIFO на запис.*/
      if((fd = open(name, O_WRONLY)) < 0){
         /* Якщо відкрити FIFO не вдалося, друкується про це  повідомлення й
       припиняється робота */
         printf("Can\'t open FIFO for writing\n");
         exit(-1);
      }
      /* Спроба записати в FIFO 14 байт, тобто весь рядок
      "Hello, world!" разом з ознакою кінця рядка */
      size = write(fd, "Hello, world!", 14);
      if(size != 14){
      /* Якщо записалася менша кількість байт,то повідомляється
      про помилку й завершується робота */
         printf("Can\'t write all string to FIFO\n");
         exit(-1);
      }
      /* Закривається вхідний потік даних і на цьому батько  припиняє роботу */
      close(fd);
      printf("Parent exit\n");
   } else {
      /* Робота в породженому процесі, що буде
      одержувати інформацію від процесу-батька. Відкривається  FIFO на читання.*/
      if((fd = open(name, O_RDONLY)) < 0){
         /* Якщо відкрити FIFO не вдалося, друкується про цьому
         повідомлення й припиняється робота */
         printf("Can\'t open FIFO for reading\n");
         exit(-1);
      }
      /* Спроба прочитати з FIFO 14 байт у масив, тобто  весь записаний рядок */
      size = read(fd, resstring, 14);
      if(size < 0){
         /* Якщо прочитати не вдалося, повідомляється про помилку  й завершується
        робота */
         printf("Can\'t read string\n");
         exit(-1);
      }
      /* Друкується прочитаний рядок */
      printf("%s\n",resstring);
      /* Закривається вхідний потік і завершується роботу */
      close(fd);
   }
   return 0;
}
```

Повторний запуск цієї програми приведе до помилки при спробі створення FIFO, тому що файл із заданим ім'ям уже існує. Тут потрібно або видаляти його перед кожним прогоном програми з диска вручну, або модифікувати вхідний текст, додавши наприкінці виклик команди *unlink*(*name*).

Особливості поводження при роботі з *pipe*, FIFO й *socket*

Системний виклик *read*

Ситуація	Поводження

Спроба прочитати менше байт, чим є в наявності в каналі зв'язку.	Читає необхідну кількість байт і повертає значення, що відповідає прочитаній кількості. Прочитана інформація видаляється з каналу зв'язку.
У каналі зв'язку перебуває менше байт, чим викликано, але не нульова кількість.	Читає все, що є в каналі зв'язку, і повертає значення, що відповідає прочитаній кількості. Прочитана інформація видаляється з каналу зв'язку.
Спроба читати з каналу зв'язку, у якому немає інформації. Блокування виклику дозволене.	Виклик блокується доти, поки не з'явиться інформація в каналі зв'язку й поки існує процес, що може передати в нього інформацію. Якщо інформація з'явилася, то процес розблокується, і поводження виклику визначається двома попередніми рядками таблиці. Якщо в канал комусь передати дані (немає жодного процесу, у якого цей канал зв'язку відкритий для запису), то виклик повертає значення 0. Якщо канал зв'язку повністю закривається для запису під час блокування читаючого процесу, то процес розблокується, і системний виклик повертає значення 0.
Спроба читати з каналу зв'язку, у якому немає інформації. Блокування виклику не дозволене.	Якщо є процеси, у яких канал зв'язку відкритий для запису, системний виклик повертає значення -1 і встановлює змінну *errno* у значення EAGAIN. Якщо таких процесів ні, системний виклик повертає значення 0.

Системний виклик *write*

Ситуація	Поводження
Спроба записати в канал зв'язку менше байт, чим залишилося до його заповнення.	Необхідна кількість байт поміщається в канал зв'язку, повертається записана кількість байт.
Спроба записати в канал зв'язку більше байт, чим залишилося до його заповнення. Блокування виклику дозволене.	Виклик блокується доти, поки всі дані не будуть поміщені в канал зв'язку. Якщо розмір буфера каналу зв'язку менше, ніж передана кількість інформації, то виклик тим самим буде чекати, поки частина інформації не буде зчитана з каналу зв'язку. Повертається записана кількість байт.
Спроба записати в канал зв'язку більше байт, чим залишилося до його заповнення, але менше, ніж розмір буфера каналу зв'язку. Блокування виклику заборонене.	Системний виклик повертає значення *-1* і встановлює змінну errno у значення EAGAIN.
У каналі зв'язку є місце. Спроба записати в канал зв'язку більше байт, чим залишилося до його заповнення, і більше, ніж розмір буфера каналу зв'язку. Блокування виклику заборонене.	Записується стільки байт, скільки залишилося до заповнення каналу. Системний виклик повертає кількість записаних байт.
Спроба запису в канал зв'язку, у якому немає місця. Блокування виклику не дозволене.	Системний виклик повертає значення -1 і встановлює змінну *errno* у значення EAGAIN.
Спроба запису в канал зв'язку, з якого немає кому більше читати, або повне закриття каналу на читання під час блокування системного виклику.	Якщо виклик був заблокований, то він розблокується. Процес одержує сигнал SIGPIPE. Якщо цей сигнал обробляється користувачем, то системний виклик поверне значення *-1* й установить змінну *errno* у значення EPIPE.

Необхідно відзначити додаткову особливість системного виклику *write* при роботі з *pipe*'ами й FIFO. Запис інформації, розмір якої не перевищує розмір буфера, повинна здійснюватися атомарно - одним підряд лежачим шматком. Цим порозумівається ряд блокувань і помилок у попередньому переліку.

Завдання

Написати програму, яку можна запускати у двох терміналах (0 й 1). У терміналах по черзі пишуться повідомлення. Якщо написали повідомлення в терміналі 0, воно повинне бути надруковане в терміналі 1, і навпаки.

Програма створює два іменованих канали, перший призначений для читання, а другий для запису, або навпаки, залежно від черги посилати повідомлення або читати повідомлення. Програма працює, поки як повідомлення не буде отримане спеціальне слово завершення.

Контрольні питання

1. Опишіть потокову модель передасі даних між процесами.
2. Опишіть системний виклик *pipe*.
3. Опишіть функцію *fork*().
4. Опишіь, що отримує процес-нащадок від процесу-батька.

Лабораторна робота №5
Взаємодія процесів за допомогою черг повідомлень

Мета роботи: навчитися використовувати функції для створення черг повідомлень і пересилання повідомлень.

Теоретичні відомості

Черги повідомлень, як і семафори, і поділювана пам'ять, є засобом зв'язку з непрямою адресацією, вимагають ініціалізації для організації взаємодії процесів і спеціальних дій для звільнення системних ресурсів по закінченню взаємодії.

Зовнішнє ім'я черги повідомлень є цілим числом, що міститься в таблиці імен IPC. Цей ключ указується при створенні черги повідомлень і при кожнім звертанні до неї для використання.

Створення черги повідомлень

Функція *msgget*() має дві мети. Вона використовується для одержання доступу до існуючої черги й може створити неіснуючу чергу. Щоб створити нову чергу, потрібно викликати *msgget* () з наступними аргументами:

key_t *key*	Цілочисленнй ключ, що не визначений у цей момент.
int *msgflag*	Набір прапорів, що включає IPC_CREAT і може включати IPC_EXCL. Це значення також містить біти прав доступу.

Цілочислене значення, що повертаєся у випадку успішного завершення системного виклику, є ідентифікатор черги повідомлень (*msqid*). У випадку невдачі результат дорівнює -1.

Новий ідентифікатор *msqid*, черга повідомлень й асоційована з нею структура даних виділяються в кожному із двох випадків:

− Значення ключа *key* дорівнює IPC_PRIVATE.

− Ключ *key* ще не має асоційованого з ним ідентифікатора черги повідомлень і вираження (*msgflg* & IPC_CREAT) істинно.

Бібліотечний виклик *ftok* використовує інформацію inode заданого файлу й одержує унікальний ідентифікатор для створення ключа, що не змінюється увесь час, тобто поки існує файл, ідентифікатор залишається постійним. Таким чином, два процеси можуть використати один файл для створення однакового IPC-ключа.

При використанні прапора IPC_EXCL у сполученні з IPC_CREAT системний виклик *msgget* завершується невдачею в тім і тільки в тому випадку, коли із зазначеним ключем *key* уже асоційований ідентифікатор. Прапор IPC_EXCL необхідний, щоб запобігти ситуації, коли процес думає, що одержав новий (унікальний) ідентифікатор черги повідомлень, у той час як це не так. Іншими словами, коли використаються й IPC_CREAT й IPC_EXCL, при успішному завершенні системного виклику обов'язково повертається новий ідентифікатор *msqid*.

Ціле число, передане як аргумент *msgflg*, зручно розглядати як восьмеричне. Воно задає права на виконання операцій і прапори.

Права на виконання операцій є права на читання із черги й запис у неї (тобто на прийом/посилку повідомлень) для власника, членів групи й інших користувачів. У наступній таблиці зведені можливі елементарні права й відповідні їм восьмеричні значення:

Права на операції	Восьмеричне значення	Права на операції	Восьмеричне значення
Читання для власника	0400	Запис для групи	0020
Запис для власника	0200	Читання для інших	0004
Читання для групи	0040	Запис для інших	0002

Потрібна комбінація прав задається як результат побітного АБО значень, що відповідають елементарним правам. Так, правам на читання / запис для власника й на читання для членів групи й інших користувачів відповідає восьмеричне число 0644 (аналогія із правами доступу до файлів).

Доступ до існуючої черги

Коли програма очікує, що черга існує, вона викликає msgget(), передаючи очікуване значення ключа й пропускаючи прапор IPC_CREAT. Якщо черга не існує або для ідентифікатора користувача й групи процесу не дозволений доступ до черги, повертається помилка.

Зміна черги повідомлень

Функція *msgctl*() можна використати для зміни чотирьох атрибутів черги після її створення або доступу до неї:

− ідентифікатор користувача
− ідентифікатор групи, який належить черга
− право доступу
− обмеження на загальний розмір всіх повідомлень у черзі.

Обмеження розміру для нової черги встановлюється рівним системному обмеженню. Це визначає, скільки повідомлень у черзі можуть очікувати до прийому. Це, у свою чергу, визначає, наскільки процес відправлення повідомлень може випереджати процес читання повідомлень. Можна зменшити межу, щоб наблизити швидкість процесу, що відправляє, до швидкості приймаючого процесу.

Видалення черги повідомлень

Чергу повідомлень можна видалити за допомогою команди ipcrm або шляхом виклику *msgctl* () і передачі коду команди IPC_RMID. У багатьох випадках черга повідомлень призначена для використання тільки в рамках однієї програми й повинна бути вилучена при завершенні роботи програми.

Відправлення повідомлення

Для відправлення повідомлення використовується функція *msgsnd*() з наступними аргументами:

int *msqid*	ідентифікатор черги повідомлень, повернутий функцією *msgget*
void **msgptr*	покажчик на одержуване повідомлення
size_t *msg_sz*	розмір повідомлення, на яке вказує *msgptr*
int *msgflg*	прапор

Покажчик *msgptr* указує на структуру наступного шаблона:

```
struct msgbuf {
        long mtype; /* тип повідомлення, повинен бути > 0 */
        char mtext[L]; /* дані */
};
```

Прапор може містити IPC_NOWAIT. Повідомлення в черзі зберігаються в послідовності прибуття в межах типів.

Повідомлення копіюється з буфера викликаючого об'єкта, тому буфер можна використовувати повторно відразу після успішного відправлення. Якщо черга заповнена, функція *msgsnd*() блокується, якщо не переданий прапор IPC_NOWAIT.

Одержання повідомлення

Для одержання повідомлення використовується функція *msgrcv*() з наступними аргументами:

int *msqid*	ідентифікатор черги повідомлень, повернутий функцією *msgget*
void **msgptr*	покажчик на одержуване повідомлення
size_t *msg_sz*	розмір повідомлення, на яке вказує *msgptr*
long int *msgtype*	тип повідомлення
int *msgflg*	прапор

Параметр *msgtype* дозволяє реалізувати просту форму пріоритетного одержання. Значення типу може бути 0, щоб указати «будь-який тип», або це може бути певний (позитивний) номер типу n, щоб вибрати перше повідомлення (саме «старе») цього типу. Також це може бути негативне значення -n для вказівки «будь-якого типу більше або рівного n».

Якщо черга порожня, функція *msgrcv*() блокується, якщо не передається прапор IPC_NOWAIT.

Якщо розмір буфера повідомлення менше розміру повідомлення, повертається помилка, якщо не переданий прапор IPC_NOERROR. Потім повідомлення просто усікається, щоб відповідати буферу.

Приклад 1.

Програма дозволяє створити чергу повідомлень із командного рядка. Підтримуються наступні аргументи командного рядка:

-k *key*	Ключ черги повідомлень, наприклад, -k 99.
-p *perms*	Права доступу для установки, наприклад, -p 0664.
-x	Використати прапор IPC_EXCL з *msgget*().
-c	Використати прапор IPC_CREAT з *msgget*().

Якщо аргумент *-k* опущений, програма використає закритий ключ й у такий спосіб створює чергу повідомлень, яку можна використовувати тільки із цієї програми.

```
#include <sys/msg.h>
#include <unistd.h>     /* for getopt() */
#include <errno.h>
#include <stdio.h>
int main(int argc, char **argv)
{
  key_t key = IPC_PRIVATE;
  int perms = 0600;
  int msgflg = 0;
  int msqid;
  struct msqid_ds buf;
  int c;
```

```
  while ( -1 != (c = getopt(argc,argv,"k:p:xc")) )
  {
    switch (c)
    {
    case 'k': /* ключ */
      key = (key_t) strtoul(optarg, NULL, 0);
      break;
    case 'p': /* доступ */
      perms = (int) strtoul(optarg, NULL, 0);
      break;
    case 'c':
      msgflg |= IPC_CREAT;
      break;
    case 'x':
      msgflg |= IPC_EXCL;
      break;
    default:
      return -1;
    }
  }
  msqid = msgget (key, msgflg|perms);
  if (-1 != msqid)
  {
    printf("msqid = 0x%04x. ",msqid);
    if (-1 != msgctl(msqid,IPC_STAT,&buf))
    {
      printf("owner = %d. %d, perms = %04o, max bytes = %d\n",
            buf.msg_perm.uid,
            buf.msg_perm.gid,
            buf.msg_perm.mode,
            buf.msg_qbytes);
      printf("%d msgs = %d bytes on queue\n",
            buf.msg_qnum, buf.msg_cbytes);
    }
    else
      perror("\nmsgctl()");
  }
  else
    perror("msgget()");
}
```

Приклад 2.

Програма відправляє одне або кілька повідомлень заданої довжини й типу в чергу повідомлень. Підтримуються наступні аргументи командного рядка:

-k *key*	Ключ черги повідомлень, наприклад, -k 99.
-i *id*	ID черги повідомлень, альтернатива ключу, наприклад, -i 80.
-c *count*	Кількість повідомлень, що відправляють. За замовчуванням 1.
-t *type*	Числовий тип повідомлення, що відправляється. Типи менше 1 відхиляються *msgsnd*().
-b *bytes*	Розмір кожного повідомлення, наприклад, -b 0x200.

-n	Використати прапор IPC_NOWAIT з msgsnd().

Програма відправляє стільки повідомлень, скільки зазначене, кожне із зазначеним типом і розміром. Перші 32 байта кожного повідомлення - це рядок, що друкується, який містить порядковий номер, дату й час. Повідомлення доповнюється до зазначеного розміру двійковим кодом 0.

```
#include <sys/msg.h>
#include <unistd.h>
#include <errno.h>
#include <time.h>
#include <stdio.h>
int main(int argc, char **argv)
{
  key_t key;
  int msqid = -1;
  int msgflg = 0;
  long type = 1;
  size_t bytes = 64;
  int count = 1;
  int c;
  struct msgspace { long type; char text[32]; } *msg;
  while ( -1 != (c = getopt(argc,argv,"k:i:t:b:c:n")) )
  {
    switch (c)
    {
    case 'k':
      key = (key_t) strtoul(optarg, NULL, 0);
      break;
    case 'i':
      msqid = (int) strtoul(optarg, NULL, 0);
      break;
    case 't':
      type = strtoul(optarg, NULL, 0);
      break;
    case 'b':
      bytes = strtoul(optarg, NULL, 0);
      if (bytes<32) bytes = 32;
      break;
    case 'c':
      count = strtoul(optarg, NULL, 0);
      if (count > 99999) count = 99999;
      break;
    case 'n':
      msgflg |= IPC_NOWAIT;
      break;
    default:
      return -1;
    }
  }
  msg = (struct msgspace *)calloc(1,sizeof(long)+bytes);
  if (-1 == msqid) /* no id given, try key */
    msqid = msgget (key, 0);
```

```
  if (-1 != msqid)
  {
    const time_t tm = time(NULL);
    char stime[26];
    (void)ctime_r (&tm,stime);
    stime[24] = '\0';
    for( c=1; c<=count; ++c)
    {
      msg->type = type;
      sprintf(msg->text,"%05d %s",c,stime);
      if (-1 == msgsnd(msqid,msg,bytes,msgflg))
      {
        perror("msgsnd()");
        break;
      }
    }
  }
  else
    perror("msgget()");
}
```

Приклад 3.

Програма одержує повідомлення із зазначеної черги. Наступні аргументи використовуються в декількох програмах:

-k *key*	Ключ черги повідомлень, наприклад, -k 99.
-i *id*	ID черги повідомлень, альтернатива ключу, наприклад, -i 80.
-c *count*	Кількість повідомлень, які потрібно спробувати одержати.
-b *bytes*	Максимальний розмір повідомлення, наприклад, -b 0x200.
-n	Використати прапор IPC_NOWAIT з *msgrcv*().
-e	Використати прапор MSG_NOERROR з *msgrcv*(),усікання повідомлень длиніше, чим *bytes*.
-q	Не показувати отримане повідомлення (використовується для тестування продуктивності)

При одержанні кожного повідомлення відображаються його порядковий номер, тип повідомлення, перші 32 байта тексту, якщо повідомлення починається з ASCII.

```
#include <sys/msg.h>
#include <unistd.h>
#include <errno.h>
#include <ctype.h>
#include <stdio.h>
int main(int argc, char **argv)
{
  key_t key;
  int msqid = -1;
  int msgflg = 0;
  long type = 0;
  size_t bytes = 64;
  int count = 1;
  int quiet = 0;
  int c;
```

```
struct msgspace { long type; char text[32]; } *msg;
while ( -1 != (c = getopt(argc,argv,"k:i:t:b:c:enq")) )
{
  switch (c)
  {
  case 'k':
    key = (key_t) strtoul(optarg, NULL, 0);
    break;
  case 'i':
    msqid = (int) strtoul(optarg, NULL, 0);
    break;
  case 't':
    type = strtol(optarg, NULL, 0);
    break;
  case 'b':
    bytes = strtoul(optarg, NULL, 0);
    break;
  case 'c':
    count = strtoul(optarg, NULL, 0);
    break;
  case 'n':
    msgflg |= IPC_NOWAIT;
    break;
  case 'e':
    msgflg |= MSG_NOERROR;
    break;
  case 'q':
    quiet = 1;
    break;
  default:        return -1;
  }
}
if (-1 == msqid) /* no id given, try key */
  msqid = msgget (key, 0);
msg = (struct msgspace *)calloc(1,sizeof(long)+bytes);
if (-1 != msqid)
{
  for( c=1; c<=count; ++c)
  {
    int ret = msgrcv(msqid,msg,bytes,type,msgflg);
    if (ret >= 0) /* got a message */
    {
      if (!quiet)
      {
        if (isascii(msg->text[0]))
          printf("%d: type %ld  len %d text %-32.32s\n",
                c,   msg->type,  ret,      msg->text);
        else
          printf("%d: type %ld len %d (nonascii)\n",
                c,   msg->type,  ret);
      }
    }
```

```
        else /* an error, end loop */
        {
          perror("msgrcv()");
          break;
        }
      } /* for c<=count */
    } /* good msgget */
    else
      perror("msgget()");
}
```

Завдання

Написати код для реалізації системи Клієнт/Сервер.

Сервер отримує повідомлення від клієнтів, тип повідомлення задається при запуску клієнта (число від 1 до 10).

Сервер друкує отримане повідомлення із вказівкою типу й часу одержання.

Сервер підтверджує одержання повідомлення від клієнта, переславши йому відповідне повідомлення. Після підтвердження клієнт передає серверу наступне повідомлення.

Сервер припиняє спілкування із кожним клієнтом після одержання N (задається при запуску сервера) повідомлень від клієнта, послави клієнтові повідомлення з визначеним текстом.

Клієнт завершує роботу після одержання від сервера повідомлення з визначеним текстом.

Контрольні питання

5. Назвіть способи комунікації між процесами.
6. Опишіть, яким чином створюються черги повідомлень.
7. Назвіть атрибути черги, які можна змінювати після її створення.
8. Опишіть функцію для відправлення повідомлень в чергу.
9. Опишіть функцію для отримання повідомлень в черзі.

Лабораторна робота №6
Робота с файлами

Мета роботи: навчитися працювати з файлами.

Теоретичні відомості

Для програміста відкритий файл представляється як послідовність зчитувальних або записуваних даних. При відкритті файлу з ним зв'язується потік вводу-виводу. Інформація, яка виводиться, записується в потік, інформація, яка вводиться, зчитується з потоку.

Коли потік відкривається для вводу-виводу, він зв'язується зі стандартною структурою типу FILE, що визначена в <stdio.h>. Структура FILE містить необхідну інформацію про файл.

Відкриття й закриття файлу

Відкриття файлу здійснюється за допомогою функції fopen(), що повертає покажчик на структуру типу FILE, якому можна використовувати для наступних операцій з файлом.
FILE *fopen(const char *name, const char *type);

name ім'я файлу, що відкривається, (включаючи шлях),

type покажчик на рядок символів, що визначають спосіб доступу до файлу:

режим			опис	починає з..…
r	rb		відкриває для читання	початку
w	wb		відкриває для запису (створює файл у випадку його відсутності). Видаляє вміст і перезаписує файл.	початку
a	ab		відкриває для додавання (створює файл у випадку його відсутності)	кінця
r+	rb+	r+b	відкриває для читання й запису	початку
w+	wb+	w+b	відкриває для читання й запису. Видаляє вміст і перезаписує файл.	початку
a+	ab+	a+b	відкриває для читання й запису (додає у випадку існування файлу)	кінця

Значення «b» зарезервовано для двійкового режиму С.

Значення, що повертається, покажчик на відкритий потік. Якщо виявлено помилку, то повертається значення NULL.

Функція fclose(FILE *fp) закриває потік fp, пов'язаний з відкритим за допомогою функції fopen() файлом.

Значення, що повертається: 0, якщо потік успішно закритий; константа EOF, якщо відбулася помилка.

```
#include <stdio.h>
int main() {
  FILE *fp;
  char name[]="s1.txt";
  if((fp = fopen(name, "r"))!=NULL) { // відкрити файл удалося?
...                 // необхідні дії над даними
  fclose(fp);
 }
 else
     printf("Не вдалося відкрити файл");
  return 0;
}
```

Читання і запис

Читання символу виконується за допомогою функції

char fgetc(FILE *fp);

Функція повертає код зчитаного символу. Якщо досягнуть кінець файлу або виникла помилка, повертається константа EOF.

Запис символу у файл:

fputc(int c, FILE *fp);

Функція повертає код зчитаного символу c з потоку fp.

Функції fscanf() і fprintf() аналогічні функціям scanf() і printf(), але працюють із файлами даних, і мають перший аргумент покажчик на файл.

fscanf(FILE *fp, "ФорматВводу ", аргументи);

fprintf(FILE *fp, "ФорматВиводу", аргументи);

Функції fgets() і fputs() призначені для вводу-виводу рядків, вони є аналогами функцій gets() і puts() для роботи з файлами.

fgets(char *s, int n, FILE *fp);

Символи читаються з потоку доти, поки не буде прочитаний символ нового рядка '\n', що включається в рядок, або поки не наступить кінець потоку EOF або не буде прочитане максимально символів n. Результат поміщається в покажчик на рядок і закінчується нуль-символом '\0'. Функція повертає адресу рядка.

fputs(const char *s, FILE *fp);

Копіює рядок s у потік fp з поточної позиції. Завершальний нуль-символ не копіюється.

Приклад 1.

Увести число й зберегти його у файлі s1.txt. Зчитати число з файлу s1.txt, збільшити його на 3 і зберегти у файлі s2.txt.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    FILE *S1, *S2;
    int x, y;
    printf("Уведіть число: ");
    scanf("%d", &x);
    S1 = fopen("S1.txt", "w");
    fprintf(S1, "%d", x);
    fclose(S1);
    S1 = fopen("S1.txt", "r");
    S2 = fopen("S2.txt", "w");
    fscanf(S1, "%d", &y);
    y += 3;
    fclose(S1);
    fprintf(S2, "%d\n", y);
    fclose(S2);
    return 0;
}
```

Результат виконання 2 файли.

Файлова система ANSI C надає дві функції, що дозволяють читати й писати блоки даних *fread*() і *fwrite*(). Вони мають наступні прототипи:

size_t fread(void **буфер*, size_*t число_байт*, size_t *обсяг*, FILE **fp*);

size_t fwrite(const void **буфер*, size_t *число_байт*, size_t *обсяг*, FILE **fp*);

У випадку *fread*() буфер це покажчик на область пам'яті, що одержує дані з файлу. У випадку *fwrite*() *буфер* це покажчик на інформацію, записувану у файл. Довжина кожного

елемента в байтах визначається в *число_байт*. Аргумент *обсяг* визначає, скільки елементів (кожен довжиною число_байт) буде прочитане або записане. Нарешті, *fp* це файловий покажчик на раніше відкритий потік.

Функція *fread*() повертає число прочитаних елементів. Дане значення може бути менше, ніж обсяг, якщо був досягнутий кінець файлу або відбулася помилка. Функція *fwrite*() повертає число записаних елементів. Дане значення дорівнює обсяг, якщо не виникла помилка.

Якщо файл був відкритий для двійкових даних, то *fread*() і *fwrite*() можуть читати й писати будь-який тип інформації.

Приклад 2.

Записати float у файл:

```
/* запис речовинних чисел у файл */
#include <stdio.h>
int main(void)
{
FILE *fp;
float f = 12.23;
if((fp=fopen("s1.txt", "wb"))==NULL) {
printf("Cannot open file.");
return 1;
}
fwrite(&f, sizeof (float), 1, fp);
fclose (fp);
return 0;
}
```

Як ілюструє дана програма, буфер може бути (а найчастіше це так й є) простою змінною.

Одне з найбільш корисних застосувань fread() і fwrite() це читання й запис блоків даних типу масивів або структур.

Приклад 3.

Записати вміст масиву речовинних чисел balance у файл balance, використовуючи один оператор fwrite(). Далі прочитати масив, використовуючи один оператор fread(), і вивести його вміст.

```
#include <stdio.h>
int main(void)
{
int i;
FILE *fp;
float balance[100];
/* відкриття на запис */
if((fp=fopen("balance.txt", "wb"))==NULL) {
printf("Cannot open file.");
return 1;
}
for(i=0; i<100; i++) balance[i] = (float) i;
/* збереження за раз усього масиву balance */
fwrite(balance, sizeof(balance), 1, fp) ;
fclose(fp);
/* обнуління масиву */
for(i=0; i<100; i++) balance[i] = 0.0;
/* відкриття для читання */
if((fp=fopen("balance.txt","rb"))==NULL) {
```

```
printf("cannot open file");
return 1;
}
/* читання за раз усього масиву balance */
fread(balance, sizeof( balance), 1, fp);
/* вивід умісту масиву */
for(i=0; i<100; i++) printf("%f  ", balance [i]);
fclose(fp);
return 0;
}
```

Використання *fread*() і *fwrite*() для читання або запису складних даних більш ефективно, ніж використання повторюваних викликів *getc*() і *putc*().

Установка позиції в потоці даних

Виконується функцією

int fseek(FILE *stream, long int offset, int whence);

Аргументи:

stream покажчик на керуючу таблицю потоку даних.

offset зсув позиції.

whence крапка відліку зсуву.

Значення, що повертається:

0 при успішній установці позиції.

Відмінне від нуля значення, якщо при роботі функції відбулися помилки. При цьому змінній ernno буде привласнений код помилки:

[EINVAL] невірне значення аргументу whence

[ESPIPE] неприпустиме значення аргументу offset

Функція *fseek*() установлює позицію в потоці даних, заданим аргументом *stream*. Щодо встановленої позиції буде здійснюватися читання й запис даних.

Точка відліку встановлюваної позиції визначається аргументом *whence*, що може приймати значення:

SEEK_SET зсув відраховує від початку файлу

SEEK_CUR зсув відраховує від поточної позиції

SEEK_END зсув відраховує від кінця файлу

Зсув задається аргументом *offset*, причому позитивне значення аргументу означає зсув вправо від зазначеної аргументом whence позиції, негативне зсув уліво.

Для двійкових потоків даних, зсув (*offset*) це кількість байт.

Для текстових потоків даних зсув (*offset*) повинен дорівнювати 0, або отриманий за допомогою функції *ftell*(), при цьому крапка відліку (*whence*) повинна мати значення SEEK_SET.

Приклад 4.

Зчитати рядок з початку файлу, а потім з позиції, зміщеної відносно початку файлу на 5 байт. Зсув задається за допомогою функції *fseek*. У файлі записаний наступний рядок: 123456789.

```
#include <stdio.h>
int main (void)
{
  FILE *mf;
  char str[50];

  printf ("File open: ");
  mf=fopen ("test.txt","r+");
  if (mf == NULL) printf ("error\n");
```

```
  else printf (" done\n");

  printf ("Set on start position:");
  if (fseek (mf,0,SEEK_SET)==0)
    printf (" done\n");
  else
    printf (" error\n");

  printf ("Read string: ");
  if (fgets (str, sizeof (str), mf)==NULL)
    printf (" no read\n");
  else
    printf ("%s\n",str);

  printf ("Set on bite No.5: ");
  if (fseek (mf,5,SEEK_SET)==0)
    printf (" done\n");
  else
    printf ("error\n");

  printf ("Read string: ");
  if (fgets (str, sizeof (str), mf)==NULL)
    printf (" no read\n");
  else
    printf (" %s\n",str);

  printf ("File close: ");
  if ( fclose (mf) == EOF) printf ("error\n");
  else printf (" done\n");

  return 0;
}
```

Завдання

Написати утиліту мовою С, що працює в середовищі Linux.

Програма виводить на екран уміст текстових файлів, які користувач указує як аргументи під час запуску програми.

>./programma file1.txt file2.txt file3.txt (довільна кількість)

Програма повинні задовольняти наступним вимогам:

− аналізувати кількість аргументів командного рядка (argc) і завершувати роботу із сигналізацією помилки, якщо вони відсутні (exit);

− для кожного отриманого аргументу (argv[]) запускати свою функцію по виводу вмісту файлу, передаючи їй це ім'я;

− функція повинна вивести на екран ім'я файлу, що вона обробляє;

− функція виводу повинна відкривати файл для читання в текстовому режимі, обробляючи можливі помилки;

− функція виконує читання тексту з файлу й виводить його на екран;

− якщо зазначено ключ -Z n, те на екран виводиться кожен n-й символ файлу;

− прочитаний до кінця файл необхідно закрити.

Контрольні питання

10. Опишіть поняття файлу з точки зору операційної системи.
11. Назвіть операції, які можна виконувати над файлом.

12. Опишіть функції для читання та запису з файлу.
13. Назвіть варіанти та способи встановлення точки відліку у файлі.

Лабораторна робота №7
Сигнали в UNIX

Мета роботи: навчитися генерувати й обробляти сигнали.

Теоретичні відомості

Апаратні переривання (interrupt), виключення (exception), програмні переривання (trap, software interrupt). Їхня обробка

Після видачі запиту вводу-виводу в процесора існує два способи довідатися про те, що обробка запиту пристроєм завершена. Перший спосіб полягає в регулярній перевірці процесором біта зайнятості в регістрі стану контролера відповідного пристрою (polling). Другий спосіб складається у використанні переривань. При другому способі процесор має спеціальний вхід, на який пристрою вводу-виводу безпосередньо, або використовуючи контролер переривань, виставляють сигнал запиту переривання (interrupt request) при завершенні операції вводу-виводу. При наявності такого сигналу, процесор, після виконання поточної команди, не виконує наступну, а, зберігши стан ряду регістрів й, можливо, завантаживши в частину регістрів нові значення, переходить на виконання команд, розташованих по деяких фіксованих адресах. Після закінчення обробки переривання можна відновити стан процесора й продовжити його роботу з команди, виконання якої було відкладено.

Аналогічний механізм часто використається при обробці виняткових ситуацій (exception), які виникають при виконанні команди процесором (неправильна адреса в команді, захист пам'яті, виникло ділення на нуль і т.д.). У цьому випадку процесор не закінчує виконання команди, а робить, як і при перериванні, зберігаючи свій стан до моменту початку її виконання. Цим же механізмом часто користуються й для реалізації так званих програмних переривань (software interrupt, trap), використовуваних, наприклад, для перемикання процесора з режиму користувача в режим ядра усередині системних викликів. Для виконання дій, аналогічних діям по обробці переривання, процесор повинен виконати спеціальну команду.

Як правило, обробку апаратних переривань від пристроїв вводу-виводу робить сама операційна система, не довіряючи роботу із системними ресурсами процесам користувача. Обробка ж виняткових ситуацій і деяких програмних переривань цілком може бути покладена на користувальницький процес через механізм сигналів.

Поняття сигналу. Способи виникнення сигналів і види їхньої обробки

Сигнали - це засіб, за допомогою якого процесам можна передати повідомлення про деякі події в системі. Самі процеси теж можуть генерувати сигнали, за допомогою яких вони передають певні повідомлення ядру й іншим процесам. За допомогою сигналів можна здійснювати такі акції керування процесами, як припинення процесу, запуск припиненого процесу, завершення роботи процесу. Усього в Linux існує 63 різних сигналів, їхній перелік можна подивитися по команді

```
$ kill -l
```

З погляду користувача одержання процесом сигналу виглядає як виникнення переривання. Процес припиняє своє регулярне виконання, і керування передається механізму обробки сигналу. По закінченні обробки сигналу процес може відновити регулярне виконання. Сигнали прийнято позначати номерами або символічними іменами. Усі імена починаються на SIG, але цю приставку іноді опускають: наприклад, сигнал з номером 1 позначають або як SIGHUP, або просто як HUP.

Процес може одержати сигнал від:

1. Hardware (при виникненні виняткової ситуації).
2. Іншого процесу, що виконав системний виклик передачі сигналу.
3. Операційної системи (при настанні деяких подій).

4. Термінала (при натисканні певної комбінації клавіш).
5. Системи керування завданнями (при виконанні команди kill).

Існує **три варіанти реакції процесу на сигнал**:

1.Примусово проігнорувати сигнал.

2.Зробити обробку за замовчуванням (проігнорувати, зупинити процес (перевести в стан очікування до одержання іншого спеціального сигналу), або завершити роботу з утворення core файлу або без оного).

3.Виконати обробку сигналу, специфіковану користувачем.

Ігнорований сигнал просто відкидається процесом і не робить на нього ніякого впливу. Блокований сигнал ставиться в чергу на видачу, але ядро не жадає від процесу ніяких дій до розблокування сигналу. Після розблокування сигналу програма його обробки викликається тільки один раз, навіть якщо протягом періоду блокування даний сигнал надходив кілька разів.

Змінити реакцію процесу на сигнал можна за допомогою спеціальних системних викликів. Реакція на деякі сигнали не допускає змін, і вони можуть бути оброблені тільки за замовчуванням. Так, наприклад, сигнал з номером 9 - SIGKILL обробляється тільки за замовчуванням і завжди приводить до завершення процесу.

Сигнали

N	Ім'я	Опис	Можна перехо-плюва-ти	Можна блоку-вати	Комбіна-ція клавіш
1	HUP	Hangup. Відбій	Так	Так	
2	INT	Interrupt. У випадку виконання простих команд викликає припинення виконання, в інтерактивних програмах - припинення активного процесу	Так	Так	‹Ctrl›+‹C› або ‹Del›
3	QUIT	Як правило, сильніше сигналу Interrupt	Так	Так	‹Ctrl›+‹\›
4	ILL	Illegal Instruction. Центральний процесор зштовхнувся з незнайомою командою (у більшості випадків це означає, що допущено програмну помилку). Сигнал відправляється програмі, у якій виникла проблема	Так	Так	
8	FPE	Floating Point Exception. Обчислювальна помилка, наприклад, ділення на нуль	Так	Так	
9	KILL	Завжди припиняє виконання процесу	Ні	Ні	
11	SEGV	Segmentation Violation. Доступ до недозволеної області пам'яті	Так	Так	
13	PIPE	Була почата спроба передачі даних за допомогою конвеєра або черги FIFO, однак не існує процесу, здатного прийняти ці дані	Так	Так	
15	TERM	Software Termination. Вимога закінчити процес (програмне завершення)	Так	Так	
17	CHLD	Зміна статусу породженого процесу	Так	Так	
18	CONT	Продовження виконання припиненого процесу	Так	Так	
19	STOP	Припинення виконання процесу	Ні	Ні	
20	TSTR	Сигнал останова, який генерується клавіатурою. Переводить процес у фоновий режим	Так	Так	‹Ctrl›+‹Z›

Важливим питанням при програмуванні з використанням сигналів є питання про збереження реакції на них при народженні нового процесу або заміні його користувальницького контексту. При системному виклику *fork*() всі встановлені реакції на

сигнали успадковуються породженим процесом. При системному виклику exec() зберігаються реакції тільки для тих сигналів, які ігнорувалися або оброблялися за замовчуванням. Одержання будь-якого сигналу, що до виклику exec() оброблявся користувачем, приведе до завершення процесу.

Системний виклик *kill* і команда *kill*()

Послати сигнал процесу можна двома способами - за допомогою команди *kill* і за допомогою системного виклику *kill*(). Команда *kill* звичайно використається в наступній формі:

kill [-*номер*] *pid*

Тут *pid* - це ідентифікатор процесу, якому посилається сигнал, а "*номер*" - номер сигналу, що посилається процесу. Послати сигнал (якщо немає повноважень суперкористувача) можна тільки процесу, у якого ефективний ідентифікатор користувача збігається з ідентифікатором користувача, що посилає сигнал. Якщо параметр -номер відсутній, то посилає сигнал SIGTERM, що звичайно має номер 15 і реакція на нього за замовчуванням - завершити роботу процесу, що одержав сигнал.

При використанні системного виклику *kill*() послати сигнал (якщо немає повноважень суперкористувача) можна тільки процесу або процесам, у яких ефективний ідентифікатор користувача збігається з ефективним ідентифікатором користувача процесу, що посилає сигнал.

Вивчення особливостей одержання термінальних сигналів поточною й фоновою групою процесів

Нижче наведена програма, у якій процес породжує дитину й вони обоє зациклюються.

Приклад 1.

```
#include <unistd.h>
int main(void){
 (void)fork();
        while(1);
        return 0;
}
```

Команда

```
ps -ej
```

дозволяє одержати інформацію про всі процеси в системі й довідатися їхні ідентифікатори, ідентифікатори груп процесів і сеансів, термінал сеансу, що управляє, й до якої групи процесів він приписаний.

Після виконання команди буде отриманий список всіх процесів у системі. Колонка PID містить ідентифікатори процесів, колонка PGID - ідентифікатори груп, до яких вони належать, колонка SID - ідентифікатори сеансів, колонка TTY - номер відповідного керуючого термінала, колонка TPGID (може бути присутнім не для всіх версій UNIX, але в Linux є) - до якої групи процесів приписаний керуючий термінал.

Якщо запустити програму на виконання й запустити команду “ps -ej” з іншого екрана, можна побачити два запущених процеси. Можна перевірити реакцію програми на сигнали SIGINT - клавіші <CTRL> й <C> - і сигнал SIGQUIT - клавіші <CTRL> й <4>.

Ліквідувати процеси можна за допомогою команди *kill*.

Системний виклик *signal*(). Установка власного оброблювача сигналу

Одним зі способів зміни поводження процесу при одержанні сигналу в операційній системі UNIX є використання системного виклику *signal*().

Цей системний виклик має два параметри: один з них задає номер сигналу, реакцію процесу на який потрібно змінити, а другий визначає, як саме потрібно її змінювати. Для першого варіанта реакції процесу на сигнал - його ігнорування - застосовується спеціальне значення цього параметра SIG_IGN. Наприклад, якщо потрібно ігнорувати сигнал SIGINT, починаючи з деякого місця роботи програми, у цьому місці програми варто вжити конструкцію

(void) signal(SIGINT, SIG_IGN);

Для другого варіанта реакції процесу на сигнал - відновлення його обробки за замовчуванням - застосовується спеціальне значення цього параметра SIG_DFL. Для третього варіанта реакції процесу на сигнал як значення параметра підставляється покажчик на користувальницьку функцію обробки сигналу, що повинна мати прототип виду

void *handler(int);

Нижче наведений приклад конструкції для користувальницької обробки сигналу SIGHUP.

Приклад 2.

```
void *my_handler(int nsig) {
<обробка сигналу>
}
int main() {
        ...
(void)signal(SIGHUP, my_handler);
        ...
}
```

Приклад 3.

```
#include <signal.h>
int main(void){
        /* Установка реакції процесу на сигнал SIGINT */
         (void)signal(SIGINT, SIG_IGN);
        /*Починаючи із цього місця, процес буде ігнорувати виникнення сигналу SIGINT */
        while(1);
        return 0;
}
```

Як значення параметра в користувальницьку функцію обробки сигналу (у прикладі - параметр *nsig*) передається номер виниклого сигналу, отже, та сама функція може бути використана для обробки декількох сигналів.

Приклад 4.

```
#include <signal.h>
#include <stdio.h>
/* Функція my_handler - користувальницький оброблювач сигналу */
void my_handler(int nsig){
        printf("Receive signal %d, CTRL-C pressed\n", nsig);
}
int main(void){
 (void)signal(SIGINT, my_handler);
        while(1);
        return 0;
}
```

Відновлення попередньої реакції на сигнал

Системний виклик *signal*() повертає покажчик на попередній оброблювач сигналу, що дозволяє відновлювати перевизначену реакцію на сигнал.

Приклад 5

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

int i=0; /* Лічильник числа обробок сигналу */
void (*p)(int); /* Покажчик, у який буде занесена адреса попереднього оброблювача сигналу */
void my_handler(int nsig){
        printf("Receive signal %d, CTRL-C pressed\n",i++);
        /* Після 5-й обробки повертаємо первісну реакцію на сигнал */
        if(i == 5) (void)signal(SIGINT, p);
}
int main(void){
        /* Установка своєї реакції процесу на сигнал SIGINT (запам'ятовуючи адресу
        попереднього оброблювача) */
        p = signal(SIGINT, my_handler);
        /*Починаючи із цього місця, процес буде 5 разів друкувати повідомлення про
        виникнення сигналу SIGINT */
        while(1);
        return 0;
      }
```

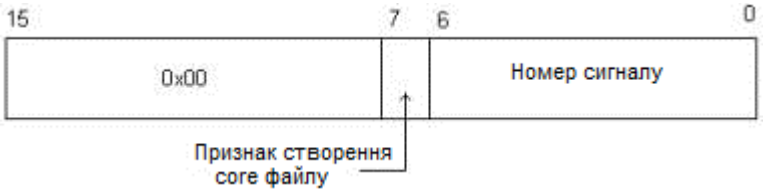
Завершення породженого процесу. Системний виклик *waitpid*(). Сигнал SIGCHLD

Якщо процес-дитина завершує свою роботу раніше процесу-батька, і процес-батько явно не вказав, що він не зацікавлений в одержанні інформації про статус завершення процесу-дитини, то процес, що завершився, не зникає із системи остаточно, а залишається в стані ***закінчив виконання*** (процес-зомбі) або до завершення процесу-батька, або до того моменту, коли батько вирішить отримати цю інформацію.

Для одержання такої інформації процес-батько може скористатися системним викликом *waitpid*() або його спрощеною формою *wait*(). Системний виклик waitpid() дозволяє батьку-процесу-батькові синхронно отримати дані про статус процесу-дитини, що завершилося, або блокуючи процес-батько до завершення процесу-дитини, або без блокування при його періодичному виклику з опцією WNOHANG. Ці дані займають 16 битів і можуть бути розшифровані в такий спосіб:

Якщо процес завершився за допомогою явного або неявного виклику функції *exit*(), то дані виглядають так (старший біт перебуває ліворуч)

Якщо процес був завершений сигналом, то дані виглядають так (старший біт перебуває ліворуч)



Кожна процес-дитина при завершенні роботи посилає своєму процесу-батькові спеціальний сигнал SIGCHLD, на який у всіх процесів за замовчуванням установлена реакція "ігнорувати сигнал". Наявність такого сигналу разом із системним викликом

waitpid() дозволяє організувати асинхронний збір інформації про статус породжених процесів, що завершилися, процесом-батьком.

Використовуючи системний виклик *signal*(), можна явно встановити ігнорування цього сигналу (SIG_IGN), тим самим проінформувавши систему, що нас не цікавить, яким образом завершаться породжені процеси. У цьому випадку зомбі-процесів виникати не буде, але й застосування системних викликів *wait*() і waitpid() буде заборонено.

Прототипи системних викликів

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

Системний виклик *waitpid*() блокує виконання поточного процесу доти, поки або не завершиться породжений їм процес, обумовлений значенням параметра *pid*, або поки поточний процес не одержить сигнал, для якого встановлена реакція за замовчуванням "завершити процес" або реакція обробки користувальницькою функцією. Якщо породжений процес, заданий параметром pid, до моменту системного виклику перебуває в стані ***закінчив виконання***, системний виклик повертається негайно без блокування поточного процесу.

Параметр *pid* визначає породжений процес, завершення якого чекає процес-батько, у такий спосіб:

− Якщо *pid* > 0, очікується завершення процесу з ідентифікатором pid.

− Якщо *pid* = 0, то очікується завершення будь-якого породженого процесу в групі, до якої належить процес-батько.

− Якщо *pid* = -1, то очікується завершення будь-якого породженого процесу.

− Якщо *pid* < 0, але не -1, то очікується завершення будь-якого породженого процесу із групи, ідентифікатор якої дорівнює абсолютному значенню параметра pid.

Параметр *options* може приймати два значення: 0 й WNOHANG. Значення WNOHANG вимагає негайного повернення з виклику без блокування поточного процесу в кожному разі.

Якщо системний виклик виявив породжений процес, що завершився, із числа специфікованих параметром *pid*, то цей процес видаляється з обчислювальної системи, а за адресою, зазначеною в параметрі *status*, заноситься інформація про статус його завершення. Параметр *status* може бути заданий рівним NULL, якщо ця інформація не представляє інтересу.

При виявленні процесу, що завершився, системний виклик повертає його ідентифікатор. Якщо виклик був зроблений із установленою опцією WNOHANG, породжений процес, специфікований параметром *pid*, існує, але ще не завершився, то системний виклик поверне значення 0. У всіх інших випадках він повертає негативне значення. Повернення з виклику, пов'язане з виникненням обробленого користувачем сигналу, може бути в цьому випадку ідентифіковане за значенням системної змінної errno == EINTR, і виклик може бути зроблений знову.

Системний виклик wait є синонімом для системного виклику *waitpid* зі значеннями параметрів *pid* = -1, *options* = 0.

Приклад 6.

```
#include <sys/types.h>
#include <unistd.h>
#include <wait.h>
#include <signal.h>
#include <stdio.h>
/* Функція my_handler - оброблювач сигналу SIGCHLD */
```

```
void my_handler(int nsig){
int status;
pid_t pid;
/* Запит статусу процесу, що завершився, і одночасне одержання його ідентифікатора */
        if((pid = waitpid(-1, &status, 0)) < 0){
                /* Якщо виникла помилка - повідомлення про неї й продовження роботи */
                printf("Some error on waitpid errno = %d\n", errno);
        } else {
        /* Інакше аналіз статусу процесу, що завершився, */
        if ((status & 0xff) == 0) {
                /* Процес завершився з явним або неявним викликом функції exit() */
                printf("Process %dwasexitedwithstatus%d\n",pid, status>>8);
                }elseif ((status&0xff00) == 0){
                        /*Процес був завершений за допомогою сигналу */
                        printf("Process %d killed by signal %d %s\n", pid, status &0x7f,
                         (status & 0x80) ? "with core file" : "without core file");
                }
        }
}

int main(void){
pid_t pid;
/* Установка оброблювача для сигналу SIGCHLD */
(void) signal(SIGCHLD, my_handler);
        /* Породження Child 1 */
        if((pid = fork()) < 0){
                printf("Can\'t fork child 1\n");
                exit(1);
        } elseif (pid==0){
        /*Child1-завершується з кодом 200 */
        exit(200);
        }

        /* Продовження процесу-батька - породжується Child 2 */
        if((pid = fork()) < 0){
                printf("Can\'t fork child 2\n");
                exit(1);
        } else if (pid == 0){
                /* Child 2 - циклиться, необхідно вбивати! */
                while(1);
        }
        /* Продовження процесу-батька - відхід у цикл */
        while(1);
        return 0;
}
```

У цій програмі батько породжує 2 процеси. Один з них завершується з кодом 200, а другий - циклиться. Перед породженням процесів батько встановлює оброблювач переривання для сигналу SIGCHLD, а після їхнього породження йде в нескінченний цикл. В оброблювачі переривання викликається *waitpid*() для будь-якого породженого процесу. Оскільки в оброблювач програма попадає, коли який-небудь із процесів завершився, то системний виклик не блокується, і можна одержати інформацію про ідентифікатор процесу, що завершився, і причині його завершення.

Роботу програми можна перевірити, завершуючи другий породжений процес за допомогою команди *kill* с яким-небудь номером сигналу. Батьківський процес також необхідно завершувати командою *kill*.

Завдання

Модифікувати програму із приклада 3 так, щоб вона перестала реагувати й на натискання клавіш <ctrl> й <4>. Переконатися у відсутності її реакцій на зовнішні подразники (знімати програму з іншого термінала командою kill).

Модифікувати програму із приклада 4 так, щоб вона друкувала повідомлення й про натискання клавіш <ctrl> й <4>. Використовувати ту саму функцію для обробки сигналів SIGINT й SIGQUIT.

Перевірити роботу програми із приклада 6.

Самостійно подивитися описи функцій і системних викликів sigemptyset(), sigaddset(), sigaction(), sigprocmask(). Написати програму, що за допомогою перерахованих функцій додає 3 сигнали для обробки, запам'ятовує сигнали в оброблювачі й наприкінці програми друкує список всіх оброблених сигналів.

Контрольні питання

14. Дайте визначення системного виклику.
15. Наведіть приклади сигналів.
16. Опишіть, яким чином можна викликати очікування одним процесом завершення іншого процесу.

Література

1. Silberschatz A., Gagne G., Galvin P. B. Operating System Concepts, 10th Edition. - John Wiley & Sons, Inc., 2018. - 1040 p.
2. Задерейко О. В., Зіноватна С. Л., Толокнов А. А. Операційні системи : навчальний посібник [Електронне видання] / О. В. Задерейко, С. Л. Зіноватна, А. А. Толокнов. – Одеса : Фенікс, 2022. – 140 с.
3. Зайцев В. Г., Дробязко І. П. Операційні системи: навч. посіб. – Київ: КПІ ім. Ігоря Сікорського, 2019. – 240 с.
4. Авраменко В. С., Авраменко А. С. Основи операційних систем. Навчальний посібник. – Черкаси: ЧНУ імені Богдана Хмельницького, 2018. – 524 с.
5. C Programming tutorial. https://www.unf.edu/~wkloster/2220/ppts/cprogramming_tutorial.pdf
6. Kauffman C. CSCI 4061: Signals and Signal Handlers. 2021. - https://www-users.cse.umn.edu/~kauffman/4061/08-signals.pdf

Навчальне видання

Методичні вказівки до лабораторних робіт з дисципліни «Операційні системи» для здобувачів першого (бакалаврського) рівня вищої освіти за спеціальністю 121 – Інженерія програмного забезпечення

Укладачі: Светлана Леонідівна Зіноватна

Ірина Валентинівна Єгоращенко