Міністерство освіти і науки України
Центральноукраїнський національний технічний університет
Механіко-технологічний факультет
Кафедра кібербезпеки та програмного забезпечення

Операційні системи

Методичні вказівки до виконання лабораторних робіт для студентів денної форми навчання за спеціальністями 122 "Комп'ютерні науки", 123 "Комп'ютерна інженерія" 125 "Кібербезпека та захист інформації"

ЗАТВЕРДЖЕНО
на засіданні кафедри кібербезпеки та програмного забезпечення, протокол №1 від 25.08.2025.

Кропивницький 2025

УДК 004.4
Операційні системи: методичні вказівки до виконання лабораторних робіт для студентів за спеціальністями 122«Комп'ютерні науки», 123«Комп'ютерна інженерія» та 125 «Кібербезпека та захист інформації» / М-во освіти і науки України, Центральноукр. нац. техн. ун-т; [уклад. П. С. Усік, Н.Л. Козірова, Л.І. Поліщук, Р.О. Ткачук] – Кропивницький: ЦНТУ, 2025. – 81 с.

Укладачі: Усік П. С., доктор філософії, старший викладач кафедри кібербезпеки та програмного забезпечення;
Козірова Н.Л., викладач кафедри кібербезпеки та програмного забезпечення, програміст бази даних в I&U Group;
Поліщук Л.І., старший викладач кафедри кібербезпеки та програмного забезпечення;
Ткачук Р.О. асистент кафедри кібербезпеки та програмного забезпечення, Senior DevOps Engineer at Epam

Рецензенти: Смірнов О. А., докт. техн. наук, професор;
Улічев О. С., канд. техн. наук, старший викладач.

ЗМІСТ

ВСТУП

Метою викладання дисципліни «Операційні системи» є формування у здобувачів теоретичної та практичної бази знань в області використання та функціонування операційних систем. Отримання теоретичних понять про механізми функціонування операційних систем. Підсилити практичні навички користування та обслуговування(адміністрування) сучасних найбільш поширених операційних систем.

Завдання:

– Вивчення теоретичних основ функціонування операційних систем.

– Вивчення архітектури, компонентів та типів операційних систем.

– Вивчення механізмів управління процесами, пам’яттю, файлами та пристроями введення/виведення.

– Набуття практичних навичок роботи з операційними системами.

У результаті вивчення навчальної дисципліни студент повинен:

– знати новітні технології в галузі інформаційних технологій; наукові положення, що лежать в основі функціонування комп’ютерних засобів, систем та мереж.

– вміти застосовувати знання для ідентифікації, формулювання і розв’язування технічних задач спеціальності, використовуючи методи, що є найбільш придатними для досягнення поставлених цілей; розв’язувати задачі аналізу та синтезу засобів, характерних для спеціальності; застосовувати знання технічних характеристик, конструктивних особливостей, призначення і правил експлуатації програмно-технічних засобів комп’ютерних систем та мереж для вирішення технічних задач спеціальності; розробляти програмне забезпечення для вбудованих і розподілених застосувань, мобільних і гібридних систем, розраховувати, експлуатувати, типове для спеціальності обладнання; розробляти програмне забезпечення для вбудованих і розподілених застосувань, мобільних і гібридних систем, розраховувати, експлуатувати, типове для спеціальності обладнання.

Структурно логічна схема підготовки бакалавра.

Враховуючи послідовність накопичення знань та інформації, дисципліна вивчається після викладання наступних дисциплін: «Базові методології та технології програмування», «Об’єктно орієнтоване програмування», «Системне програмування».

Для опанування матеріалу дисципліни «Операційні системи» окрім лекційних та лабораторних занять, тобто аудиторного навантаження, значна увага приділяється самостійній роботі.

До основних видів самостійної роботи студента відносимо:

1. Вивчення лекційного матеріалу.

2. Робота з літературними джерелами.

3. Розв’язання практичних задач.

4. Підготовка до модульних, підсумкового контролю, екзамену (денна) та заліку (заочна).

5. Виконання контрольної роботи для заочної форми навчання.

У дисципліні «Операційні системи» використовуються комбіновані методи навчання, що поєднують теоретичний матеріал і практичні навички.

Словесні методи: пояснення теоретичного матеріалу, введення основних понять і принципів роботи операційних систем, відповіді на запитання, короткі обговорення технічних аспектів.

Наочні методи: використання мультимедійних презентацій, демонстрація та розбір основних схем, моделей і структурних компонентів операційних систем.

Практичні методи: виконання лабораторних робіт під керівництвом викладача.

Оцінювання результатів навчання здійснюється через поточний і підсумковий контроль, кожен з яких має свої особливості та критерії.

Поточний контроль передбачає оцінювання кожної лабораторної роботи окремо. Основними критеріями є правильність виконання завдання, якість захисту роботи та дотримання встановлених термінів. У процесі оцінювання

враховується рівень розуміння теоретичного матеріалу та сформованість практичних навичок роботи з комп'ютерними системами.

Підсумковий контроль здійснюється у формі екзамену, який оцінює ступінь засвоєння теоретичних положень дисципліни та здатність студента застосовувати отримані знання на практиці.

Шкала оцінювання: національна та ЄКТС

Сума балів за всі види навчальної діяльності	Оцінка ЄКТС	Оцінка за національною шкалою
		для екзамену
90-100	A	відмінно
82-89	B	добре
74-81	C	
64-73	D	задовільно
60-63	E	
35-59	FX	незадовільно з можливістю повторного складання
1-34	F	незадовільно з обов'язковим повторним вивченням дисципліни

Вибравши предметну область, над якою ви будете працювати, ви повинні виконати завдання до лабораторних робіт, а також відповісти на питання в кінці кожної лабораторної роботи. Звіт повинен містити хід виконання завдань, а також графічні матеріали, що підтверджують виконання цих завдань.

Лабораторна робота №1. Структура файлової системи Linux, основні команди, команди роботи з файлами

Мета: Оволодіння практичними навичками роботи в системі Linux. Знайомство із структурою файлової системи, основними командами роботи з файлами.

Linux — це не повноцінна операційна система, а **ядро** (kernel), тобто центральна частина ОС, що відповідає за безпосередню взаємодію з апаратним забезпеченням. Ядро є проміжною ланкою між апаратними компонентами комп'ютера та прикладними програмами.

Завдання ядра Linux включають:

1. **Управління процесами**

   Ядро створює, планує та завершує процеси, контролює їхню взаємодію та розподіл процесорного часу.

2. **Управління пам'яттю**

   Забезпечує виділення та звільнення оперативної пам'яті, керує віртуальною пам'яттю, сторінками та механізмом підкачки.

3. **Керування пристроями**

   Надає драйвери для роботи з периферією: диски, мережеві інтерфейси, клавіатура, миша, графічні адаптери тощо.

4. **Файлова система**

   Реалізує доступ до даних на різних носіях, підтримує різні файлові системи (ext4, XFS, Btrfs та інші) та забезпечує єдиний інтерфейс роботи з файлами.

5. **Мережеві функції**

   Обробляє роботу мережевих протоколів, маршрутизацію, сокети та інші механізми мережевої взаємодії.

Linux-дистрибутиви як повноцінні операційні системи

Хоча Linux є лише ядром, користувачі зазвичай працюють із повноцінними операційними системами, які будуються **на основі ядра Linux**. Такі системи називаються **дистрибутивами Linux**.

Дистрибутив включає:

- ядро Linux;
- системні утиліти (командна оболонка, менеджери пакетів, драйвери);
- графічне середовище (GNOME, KDE тощо);
- прикладні програми.

Приклади популярних дистрибутивів: **Ubuntu, Debian, Fedora, Arch Linux, Mint, CentOS**.

Таким чином, **Linux — це ядро**, а **Ubuntu чи інші дистрибутиви — це операційні системи на його основі**.

Операційні системи на основі ядра Linux мають низку особливостей, які відрізняють їх від Windows, macOS та інших пропрієтарних ОС. Ці відмінності визначають специфіку роботи користувача, спосіб керування системою та підхід до адміністрування.

1. Відкритий вихідний код

Linux поширюється за ліцензією GPL, що дає можливість:

- вільно переглядати та змінювати вихідний код;
- створювати власні версії ОС;
- розповсюджувати модифіковані варіанти.

Це створює широку спільноту розробників і високу прозорість системи.

2. Велика кількість дистрибутивів

На основі одного ядра існують сотні дистрибутивів, кожен з яких має:

- власний пакетний менеджер (apt, dnf, pacman тощо);
- різну філософію роботи (стабільність, мінімалізм, універсальність);
- різний набір утиліт та графічних середовищ.

Це дозволяє підібрати систему під конкретні задачі — від серверів до смартфонів.

3. Багатокористувацькість за замовчуванням

Linux з самого початку створювався як багатокористувацька система. Тому в ньому:

- кожен користувач має окремі права і домашній каталог;
- існує чітке розмежування прав доступу;
- виконання критичних дій можливе лише через суперкористувача (root).

Це підвищує безпеку та стабільність системи.

4. Орієнтація на командний рядок

У Linux великий акцент зроблено на роботу через термінал.
Командна оболонка (bash, zsh тощо) є потужним інструментом для:

- роботи з файлами;
- автоматизації задач (скрипти);
- адміністрування системи.

Графічні інтерфейси існують, але більшість системних завдань простіше виконувати через командний рядок.

5. Гнучкість і модульність

У Linux майже всі компоненти можна замінити:

- графічне середовище;
- менеджер вікон;

- системні служби;
- файлову систему.

Це дозволяє кастомізувати систему під конкретні потреби та оптимізувати її для обмежених ресурсів.

6. Пакетна система встановлення програм

На відміну від Windows, де ПО часто встановлюється окремими інсталляторами, Linux використовує репозиторії та пакетні менеджери. Це забезпечує:

- єдиний спосіб встановлення, оновлення і видалення програм;
- централізовані офіційні джерела програмного забезпечення;
- автоматичне вирішення залежностей.

7. Висока стабільність та безпека

Linux широко використовується у серверних середовищах завдяки:

- захищеній системі прав доступу;
- активному оновленню безпеки;
- мінімальному ризику вірусів;
- можливості працювати роками без перезавантаження.

8. Підтримка більшості платформ

Linux може працювати на:

- ПК;
- серверах;
- смартфонах (Android);
- маршрутизаторах, IoT-пристроях;
- суперкомп’ютерах.

Це універсальна платформа з широким спектром застосувань.

Командний рядок, командна оболонка та термінал: у чому різниця

У повсякденній мові терміни *командний рядок*, *термінал* і *shell* часто використовують як синоніми, однак у Linux вони позначають різні речі.

Командна оболонка (shell)

Командна оболонка — це програма, яка інтерпретує команди, введені користувачем. Саме вона приймає текст команди, аналізує її та передає операційній системі для виконання.
У більшості дистрибутивів Linux за умовчанням використовується оболонка **bash** (Bourne Again Shell) — покращена версія класичної оболонки *sh*.

Shell — це «мозок» командного середовища, але сам по собі він не є ні графічним, ні текстовим вікном.

Термінал (емулятор терміналу)

У сучасних графічних дистрибутивах Linux користувач отримує доступ до командної оболонки через **емулятор терміналу** — спеціальну програму, яка відкриває вікно для введення команд.
Приклади популярних емуляторів:

- **Konsole** (KDE),
- **gnome-terminal** (GNOME),
- інші: xterm, Terminator, Kitty тощо.

Емулятор терміналу — це **вікно**, у яке ви вводите команди, але він лише надає інтерфейс для роботи з оболонкою.

Командний рядок

Командний рядок — це **інтерфейс взаємодії**, тобто режим роботи, у якому користувач вводить текстові команди.

Фактично командний рядок — це **сеанс роботи оболонки**, який відображається всередині терміналу.

Коротко: як вони пов'язані

- **Термінал** -це вікно/програма, через яке ви взаємодієте з оболонкою.
- **Shell** - програма, яка виконує команди.
- **Командний рядок** - процес введення команд у shell через термінал.

Разом вони створюють середовище, що дозволяє працювати з Linux без графічних інструментів - напряму, через текстові команди.

Робота з терміналом та командною оболонкою в Linux

При встановленні дистрибутива Linux на персональний комп'ютер система, як правило, налаштовується на роботу з **графічним середовищем** (GNOME, KDE, XFCE тощо). Вхід у систему відбувається через графічний менеджер входу, і лише після цього користувач за потреби може перейти до **терміналу** та інтерфейсу командного рядка.

У цьому практикумі ми широко використовуватимемо можливості командної оболонки, тому розглянемо, як саме організована робота терміналів у Linux.

Віртуальні консолі та графічні термінали

У Linux одночасно існує кілька способів доступу до командного рядка:

Віртуальні текстові консолі - Це незалежні текстові робочі середовища, між якими можна переключатися комбінаціями:

- **Alt + F1…F6** — у більшості дистрибутивів переходи між текстовими консолями
- **Ctrl + Alt + F1…F6** — перехід у текстовий режим із графічного середовища

- **Ctrl + Alt + F7/F1** — повернення до графічного середовища (залежить від дистрибутива)

Користувач може працювати паралельно в кількох консолях, кожна з яких має власну сесію оболонки.

• Емулятори терміналу в графічному середовищі

У графічному інтерфейсі доступ до командної оболонки надається спеціальними програмами:

- **Konsole** (KDE),
- **GNOME Terminal** (GNOME),
- а також десятками інших — xterm, terminator, kitty, alacritty тощо.

Емулятор терміналу — це **лише вікно**, у якому працює командна оболонка; всі команди виконує саме shell.

Вхід до системи в текстовому терміналі

При переході у текстову консоль користувач бачить запрошення на введення облікових даних:

login:

Якщо цього запрошення не екрані немає (і не діє екранна заставка – screensaver), то це означає, що даний термінал не очікує входу користувача. Три найтиповіші причини:

1. Вхід вже здійснено. Для виходу можна ввести команду logout. Або можна ввести команду login – тоді після завершення сеансу нового користувача термінал повернеться до роботи з попереднім.

2. Термінал апаратно заблоковано клавішею “Scroll Lock” (це можна визначити за відповідним індикатором) – розблокуйте термінал.

3. Термінал зайнятий, тобто з ним пов’язана деяка програма – слід вийти з цієї програми і з командної оболонки, для чого потрібен певний досвід (можуть

спрацювати клавіша q, комбінації клавіш Ctrl+C, Ctrl+D, але іноді це не допоможе). У загальному випадку не слід застосовувати комбінацію Ctrl+Z – при цьому дійсно з'явиться запрошення на введення команди, але програма, що виконувалась, не буде зупинена, а лише перейде у фоновий режим, забираючи ресурси вашого комп'ютера. Якщо таких програм буде багато, вони можуть суттєво погіршити працездатність вашого комп'ютера.

4. Термінал використовується винятково для виведення на екран важливих системних подій, при цьому на ньому можна вводити команди і будь-які символи, але реакції на це не буде – слід перейти на іншу консоль комбінацією клавіш Alt+F#, де F# – одна з функціональних клавіш, крім F1.

Після введення ідентифікатора система запитує в користувача пароль:

Password:

Під час введення пароля символи на екрані не відображаються. Якщо ідентифікатор і пароль користувача були введені правильно, система здійснює авторизацію користувача, тобто, надає йому певні повноваження, необхідні для роботи в системі.

Після запуску емулятора терміналу на екрані з'являється вікно з **рядком запрошення** (shell prompt). Це спеціальне повідомлення, яке показує, що командна оболонка готова приймати команди. У різних дистрибутивах вигляд запрошення може дещо відрізнятися, але найчастіше воно містить ім'я користувача, ім'я комп'ютера та назву поточного каталогу, наприклад:

```
user@hostname:~$
```

Завершальний символ запрошення має значення. Якщо в кінці стоїть **$**, це означає, що працює звичайний користувач. Якщо ж останнім символом є #, то сеанс відкрито з правами адміністратора (superuser). Це може бути результатом входу під обліковим записом *root* або запуску терміналу, який автоматично надає підвищені привілеї.

Командна оболонка приймає команди, що вводяться з клавіатури, інтерпретує їх і виконує відповідні дії. Ці дії можуть полягати у запуску певних

утиліт із заданими у командному рядку параметрами. Крім того, командна оболонка надає користувачеві певний додатковий сервіс, наприклад, дозволяє виконувати редагування команди (курсор можна переміщати вправо чи вліво, додавати або знищувати символи під курсором), в деяких оболонках можна легко відтворити попередні команди (клавіші переміщення курсору вгору та вниз), а також користуватись підказками щодо імен наявних файлів (клавіша Tab). Докладніше про ці та інші сервісні можливості можна дізнатись в довідковій системі man, а також в будь-якій доступній книзі про системи UNIX чи Linux.

В кожній системі UNIX є в наявності кілька різних командних оболонок. Практично в кожній системі можна знайти звичну оболонку, або подібну до неї. Найбільш стандартною, що присутня в усіх системах, є оболонка Bourne (Bourne shell), яка стала основою стандарту POSIX shell. Ця оболонка пропонує мінімальний сервіс для користувача, і для інтерактивної роботи незручна. Її файл – /bin/sh. Існують альтернативні оболонки. Одна з них – C shell (/bin/csh), вона досить сильно відрізняється синтаксисом багатьох команд, але дуже зручна для користувача і програміста, особливо для тих, хто добре знайомий з мовою програмування C. Як правило, нею користуються адміністратори систем BSD. У сучасних системах, зокрема в Linux і в Mac OS X, стандартною оболонкою є Bourne again shell (/bin/bash або /usr/bin/bash) – розвиток Bourne shell, що зберігає програмну сумісність з sh, але включає в себе багато нових можливостей.

В подальшому ми будемо використовувати синтаксис оболонки sh (запрошення має вигляд ‘$’), а в окремих випадках порівнювати її з csh (запрошення має вигляд ‘>’).

Командна оболонка, в якій розпочинає роботу користувач після входу в систему, визначається з файлу /etc/passwd. Це один з найголовніших конфігураційних файлів системи, який містить параметри облікових записів користувачів. Кожний рядок файлу відповідає певному користувачу, точніше, обліковому запису користувача, що розрізняється за і ифікатором login або

userid. Користувач в процесі роботи може запустити іншу командну оболонку, просто набравши її ім’я.

Усікоманди, які можна ввести у рядку запрошення оболонки, належать до одної з таких категорій:

- вбудовані функції,
- функції, що визначені користувачем,
- зовнішні програми й утиліти.

Вбудовані функції реалізуються фрагментами програмного коду оболонки і виконуються найшвидше. Користувач може визначити свої функції (хоча таку можливість використовують нечасто). Якщо команда не є вбудованою функцією і не визначена як функція користувачем, тоді оболонка буде шукати файл з відповідним ім’ям і намагатись запустити його на виконання. Якщо ім’я файлу задано із шляхом до нього, то система намагається знайти його саме у тому каталозі, який явно задано у команді, але якщо ім’я файлу задано без шляху, то пошук файла здійснюється лише у тих каталогах, які задані системною змінною PATH. Для перегляду значення змінних їх слід набрати зі знаком $ попереду. Зокрема, якщо у змінній PATH не задано пошук у поточному каталозі, і поточний каталог не є одним із тих, що явно задано у цій змінній, то оболонка не побачить виконуваних файлів з поточного каталогу.

Основні команди та робота з терміналом

Після входу в систему користувач потрапляє у середовище командної оболонки (shell), де з’являється запрошення командного рядка. Тут можна вводити команди, які оболонка інтерпретує та виконує. Наприклад, якщо ввести випадкову послідовність символів:

```
$ kaekfjaeifj38
```

оболонка повідомить, що команда не знайдена, і знову виведе запрошення:

```
bash: kaekfjaeifj: команда не знайдена
$
```

Це дозволяє перевірити правильність введеної команди.

Історія команд та редагування

Історія команд дозволяє швидко переглядати раніше введені команди. Натискання стрілки вгору відображає попередню команду, стрілка вниз повертає до більш нових записів. Клавіші ліворуч і праворуч переміщують курсор у межах рядка, що дозволяє зручно редагувати команду.

Основні системні команди

Деякі базові команди, які користувач повинен пам’ятати, включають:

- `man` – відкриває сторінки довідкової системи. Можна вказати розділ для пошуку (наприклад, `man 7 mdoc`).
- `passwd` – змінює пароль користувача. Спочатку слід підтвердити старий пароль, потім ввести новий двічі.
- `who` – показує, хто зараз працює в системі. Команда `who am i` нагадує ваш login.
- `date` – виводить поточні дату і час.
- `cal` – відображає календар поточного або будь-якого обраного місяця.
- `cat <ім’я файлу>` – перегляд вмісту текстового файлу.
  `more <ім’я файлу>` – посторінковий перегляд, дозволяє перегортати сторінки вперед і назад.
- `wc <ім’я файлу>` – підрахунок рядків (`-l`), слів (`-w`) і символів (`-c`) у файлі.
- `touch <ім’я файлу>` – створення нового порожнього текстового файлу.

Перегляд інформації про систему

Для перевірки зайнятого і вільного місця на дисках використовується команда:

df

Аналогічно команда `free` показує обсяг використаної та доступної оперативної пам’яті й підкачки.

Завершення роботи з терміналом

Сеанс роботи з терміналом можна завершити або закривши вікно емулятора, або виконавши команду:

exit

Розглянемо особливості файлової системи UNIX. Вся файлова система поєднується в єдине дерево каталогів, які починаються з кореневого каталогу, що має позначення ‘/’. Всі зовнішні файлові системи (змінні носії інформації, мережеві диски і таке інше) монтуються у визначенні місця єдиного дерева файлової системи.

Як і в інших ієрархічних файлових системах, у файловій системі UNIX ім’я файлу повинно бути унікальним лише в межах одного каталогу (на відміну від MS-DOS/Windows, UNIX розрізняє великі і малі літери в назвах файлів). Для однозначної ідентифікації файлу в дереві каталогів слід указувати повний шлях до файлу. Якщо шлях починається з символу ‘/’ (наприклад, /usr/bin/cal), то він відраховується від кореневого каталогу (абсолютний шлях), а якщо з іншого символу – то від поточного каталогу, тобто того, в якому користувач знаходиться в поточний момент (відносний шлях). Крім того, поточний каталог позначається символом ‘.’ (крапка), каталог, що знаходиться на один рівень вище, тобто батьківський каталог – символом ‘..’ (дві крапки). Крім того, існує спеціальне позначення для так званого домашнього каталогу користувача, тобто каталогу, з якого він починає свою роботу – ‘~’ (тильда). Домашній каталог для кожного користувача також задається у файлі /etc/passwd, за умовчанням це /home/<login>.

Для переходу з каталогу в каталог існує команда cd <новий каталог> (change directory – змінити каталог). Якщо використати цю команду без параметрів, відбудеться перехід в домашній каталог користувача.

Слід зазначити, що відносні шляхи слід використовувати з обережністю. Для того, щоби перевірити, в якому каталозі знаходиться користувач, можна скористатись командою pwd.

Перегляд вмісту каталогів здійснюється за допомогою команди ls, а розширений варіант цієї команди ls -l дає також інформацію з таблиці індексних дескрипторів (див. Роботу №2). Щоби скопіювати файл, використовується команда cp <файл-джерело> <призначення>. Для перенесення файлу з каталогу в каталог, а також для перейменування файлу, використовується команда mv <файл-джерело> <призначення>. В обох командах в якості параметра <призначення> може задаватись каталог призначення або ім'я файлу призначення. Крім того, число параметрів може бути більше двох. В такому випадку всі параметри, крім останнього, розглядаються як список імен файлів-джерел, а останній параметр може бути лише каталогом призначення. Створити каталог можна командою mkdir, видалити файл – командою rm, видалити каталог – командою rmdir або rm -r.

Крім звичайних файлів існують різні типи спеціальних файлів. З одним із них ми вже познайомились – це каталоги. Ще одним типом спеціальних файлів є так звані зв'язки або посилання (рос. – ссылка, англ. – link). В системі UNIX розрізняють два принципово різних типи посилань, хоча створюються вони однією командою – ln. Перший тип – це так звані жорсткі посилання. Фактично вони є абсолютно рівноправними новими іменами вже існуючого файлу. Після створення такого посилання система не розрізняє, яке ім'я було первинне, а яке було створене як посилання. Спроба видалити такий файл призводить до того, що одне з його імен (те, за яким ми намагаємось його видалити), знищується, а інші (як і сам файл) залишаються. Тільки після видалення останнього з імен фактично знищується сам файл. Другий тип посилання – символічне посилання, яке створюють командою ln –l. Це спеціальний тип файлу, який містить в собі

ім’я того файлу (або каталогу), на який він посилається. Символічні посилання дуже широко застосовуються в системах UNIX і Linux для забезпечення сумісності програм з різними системами.

Більшість команд, що застосовуються по відношенню до символічного посилання, діють безпосередньо на файл, на який посилання здійснене. Натомість, деякі команди, наприклад mv і rm, діють не на файл, а на посилання. При цьому деякі послідовності команд можуть привести до небажаних наслідків. Наприклад, маємо файл oldfile і бажаємо перейменувати його в newfile. Це можна зробити як командою

mv oldfile newfile

так і послідовністю команд

cp oldfile newfile rm oldfile

Результати будуть однакові. До речі, в одному командному рядку можна задати декілька команд, розділивши їх знаком ‘;’, ці команди будуть виконуватись послідовно:

cp oldfile newfile; rm oldfile

Тепер уявимо, що маємо файл targetfile і посилання на нього

oldfile. Команда

mv oldfile newfile

перейменує посилання, тобто тепер newfile буде посилатись на targetfile. Команда

cp oldfile newfile

скопіює не посилання, а сам файл targetfile, тобто під іменем

newfile буде створено новий файл – копію targetfile.

Наступна команда

rm oldfile

знищить старе посилання, не пошкодивши при цьому файл targetfile. Тобто замість одного файлу з посиланням на нього у нас утворилися два ідентичних файла, які абсолютно не пов’язані між собою.

Завдання

1. Завантажтеся в систему під вашим користувацьким ім'ям.
2. Поміняйте ваш пароль. Ваш новий пароль повинен включати в себе як частину номер Вашої залікової книжки.
3. Виведіть системну дату.
4. Підрахуйте кількість рядків у файлі:

Варіант	Файл
1	/etc/passwd
2	/etc/group
3	/etc/profile
4	/etc/pam.conf
5	/etc/rsyslog.conf
6	/etc/crontab
7	/etc/modules
8	/etc/networks
9	/etc/protocols
10	/etc/fstab

5. Виведіть на екран вміст відповідного файлу.
6. Виведіть календар на <1995+№варіанту> рік.
7. Виведіть календар на 1752 рік. Чи не помічаєте що-небудь цікаве у вересні? Поясніть.
8. Визначте, хто ще завантажений у систему.
9. Наберіть команду ping. Поясніть результат.
10. **Скопіюйте (скопіюйте, а не перемістіть, бо система перестане працювати коректно!) файли**

варіант	**файл 1**	**файл 2**
1	**/bin/cat**	**/bin/grep**
2	**/bin/echo**	**/bin/chmod**
3	**/bin/ls**	**/bin/chown**

4	**/bin/touch**	**/bin/kill**
5	**/bin/more**	**/bin/gzip**
6	**/bin/date**	**/bin/more**
7	**/bin/cp**	**/bin/ps**
8	**/bin/mv**	**/bin/bash**
9	**/bin/login**	**/bin/sh**
10	**/bin/less**	**/bin/rm**

у ваш домашній каталог різними способами.

11. Створіть каталог lab_1.

12. Скопіюйте в нього з вашого домашнього каталогу копію файлу 1, яку ви отримали в п.10, під ім'ям my_<ім’я файлу 1>. Перемістіть в цей каталог з вашого домашнього каталогу копію файлу 2, яку ви отримали в п.10, перейменувавши його при цьому в my_<ім’я вихідного файлу 2>. За ім’я вихідного файлу слід брати саме ім’я файлу, без імен каталогів і шляху до файлу (інакше символ "/" буде проінтерпретований операційною системою зовсім не так, як Ви очікуєте).

13. Перейдіть у свій домашній каталог і переконайтеся в тому, що все зроблено правильно.

14. Створіть каталог lab_1_<№варіанту> і перейдіть в нього.

15. Скопіюйте в каталог lab_1_<№варіанту> файл з п.4 під ім'ям n<ім’я вихідного файлу>.

16. За допомогою команд cat і less перегляньте його вміст.

17. Перейдіть у свій домашній каталог.

18. Видаліть каталог lab_1_<№варіанту>.

Лабораторна робота №2. Система розмежування доступу в UNIX і Linux, права доступу до файлів і керування ними

Мета: Оволодіння практичними навичками керування правами доступу до файлів і їхній аналіз в ОС UNIX та Linux.

Однією з основних складових захищеної системи є система контролю доступу, яка відіграє важливу роль у роботі з файловою системою у UNIX-подібних операційних системах. Кожен файл має свого власника і належить певній групі. Уся детальна інформація про файл, включаючи його тип, ідентифікатори власника та групи, розмір, час останньої зміни, права доступу та розташування (номери блоків), зберігається не в каталогах, як це зроблено в багатьох інших файлових системах, таких як NTFS, а в системній таблиці індексних дескрипторів (**i-node**).

Прямий доступ до цієї таблиці заборонений. Каталоги містять лише ім'я файлу та індекс — посилання на відповідний запис у таблиці **i-node**. Завдяки такій організації файли у файловій системі UNIX можуть мати декілька рівноправних імен (жорстких посилань), які можуть зберігатися в різних каталогах. Переглянути інформацію з таблиці **i-node** можна за допомогою команди ls –l, яка виводить дані про файл у вигляді одного рядка. Перший символ вказує на тип файлу: '**–**' – звичайний файл (текстовий або бінарний), **d** – каталог (*directory*), **l** – символічне посилання (**link**), **c** – символьний пристрій (*character device*), **b** – блочний пристрій (*block device*). Наступні дев'ять символів відображають права доступу. Далі йде інформація про кількість жорстких посилань на файл, власника, групу, розмір файлу, дату останньої зміни, а останнє поле – ім'я файлу.

UNIX використовує дискреційну модель контролю доступу, за якою для кожного файлу визначаються права доступу для всіх користувачів. Кожен файл пов'язаний зі спеціальною інформацією, яка містить ідентифікатор власника файлу, ідентифікатор групи та права доступу. Права доступу поділяються на три категорії: для власника, для групи та для всіх інших користувачів. У кожній категорії є три біти, що відповідають правам на читання, запис і виконання (**r**,

w, **x**). Для каталогів право на виконання означає можливість доступу до таблиці індексних дескрипторів для читання та запису. Без цього права неможливо зробити каталог поточним (перейти до каталогу через *cd*), переглянути або змінити права доступу до його об'єктів, хоча переглядати вміст каталогу можна за наявності права на читання. Навіть якщо є право на запис, без права на виконання не можна змінювати вміст каталогу. Водночас, якщо є право виконання, але немає права читання, неможливо переглянути вміст каталогу, однак можна переходити в його підкаталоги та звертатися до файлів, якщо відомі їхні імена.

Є ще три біти, що впливають на доступ до файлу. Перший з них називається **SUID** (Set-User-ID-on-execution bit), і якщо він установлений для файлу, що виконується, то цей файл для будь-якого користувача виконується не з його правами, а з правами власника цього файлу — зазвичай, власником є **root**. Другий біт — **SGID** (Set-Group-ID-on-execution bit), якщо він встановлений, то ця програма виконується для будь-якого користувача із правами членів групи цього файлу. Для каталогів **SGID** визначає, що для усіх файлів, створюваних у цьому каталозі, ідентифікатор групи буде встановлений такий же, як у каталогу, а не такий, що визначає первинну групу користувача (ці правила залежать від версії UNIX). Останній біт називається **Sticky**, він використовується тільки для каталогів і визначає, що користувачі, які мають право на записування в каталог, не мають права видаляти чи перейменовувати файли інших користувачів у цьому каталозі. Це необхідно для каталогів загального використання, наприклад **/tmp**.

Права доступу до файлів задаються при створенні файлу і в подальшому можуть бути змінені командою *chmod <нові права> <файл(и)>*

<нові права> задаються двома способами. Перший – символьний. Використовуються такі позначення: **u** – власник файлу (*user*), **g** – група (*group*), **o** – всі інші (*others*), **a** – всі категорії користувачів одночасно (*all*). Після категорії користувачів задається дія: ‘**+**’ – додати права до існуючих, ‘**-**’ – відібрати права (якщо існують), ‘**=**’ – встановити права замість існуючих. Далі

позначаються права: **r** – зчитувати (*Read*), **w** – записувати (*Write*), **x** – виконувати (*eXecute*). Можна формувати список зміни прав, розділяючи окремі категорії користувачів комами (без пробілів). Наприклад, команда *chmod* u+rw,g=rx,o-w my_file встановить для користувача права на зчитування і записування (право на виконання для користувача буде залежати від того, чи було воно встановлено раніше), для групи встановить права на зчитування і виконання (незалежно від того, які права були раніше), а для всіх інших гарантовано заборонить записування (права на зчитування і виконання будуть встановлені в залежності від того, чи були вони встановлені раніше).

Другий спосіб встановлення прав доступу – числовий – є зручнішим, коли потрібно задати нові права незалежно від попередніх. Для цього використовується восьмеричне представлення, яке можна зрозуміти за допомогою наступної таблиці:

Вісімковий запис	Двійковий запис	Права доступу
0	000	---
1	001	--x
2	010	-w-
3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwx

Для наглядного прикладу зверніть увагу на рис1 репрезентацію прав у системі:

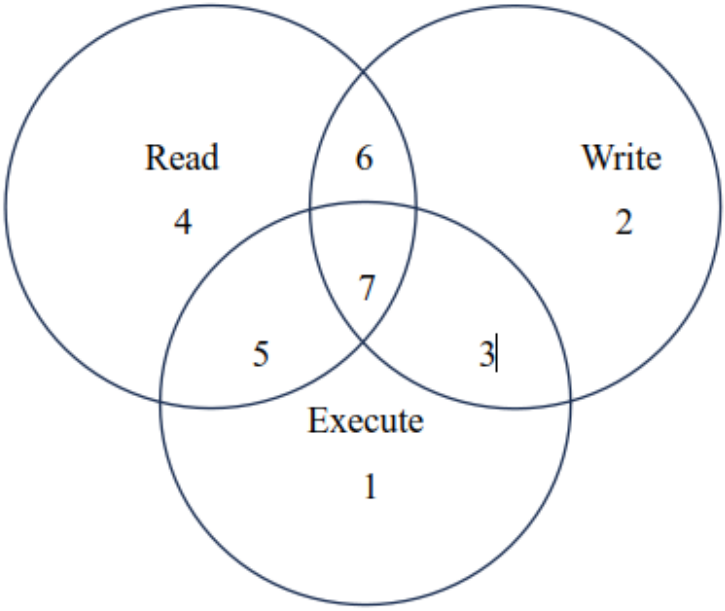

Рис1 Схема формування прав доступу

Наприклад, команда chmod 751 my_file встановлює такі права доступу до файлу my_file: власнику – зчитування, записування, виконання, групі – зчитування і виконання, а всім іншим – лише виконання. Команда ls -l покаже для цього файлу таку інформацію про права доступу:

-rwxr-x--x (перший “мінус” означає, що це звичайний файл).

За умовчанням для всіх нових файлів, що створюються, встановлюються такі права: для файлів 666, тобто rw-rw-rw-, для каталогів – 777, тобто rwxrwxrwx (ми вже бачили вище, що наявність права на виконання дуже важлива для каталогів). Є особлива команда – *umask*, яка дозволяє зменшити права доступу, що встановлюються для нових файлів. Параметром цієї команди є вісімкова маска, аналогічна тій, що використовується при числовому способі задавання параметрів команди *chmod*, але у випадку команди *umask* права доступу, що задані нею, будуть відніматись від прав доступу, що були задані по замовчуванню для вже утвореного файлу. Команда *umask* не може додавати прав, тому нові файли ніколи автоматично не отримають право на виконання, в разі необхідності його треба буде додати вручну. Дія команди *umask* розповсюджується на всі файли всіх типів, що утворюються протягом поточної сесії користувача після цієї команди. Наприклад, після команди *umask* 123 всі

файли будуть утворюватись з параметрами доступу rw-r--r-- (644), а всі каталоги – rw-r-xr-- (654). За замовчуванням рекомендується використовувати команду *umask* 22 (інтерпретується як *umask* 022).

Для зміни власника файлу існує команда *chown*. З міркувань безпеки, ця команда дозволяє встановлювати власником файла будь-кого лише системному адміністратору (root). Інші користувачі можуть лише привласнити файл собі, якщо вони мають на це права, встановлені правами доступу до файлу і каталогу.

Списки ACL

Стандартні права доступу в UNIX і Linux не мають достатньої гнучкості і в деяких випадках не дозволяють встановити потрібні правила доступу. Розглянемо приклади.

Припустимо, у нас є власник файлу з повними правами доступу і потрібно надати доступ на зчитування і записування у файл ще одному користувачеві (всім решті — лише на зчитування). Для цього можна створити нову групу і вказати, що ця група володіє файлом.

addgroup <new_group>

chgrp <new_group> <this_file> *chmod* g=rw <this_file>

Далі треба відредагувати файл /etc/group, додавши у нову групу потрібного нам користувача.

А тепер припустимо, що нам треба надати права зчитування, записування та запуску на виконання двом різним користувачам, ще двом – права зчитування й записування, ще одному – тільки зчитування, а усім решті – взагалі не дати доступу до цього файлу. Класична система розмежування доступу UNIX не дозволяє налаштувати такі права.

Але на певному етапі розвитку в UNIX-подібних системах (спочатку в **Solaris**, далі – у **FreeBSD**, а згодом і в **Linux**) впровадили потрібний механізм. Це списки керування доступом до об’єктів файлової системи, access control lists (ACLs). Іноді їх називають списками контролю доступу, що є неточним перекладом з англійської. За допомогою ACL можна призначити як права

доступу власника файлу, групи файлу і усіх інших, так і окремі права для певних користувачів і груп, а також максимальні права доступу за умовчанням, які дозволено мати будь-якому користувачу.

Далі ми будемо називати ACL для файлів і каталогів «розширеними правами доступу». Розширені права доступу встановлюються командою *setfacl*. Аргументи команди задаються як ряд полів, що розділені двокрапками:

<entry_type>:[<uid>|<gid>]:<perms>

Тут *<entry_type>* – тип розширеного права доступу (u – **user**, g – **group**, o – **other**, m – **mask**), *<uid>* і *<gid>* – ідентифікатори користувача і групи, відповідно, *<perms>* – права доступу, що призначають (r – **read**, w – **write**, x – **execute**).

Розширені права доступу до файлу можуть бути такими, як вказано у Таблиці:

u[ser]::<perms>	права доступу власника файлу
g[roup]::<perms>	права доступу групи файлу
o[ther]:<perms>	права усіх інших
m[ask]:<perms>	маска ACL
u[ser]:<uid>:<perms>	права доступу вказаного користувача; в якості uid можна вказати і UID, і ім’я (login) користувача
g[roup]:<gid>:<perms>	права доступу вказаної групи; в якості gid можно вказати і GID, і ім’я групи

Встановлюючи права доступу за умовчанням для конкретних користувачів і груп (два останніх рядка таблиці), необхідно також встановити права доступу за умовчанням для власника і групи файлу, усіх інших, а також маску ACL (тобто, чотири перших рядка таблиці у цьому випадку є обов’язковими).

Встановлення розширених прав доступу здійснюється командою *setfacl* з ключем –s (**set**). Ключ -m вказує на додавання прав доступа замість їх заміни.

Ефективні (тобто ті, що будуть насправді застосовані) права доступу користувачів визначаються як їхніми персональними правами доступу до цього

файлу, так і встановленою маскою. З персональних прав і маски обираються найбільш строгі обмеження.

Команда *ls –l* не показує розширені права доступу, але показує наявність створеного ACL – символ «+» (плюс) після стандартних прав доступу.

Переглянути розширені права доступу можна командою

getfacl.

Детальнішу інформацію про параметри команд *setfacl* і

getfacl можна отримати з *man*.

Завдання

1. Створіть каталог lab_2.
2. Скопіюйте в каталог lab_2 файл /bin/cat під назвою my_cat.
3. За допомогою файлу my_cat, що знаходиться в каталозі lab_2, перегляньте уміст файлу .profile (ви знаходитесь у домашньому каталозі).
4. Перегляньте список файлів у каталозі lab_2. Потім перегляньте список усіх файлів, включаючи приховані, з повною інформацією про файли. Зверніть увагу на права доступу, власника, дату модифікації файлу, що ви тільки скопіювали. Потім перегляньте цю інформацію про оригінальний файл (той, який копіювали) і порівняйте два результати.
5. Змініть права доступу до файлу my_cat так, щоб власник міг тільки читати цей файл.
6. Переконайтеся в тому, що ви зробили ці зміни і повторіть п.3.
7. Визначте права на файл my_cat таким чином, щоб ви могли робити з файлом усе, що завгодно, а всі інші — нічого не могли робити.
8. Поверніться в домашній каталог. Змініть права доступу до каталогу lab_2 так, щоб ви могли його тільки читати.
9. Спробуйте переглянути простий список файлів у цьому каталозі. Спробуйте переглянути список файлів з повною інформацією про них. Спробуйте запустити і видалити файл my_cat з цього каталогу.

10. Поясніть отримані результати. Результати виконання п.8 можуть бути різними в різних версіях UNIX, зокрема, Linux і FreeBSD. Прокоментуйте отримані результати у висновках.

11. За допомогою команди su <user name>, завантажтесь в систему, користуючись обліковим записом іншого користувача. (Вам потрібно знати пароль цього користувача.) Спробуйте отримати доступ до Вашого каталогу lab_2. Перевірте, чи правильно зроблено завдання попереднього пункту. Створіть каталог lab_2_2.

12. Знову завантажтесь в систему, користуючись своїм обліковим записом4. Спробуйте зробити власником каталогу lab_2 іншого користувача. Спробуйте зробити себе власником каталогу lab_2_2. Поясніть результати.

13. Зайдіть у каталог lab_2. Зробіть так, щоб нові створені файли і каталоги діставали права доступу згідно Таблиці. Створіть новий файл і каталог і переконайтеся в правильності ваших установок.

варіант	Права для файлів	Права для каталогів
1	644	754
2	664	774
3	6-4	7-5
4	62-	73-
5	644	745
6	664	764
7	6-4	715
8	62-	63-
9	644	744
10	664	765

14. Поверніть собі права читати, писати, та переглядати вміст каталогів.

15. Створіть у каталозі lab_2 каталог acl_test та у ньому файли file1, file2. Після час створення file1 додайте у нього довільний текст.

16. Виведіть ACL для file1

17. Змінить права доступу на file1 так, щоб тільки власник мав право на читання.

18. Увійдіть до системи під іншим обліковим записом та спробуйте прочитати вміст file1. Що отримаємо? Поверніться до свого облікового запису.

19. За допомогою команди setfacl додайте право на читання іншому обраному користувачу для file1. Перевірте, що створився новий ACL для file1.

20. Увійдіть до системи під іншим обліковим записом та спробуйте прочитати вміст file1. Що отримаємо? Поверніться до свого облікового запису.

21. За допомогою команди setfacl встановіть значення маски таким чином щоб дозволити читати вміст file1 іншому користувачу. Виведіть ACL для file1

22. Увійдіть до системи під іншим обліковим записом, та спробуйте прочитати вміст file1. Ви повинні мати таку змогу.

Лабораторна робота №3 Командна оболонка shell, стандартні потоки введення/виведення, фільтри і конвеєри

Мета: Оволодіння практичними навичками перенаправлення стандартних потоків, роботи з фільтрами і організації конвеєрів.

Теоретична частина

У цій роботі ми розглянемо деякі функції командних оболонок, які дозволяють користувачам гнучко налаштовувати завдання для виконання операційною системою. Під час запуску кожна командна оболонка виконує налаштування системного середовища (ініціалізує системні та власні змінні), для чого використовуються певні файли. Зазвичай, їх щонайменше два – глобальний і локальний (деякі оболонки можуть використовувати більше конфігураційних файлів). Глобальні файли конфігурацій для всіх оболонок розташовані в каталозі /etc, тоді як локальні знаходяться в домашньому каталозі користувача. Назви конфігураційних файлів зазвичай закінчуються на 'rc'. Локальні файли приховуються, щоб не заважати користувачу в повсякденній роботі (приховані файли починаються з символу '.', і команда ls без додаткових параметрів їх не показує).Приклади конфігураційних файлів: для sh – /etc/profile і ~/.profile, для csh – /etc/cshrc і ~/.cshrc, для tcsh – /etc/cshrc і ~/.cshrc, а також /etc/tcshrc і ~/.tcshrc (двох останніх може і не бути).

Часто, як параметр команди, потрібно вказати не один файл, а кілька, назви яких мають спільні характеристики. У таких випадках використовують так звані маски пошуку. Спеціальний символ '?' у масці позначає один будь-який символ, а символ '*' – будь-яку послідовність символів. Наприклад, команда ls /bin/???? виведе всі файли з каталогу /bin, чиї назви складаються з чотирьох символів, а команда ls /etc/d* покаже всі файли з каталогу /etc, назви яких починаються з літери 'd'. Також можна задати набір символів, наприклад, маска [abc]??? означає ім'я з чотирьох символів, де перша літера – a, b або c.

Для пошуку файлів за певними ознаками можна використовувати команду find. Перший параметр цієї команди (обов’язковий) – це каталог, з якого починається пошук (наприклад, / – кореневий каталог), далі – параметр, що задає ознаку пошуку (наприклад, -name – пошук файлів, імена яких відповідають заданій масці, –atime – пошук файлів, дата модифікації яких відповідає заданій умові), далі іде власне маска пошуку, а далі – дія. Найтиповіша дія – –print, вивід результатів пошуку на екран. Якщо цей параметр не вказати, пошук відбуватись буде, а от результатів його видно не буде.

Ще одна корисна функція оболонок – це перенаправлення потоків введення-виведення. Зазвичай більшість команд або утиліт отримують інформацію з клавіатури або з файлу, якщо він вказаний як параметр, і виводять результати на екран. Однак, насправді вони працюють зі стандартними потоками введення та виведення, які пов'язані з певними файлами. У UNIX-подібних системах файл розглядається як потік байтів. Оскільки пристрої також трактуються як файли, а операції введення-виведення — як зчитування та записування у відповідні файли, це забезпечує легке перенаправлення потоків між файлами та пристроями.

За замовчуванням, стандартний потік введення отримується з клавіатури. Якщо вказано ім'я файлу, утиліта буде використовувати його як джерело вхідного потоку замість клавіатури (але це не стосується всіх команд). Є два вихідні потоки: стандартний потік виведення (як правило, виводиться на екран) і потік повідомлень про помилки (також виводиться на екран за замовчуванням).

Оболонка дає змогу перенаправити потоки у заданий файл. Символ ‘<’ перенаправляє вхідний потік. Після цього символу очікується ім’я файлу або пристрою, з якого буде братись вхідний потік.

Наприклад, команда cat без параметрів очікує введення з клавіатури, і кожний рядок передає на екран монітора. Команда cat my_file замість введення з

клавіатури виведе на екран вміст файлу my_file, якщо такий існує, і повідомлення про помилку, якщо такого не існує.

Команда cat < my_file на перший погляд виглядає аналогічно попередній, але з точки зору операційної системи і самої утиліти cat процеси відбуваються інакше. У першому випадку командна оболонка запускала cat і передавала їй параметр командного рядка my_file, який утиліта інтерпретувала як ім'я вхідного файлу. У другому випадку оболонка запускає cat без параметрів, але передає вміст файлу my_file як вхідний потік.

Команда cat > my_file перенаправляє вихідний потік. Оскільки вхідний потік не перенаправлено, введення очікується з клавіатури. У результаті цієї команди буде створено файл my_file, якщо його не існує, або ж вміст існуючого файлу буде знищено, і все, що вводиться з клавіатури, буде записуватись у цей файл до отримання символу кінця файлу (EOF, Ctrl+D). Існує також можливість дописати дані в кінець файлу без знищення його попереднього вмісту, використовуючи команду cat >> my_file.

Команда cat < my_file1 > my_file2 перенаправляє як вхідний, так і вихідний потік. Якщо файл my_file1 існує, то його вміст буде записано у файл my_file2. Якщо не існує, то повідомлення про помилку буде виведено на екран. Потік повідомлень про помилки в оболонці sh (і bash) перенаправляється окремо, він позначається 2>. Наприклад, команда cat < my_file1 > my_file2 2> my_file3 у разі, якщо файл my_file1 існує, запише його вміст в файл my_file2, а якщо не існує, то запише повідомлення про помилку в файл my_file3. Важливо наголосити, що оболонки csh і tcsh (стандартні для систем BSD) не мають засобів безпосередньо перенаправляти потік повідомлень про помилки. Тим не менше, засобами цих оболонок також можна організувати окреме перенаправлення потоків завдяки хитрим прийомам програмування.

Дуже часто потік помилок намагаються взагалі “загасити”. Для того, щоби знищити якийсь потік, існує спеціальний пристрій /dev/null. Все, що в нього направляється, зникає безслідно

Перенаправлення введення-виведення широко використовується в двох основних випадках. Перший – це запуск утиліт у фоновому режимі. Щоб робота фонових програм не заважала взаємодії користувача з терміналом, потрібно перенаправити потоки введення-виведення таким чином, щоб вони працювали виключно з файлами. Другий випадок – це використання спеціальних утиліт, призначених для отримання інформації з одного файлу, її обробки та запису результату в інший файл. Такі утиліти називаються фільтрами. Утиліта cat, приклади використання якої з перенаправленням потоків розглянуто раніше, є простим фільтром. Вона практично не обробляє інформацію, а лише може об'єднувати кілька файлів в один. Інші корисні фільтри – це cut, grep, sort.

Утиліта cut переглядає вхідний файл і виділяє з кожного рядка інформацію на основі її розташування в певних колонках або полях. Наприклад, рядки файлу /etc/passwd розділені на поля символом ‘:’. Перше поле містить логін, а п'яте – інформацію про користувача. Якщо ми хочемо вивести лише цю інформацію, можна скористатися такою командою: cut –d: –f1,5 < /etc/passwd

Ключ –d задає символ-роздільник полів (у цьому випадку ‘:’), ключ –f – список полів, що треба роздрукувати (у цьому випадку 1 і 5).

Утиліта grep відображає тільки ті рядки, в яких присутній вказаний рядок пошуку. Утиліта sort виконує сортування вхідного потоку, наприклад, за алфавітом. Детальніше про ці та інші фільтри можна дізнатися з довідкової системи man.

Також є можливість перенаправляти вихідний потік однієї утиліти безпосередньо у вхідний потік іншої, без створення тимчасових файлів. Така техніка називається конвеєром (pipe). У UNIX-подібних системах всі утиліти, поєднані в конвеєр, працюють паралельно, обробляючи дані у міру їх надходження. Конвеєр створюється за допомогою символу |, наприклад: util1 | util2 | util3

При утворені конвеєра окремо перенаправляти вхідні й вихідні потоки на проміжних стадіях не можна – це буде або сприйнято як синтаксична помилка, або результат може бути непередбачуваним.

Приклад конвеєру: ps –al | grep root | more

Команда ps з ключами –al направить у вихідний потік список всіх процесів у системі, grep root вибере з них лише ті, які виконуються від імені root, more забезпечить їх виведення на екран посторінково.

Інший приклад:cat /etc/passwd | cut –d: –f1,5 | more

Ця команда зробить те ж саме, що й приклад з командою cut, що розглядався раніше, але виведення на екран буде посторінковим.

Якщо проміжні результати на деякій із стадій конвеєра бажано зберегти, можна скористатись командою tee my_file. Ця команда візьме вхідний потік, передасть його без змін у вихідний потік і одночасно продублює у файл my_file6. Наприклад, так можна модифікувати один із розглянутих вище прикладів:

ps –ef | grep root | tee my_file | more

Тепер ми не лише побачимо на екрані посторінково виведений список всіх процесів root, але й збережемо його у файлі my_file.

Завдання

1. Перейдіть у каталог /bin. Перегляньте список усіх файлів, що починаються із символу, який визначено в таблиці індивідуальних завдань.

2. Перегляньте список файлів, імена яких складаються з визначеної у таблиці індивідуальних завдань кількості символів.

3. Перегляньте список файлів, імена яких починаються із символів, які визначено в таблиці індивідуальних завдань. Зробіть це декількома способами.

4. Створіть у вашому домашньому каталозі підкаталог lab_4

і перейдіть в нього.

5. За допомогою команди cat створіть файл my_text і запишіть у нього кілька рядків. Потім за допомогою команди cat допишіть у нього ще кілька рядків.

6. Підрахуйте кількість файлів у каталозі, визначеному з таблиці індивідуальних завдань, використовуючи і не використовуючи конвеєри. Порівняйте результат.

Таблиця індивідуальних завдань

варіант	п.1	п.2	п.3	п.6, 7
1	a	2	a, b, c, d	/bin
2	b	3	e, f, g, h	/usr
3	c	4	i, j, k, l	/usr/bin
4	d	5	m, n, o, p	/home
5	f	2	q, r, s, t	/var
6	g	3	u, v, w	/
7	h	4	x, y, z	Ваш домашній каталог
8	k	5	a, d, k, l	/tmp (або /var/tmp)
9	l	3	m, g, y	/sbin
10	n	2	x, z, r, q	/usr/sbin

7. Підрахуйте кількість файлів у каталозі, визначеному з таблиці індивідуальних завдань, при цьому зберігши список файлів у файлі filelist, використовуючи команду tee.

8. Починаючи з вашого домашнього каталогу, виведіть на екран у повному форматі назви усіх файлів і каталогів, що починаються з ‘m’. При цьому перед виведенням кожної назви на екран повинен виводитися запит на його підтвердження.

9. Починаючи з кореневого каталогу, виведіть на екран імена всіх каталогів, що останній раз змінювалися більше 15 днів назад.

10. Виведіть на екран тільки час, що повертається командою date.

11. Виведіть на екран список усіх користувачів системи, тобто перші поля кожного рядка файлу /etc/passwd (роздільник полів — символ ‘:’).

12. Виведіть на екран імена усіх файлів у каталозі /bin, що містять слова Software чи software. Потік помилок при цьому не повинний виводитися на екран.

Увага!!! у цьому завданні мова йде про те, що слова Software чи software містяться не у назві файлу (таких файлів там не повинно бути), а у самому файлі (а таких файлів має бути достатньо).

13. Відсортуйте конфігураційний файл вашої оболонки (.profile, .cshrc) відповідно до кодової таблиці ASCII так, щоб при цьому ігнорувалися пробіли на початку рядків. Робіть це з копією файлу, щоби не порушити нормальну працездатність вашої оболонки.

Лабораторна робота №4 Розробка сценаріїв командної оболонки

Мета: Оволодіння практичними навичками професійної роботи з командною оболонкою shell – використання змінних і створення командних файлів.

Теоретичні відомості

Усі змінні вашого оточення виводяться за допомогою команди set, ознайомтесь з ними.

Сценарій командної оболонки (shell-скрипт) — це текстовий файл, що містить послідовність команд, які виконуються командною оболонкою. Оболонка зчитує файл сценарію та виконує команди так, ніби вони вводяться користувачем вручну в командному рядку.

Командна оболонка одночасно є:

- потужним інтерфейсом командного рядка до операційної системи;
- інтерпретатором мови сценаріїв.

Більшість дій, які можна виконати безпосередньо у командному рядку, також можна реалізувати у сценаріях. Аналогічно, багато конструкцій сценаріїв можуть бути виконані безпосередньо у командному рядку. Раніше увага приділялася переважно інтерактивній роботі з оболонкою, проте shell також має численні засоби, орієнтовані на створення програм та автоматизацію роботи.

Створення сценарію командної оболонки

Для створення та запуску сценарію командної оболонки необхідно виконати такі кроки:

1. **Написати сценарій**

Сценарії командної оболонки є звичайними текстовими файлами. Для їх

створення використовується текстовий редактор. Рекомендується застосовувати редактори з підсвіткою синтаксису, оскільки вона допомагає виявляти типові помилки. Для написання сценаріїв зручно використовувати редактори `vim`, `gedit`, `kate` та інші.

2. **Надати файлу право на виконання**

   Операційна система не дозволяє виконувати будь-який текстовий файл як програму. Тому для запуску сценарію необхідно надати відповідні права доступу, зокрема право на виконання.

3. **Розмістити сценарій у каталозі, доступному для пошуку оболонкою**
   Якщо шлях до файлу не вказаний явно, командна оболонка шукає виконувані файли лише в каталогах, перелічених у змінній середовища `PATH`. Для зручності сценарії доцільно розміщувати саме в цих каталогах.

Формат файлу сценарію

Для демонстрації структури сценарію розглянемо простий приклад програми типу *Hello World*. У текстовому редакторі створіть файл з таким вмістом:

```
#!/bin/bash
# Це наш перший сценарій.
echo 'Hello World!'
```

Останній рядок — це команда `echo`, яка виводить текст на екран. Другий рядок є коментарем.

Коментарі в сценаріях починаються із символу `#`. Усе, що розміщено після цього символу до кінця рядка, ігнорується оболонкою. Коментарі можуть розміщуватися також наприкінці рядка після команди, наприклад:

```
echo 'Hello World!' # Це теж коментар
```

Аналогічно коментарі можуть використовуватися і в командному рядку, хоча практичної потреби в цьому зазвичай немає.

Перший рядок сценарію має спеціальне призначення:

```
#!/bin/bash
```

Послідовність символів `#!` називається **shebang**. Вона вказує системі, який інтерпретатор повинен використовуватися для виконання сценарію. У цьому випадку зазначено оболонку `bash`. Кожен сценарій командної оболонки повинен містити shebang у першому рядку.

Збережіть файл сценарію, наприклад, з ім'ям `hello_world`.

Надання прав на виконання

Для того щоб сценарій можна було виконати, необхідно змінити його права доступу за допомогою команди `chmod`:

```
chmod 755 hello_world
```

Існують два найпоширеніші набори прав для сценаріїв:

- **755** — сценарій доступний для виконання всім користувачам (проте редагувати його може тільки власник);
- **700** — сценарій може виконувати (а також читати та записувати) лише власник файлу.

Зверніть увагу, що файл сценарію повинен мати право на читання, інакше його неможливо буде виконати.

Після надання прав сценарій можна запустити, вказавши відносний шлях до файлу:

```
./hello_world
```

Якщо ж спробувати виконати сценарій без зазначення шляху то оболонка повідомить, що команду не знайдено. Це пов'язано не з помилкою в сценарії, а з його розташуванням.

Командна оболонка шукає виконувані файли лише в каталогах, перелічених у змінній середовища `PATH`. Її вміст можна переглянути за допомогою команди:

```
echo $PATH
```

Змінна `PATH` містить список каталогів, розділених двокрапками. Якщо сценарій розмістити в одному з цих каталогів, його можна буде запускати без зазначення шляху.

У більшості дистрибутивів Linux у `PATH` входить каталог *`~/bin`*. Якщо створити цей каталог у домашньому каталозі користувача та помістити туди сценарій, його можна запускати як звичайну команду:

```
mkdir ~/bin
mv hello_world ~/bin
hello_world
```

Якщо потрібний каталог відсутній у змінній `PATH`, його можна додати, дописавши до файлу `.bashrc` такий рядок:

```
export PATH=~/bin:"${PATH}"
```

Щоб зміни набрали чинності в поточному сеансі, необхідно повторно зчитати файл `.bashrc`:

```
. .bashrc
```

Команда **.** (крапка) є синонімом вбудованої команди `source` і виконує вказаний файл так, ніби його вміст вводиться з клавіатури.

Вибір місця розташування сценаріїв

Каталог *~/bin* є зручним місцем для зберігання сценаріїв, призначених для **особистого використання** користувача. Такі сценарії доступні лише власнику та використовуються для автоматизації індивідуальних завдань.

Сценарії, які повинні бути доступні **всім користувачам системи**, доцільно розміщувати у традиційному системному каталозі:

/usr/local/bin

Сценарії, призначені для використання **системним адміністратором**, зазвичай розміщуються в каталозі:

/usr/local/sbin

У більшості випадків програмне забезпечення, створене локально в системі (як сценарії, так і скомпільовані програми), слід розміщувати в ієрархії каталогів */usr/local*, а не в каталогах */bin* або */usr/bin*. Згідно зі стандартом ієрархії файлової системи Linux (**Filesystem Hierarchy Standard**), каталоги */bin* і */usr/bin* призначені виключно для програм, що постачаються разом із дистрибутивом операційної системи.

Додаткові рекомендації з оформлення сценаріїв

Однією з важливих вимог до сценаріїв командної оболонки є **простота супроводу**, тобто можливість легко змінювати сценарій автором або іншими користувачами відповідно до нових потреб. Для цього необхідно приділяти увагу читабельності та зрозумілості коду сценарію.

Використання довгих імен параметрів

Багато команд командної оболонки підтримують параметри з **короткими** та **довгими** іменами. Наприклад, команда *ls* дозволяє використовувати такі еквівалентні записи:

```
ls -ad
```

та

```
ls --all --directory
```

Короткі імена параметрів доцільно використовувати в командному рядку, оскільки вони зменшують обсяг ручного введення. Водночас **довгі імена параметрів** підвищують зрозумілість команд і рекомендуються для використання у сценаріях, особливо якщо сценарій буде читатися або змінюватися іншими користувачами.

У разі використання довгих або складних команд їх читабельність можна значно підвищити, розбивши команду на кілька рядків. Для цього використовується символ зворотної похилої риски \, який позначає **продовження рядка**.

Наприклад, часто адміністраторам операційної системи Linux доводиться визначати список усіх відкритих назовні портів. Для цього можна використати складний bash-сценарій в один рядок:

```
netstat -tulpn | grep -v -e "127.0.0.1" -e ":::" | awk '{print $4}' | awk -F':'
'{print $2}' | awk 'NF' | uniq
```

З першого погляду початківцю досить важно зрозуміти суть цього сценарію. Для того, аби зробити його більш зрозумілим, як правило використовується розбиття на декілька рядків:

```
netstat -tulpn | \
    grep -v -e "127.0.0.1" -e ":::" | \
    awk '{print $4}' | \
    awk -F':' '{print $2}' | \
    awk 'NF' | \
    uniq
```

Перша команда *netstat -tulpn* відповідає за відображення на екрані списку усіх портів та процесів, що їх використовують. Далі використовується такий механізм, як pipeline (конвеєр), який використовує результат роботи минулої команди, та відправляє його в наступну команду але вже як дані з якими необхідно працювати. Таким чином, наступна команда *grep -v -e "127.0.0.1" -e "::"* прибере усі записи, які пов'язані з ipv6 чи локалхостом. Після цього команда *awk '{print $4}'* залишить лише записи у форматі *адреса:порт*, а наступі awk-команди відфільтрують лише порти та пусті рядки. Під кінець команда *uniq* залишить лише унікальні записи. Таким чином можна легко отримати список усіх публічних портів.

Використання продовження рядків разом з відступами дозволяє чітко побачити логіку виконання складного сценарію та полегшує його аналіз.

Такий підхід можливий і в командному рядку, однак використовується рідко через незручність введення та редагування. Однією з відмінностей сценаріїв від інтерактивного режиму є можливість застосування **символів табуляції** для оформлення відступів. У командному рядку це неможливо, оскільки клавіша Tab використовується для автодоповнення команд і шляхів.

Зверніть увагу на відмінності між різними програмними оболонками shell, оскільки вони мають важливе значення для програмування. Під час виконання завдань переконайтеся, в якій оболонці ви працюєте (найчастіше в Linux це bash), а також яка оболонка буде використовуватися для запуску вашого командного файлу (це визначається першим рядком вашого командного

файлу). Уважно ознайомтеся з правилами використання операторів if та формування умов для перевірки. Особливу увагу приділіть команді test.

У сценаріях роботи з bash, змінні грають важливу роль. Особливо коли необхідно «запам'ятати» динамічні дані під час роботи сценарію. До прикладу можна віднести невеликий сценарій, який дозволяє знайти усі процеси, які були запущені від імені заданого користувача.

```
[ -z ${1} ] && echo "Provide user name" && exit 1

user=${1}
ps aux | grep "^${user}"
```

В першому рядку використовуючи логічну операцію (*-z* – перевірити існування змінної) ми перевіряємо чи існує аргумент під час запуску сценарію. Якщо файл сценарію має назву *checkUserProcess.sh*, то його запуск має відбуватися наступним чином:

```
./checkUserProcess.sh root
```

Наступний рядок запам'ятає аргумент як змінну *user*, аби вже в третьому рядку ми змогли відфільтрувати усі процеси та віднайти там ті, що належать вказаному користувачу.

Використання змінних передбачає синтаксис спеціального символу $ та фігурних дужок {} в яких зазначається ім'я змінної. Спеціальний симво ^ в даному прикладу використовується лише для пошуку рядка із результату роботи ps, де ім'я користувача знаходиться на початку рядка.

Завдання

Написати сценарій для оболонки bash згідно таких вимог:

1. Сценарій приймає три параметри з командного рядка. Перший параметр — це назва каталогу, в якому потрібно виконати пошук (рекурсивно включаючи

всі підкаталоги). Другий параметр — шаблон для пошуку. Якщо ці два параметри відсутні, сценарій має вивести повідомлення з інструкціями щодо правильного формату запуску. Третій параметр є необов’язковим — це назва файлу, в якому будуть збережені результати пошуку.

2. Сценарій здійснює пошук у вказаному каталозі та його підкаталогах всіх файлів, що містять послідовність байтів, яка відповідає пошуковому шаблону, і формує їх список. У цьому списку повинні бути зазначені: назва файлу, його повний шлях, тип файлу (за допомогою команди file), розмір у байтах та рядок, що відповідає шаблону.

3. Якщо під час виклику сценарію було вказано третій параметр (назву файлу), список має бути записаний у цей файл. Якщо цей параметр не задано, сценарій повинен запитати ім'я файлу під час виконання. У обох випадках, якщо файл для виведення вже існує, сценарій повинен запитати, що робити далі: переписати файл, дописати нові дані в його кінець або скасувати операцію.

4. Сценарій повинен коректно обробляти помилки, такі як некоректні імена файлів або шаблони пошуку, відсутність вказаного каталогу, помилки доступу (включаючи відсутність прав доступу до певних файлів) та відсутність файлів, що відповідають заданому шаблону. Усі помилки повинні супроводжуватися відповідними діагностичними повідомленнями.

Лабораторна робота №5 Процеси в ОС UNIX

Мета: Оволодіння практичними навичками роботи з процесами.

Теоретичні відомості

Багатозадачність та роль процесів в ОС UNIX

Сучасні операційні системи, зокрема UNIX-подібні, належать до класу **багатозадачних систем з розподілом часу**. Це означає, що вони створюють для користувача ілюзію одночасного виконання багатьох програм шляхом дуже швидкого перемикання процесора між окремими задачами. Центральним механізмом, за допомогою якого ядро Linux реалізує багатозадачність, є **процеси**.

Ядро операційної системи керує виконанням процесів, розподіляє між ними процесорний час, пам’ять та інші ресурси, а також може призупиняти виконання одних програм, надаючи можливість виконуватися іншим. Саме через систему процесів Linux організовує очікування програмами настання певних подій (ввід з клавіатури, надходження мережевих пакетів, завершення операцій введення/виведення тощо).

У практичній роботі користувача можуть виникати ситуації, коли комп’ютер починає працювати повільно або певна програма перестає реагувати на команди. У таких випадках аналіз стану процесів дозволяє визначити, які програми споживають надмірну кількість ресурсів, і за потреби коректно завершити їх виконання.

Для аналізу та керування процесами в UNIX-подібних операційних системах використовуються спеціальні утиліти командного рядка. До найбільш поширених належать:

- `ps` — відображення списку процесів;
- `top` — динамічний моніторинг стану процесів і системи;

- `jobs` — перегляд активних завдань оболонки;
- `bg`, `fg` — керування фоновими та активними завданнями;
- `kill`, `killall` — надсилання сигналів процесам;
- `shutdown` — завершення роботи або перезавантаження системи.

Життєвий цикл процесів

Під час завантаження операційної системи ядро ініціалізує виконання низки власних службових задач у вигляді процесів, після чого запускає спеціальний процес із назвою `init` (у сучасних дистрибутивах за це відповідає система ініціалізації `systemd`). Процес `init` відповідає за запуск сценаріїв початкового завантаження, більшу частину яких ви зможете віднайти в спеціальному каталозі */etc/system/system*. Ці сценарії забезпечують ініціалізацію та запуск усіх необхідних системних служб. Користувач може без проблем самостійно створювати такі сценарії, та визначати їх стан на початок роботи операційної системи. Нижче наведено приклад такого сценарію:

```
[Unit]
Description=Messanger service
After=network-online.target

[Service]
WorkingDirectory=/my-awesome-messanger
ExecStart=/bin/bash run.sh
User=bot
Group=bot
Restart=always
RestartSec=3
Environment="PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"

[Install]
WantedBy=default.target
```

Даний приклад системного сценарію демонструє роботу створеного серверу користувача *my-awesome-messanger*, який при старті системи буде автоматично запускати сервер від імені користувача *bot*.

Більшість системних служб реалізовано у вигляді **демонів** — програм, що працюють у фоновому режимі без безпосередньої взаємодії з користувачем. Завдяки цьому система виконує службові функції навіть за відсутності активних користувацьких сеансів.

В UNIX-подібних операційних системах діє фундаментальне правило: **кожен процес створюється іншим процесом**. Процес, який ініціює створення нового, називається **батьківським**, а створений процес — **дочірнім**. У результаті всі процеси в системі утворюють ієрархічну структуру.

Механізм створення процесів у UNIX

Створення нового процесу в UNIX-подібних операційних системах відбувається у два етапи та реалізується за допомогою системних викликів `fork()` і `exec()`.

Системний виклик `fork()` створює новий процес шляхом копіювання поточного процесу. У результаті з'являються два процеси — батьківський і дочірній, які мають різні значення ідентифікатора процесу (PID), але початково виконують один і той самий код.

Після створення дочірнього процесу за допомогою `fork()` зазвичай викликається один із системних викликів сімейства `exec()`, який замінює програмний код поточного процесу новою програмою. Саме таким чином командна оболонка запускає більшість прикладних програм: оболонка створює дочірній процес і замінює його програмний образ на виконуваний файл потрібної команди.

Ядро операційної системи зберігає інформацію про всі процеси, зокрема:

- ідентифікатор процесу;
- ідентифікатор батьківського процесу;
- інформацію про виділену пам'ять;

- поточний стан процесу;
- ідентифікатори користувача та власника процесу.

Кожному процесу присвоюється унікальний **ідентифікатор процесу (PID, Process ID)**. Ідентифікатори надаються у порядку зростання, при цьому процес `init` завжди має значення PID = 1.

Перегляд списку процесів за допомогою команди ps

Однією з основних утиліт для перегляду процесів є команда `ps`. У базовому режимі вона відображає обмежену інформацію лише про процеси, пов'язані з поточним сеансом користувача. Типовий вивід команди містить такі поля:

- **PID** — ідентифікатор процесу;
- **TTY** — керуючий термінал процесу;
- **TIME** — сумарний час процесора, використаний процесом;
- **CMD** — команда або програма, що виконується.

Поле **TTY** (скорочення від *teletype*) містить інформацію про термінал, з яким пов'язаний процес. Поле **TIME** відображає загальний обсяг процесорного часу, спожитого процесом з моменту запуску.

Під час використання додаткового параметра `x` команда `ps` відображає всі процеси користувача, незалежно від наявності керуючого терміналу. У цьому випадку в полі **TTY** може з'являтися символ `?`, який означає відсутність керуючого терміналу (типово для фонових служб та демонів).

У розширеному виводі команди `ps` з'являється стовпчик **STAT** (state), що відображає поточний стан процесу.

Стани процесів у ядрі операційної системи

У процесі виконання кожен процес може перебувати в одному з кількох станів, які визначаються ядром операційної системи. Перехід між станами

відбувається залежно від подій, таких як виділення процесорного часу, очікування введення/виведення або отримання сигналів. Аналіз станів процесів дозволяє оцінити поточне навантаження системи та виявити причини зниження її продуктивності.

Основні стани процесів

Позначення	Значення
R	Процес виконується або готовий до виконання
S	Процес призупинений та очікує настання події
D	Процес перебуває в безперервному очікуванні операції введення/виведення
T	Процес зупинений примусово
Z	Процес-зомбі (дочірній процес завершився, але не був видалений батьківським)

Крім основного символу стану, можуть використовуватися додаткові позначення, які характеризують пріоритет процесу:

Позначення	Значення
<	Високопріоритетний процес
N	Низькопріоритетний процес

Поняття *niceness* визначає, наскільки охоче процес поступається процесорним часом іншим процесам. Процеси з високим пріоритетом отримують більше процесорного часу, ніж процеси з низьким пріоритетом.

Планування процесів і роль планувальника

Розподіл процесорного часу між процесами здійснюється спеціальним компонентом ядра — планувальником (scheduler). Планувальник визначає, який процес і на який проміжок часу отримає доступ до процесора. Такий проміжок часу називається квантом процесорного часу.

Планувальник враховує пріоритет процесу, значення його параметра niceness, а також загальне навантаження системи. Процеси з вищим пріоритетом або нижчим значенням niceness отримують більше процесорного часу. Показник load average, який відображається у команді top, дозволяє оцінити кількість процесів, що одночасно претендують на виконання.

Перегляд процесів у стилі BSD. Команда ps aux

Комбінація параметрів `aux` дозволяє переглянути процеси, що належать усім користувачам системи. При цьому команда `ps` працює у так званому стилі BSD. У виводі з'являються додаткові стовпці, наведені в таблиці.

Заголовок	Опис
USER	Користувач — власник процесу
%CPU	Частка використання процесора, %
%MEM	Частка використання оперативної пам'яті, %
VSZ	Обсяг віртуальної пам'яті, КБ
RSS	Обсяг фізичної пам'яті (ОЗУ), використаної процесом, КБ
START	Час або дата запуску процесу

Приклад результату роботи *ps aux | grep www-data*, для виводу усіх процесів, що пов'язані з користувачем *www-data*. В даному випадку *grep* використовується лише для фільтрування виводу команди.

```
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
www-data 1383102  0.0  0.0  22752  9032 ?        S     2025   0:03 nginx: worker
www-data 1383103  0.0  0.0  22888  9032 ?        S     2025   0:04 nginx: worker
www-data 1383104  0.0  0.0  22752  9088 ?        S     2025   0:04 nginx: worker
www-data 1383105  0.0  0.0  22752  9016 ?        S     2025   0:05 nginx: worker
www-data 1383106  0.0  0.0  22896  9160 ?        S     2025   0:04 nginx: worker
www-data 1383108  0.0  0.0  22892  9104 ?        S     2025   0:04 nginx: worker
www-data 1383109  0.0  0.0  22752  9064 ?        S     2025   0:04 nginx: worker
www-data 1383110  0.0  0.0  22752  8804 ?        S     2025   0:04 nginx: worker
```

В даному прикладі можна порівняти результат роботи команди *ps* та зрозуміти, що на даний момент сервіс *nginx* запущений від імені користувача *www-data* взагалі майже не використовує ресурсів(%CPU та %MEM значення близькі до нуля), а його дочірні процеси знаходяться у стані очікування події (S).

Динамічний моніторинг процесів за допомогою команди top

На відміну від команди `ps`, яка надає лише миттєвий знімок стану системи, команда `top` дозволяє спостерігати за процесами в динаміці. Вона періодично оновлює інформацію про систему та відображає найбільш активні процеси.

Вивід команди `top` умовно поділяється на дві частини:

1. **Зведена інформація про стан системи**;
2. **Таблиця процесів**, відсортована за рівнем використання системних ресурсів.

Основні поля зведеної інформації команди top

Поле	Опис
up	Час безперервної роботи системи з моменту завантаження

users	Кількість активних користувачів
load average	Середнє навантаження системи за 1, 5 та 15 хвилин
%us	Частка процесорного часу, витраченого на користувацькі процеси
%sy	Частка процесорного часу, витраченого на системні процеси
%ni	Частка часу, витраченого на низькопріоритетні процеси
%id	Частка часу простою процесора
Mem	Використання оперативної пам’яті
Swap	Використання простору підкачки

Нижче наведено приклад роботи команди *top*

```
top - 20:50:18 up 38 days, 20:16,  1 user,  load average: 0.00, 0.01, 0.02
Tasks: 526 total,   2 running, 524 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.5 us,  0.1 sy,  0.0 ni, 99.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem : 193366.6 total, 167260.0 free,   4776.1 used,  22770.7 buff/cache
MiB Swap:  15258.0 total,  15258.0 free,      0.0 used. 188590.4 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   2172 root      20   0 1350796  85744  24488 S  14.9   0.0     67,54 cadvisor
 179161 root      20   0   11176   6904   5404 D   2.3   0.0   0:00.07 ipmi-sensors
   2938 nobody    20   0 3322432 290860  69352 S   1.7   0.1     10,25 prometheus
   1071 root      20   0  714652  25732   7772 S   0.7   0.0 223:19.35
ipmi_exporter
1811653 root      20   0 2905696  65616  32684 S   0.7   0.0 177:43.91 containerd
     16 root      20   0       0      0      0 I   0.3   0.0  68:52.18 rcu_preempt
 176260 root      20   0       0      0      0 I   0.3   0.0   0:00.03
kworker/27:0-events
```

Команда `top` дозволяє оперативно оцінити загальний стан системи, виявити процеси, що надмірно використовують ресурси, та зробити висновки щодо продуктивності роботи операційної системи. В даному прикладі можна визначити стан системи за вже зрозумілими нам позначками:

Позначка	Визначення
Up 38 days	Визначає, що система наразі працює понад місяць
1 user	Наразі в системі знаходиться активним 1 користувач
Load average 0.00, 0.01, 0.02	Дуже важливий показник. Значення наведені не у відсотках, проте у завантаженні на одне активне ядро. До прикладу – у системи наразі 48 ядер. Це означає, що число *48.0* буде означати 100% навантаження системи. За останню хвилину в середньому навантаження майже не було, проте за останні 15 хвилин (0.02), навантаження було не більше 2% на єдине ядро. Це означає, що загальне навантаження системи у відсотках за цей час було ~0.0004 -> 0.04%. Це можна дізнатися якщо значення ми розділимо на кількість ядер. Важливо розуміти, що якщо у нас Load average більше за кількість ядер – це означає що система навантажена більше ніж на 100%, та більша частина дій очікує у черзі.
%Cpu(s)	Загальна інформація про навантаження на процесор(и). Більше всього нас цікавлять us(час процесору витрачений на користувацькі процеси), sy(час використаний на системні процеси), id(час простою процесору). Усі значення надані у відсотках. На даний момент система у більшості відпочиває (id/idle показує 99.3%).

Mem/Swap	Усього у системі 192 Гб оперативної пам’яті (значення total), проте використано лише 4.7 Гб (значення used). Значення free визначається різницею між total та used + buff. Пам’ять зайнята кешем (buff) у разі потреби може бути вивільнена системою автоматично. У випадку, коли системі не вистачає оперативної пам’яті *top* продемонструє нам зміни у показника Swap, а саме поле used. Оскільки зараз біля позначки used нулі – то системі вистачає для усіх своїх задач оперативної пам’яті.

Керування завданнями оболонки

Командна оболонка UNIX-подібних систем реалізує механізм керування завданнями (job control), який дозволяє користувачу керувати процесами, запущеними з поточного термінального сеансу. Завданням (job) називається група процесів, об’єднаних оболонкою та пов’язаних з одним терміналом.

Користувач може переводити завдання у фоновий або активний режим, призупиняти їх виконання та відновлювати роботу без завершення процесів. Команди `jobs`, `bg` і `fg` дозволяють керувати такими завданнями, тоді як самі процеси продовжують існувати незалежно від режиму їх виконання.

Команда KILL:

```
kill [-<signal>] <PID>
```

Параметр <signal> може бути мнемонічним або числовим позначенням сигналу (наприклад, STOP, TERM, CONT, 9), а <PID> — це ідентифікатор процесу, якому надсилається сигнал. Якщо параметр <signal> не вказаний, за замовчуванням відправляється сигнал завершення процесу TERM (15). Цей

сигнал може бути перехоплений процесом. Однак сигнал KILL (9) не може бути перехоплений і примусово знищує процес (за умови, що у користувача є достатні права). Це дозволяє зупинити будь-який процес, якщо над ним втрачено контроль (наприклад, якщо процес завис або користувач не знає, яку команду для зупинки процесу використати). У таких випадках потрібно відкрити інший термінал (віртуальний або фізичний) і виконати команду kill -9 <PID>, де <PID> можна дізнатися за допомогою команди ps.

 SIGHUP (1) — сигнал для повідомлення про втрату з'єднання з керівним терміналом користувача;

 SIGINT (2) — сигнал переривання процесу з клавіатури (зазвичай при натисканні комбінації клавіш **Ctrl + C**);

 SIGQUIT (3) — сигнал, аналогічний **SIGINT**, але викликається символом **QUIT** і під час завершення процесу створює дамп ядра;

 SIGABRT (6) — сигнал, який процес надсилає сам собі під час аварійного завершення з виконанням дампа пам'яті у разі неможливості подальшого виконання програми;

 SIGKILL (9) — сигнал, що використовується для негайного завершення програми (не може бути оброблений, проігнорований або заблокований);

 SIGTERM (15) — сигнал для завершення роботи програми, який можна блокувати, обробляти та ігнорувати (використовується як сигнал за замовчуванням);

 SIGSTOP (17, 19, 23) — сигнал для призупинення процесу (процес не буде відновлено автоматично, доки йому не буде надіслано сигнал **SIGCONT** — дозволяє призупинити процес без його завершення).

Права доступу до процесів

Кожен процес у UNIX-подібній операційній системі виконується від імені певного користувача та має відповідні ідентифікатори UID і GID. Керування процесом, зокрема надсилання йому сигналів, можливе лише для користувача — власника процесу або для суперкористувача (root).

Якщо користувач не має достатніх прав доступу, спроба завершити або призупинити процес за допомогою команди kill завершиться помилкою. Для виконання адміністративних дій над процесами інших користувачів зазвичай використовується команда sudo.

Однак варто пам'ятати, що фонова задача може виводити інформацію на екран, що перешкоджатиме виведенню пріоритетного процесу (наприклад, при фоновому запуску команди ping). Тому слід перенаправляти виведення фонових завдань до файлів. Завдання, запущене в пріоритетному режимі, можна перевести у фоновий, спочатку призупинивши його за допомогою комбінації клавіш Ctrl+Z. Потім завдання можна відновити в пріоритетному режимі за допомогою команди fg або в фоновому — за допомогою команди bg. Кожне завдання має свій номер, який можна переглянути через команду jobs.

Однією з корисних функцій системи є можливість запускати завдання за розкладом, що реалізується за допомогою демона crond. Він зчитує файли crontab для кожного користувача і запускає завдання відповідно до розкладу від імені користувача, чий файл використовується. Щоб змінити розклад завдань, потрібно відредагувати файл crontab, після чого перезапустити crond. Це може зробити лише root або користувачі, які мають відповідні права. Для зручності рекомендується використовувати команду crontab, яка автоматично перезапускає crond після редагування.

Висновки

Механізми керування процесами є фундаментальною складовою UNIX-подібних операційних систем. Завдяки процесам ядро забезпечує

багатозадачність, ефективний розподіл ресурсів та стабільну роботу системи. Використання утиліт командного рядка дозволяє адміністратору та користувачу оперативно аналізувати стан системи, виявляти проблемні процеси та коректно керувати їх виконанням.

Завдання

1. Перегляньте список процесів користувача.

2. Перегляньте повний список процесів, запущених у системі. При цьому гарантуйте збереження інформації від "утікання" з екрана (якщо процесів багато). Зверніть увагу на ієрархію процесів. Простежте через поля PID і PPID всю ієрархію процесів тільки-но запущеної вами команди, починаючи з початкового процесу init. Зверніть увагу на формування інших полів виводу.

3. Запустіть ще одну оболонку shell. Перегляньте повний список процесів, запущених вами, при цьому зверніть увагу на ієрархію процесів і на їхній зв'язок з терміналом. Використовуючи команду kill, завершіть роботу в цій оболонці.

4. Перегляньте список задач у системі і проаналізуйте їхній стан.

5. Запустіть фоновий процес командою find / -name "*c*" -print > file 2> /dev/null & 8

6. Визначте його номер. Відправте сигнал призупинення процесу. Перегляньте список задач у системі і проаналізуйте їхній стан. Продовжить виконання процесу. Знову перегляньте список задач у системі і проаналізуйте його зміну. Переведіть процес в активний режим, а потім знову у фоновий. Запустіть цей процес із пріоритетом 5.

7. Виведіть на екран список усіх процесів, запущених не користувачем root.

8. Організуйте виведення на екран календаря <2010+№варіанту> року через 1 хвилину після поточного моменту часу.

9. Організуйте періодичне видалення в домашньому каталозі усіх файлів з розширенням *.bak і *.tmp.

Лабораторна робота №6 Основи роботи з потоками у Linux

Мета: Оволодіння практичними навичками роботи з потоками POSIX у Linux з використанням бібліотеки pthread.

Теоретичні відомості

Загальні відомості про потоки виконання

Потік виконання (thread) є найменшою одиницею виконання в операційній системі, яка може плануватися ядром процесора. На відміну від процесів, потоки одного процесу спільно використовують адресний простір пам’яті, відкриті файлові дескриптори, сигнали та інші ресурси, що забезпечує більш ефективну взаємодію між ними та зменшує накладні витрати на створення й перемикання контексту.

Реалізація потоків у Linux

Початково у Linux механізм багатопотоковості реалізовувався за допомогою системного виклику `clone()`. Цей виклик є розширенням концепції `fork()` і дозволяє точно визначати, які саме ресурси (адресний простір, файлові дескриптори, таблиця сигналів тощо) будуть спільними між батьківським та дочірнім контекстами виконання. У результаті виклику `clone()` фактично створювався новий процес, який мав спільні ресурси з батьківським, що за своєю поведінкою наближало його до потоку, однак формально він залишався процесом.

Для підтримки потоків на рівні користувацького простору в Linux спочатку була розроблена бібліотека LinuxThreads. Згодом її було замінено більш досконалою реалізацією — NPTL (Native POSIX Thread Library), яка забезпечує повнішу відповідність стандарту POSIX та значно кращу продуктивність і масштабованість.

Важливо зазначити, що на відміну від операційних систем сімейства Solaris або Windows, де процеси виступають контейнерами для потоків, у Linux потоки реалізовані як так звані «полегшені процеси» (lightweight processes). З погляду ядра кожен потік має власний ідентифікатор і може плануватися незалежно, проте з програмної точки зору вони поводяться як потоки стандарту POSIX.

Бібліотека POSIX Threads (pthread)

Інтерфейс для роботи з потоками у Linux представлений бібліотекою POSIX Threads (pthread). Вона надає стандартизований набір функцій для створення, керування, синхронізації та завершення потоків. Бібліотека `pthread` є кросплатформною та доступною для багатьох операційних систем, зокрема Linux, BSD та Windows (у вигляді відповідних реалізацій).

Використання стандарту POSIX Threads дозволяє розробляти переносимі багатопотокові програми, навички роботи з якими можуть бути застосовані в різних операційних середовищах.

Основні функції бібліотеки pthread

У межах даної лабораторної роботи передбачається ознайомлення з такими базовими функціями бібліотеки `pthread`:

- pthread_create() — використовується для створення нового потоку виконання та запуску функції, яка буде виконуватися в цьому потоці;
- pthread_exit() — завершує виконання поточного потоку. Особливу увагу слід звернути на використання цієї функції в основному потоці програми (функції `main`), оскільки її виклик не призводить до завершення всього процесу;
- pthread_join() — забезпечує очікування завершення заданого потоку та отримання коду його завершення, реалізуючи механізм синхронізації між потоками;

- pthread_cancel() — ініціює запит на примусове завершення потоку, з урахуванням його стану та встановленого режиму скасування.

Розуміння призначення та особливостей використання зазначених функцій є необхідною передумовою для подальшого вивчення механізмів багатопотокового програмування у Linux.

Створення потоків у бібліотеці pthread

У стандарті POSIX для роботи з потоками використовується бібліотека `pthread` (POSIX Threads). Потік є окремою послідовністю виконання всередині одного процесу, яка має власний лічильник команд, стек та регістри, але спільно використовує адресний простір процесу, файлові дескриптори та інші ресурси.

Створення нового потоку здійснюється за допомогою функції `pthread_create()`. Під час створення потоку вказується функція, з якої починається його виконання. Ця функція повинна мати спеціальний сигнатурний вигляд, визначений стандартом POSIX, та повертати значення типу `void *`.

Функція `pthread_create()` також дозволяє передати параметр до потоку у вигляді вказівника типу `void *`. Це дає змогу використовувати одну й ту саму функцію для кількох потоків, передаючи їм різні вхідні дані.

Якщо параметр атрибутів потоку має значення `NULL`, то потік створюється з атрибутами за умовчанням. У цьому випадку потік є приєднуваним (joinable), має стандартний розмір стеку та використовує типове планування операційної системи.

Основний потік програми (той, що виконує функцію `main`) після створення дочірнього потоку продовжує виконання незалежно від нього. Порядок виконання інструкцій у різних потоках не є визначеним і залежить від планувальника операційної системи.

Синхронізація потоків за допомогою pthread_join

Оскільки потоки виконуються конкурентно, порядок завершення основного та дочірніх потоків не гарантується. У деяких випадках необхідно забезпечити, щоб один потік завершив свою роботу лише після завершення іншого. Для цього використовується функція `pthread_join()`.

Функція `pthread_join()` блокує виконання потоку, який її викликає, до моменту завершення вказаного потоку. Це дозволяє організувати синхронізацію між потоками та впорядкувати виконання програми.

Використання `pthread_join()` є аналогом очікування завершення дочірнього процесу в моделі процесів UNIX. Застосування цієї функції особливо важливе у випадках, коли результат роботи дочірнього потоку необхідний для подальших обчислень у батьківському потоці.

Передавання даних потокам

Для передачі даних до потоку використовується параметр типу `void *`, який передається у функцію потоку під час її створення. Усередині функції потоку цей параметр може бути приведений до необхідного типу.

Такий підхід дозволяє створювати кілька потоків, які виконують одну й ту саму функцію, але працюють з різними наборами даних. Це є типовим сценарієм паралельних обчислень, коли один алгоритм застосовується до різних вхідних даних.

Під час передавання параметрів необхідно враховувати час життя даних. Дані, на які посилається переданий вказівник, повинні залишатися доступними протягом усього часу виконання потоку. Неправильне керування пам’яттю може призвести до непередбачуваної поведінки програми.

Примусове завершення потоку

У деяких ситуаціях виникає потреба примусово зупинити виконання потоку. Для цього в бібліотеці `pthread` передбачена функція `pthread_cancel()`.

Функція `pthread_cancel()` надсилає потоку запит на скасування виконання. Саме завершення потоку відбувається не миттєво, а під час досягнення так званих точок скасування. До таких точок належать, зокрема, функції очікування, введення/виведення та інші системні виклики.

Поведінка потоку під час скасування залежить від його стану та налаштувань. Потік може бути завершений коректно або аварійно, якщо не передбачено відповідної обробки завершення.

Cleanup-обробники потоків

Для забезпечення коректного завершення потоку, зокрема у випадку його примусового скасування, стандарт POSIX передбачає механізм cleanup-обробників. Вони реєструються за допомогою функції `pthread_cleanup_push()` та автоматично виконуються перед завершенням потоку.

Cleanup-обробники дозволяють виконати необхідні дії, такі як виведення повідомлення, звільнення пам’яті або закриття ресурсів, незалежно від причини завершення потоку. Це особливо важливо у багатопотокових програмах, де некоректне завершення одного потоку може призвести до витоків ресурсів або нестабільної роботи всієї програми.

Використання cleanup-обробників є рекомендованою практикою під час розробки програм, що використовують механізм скасування потоків.

Завдання

1. Напишіть програму, що створює потік. Застосуйте атрибути за умовчанням. Батьківський і дочірній потоки мають роздрукувати по десять рядків тексту.

2. Модифікуйте програму так, щоби батьківський поток здійснював роздрукування після завершення дочірнього (функція pthread_join()).

3. Напишіть програму, що створює чотири потоки, що виконують одну й ту саму функцію. Ця функція має роздруковувати послідовність текстових рядків, переданих як параметр. Кожний зі створених потоків має роздруковувати різні послідовності рядків.

4. Дочірній потік має роздруковувати текст на екран. Через дві секунди після створення дочірнього потоку, батіківський потік має перервати його (функція pthread_cancel()).

5. Модифікуйте програму так, щоби дочірній потік перед завершенням роздруковував повідомлення про це (pthread_cleanup_push()).

Лабораторна робота №7 Засоби синхронізації і взаємодії процесів

Мета: Оволодіння практичними навичками використання засобів міжпроцесової взаємодії в Linux.

Теоретичні відомості

Міжпроцесна взаємодія (Inter-process communication, IPC) — це набір методів і засобів, що дозволяють процесам обмінюватися даними та координувати свою роботу. IPC необхідна для того, щоб процеси з окремими адресними просторами могли ефективно обмінюватися інформацією і синхронізуватися. Використання IPC не зводиться лише до передачі даних — важливим аспектом є правильна синхронізація, яка дозволяє уникнути проблем типу гонки даних. Наприклад, розділювана пам’ять без контролю доступу через семафори може призвести до одночасного запису різними процесами в один сегмент, що зробить дані некоректними.

Передумови

Перед вивченням IPC варто пам’ятати базові системні виклики, які застосовуються для роботи з процесами:

- `fork()` — створення нового процесу
- `wait()` — очікування завершення дочірнього процесу
- `getpid()`, `getppid()` — отримання ідентифікаторів процесу

Для базової синхронізації можна використовувати:

- `signal()`, `kill()` — найпростіший спосіб передати сигнал іншому процесу
- `sleep()`, `alarm()`, `pause()` — прості механізми керування часом виконання

Ці виклики дозволяють створювати і координувати роботу процесів, але не забезпечують безпечного обміну великими обсягами даних.

Класифікація IPC

Сучасні механізми IPC поділяються на кілька груп:

1. Сигнали (Signals)

Сигнали дозволяють процесам повідомляти один одному про певні події або вимагати виконання дій. Це простий механізм синхронізації, але він обмежений передачею коротких повідомлень (подія/сигнал).

2. Потокові канали (Pipes)

Pipe - це однонаправлений канал, який дозволяє передавати потік байтів між двома процесами. В основному використовується між батьківським і дочірнім процесами. Перевага - простота та швидкість, недолік - обмеження на кількість процесів і однонаправленість.

3. Черги повідомлень (Message Queues)

Message queue дозволяє організувати структурований обмін даними між процесами. Кожне повідомлення має тип і вміст, процеси можуть вибірково отримувати повідомлення. Перевага — контроль і черговість обробки повідомлень, недолік — повільніше, ніж shared memory.

4. Розділювана пам’ять (Shared Memory)

Shared memory дозволяє двом і більше процесам швидко обмінюватися великими обсягами даних без копіювання. Перевага — швидкість, недолік — потреба в синхронізації, інакше можливі гонки даних.

5. Семафори (Semaphores)

Семафори забезпечують безпечну синхронізацію доступу до спільних ресурсів. Використовуються для організації критичних секцій, контролю одночасного доступу до пам’яті або файлів.

Роль синхронізації

Синхронізація дозволяє процесам працювати узгоджено. Без синхронізації навіть надійні механізми передачі даних можуть призвести до

помилок через гонки даних. Наприклад, двоє процесів одночасно читають і змінюють спільний сегмент пам’яті — результат буде непередбачуваним. Тому для shared memory завжди використовують семафори або mutex.

Порівняльний підхід

Pipe vs Message Queue: Pipe швидший і простіший, але обмежений двома процесами та однонапрямковий. Message queue дозволяє працювати з великою кількістю процесів і контролювати черговість обробки повідомлень.

Message Queue vs Shared Memory: Message queue зручна для структурованих повідомлень, але швидкість нижча. Shared memory ефективна для обміну великими обсягами даних, але потребує синхронізації.

Shared Memory + Semaphore: Комбінація дозволяє безпечно організувати швидкий обмін великими обсягами даних між багатьма процесами. Семафор контролює доступ, уникнувши гонок даних.

Системні виклики IPC

Для реалізації IPC використовуються такі виклики:

- **Pipe:** `pipe()`, `read()`, `write()`
- **Семафори:** `semget()`, `semop()`, `semctl()` (`<sys/ipc.h>`, `<sys/sem.h>`)
- **Черги повідомлень:** `msgget()`, `msgsnd()`, `msgrcv()`, `msgctl()` (`<sys/ipc.h>`, `<sys/msg.h>`)
- **Shared memory:** `shmget()`, `shmat()`, `shmdt()`, `shmctl()` (`<sys/ipc.h>`, `<sys/shm.h>`)

Порівняльна таблиця

Механізм	**Переваги**	**Недоліки**	**Рекомендоване використання**
pipe	Простота, швидкий	Тільки два	Невеликі обсяги даних

Механізм	**Переваги**	**Недоліки**	**Рекомендоване використання**
	обмін	процеси, однонапрямок	між батьком та дитиною
message queue	Структуровані повідомлення, вибірка за типом	Повільніше, ніж shared memory	Коли потрібна черговість та контроль
shared memory	Швидкий обмін великими обсягами	Потрібна синхронізація	Ефективна передача великих даних
shared memory + semaphore	Швидко та безпечно	Потребує правильного налаштування	Критичні секції, складні системи з багатьма процесами

Міжпроцесна взаємодія (Inter-process communication, IPC) — це набір методів та інструментів, що дозволяють процесам обмінюватися даними. Також можна згадати взаємодію потоків, оскільки деякі засоби (наприклад, семафори) можуть використовуватися для синхронізації потоків одного процесу. Проте у цьому випадку увага приділяється засобам, які операційна система надає потокам різних процесів, що мають окремі адресні простори і можуть виконуватись навіть на різних комп'ютерах, з'єднаних мережею. Серед прикладів механізмів IPC: сигнали, сокети, семафори, файли, повідомлення, канали та спільна пам'ять. Для їх реалізації використовуються відповідні системні виклики.

Зверніть увагу на наступні системні виклики, які можуть знадобитися (деякі з них, можливо, ви вже використовували раніше):

Створення та завершення процесу, отримання інформації про нього: fork(), exit(), getpid(), getppid().

Синхронізація процесів: signal(), kill(), sleep(), alarm(), wait(), pause().

Створення інформаційного каналу та робота з ним: pipe(), read(), write().

Для роботи з семафорами існують три основні системні виклики:

semget() для створення або отримання доступу до набору семафорів.

semop() для маніпуляцій значеннями семафорів, що дозволяє процесам синхронізуватися.

semctl() для виконання різних операцій з керування набором семафорів.

Прототипи цих викликів описані у файлах:

#include <sys/ipc.h>

#include <sys/sem.h>

Для обміну повідомленнями між процесами застосовується механізм черг, що підтримується наступними викликами:

msgget() для створення нової черги або отримання дескриптора існуючої черги повідомлень.

msgsnd() для додавання повідомлення до черги.

msgrcv() для отримання повідомлення з черги.

msgctl() для виконання операцій керування чергою.

Прототипи цих системних викликів описані у файлах:

#include <sys/ipc.h>

#include <sys/msg.h>

Для роботи зі спільною пам'яттю використовуються наступні системні виклики:

shmget() створює новий сегмент спільної пам'яті або знаходить існуючий за ключем.

shmat() підключає сегмент до віртуального адресного простору процесу.

shmdt() відключає сегмент від віртуального адресного простору процесу.

shmctl() керує параметрами сегмента спільної пам'яті.

Прототипи цих викликів описані у файлах:

#include <sys/ipc.h>

#include <sys/shm.h>

Завдання

1. Два дочірніх процеси виконують деякі цикли робіт, передаючи після закінчення чергового циклу через чергу повідомлень батьківському процесові чергові чотири рядка деякого віршу, при цьому перший процес передає непарні чотиривірші, а другий — парні. Цикли робіт процесів не збалансовані у часі. Батьківський процес компонує з фрагментів, що йому передають, закінчений вірш, і виводить його по завершенню роботи обох дочірніх процесів. Розв’язати задачу з використанням апарату семафорів.

2. Ознайомитись з основними поняттями механізму IPC.

3. Розібратисьз набором системних викликів, що забезпечують розв’язання завдання.

4. Налагодити складену програму, використовуючи інструментарій ОС Linux.

Лабораторна робота №8. Інтерфейс файлової системи в ОС Linux.

Мета: Оволодіння практичними навичками використання файлової системи в Linux.

В операційній системі Linux взаємодія прикладних програм з даними на носіях інформації організована через рівень абстракції, що називається Віртуальною Файловою Системою (VFS). Наявність VFS дозволяє ядру підтримувати одночасно множину різних файлових систем, таких як ext4, NTFS або мережеві файлові системи. Процеси користувача звертаються до файлів за допомогою універсальних системних викликів, а ядро встановлює відповідність між цим загальним набором операцій та специфічними функціями конкретної файлової системи.

З точки зору внутрішньої організації ядра, системні функції класифікують на кілька категорій, хоча деякі з них можуть належати одразу до кількох груп. До першої категорії належать функції, які повертають дескриптори файлів для подальшого використання (наприклад, open, dup, pipe). Друга категорія включає функції, що використовують алгоритм namei для аналізу імені шляху пошуку (open, stat, unlink). Третя категорія — це функції, які працюють з індексними дескрипторами, використовуючи алгоритми ialloc та ifree для їх виділення та звільнення. Четверта категорія відповідає за зміну атрибутів файлу (chmod, chown). П'ята категорія дозволяє проводити введення-виведення з використанням алгоритмів alloc і free та алгоритмів виділення буферів (read, write). Останні категорії включають функції, що змінюють структуру файлової системи (mount, link) або уявлення процесу про дерево каталогів (chdir, chroot).

Початок роботи з файлом завжди пов'язаний із системним викликом open, який перетворює шлях до файлу у числовий дескриптор — перший крок для звернення до даних. Якщо файл не існує, його можна створити за допомогою функції creat або, у випадку спеціальних файлів (каналів, файлів пристроїв), функції mknod. Після отримання дескриптора процес може виконувати операції читання та запису за допомогою функцій read та write. Ці функції зазвичай забезпечують послідовний доступ, проте використання виклику lseek дозволяє

змінити поточну позицію вводу-виводу, реалізуючи довільний доступ до даних. Коли робота з файлом завершена, виклик close звільняє дескриптор.

Окрему групу становлять функції роботи з метаданими та структурою. Оскільки зміна власника або режиму доступу здійснюється над індексним дескриптором, а не над файлом як таким, операції chown та chmod змінюють відповідні атрибути безпосередньо в inode. Функції stat та fstat дозволяють отримати повну інформацію про статус файлу: тип, власника, права доступу, розмір, кількість зв'язків та часові мітки. Керування ієрархією файлової системи здійснюється через монтування: функція mount пов'язує файлову систему з розділу диска з існуючим деревом каталогів, а umount — вилучає її. Зв'язок імені файлу з його даними регулюється функціями link, яка створює нове ім'я (жорстке посилання) у каталозі для існуючого індексного дескриптора, та unlink, яка видаляє точку входу для файлу з каталогу. Системна функція dup копіює дескриптор файлу в перше вільне місце в таблиці дескрипторів, дозволяючи мати кілька посилань на один відкритий файл.

Для реалізації цих операцій ядро Linux використовує три взаємопов'язані структури даних. Першою є масив файлових дескрипторів процесу (fd), що створюється для кожного процесу відразу після його породження. Кожен елемент цього масиву містить вказівник на місце розташування відповідного елементу в другій структурі — таблиці відкритих файлів (file table). Кожен елемент таблиці відкритих файлів містить інформацію про режим відкриття файлу та, що найважливіше, поточне положення покажчика читання-запису. Третьою структурою є таблиця індексних дескрипторів (inode), яка містить фізичні атрибути файлу. В реалізації VFS у Linux посилання між таблицею відкритих файлів та індексним дескриптором часто здійснюється через об’єкти елементів каталогу (dentry), впроваджені для кешування шляхів.

Розуміння цієї трирівневої структури важливе для пояснення взаємодії процесів. Якщо один і той самий файл відкривається кількома незалежними процесами, йому відповідає один індексний дескриптор, але стільки елементів таблиці відкритих файлів, скільки разів він був відкритий. Це означає, що

кожен процес має власне зміщення (курсор) у файлі. Однак є виняток: коли файл відкривається процесом, а потім цей процес породжує нащадка через системний виклик fork, процес-нащадок успадковує копію масиву дескрипторів батька. У цьому випадку обидва процеси посилаються на один і той самий елемент таблиці відкритих файлів, спільно використовуючи положення покажчика читання-запису.

Завдання

Варіант 1

Необхідно написати програму, що ілюструє зміни в таблиці файлових дескрипторів при маніпуляціях з файлами. Процес відкриває N файлів (існуючих або нових).

Сценарій виконання:

1. Відкриття першого файлу користувача.
2. Відкриття другого файлу.
3. Відкриття третього файлу.
4. Усічення третього файлу до 0 байт (зміна розміру).
5. Копіювання вмісту другого файлу в третій.

Вимога: Після кожного кроку виводити поточний стан таблиці дескрипторів.

Варіант 2

Реалізувати емуляцію перенаправлення стандартного виводу (stdout) у файл. Програма має відображати зміни у трьох структурах: таблиці дескрипторів, таблиці відкритих файлів та масиві дескрипторів процесу.

Сценарій виконання:

1. Ініціалізація стандартних потоків (0, 1, 2).
2. Відкриття нового файлу користувача.
3. Закриття стандартного виводу (імітація close(1)).
4. Дублювання дескриптора файлу користувача на місце звільненого (імітація dup(fd)).
5. Закриття початкового дескриптора файлу користувача.

Вимога: Виводити стан усіх трьох таблиць після кожного етапу.

Варіант 3

Змоделювати ситуацію конкурентного доступу до одного файлу двома процесами: один читає, інший пише. Необхідно продемонструвати механізм блокування ("заморожування") покажчика одного процесу, коли обидва намагаються працювати з одним і тим же фізичним блоком диска.

Сценарій: Два процеси звертаються до одного файлу, виникає колізія на рівні блоку.

Вимога: Відобразити динаміку створення та зміни таблиць відкритих файлів та масивів дескрипторів для обох процесів під час блокування та після нього.

Варіант 4

N процесів працюють з одним спільним файлом, але використовують різні режими доступу. Програма повинна показати, як змінюються поля (зокрема зміщення/offset) у таблиці відкритих файлів.

Сценарій виконання:

1. Процес 0 відкриває файл для читання (O_RDONLY).
2. Процес 1 відкриває той самий файл для запису (O_WRONLY).
3. Процес 2 відкриває файл для дозапису в кінець (O_APPEND).
4. Процес 0 зчитує X байт.
5. Процес 1 записує Y байт.
6. Процес 2 додає Z байт.

Вимога: Виводити вміст таблиць відкритих файлів для всіх процесів після кожної операції I/O.

Варіант 5

Продемонструвати базову роботу підсистеми вводу-виводу Linux для одного процесу. Акцент на зміні внутрішніх зміщень (курсорів) у таблицях.

Сценарій виконання:

1. Ініціалізація стандартних потоків (stdin, stdout, stderr).
2. Відкриття файлу А.
3. Відкриття файлу Б.
4. Запис 20 байт у файл А.

5. Читання 15 байт з файлу Б.
6. Запис ще 45 байт у файл А.

Вимога: Відображати стан таблиці дескрипторів, таблиці відкритих файлів та масиву дескрипторів процесу на кожному етапі.

Варіант 6

Створити програму, яка візуалізує процес заповнення системних таблиць при послідовному відкритті файлів.

Сценарій виконання:

1. Автоматичне відкриття стандартних потоків (0, 1, 2).
2. Явне відкриття першого файлу.
3. Явне відкриття другого файлу.
4. Явне відкриття третього файлу.

Вимога: Після кожного відкриття друкувати вміст таблиці дескрипторів, таблиці відкритих файлів та масиву дескрипторів процесу.

Варіант 7

Складний сценарій з ієрархією процесів. Є N процесів, що відкривають файли. Частина з них (M) породжує нащадків через fork(), а деякі нащадки (K) відкривають додаткові файли.

Сценарій виконання:

1. Процеси 0, 1, 2 відкривають по 2 файли кожен.
2. Процес 0 створює нащадка (Процес 3) — демонструється успадкування таблиць.
3. Процес 1 створює нащадка (Процес 4) — демонструється успадкування таблиць.
4. Процес 4 відкриває ще 2 нових файли.

Вимога: На кожному етапі виводити масиви дескрипторів для всіх активних процесів.

Варіант 8

Реалізувати емуляцію перенаправлення стандартного вводу (stdin) з файлу.

Сценарій виконання:

1. Ініціалізація стандартних потоків.
2. Читання 5 байт зі стандартного вводу (клавіатури).
3. Відкриття файлу користувача.
4. Закриття стандартного вводу (імітація close(0)).
5. Дублювання дескриптора файлу на місце 0 (імітація dup(fd)).
6. Закриття початкового дескриптора файлу.
7. Читання 10 байт зі "стандартного" вводу (який тепер пов'язаний з файлом).

Вимога: Виводити зміни в таблиці дескрипторів та таблиці відкритих файлів.

Варіант 9

Дослідити життєвий цикл дескрипторів при створенні та завершенні процесу-нащадка.

Сценарій виконання:

1. Процес-предок відкриває стандартні потоки + 4 файли.
2. Процес-предок закриває 2 файли.
3. Виклик fork(): нащадок успадковує стан таблиць.
4. Процес-нащадок закриває ще частину успадкованих файлів.
5. Завершення процесу-нащадка (предок очікує).

Вимога: Фіксувати зміни в таблицях відкритих файлів та масивах дескрипторів обох процесів.

Варіант 10

Дослідити вплив операцій переміщення курсору (lseek) та дублювання дескрипторів на спільне використання таблиці відкритих файлів. Програма має показати різницю між незалежним відкриттям файлу та його дублюванням.

Сценарій виконання:

1. Відкриття файлу A (отримання fd1).
2. Запис 10 байт у fd1.
3. Створення копії дескриптора через dup(fd1) -> отримання fd2.
4. Переміщення курсору (lseek) для fd2 на 50 байт вперед.
5. Відкриття того ж самого файлу A заново через open() -> отримання fd3.

6. Запис 10 байт у fd3.

Вимога: Після кожного кроку друкувати таблицю відкритих файлів (особливу увагу приділити полю "offset/зміщення") та пояснити, чому зміна зміщення в fd2 вплинула (або не вплинула) на fd1 та fd3.

Список використаної літератури

1. Andrew S. Tanenbaum, Herbert Bos: Modern Operating Systems (Fourth Edition). March 10, 2014. Pearson Education, Inc. USA. May 2013. ISBN : 978-0-13-359162-0. 1136 pages.
2. Авраменко В. С., Авраменко А. С. Основи операційних систем. Навчальний посібник. – Черкаси: ЧНУ імені Богдана Хмельницького, 2018. – 524 с.: іл. ISBN 966-552-157-8
3. Brian W. Kernigan, Rob Pike: UNIX Programming Enviornment. 1984. Bell Laboratories; PRENTICE-HALL; Whitehall Books. ISBN : 0-13-937699-2. 376 pages.
4. William Stallings: Operating Systems: Internals and Design Principles, 7th Editions. 2012. ISBN : 978-0-13-230998-1. 820 pages.
5. William Stallings: Operating Systems: Internals and Design Principles, 4th Editions. 2000. ISBN : 5-8459-0310-6. 848 pages.
6. Harvey M. Deitel, Paul J. Deitel, David R. Choffnes: Operating Systems (3rd Edition)2003. ISBN : 978-0131828278. 1100 pages.
7. Бондаренко М.Ф., Качко О.Г. Операційні системи: Навчальний посібник. – Х.: Компанія СМІТ, 2008. – 432 с.
8. Scott Granneman: Linux Phrasebook (Developer's Library) 2nd Edition. 2015. Addison-Wesley Professional. ISBN-13 : 978-0321833884
9. Robert Love: Linux System Programming, Second Edition. 2013. Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472. ISBN: 978-1-449-33953-1
10. Robert Love: Linux Kernel Development Third Edition. 2010. ISBN: 978-0-672-32946-3
11. William E. Shotts Jr.: The Linux Command Line: A Complete Introduction. No Starch Press, 2012 / 2nd Edition 2019. ISBN: 978-1593273897