МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ОПЕРАЦІЙНІ СИСТЕМИ

Методичні вказівки до комп'ютерного практикуму

для студентів спеціальностей

113 «Прикладна математика»
125 «Кібербезпека»

Рекомендовано Вченою радою Фізико-технічного інституту НТУУ «КПІ ім. Ігоря Сікорського»

Київ
НТУУ "КПІ ім. І. Сікорського"
2019

Операційні системи. Методичні вказівки до комп’ютерного практикуму. Для студентів спеціальностей 113 «Прикладна математика», 125 «Кібербезпека» / Укладач Грайворонський М.В. — Київ, НТУУ «КПІ ім. І.Сікорського», 2019 — 74 с.

Гриф надано Вченою радою Фізико-технічного інституту НТУУ «КПІ ім. Ігоря Сікорського» (Протокол № 2/2019 від 8 лютого 2019 року)

Електронне видання

ОПЕРАЦІЙНІ СИСТЕМИ

Методичні вказівки до комп’ютерного практикуму

для студентів спеціальностей

113 «Прикладна математика»
125 «Кібербезпека»

Укладач: Микола Владленович Грайворонський, к. ф-.м. н, доц.

Відповідальний редактор: Т.В. Литвинова, к.т.н., доц.

Рецензент: Л. Ю. Гальчинський, к.т.н, доц.

Зміст

Загальні вимоги до виконання комп’ютерного практикуму

Обладнання

Для виконання практикуму вам потрібен комп'ютер, на якому встановлена операційна система із сімейства UNIX. Передбачається виконання робіт в комп’ютерній лабораторії ФТІ на системі Linux з використанням технології віртуальних машин.

Для самостійної підготовки рекомендуються вільно (безкоштовно) розповсюджувані операційні системи з сімейства Linux (наприклад, Debian, Ubuntu, або інші). Можна виконувати роботи також на системах BSD (наприклад, FreeBSD), але слід пам’ятати, що вони мають певні відмінності від Linux. Для виконання окремих завдань будуть необхідні права root. Через обмеження, що встановлені політикою безпеки, система Mac OS X може не дозволити виконання деяких завдань, тому її використання для цього практикуму не рекомендується.

Зверніть увагу, що для виконання робіт Вам не потрібна багатовіконна система X (X Window System), як і інші графічні системи (Gnome, KDE).

Серйозних вимог до апаратного забезпечення немає. Якщо не використовувати графічних систем, не буде проблем навіть із дуже застарілим комп’ютером.

Вимоги до виконання робіт

Методичні вказівки до кожної роботи складаються з завдання для самостійної підготовки й завдання для виконання. Завдання для самостійної підготовки може виконуватись без комп’ютера (якщо є роздруковані матеріали довідкової системи). Воно передбачає ознайомлення з теоретичним матеріалом і завданням до виконання, а також підготовку до виконання практичної частини роботи.

Деякі теоретичні відомості містяться безпосередньо в методичних вказівках. Але не слід обмежуватись цією далеко не повною інформацією. Зверніться до рекомендованої літератури. Багато додаткових відомостей можна знайти в Інтернеті (пошук варто починати з www.google.com, а перегляд результатів пошуку варто починати з Вікіпедії). Докладні довідкові дані по кожній (майже кожній) команді UNIX/Linux можна одержати, набравши на клавіатурі `man <необхідна_команда>`.

Роботи виконуються в комп’ютерній лабораторії в присутності викладача. При виконанні роботи студент повинен фіксувати все, що відбувається на екрані монітора комп’ютера.

Зверніть увагу на те, що в індивідуальних завданнях часто зустрічаються певні слова, рядки, вирази в кутових дужках. Це означає, що ви повинні підставити замість цього виразу разом із дужками значення цього виразу. Наприклад, якщо вас звуть Сергій, то замість `студент_<ваше ім'я>` ви повинні написати (набрати на клавіатурі) `студент_Сергій`.

У деяких випадках вам будуть запропоновані **варіанти** завдань, обирати з яких необхідно той, що відповідає номеру вашої залікової книжки (або той, що вказав викладач). Не намагайтесь удавати з себе бравого солдата Швейка і виконувати лише ті **пункти** завдання, які визначаються номером вашої залікової книжки. Виконати необхідно **усі пункти без виключення** (якщо викладач не вказав інше), а в окремих пунктах може бути явно вказано, що слід обрати один з **варіантів**.

Для захисту роботи студенти повинні надати викладачу оформлений звіт, вимоги до оформлення якого викладені далі. Також студент повинен дати правильні відповіді на додаткові запитання за матеріалами цієї лабораторної роботи. Зверніть увагу на те, що студент повинен володіти всім теоретичним і практичним матеріалом, а не тільки тією його частиною, яка безпосередньо входила до індивідуального завдання.

Вимоги до оформлення звіту

Звіт повинен містити:

1. назву роботи, тему і мету;
2. номер варіанту і зміст індивідуального завдання;
3. дату і місце виконання роботи, коротку характеристику системи, на якій було виконано роботу (клас комп'ютера, версію ОС тощо);
4. протокол виконання роботи, де чітко позначено, що вводить користувач з клавіатури, і що в результаті з'являється на екрані (необхідно показати всі результати; якщо іде запис даних в файл, вміст файлу також слід роздрукувати);
5. (**обов'язково!!!**) висновки по роботі, які повинні містити саме висновки, які студент може зробити в результаті виконання цієї роботи, а не констатацію факту "в результаті виконання роботи я навчився…"; у висновках може бути зроблена оцінка наявності певних функцій операційної системи, якості або гнучкості їх реалізації, зручності інтерфейсу тощо;
6. на кожній сторінці (в верхньому або нижньому колонтитулі) прізвище студента і номер групи або залікової книжки.

Звіт з кожної роботи повинен мати титульний лист, оформлений згідно стандартних вимог, як у наведеному далі прикладі.

Міністерство освіти і науки України
Національний технічний університет України
"Київський політехнічний інститут імені Ігоря Сікорського"
Фізико-технічний інститут

ОПЕРАЦІЙНІ СИСТЕМИ
Комп’ютерний практикум

Робота №___

Виконав
студент гр.<№групи><прізвище ініціали>

Перевірив
<прізвище ініціали викладача>

Київ - <поточний рік>

Загальні відомості про UNIX-подібні системи

Системи, які узагальнюють ознакою “UNIX-подібні”, є багатозадачними багатокористувацькими системами з розділенням часу. Насправді, це досить різні системи, які зберегли подібний інтерфейс користувача і стандартизований програмний інтерфейс (інтерфейс системних викликів), а також схожі принципи побудови файлових систем.

Всі ці системи є нащадками однієї системи, яку було розроблено на початку 70-х років XX століття, і яка отримала назву UNIX. Шляхи розвитку цієї системи спочатку розійшлися, потім було проведено велику роботу по відновленню сумісності систем. Тепер розрізняють дві гілки розвитку систем UNIX: SVR4 (System 5 Release 4) розробки підрозділів AT&T і BSD компанії Berkeley Software Distribution. Правом на торгову марку “UNIX” володіла AT&T, потім ці права (разом з патентами і правами на вихідні коди системи) були продані Novell, а далі – SCO (Santa Cruz Operation). Інші системи мають власну назву, що, як правило, дещо співзвучна UNIX (AIX, Minix, Xenix, Irix тощо). Коли бажають підкреслити, що мають на увазі різні операційні системи, що мають спільні риси, часто називають їх системи *Nix.

Фактично третім напрямком розвитку UNIX-подібних систем з початку 1990-х років стала система Linux, яка з самого початку була спробою незалежно, без застосування будь-якого захищеного майновими правами програмного коду, відтворити основні риси системи UNIX для застосування її на персональних комп’ютерах. Користуючись підтримкою численних прихильників програмного забезпечення з відкритим кодом, Linux набула великої популярності, а разом з тим – і значного технологічного удосконалення. Слід визнати, що за останні роки Linux залишила позаду більшість класичних UNIX-систем, майже всі з яких фактично зійшли зі сцени. Деякі «останні з могікан», такі як Solaris (тепер належить Oracle) і AIX (IBM) ще мають переваги у певних умовах застосування (наприклад, для великих серверів баз даних), але ринкова ніша цих систем невпинно звужується, а Linux – поширюється. Гнучкість Linux, значною мірою зумовлена відкритим програмним кодом, дозволяє успішно застосовувати її не лише на персональних комп’ютерах, для яких вона була створена, а й на великих навантажених серверах, і на потужних обчислювальних кластерах, і у мережному обладнанні (маршрутизаторах), і у смартфонах, і навіть у вбудованих пристроях керування (як, наприклад, у деяких сучасних телевізорах).

Найбільш помітні для користувача відмінності *Nix-систем:

- традиційні командні інтерпретатори – sh, ksh для SVR4; csh, tcsh для BSD; bash для Linux;
- файлові системи – S5 для SVR4, UFS для BSD, ext4 для Linux.

Також є численні відмінності в іменах і розміщенні важливих конфігураційних файлів. Найбільш відомі операційні системи, що їх можна віднести до SVR4: Solaris (розроблена Sun Microsystems, тепер належить Oracle), IRIX (Silicon Graphics), AIX (IBM), SCO UNIX. За зовнішніми ознаками до SVR4 також близька система Linux. До BSD відносяться в першу чергу системи розробки власне BSD, в тому числі відкриті проекти OpenBSD і FreeBSD. Системи BSD також покладено в основу MAC OS X.

Для того, щоб користувач міг розпочати роботу в системі, він повинен проідентифікувати себе і підтвердити ідентифікацію паролем. Для цього адміністратор повинен заздалегідь створити в системі обліковий запис (іноді кажуть – бюджет, англ. – *account*) цього користувача. Інформацією, що ідентифікує користувача є *login* або *userid* (*login* – це ім'я-ідентифікатор з одного слова, який використовується при взаємодії користувача з системою, *userid* – числовий ідентифікатор користувача, яким користується система). Інформацією, що автентифікує (підтверджує тотожність) користувача є пароль (рядок символів, на який у залежності від конфігурації накладаються деякі обмеження).

Після входу в систему користувач опиняється в графічному багато-віконному середовищі. Щоб потрапити до командної оболонки (*shell*), в якій безпосередньо будуть виконуватися всі лабораторні роботи, слід запустити термінальне вікно – Terminal. Оболонка може запускати на виконання файли, що містять програмний код. Також вона може виконувати послідовність команд, що міститься в текстових файлах – так званих пакетних файлах, файлах сценаріїв, або, як їх ще називають, скріптах (від англ. – *script*). До складу операційної системи, крім ядра, входить велика кількість окремих програмних файлів – системних утиліт. Переважна більшість команд системи UNIX є так званими зовнішніми командами, тобто фактично вони означають запуск тієї чи іншої утиліти. Утиліти використовують програмний інтерфейс системи UNIX (інтерфейс системних викликів) і звертаються до внутрішніх структур ядра системи. Вони забезпечують гнучкі можливості керування файловими системами, процесами, контролю стану системи.

Докладну довідку щодо кожної команди UNIX ви можете отримати використовуючи довідкову систему `man`, давши команду:

```
man <команда>
```

Довідковий матеріал розбито на розділи. В описах (як у довідковій системі, так і в літературі) прийнято супроводжувати посилання на певну команду, системний виклик, функцію, номером відповідного розділу довідкової системи. Наприклад, `man(1)`, `passwd(1)`, `passwd(4)`. Нумерація розділів трохи різна в SVR4 і BSD:

Розділ	BSD	SVR4
Команди загального призначення (довідник користувача)	1	1
Системні виклики (довідник програміста)	2	2
Бібліотечні функції (довідник програміста)	3	3
Інтерфейси ядра (довідник програміста)	4	7
Формати конфігураційних і системних файлів	5	4
Різнобічна інформація	7	5
Системні утиліти (довідник системного адміністратора)	8	1M

Докладнішу інформацію про історію системи UNIX, інших UNIX-подібних систем і, зокрема, Linux, можна отримати з книг, наприклад, [1].

Програмне забезпечення з відкритим кодом

Програмним забезпечення (ПЗ) з відкритим кодом називають таке ПЗ, яке:

- вільно розповсюджується;
- доступно у вигляді вихідного коду (тексту програм);
- дозволено модифікувати.

Крім цього, існує ПЗ, що використовує відкриті стандарти (тобто стандарти, які вільно розповсюджуються і опис яких доступний). Таке ПЗ може не мати відкритого коду, але при цьому розробникам іншого ПЗ легко забезпечити взаємодію з ним, тому що стандарти і протоколи обміну даними оприлюднені і доступні усім.

Також розрізняють безкоштовне ПЗ, яке також називають вільним (англ. – free) за принципом розповсюдження – вільно, без оплати.

ПЗ з відкритим кодом користується заслуженою популярністю в усьому світі, оскільки воно:

- гарантує незалежність користувача від виробників (вихідний код залишається у користувача навіть якщо виробник перестає підтримувати свій продукт);
- дає можливість кожному користувачу особисто переконатись у відсутності програмних закладок (стороннього щодо проголошеної функціональності або шкідливого коду);
- має більшу гнучкість: користувач може модифікувати вихідний код для адаптації ПЗ до своїх потреб як самостійно, так і доручивши цю роботу незалежним розробникам;
- у деяких випадках є дешевшим в експлуатації: початкова вартість може дорівнювати нулю, вартість підтримки зазвичай нижче, ніж вартість підтримки закритого ПЗ, широке відкрите для спілкування співтовариство користувачів дозволяє витрачати менше грошей на навчання і вирішення проблем.

У світі поширені найрізноманітніші програмні пакети, і одна з проблем тих, хто ними користується – сумісність форматів, в яких ці програмні пакети зберігають дані. Сучасне ПЗ як з відкритим, так і з закритим кодом зазвичай розробляють так, щоби воно було сумісним за форматами даних як з відкритими програмними пакетами, так і з розробками компаній, що продають свої продукти з закритим кодом.

Наприклад, пакет офісних програм Libre Office, функціонально дуже близький до інших популярних офісних пакетів, застосовує для зберігання

даних відкритий стандарт, але допускає збереження і імпортування даних у закритих форматах інших офісних пакетів.

Існують десятки тисяч програмних пакетів з відкритим кодом; однак значний вплив на розвиток і поширення сучасних технологій мають лише деякі з них. Серед них слід відзначити операційні системи з відкритим кодом. Більшість таких систем походять від операційної системи UNIX. Серед них у першу чергу слід назвати Linux, яка значною мірою визначає шлях розвитку усього відкритого ПЗ, також відкритими є кілька систем з гілки BSD (FreeBSD, OpenBSD), а протягом певного часу до відкритих належала і Solaris від Sun Microsystems (теперішні власники цієї системи не підтримують її відкритість, а й взагалі проголосили про згортання її підтримку).

Укладачі цього комп’ютерного практикуму закликають студентів вивчати і широко застосовувати відкриті програмні продукти – як операційні системи, так і прикладні програми.

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.

Завдання для самостійної підготовки

1. Ознайомитись з документацію, звернувши увагу на такі питання:
   - команди входу в систему, зміни пароля, одержання системної підказки, виводу календаря і зміни дати;
   - організацію і структуру файлової системи Linux, обмеження на імена файлів;
   - типи файлів, каталоги і посилання;
   - системні каталоги;
   - створення, видалення, копіювання і перегляд умісту файлів.

   Рекомендовані джерела: [2, 3]. Зокрема, [3] рекомендується скачати і роздрукувати (2 сторінки). Хоча ці джерела розроблені для Debian Linux, значні їх частини є вірними і для інших сучасних UNIX-подібних систем. Однак, для кожної конкретної системи рекомендується застосовувати документацію, розроблену конкретно для неї (за наявності)

2. Ознайомитись з такими командами UNIX-подібних систем (рекомендується застосовувати сторінки `man`):

   `man, passwd, date, cat, more, wc, who, ls, cd, cal, cp, mv, mkdir, rm, rmdir`

3. Відповідно до завдання підготувати послідовність команд для його виконання.

Довідковий матеріал

При встановленні ОС UNIX/Linux на персональному комп’ютері, вона налаштовується на роботу з графічною оболонкою. При цьому вхід користувача в систему здійснюється вже засобами графічної оболонки. Оскільки у цьому практикумі нам буде потрібне інтенсивне застосування терміналу з інтерфейсом командного рядка, розглянемо його особливості.

У системі підтримується певна кількість так званих віртуальних терміналів, між якими можна переключатись комбінаціями клавіш Alt+F#, де F# – одна з функціональних клавіш (при цьому вихід з багатовіконного

графічного інтерфейсу в один з текстових віртуальних терміналів може здійснюватись комбінацією клавіш Alt+Ctrl+F#). Користувач може працювати в системі, використовуючи одночасно кілька терміналів. Для здійснення входу в систему в текстовому терміналі, користувач повинен спочатку ввести свій ідентифікатор (*login*). Як правило, він вводиться у відповідь на запрошення системи такого вигляду:

```
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 – див. Роботу № 4 з цього практикуму). 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. У текстовому терміналі після цього користувач зазвичай опиняється в середовищі командної оболонки (англ. – *shell*). При цьому на екрані з’являється так зване запрошення командної оболонки (найчастіше – символ ‘$’ або ‘>’, також можна довільно змінити запрошення). 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 ідентифікатором *login* або *userid*. Користувач в процесі роботи може запустити іншу командну оболонку, просто набравши її ім’я.

Усі команди, 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.

Розглянемо деякі основні команди. Слід зазначити, що немає штатного засобу, який надавав би користувачеві перелік доступних йому команд, тому основні команди необхідно пам’ятати.

Команда `man` форматує і відображає на терміналі сторінки довідкової системи. Відповідно до номерів розділів даються посилання на ту чи іншу сторінку довідника. Якщо є необхідність, можна вказати, в якому розділі треба шукати потрібну сторінку (Приклад 2). Приклади використання (системні запрошення не показані):

```
man passwd
man 7 mdoc
```

Команда `passwd`, що викликає однойменну утиліту, дозволяє користувачеві змінювати свій пароль входу в систему. Спочатку необхідно підтвердити свою автентичність, набравши свій пароль, а далі можна ввести новий пароль (це здійснюється двічі для уникнення випадкових помилок).

Команда `who` дозволяє визначити, хто ще працює в поточний момент в системі. Команда `who am i` нагадає вам, який ваш *login*.

Існують зручні команди визначення поточної дати й часу (`date`), а також виводу на екран календаря на будь-який місяць будь-якого року (`cal`).

Для того, щоби переглянути вміст текстового файлу, можна скористатись командою `cat` <*ім’я файлу*>, або `more` <*ім’я файлу*> (остання команда призначена для виводу інформації на екран посторінково, вона надає можливість “перегортати сторінки” вперед і назад). Існує команда `wc` <*ім’я файлу*> (word count – підрахувати слова), яка дозволяє підрахувати кількість рядків (`wc -l`), слів (`wc -w`) і символів (`wc -c`) у файлі. Створити текстовий файл можна командою `touch` <*ім’я файлу*>.

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 до виконання

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**[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`.

[1] Якщо файл 1 або 2 не знайдено в каталозі `/bin`, шукайте його в каталогах `/usr/bin`, `/sbin` або `/usr/sbin`

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>`.

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

Завдання для самостійної підготовки

1. Вивчити (крім довідкового матеріалу, наведеного далі, рекомендується [2, пп. 1.2.3, 1.2.4]:
   - поняття “право доступу” і “метод доступу”;
   - атрибути доступу до файлів в UNIX;
   - перегляд інформації про права доступу;
   - зміна прав доступу.

2. Детально ознайомитись з довідкової системи `man` з такими командами UNIX (Linux):
   `ls –l, chmod, chown, umask`[2]**`, setfacl, getfacl.`**

3. Відповідно до завдання підготувати послідовність команд для його виконання.

Довідковий матеріал

Складовою кожної захищеної системи є система розмежування доступу. В системі UNIX контролюється доступ до файлів. Кожний файл має свого власника, а також відноситься до визначеної групи. Вся детальна інформація про файл, що включає тип файлу, ідентифікатори власника файлу та його групи, розмір файлу, час останньої модифікації файлу, інформацію щодо прав доступу до файлу, а також про його розміщення (номери блоків), міститься не в каталогах, як це зроблено в багатьох інших файлових системах, в тому числі FAT (MS-DOS, Windows), а в системній таблиці так званих індексних дескрипторів (*i-node*). Безпосереднє звернення до цієї таблиці заборонене. Каталоги містять лише ім’я файлу та індекс – посилання на відповідний запис в таблиці індексних дескрипторів. Саме завдяки такій організації кожний файл у файловій системі UNIX може мати кілька абсолютно рівноправних імен (жорстких посилань), що в загальному випадку містяться в різних каталогах (це було розглянуто в Роботі №1). Інформацію з таблиці індексних дескрипторів можна переглянути командою `ls –l`. При цьому для кожного файлу

[2] Для команди **`umask`** не існує окремої сторінки **`man`**, оскільки це – вбудована команда **`shell`**, тому її опис (хоча й не повною мірою детальний) міститься в сторінках **`man`** для кожного з **`shell`** (**`sh`**, **`bash`**, тощо).

інформація видається у вигляді одного рядка. Перший символ – 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 описують права доступу до файлу (див. далі). Далі надається інформація про кількість жорстких посилань на файл, власника файлу, групу, розмір файлу, дату останньої модифікації і останнє поле – ім’я файлу.

UNIX реалізує дискреційну[3] модель розмежування доступу, в якій для кожного файлу визначається, які права мають всі користувачі на доступ до файлу. Для цього з кожним файлом асоціюється спеціальна інформація, що містить ідентифікатор власника файлу, ідентифікатор групи файлу і права доступу до файлу. Права доступу поділяються на 3 частині: права власника, права групи і права всіх інших. У кожному класі користувачів виділено по 3 біти, що відповідають правам читання, запису й виконання (r, w, x), відповідно. Для каталогів право виконання трактується як право доступу до таблиці індексних дескрипторів на читання і запис, не маючи цього права неможливо зробити поточним цей каталог чи будь-який з його підкаталогів, неможливо ознайомитись і змінити права доступу до об’єктів цього каталогу, можна тільки переглядати його вміст, якщо є право читання. Навіть маючи право запису, без права виконання не можна змінити вміст каталогу. Навпаки, якщо є право на виконання, але не встановлено право на читання для каталогу, то неможливо переглянути вміст каталогу, але можна заходити в його підкаталоги чи звертатись до файлів, що містяться в ньому, якщо знати 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`.

[3] Більш докладну інформацію про моделі розмежування доступу ви отримаєте з курсів “Основи кібербезпеки” і “Захист інформації в інформаційно-комунікаційних системах”

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 – числовий спосіб задавання прав доступу – зручніше використовувати, коли треба встановлювати нові права незалежно від попередньо встановлених. При цьому використовується представлення у восьмеричній системі числення, яке легко зрозуміти з наведеної таблиці:

Восьмеричне представлення	**Двійкове представлення**	**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**

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 восьмерична 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

Маска задає максимальні права доступу для усіх користувачів, за винятком власника і групи. Призначення маски є найшвидшим способом обмежити фактичні (ефективні) права доступу.

Наприклад, маска **`r – –`** вказує, що користувачі і групи не можуть мати більших прав, ніж просто зчитування, навіть якщо їм призначені права доступу на зчитування й записування.

Розглянуті вище розширені права доступу можуть застосовуватись і до файлів, і до каталогів. Крім них, існують специфічні права доступа до каталогу,

а саме права доступу за умовчанням. Файли і підкаталоги, що будуть створювати у цьому каталозі, будуть автоматично отримувати такі ж росширені права доступу, які задані в розширених правах доступу за умовчанням до цього каталогу.

Розширені права доступу до каталогів за умовчанням

d[efault]:u[ser]::<perms>	права власника файлу за умовчанням
d[efault]:g[roup]::<perms>	права групи файлу за умовчанням
d[efault]:o[ther]:<perms>	права решти користувачів за умовчанням
d[efault]:m[ask]:<perms>	маска ACL за умовчанням
d[efault]:u[ser]:<uid>:<perms>	права доступу за умовчанням для вказаного користувача; в якості `uid` можна вказати і UID, і ім’я (login) користувача
d[efault]:g[roup]:<gid>:<perms>	права доступу за умовчанням для вказаної групи; в якості `gid` можна 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 заміны.

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 до виконання

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-но 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 записом[4]. 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.

[4] Ви можете одночасно користуватись різними обліковими записами, використовуючи для цього різні віртуальні консолі в текстовому режимі, або різні вікна терміналів в графічній багатовіконній системі. Віртуальні консолі в текстовому режимі переключаються комбінаціями клавіш Alt+F1 – Alt+F8.

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.

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

Завдання для самостійної підготовки

1. Вивчити (довідковий матеріал і, наприклад, [2, пп. 1.4, 1.5]):
   - командні оболонки, їх запуск, конфігураційні файли;
   - стандартні потоки і їх перенаправлення;
   - організацію конвеєрів;
   - організацію фільтрів і команди, використовувані як фільтри.

2. Ознайомитись з такими командами UNIX:
   `tee, find, cut, date, grep, sort`
   Звернути увагу на метасимволи **`*, ?, /, […], $`** і на правила інтерпретації їх при використанні одинарних та подвійних лапок **`'...'`** та **`"..."`**. Розібратись з використанням в командах операторів перенаправлення потоків і організації конвеєрів **`">", "<", "|"`** і використанням псевдопристрою **`/dev/null`**.

3. Відповідно до завдання підготувати послідовність команд для його виконання

Довідковий матеріал

У цій роботі ми розглянемо деякі можливості командних оболонок, які дозволяють користувачам гнучко формувати завдання для виконання операційною системою.

Кожна командна оболонка в процесі свого запуску робить налаштування системного оточення (виконує ініціалізацію системних змінних та змінних командної оболонки), для чого використовує визначені файли. Їх щонайменше два – глобальний і локальний (деякі оболонки можуть використовувати більшу кількість конфігураційних файлів). Глобальні конфігураційні файли для всіх оболонок знаходяться в каталозі `/etc`. Локальні конфігураційні файли для всіх оболонок знаходяться в домашньому каталозі користувача. Імена конфігураційних файлів, як правило, закінчуються на ‘`rc`’. Локальні файли роблять “прихованими”, щоби вони не заважали користувачу в його повсякденній роботі (приховані файли мають ім’я, що починається з символу ‘`.`’, команда `ls` без 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).

Конфігураційні файли є звичайними командними файлами (докладніше про командні файли див. Роботу №4).

Часто в якості параметру деякої команди нам треба вказати не один файл, а кілька файлів, назви яких мають певні спільні риси. В таких випадках використовують так звані маски пошуку. Спеціальний символ '**?**' в масці означає один будь-який символ, а спеціальний символ '*****' – будь-яку послідовність символів. Наприклад, команда `ls /bin/????` виведе на екран всі файли з каталогу `/bin`, імена яких складаються з чотирьох символів, а команда `ls /etc/d*` виведе на екран всі файли з каталогу `/etc`, імена яких починаються з літери **d**. Також можна задати список символів, наприклад, маска `[abc]???` задає ім'я з чотирьох літер, перша з яких – **a**, **b** чи **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.

Наприклад:

```
find / -name "*.c" –print
```

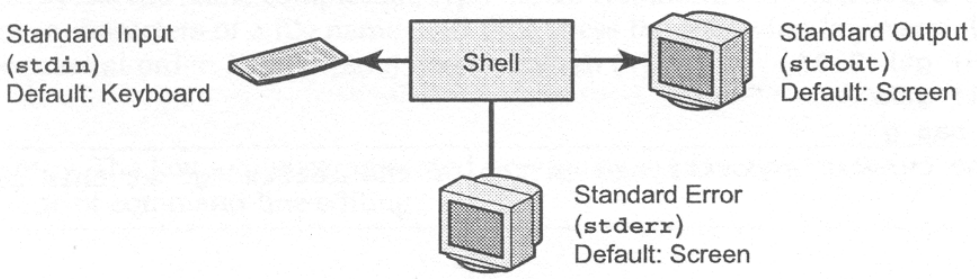
виведе на екран список усіх файлів, імена яких мають розширення .c, тобто вихідних кодів на мові програмування С. [5]

Ще одна можливість оболонок – перенаправлення потоків введення-виведення. Як правило, більшість команд (утиліт) приймає інформацію з клавіатури, або з файлу, якщо його вказано як параметр, і виводить результати на екран. Однак, фактично вони працюють із так званими стандартними потоками введення і виведення, які пов'язані з певними файлами. Файл в UNIX-подібних системах розглядається як потік байт. Оскільки пристрої в UNIX-подібних системах розглядаються як файли, а операції введення і виведення – як зчитування і записування у відповідні файли, це дозволяє легко переводити вхідний і вихідний потоки з файлів на пристрої чи навпаки.

Стандартний потік введення за умовчанням зчитується з клавіатури. Якщо в якості параметру вказано ім'я файлу, то замість стандартного введення відповідна утиліта буде організовувати вхідний потік з указаного файлу (але

[5] Якщо ваш Linux встановлено з вихідними кодами, то таких файлів буде безліч!

так діють не всі команди!) Вихідних потоків є два – стандартний потік виведення (за умовчанням у сучасних системах – на екран монітору), і потік повідомлень про помилки (за умовчанням – туди ж).



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 буде робити те ж саме, але з точки зору операційної системи і самої утиліти `cat` все буде відбуватись по-іншому. В попередньому випадку командна оболонка запускала утиліту `cat` і передавала їй параметр командного рядка `my_file`, а сама утиліта `cat` вже інтерпретувала цей параметр як ім’я вхідного файлу. В останньому випадку командна оболонка запускала утиліту `cat` без параметрів, але передавала їй вміст файлу `my_file` в якості вхідного потоку.

Команда `cat > my_file` перенаправляє вихідний потік. Оскільки вхідний потік не перенаправляється, введення буде очікуватись з клавіатури. Тому ця команда створить файл `my_file`, якщо він не існує, знищить вміст файлу `my_file`, якщо він існує, і буде записувати в цей файл усе, що буде введено з клавіатури, аж поки не поступить символ кінця файлу EOF (Ctrl+D). Існує також можливість дописати інформацію в кінець файлу, не знищуючи його вмісту. Така команда буде мати вигляд `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. Інформацію про це можна знайти в літературі.

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 у двох випадках. Перший – це запуск утиліт у фоновому режимі (це буде розглянуто в Роботі №5). Щоби робота фонових утиліт не заважала роботі користувача з терміналом, слід так перенаправити потоки введення-виведення, щоби вони працювали лише з файлами. Другий – це використання спеціальних команд-утиліт, які призначені саме для того, щоби прийняти певну інформацію з одного файлу, обробити її, а результат записати у другий файл. Такі утиліти називаються фільтрами. Утиліта `cat`, варіанти використання якої з перенаправленням потоків було розглянуто вище – це простіший фільтр. Він практично не обробляє інформацію, лише може зчіплювати кілька файлів в один. Інші корисні фільтри: `cut, grep, sort`.

Утиліта `cut` переглядає вхідний файл, і виділяє з кожного його рядка інформацію за ознаками розміщення в певних колонках або полях. Наприклад, рядки файлу `/etc/passwd` розділяються на поля за допомогою символу ‘`:`’. Перше поле – *login*, п’яте поле – інформація про користувача. Якщо ми хочемо надрукувати лише цю інформацію, ми можемо дати команду:

```
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` виводить лише ті рядки, в яких зустрічається заданий рядок пошуку. Утиліта `sort` виконує сортування вхідного потоку, наприклад, за абеткою. Докладніше про ці та інші фільтри вам слід дізнатися з довідкової системи `man`.

Існує можливість перенаправити вихідний потік однієї утиліти безпосередньо у вхідний потік іншої, без використання тимчасових файлів. Це так звані конвеєри (pipe). В UNIX-подібних системах усі утиліти, що поєднані в конвеєр, запускаються паралельно і обробляють інформацію по мірі її надходження. Конвеєр утворюється за допомогою символу ‘`|`’ таким чином:

```
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_file`[6]. 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 до виконання

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.

---

[6] Походження назви цієї команди добре пояснює її дію. Взагалі “tee” – це назва літери “T” в англійській абетці. Літера “T” якраз і використовується, щоби проілюструвати відгалуження, розщеплення потоку на основний і бічний (коли натякають на розщеплення потоку на два еквівалентні, використовують літеру “Y”).

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.

Робота №4. Розробка сценаріїв командної оболонки

Мета

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

Завдання для самостійної підготовки

1. Вивчити (наприклад, за [1, розд. 12.1], [4]):
   - організацію умовного виконання командного рядка, угруповання команд у командному рядку;
   - використання змінних shell;
   - організацію командних файлів: передачу параметрів, уведення значень, умовні розгалуження і цикли;
   - арифметичні обчислення в shell.

2. Розробити алгоритм рішення відповідно до завдання

3. Скласти програми рішення завдань

4. Підготувати тест для перевірки програм

Довідковий матеріал

У попередніх роботах ми вже познайомились з командними оболонками (*shell*). У цій роботі розглянемо прийоми професійної роботи з командними оболонками, а саме використання змінних оточення і створення командних файлів.

Змінні оточення

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

Командні файли

Командний файл, або сценарій (також часто кажуть “скріпт” від англійського *script* – сценарій) є текстовим файлом, який оформлено з дотриманням певних правил, і який містить команди, у найпростішому випадку повністю аналогічні тим командам, що вводяться з клавіатури. Командна оболонка здатна запускати такий файл на виконання і послідовно виконувати

команди, що містяться в ньому. Для користувача, що запустив цей сценарій, його виконання буде виглядати як виконання звичайної програми.

Зверніть увагу на розбіжності у різних програмних оболонках shell, які суттєві для програмування. Під час виконання роботи впевніться, в якій із програмних оболонок Ви працюєте (зазвичай, для Linux це `bash`), і яка буде запускатись для виконання Вашого командного файлу (це визначається першим рядком Вашого командного файлу). Уважно прочитайте правила використання операторів `if` і формування перевірки відповідної умови. Зверніть увагу на команду `test`.

Завдання до виконання

Виконати завдання згідно варіанту, вказаного викладачем.

Увага! Не намагайтесь автоматично обрати варіант за номером вашої заліковки, бо варіантів вистачить не усім. Натомість, зверніться до викладача.

Варіант 1

Написати сценарій для оболонки bash згідно таких вимог:

- Сценарій приймає 3 параметри командного рядка. Перший параметр — ім’я каталогу, в якому (і в підкаталогах якого рекурсивно) треба здійснювати пошук. Другий параметр — шаблон пошуку. За відсутності двох параметрів сценарій повинен виводити рядок, що описує коректний формат виклику сценарію. Третій параметр — необов’язковий — це ім’я файлу архіву, що буде створений.
- Сценарій шукає у заданому каталозі і його підкаталогах усі файли, імена яких відповідають шаблону пошуку, і формує їх список, що містить ім’я файлу, повний шлях до каталогу, тип файлу (див. команду `file`), розмір файлу у байтах. Список виводиться у стандартний потік.
- Усі знайдені файли згідно списку додаються до архіву tar, до якого також додається створений список (оберіть розумне ім’я файлу, що дає зрозуміти, що це — саме список файлів). Якщо ім’я файлу не було задано параметром виклику сценарію, на цьому етапі сценарій повинен запросити ввести його.
- Сценарій повинен коректно відпрацьовувати помилки, такі як некоректні імена файлів і шаблон пошуку, відсутність заданого каталогу, помилки доступу (зокрема, відсутність права доступу до певних файлів), відсутність жодних файлів, що відповідають заданому шаблону. При цьому сценарій повинен видавати діагностику помилок.

Варіант 2

Написати сценарій для оболонки bash згідно таких вимог:

- Сценарій приймає 2 параметри командного рядка. Перший параметр — ім’я каталогу, в якому (і в підкаталогах якого рекурсивно) треба здійснювати пошук. Другий параметр — необов’язковий — це шаблон пошуку. За відсутності першого параметру сценарій повинен виводити рядок, що описує коректний формат виклику сценарію.
- Сценарій шукає в заданому каталозі і його підкаталогах усі файли, імена яких відповідають шаблону пошуку, якщо такий був заданий. Якщо шаблон заданий не був, то відповідними вважають усі файли, крім ‘.’ і ‘..’ . Сценарій формує список усіх знайдених файлів, що містить ім’я файлу, повний шлях до каталогу, тип файлу (див. команду `file`), розмір файлу у байтах, і геш MD5. Список виводиться у стандартний потік.
- Сценарій повинен коректно відпрацьовувати помилки, такі як некоректні імена файлів і шаблон пошуку, відсутність заданого каталогу, помилки доступу (зокрема, відсутність права доступу до певних файлів), відсутність жодних файлів, що відповідають заданому шаблону. При цьому сценарій повинен видавати діагностику помилок.

Варіант 3

Написати сценарій для оболонки bash згідно таких вимог:

- Сценарій приймає 3 параметри командного рядка. Перший параметр — ім’я каталогу, в якому (і в підкаталогах якого рекурсивно) треба здійснювати пошук. Другий параметр — шаблон пошуку. За відсутності двох параметрів сценарій повинен виводити рядок, що описує коректний формат виклику сценарію. Третій параметр — необов’язковий — це ім’я файлу, що буде створений.
- Сценарій шукає у заданому каталозі і його підкаталогах усі файли, які містять послідовність байт, що відповідає шаблону пошуку, і формує їх список. Список повинен містити ім’я файлу, повний шлях до каталогу, тип файлу (див. команду `file`), розмір файлу у байтах, і рядок, що відповідає шаблону.
- Якщо третій параметр виклику сценарію (ім’я файлу) був заданий, то список має бути виведений у цей файл. Якщо параметр заданий не був — на цьому етапі сценарій повинен запросити ввести його. В обох випадках якщо такий файл для виводу вже існує, сценарій повинен видати запит, що робити далі — переписати вміст файлу, дописати нові дані у кінець файлу, або скасувати операцію.
- Сценарій повинен коректно відпрацьовувати помилки, такі як некоректні імена файлів і шаблон пошуку, відсутність заданого каталогу, помилки доступу (зокрема, відсутність права доступу до певних файлів),

відсутність жодних файлів, що відповідають заданому шаблону. При цьому сценарій повинен видавати діагностику помилок.

Варіант 4

Написати сценарій для оболонки bash створення резервних копій згідно таких вимог:

- Сценарій приймає довільну кількість параметрів командного рядка. Якщо жодного з параметрів не задано, повинна створюватись резервна копія (`tar.gz` або `tar.bz2`) домашнього каталогу користувача. Якщо параметри задані, вони розглядаються як імена каталогів, які треба додати до резервної копії (при цьому домашній каталог у цілому за умовчанням не додається).
- Якщо імена каталогів задані параметрами командного рядка, слід додати перевірку їх наявності. Сценарій повинен повідомляти про відсутність певних каталогів, але продовжувати роботу з архівації інших каталогів із списку.
- Архів повинен створюватись у підкаталозі `archives` домашнього каталогу. Сценарій повинен перевірити наявність цього каталогу, і якщо його немає, створити його. Ім’я створюваного архіву повинно містити поточну дату у форматі YYYYMMDDHHMM.
- Сценарій повинен коректно відпрацьовувати помилки, такі як некоректні імена каталогів і помилки доступу (зокрема, відсутність права доступу до певних файлів). При цьому сценарій повинен видавати діагностику помилок.

  Налаштувати періодичний запуск сценарію за допомогою `cron`.

Варіант 5

Написати сценарій для оболонки bash згідно таких вимог:

- Сценарій перевіряє активність заданого користувача. Якщо користувач працює на комп’ютері, сценарій видає йому привітання. Якщо користувач не працює (немає логіну), сценарій повинен автоматично згенерувати і відправити повідомлення електронної пошти на адресу начальника (можете обрати адресу на свій смак).
- Повідомлення повинно містити формальне вибачення і причину відсутності на роботі. Причина має обиратися рандомно з файлу, що містить щонайменше десять таких причин (наприклад, викликали до школи через погану поведінку сина, чекаєте на сантехніка бо трубу прорвало, тощо). Жодних натяків на втому чи головний біль! *Примітка: почуття гумору вітається, але завданням роботи все ж є написання сценарію, що працює, а не вправи у красному письменстві!*

Налаштувати періодичний запуск сценарію за допомогою `cron` по понеділках о 9-й годині ранку.

Варіант 6

Написати сценарій для оболонки bash згідно таких вимог:

Частина 1

- Сценарій визначає, хто є користувачем системи, та виводить результат на екран;
- За допомогою змінних оточення сценарій визначає домашній каталог користувача;
- Сценарій знаходить усі файли, які належать користувачеві у його домашньому каталозі;
- Сценарій архівує ці файли з іменем `back_up_<your_name>_<data>`;
- Якщо архів з даним іменем вже існує, то сценарій повинен виводити запит на його перезаписування;
- Сценарій встановлює права на отриманий архів тільки для зчитування.

Частина 2 (окрема від першої!)

- Сценарій розархівовує файли з архіву зі збереженням структури каталогів
- Якщо файл існує у каталозі, і той, що розпаковується з архіву, відрізняється у порівнянні з ним, то сценарій повинен виводити запит на його перезаписування.

Варіант ЩЕ ОДИН (за попередньою згодою викладача)

Написати сценарій для оболонки bash згідно таких вимог:

- Сценарій повинен робити щось розумне і корисне.
- Сценарій повинен (у деякій комбінації) застосовувати:
  - Роботу зі змінними оточення;
  - Роботу з параметрами командного рядка;
  - Введення з клавіатури, виведення на екран;
  - Перевірку наявності файлів або каталогів, або деяких їх атрибутів;
  - Запуск інших програм і перевіку коду повернення.
- Сценарій повинен коректно відпрацьовувати помилки (видавати діагностику).

Робота №5. Процеси в ОС UNIX і керування ними

Мета

Оволодіння практичними навичками роботи з процесами — створення і знищення, керування процесами та їх аналіз

Завдання для самостійної підготовки

1. Вивчити (теорія — [1, гл. 2], [5, розд. 3, 4]; моніторинг і керування — [2, розд. 9.3]):
   - поняття процесу та його характеристики;
   - виведення на екран списку процесів і його аналіз;
   - фонові й активні процеси;
   - пріоритет процесів і його зміна;
   - відправлення сигналів процесам, організація перехоплення сигналів;
   - виконання завдань у системі в заданий час і з заданою періодичністю.

2. Ознайомитись з такими командами UNIX:
   `ps, ptree, pgrep, kill, pkill, fg, bg, jobs, crontab, at`

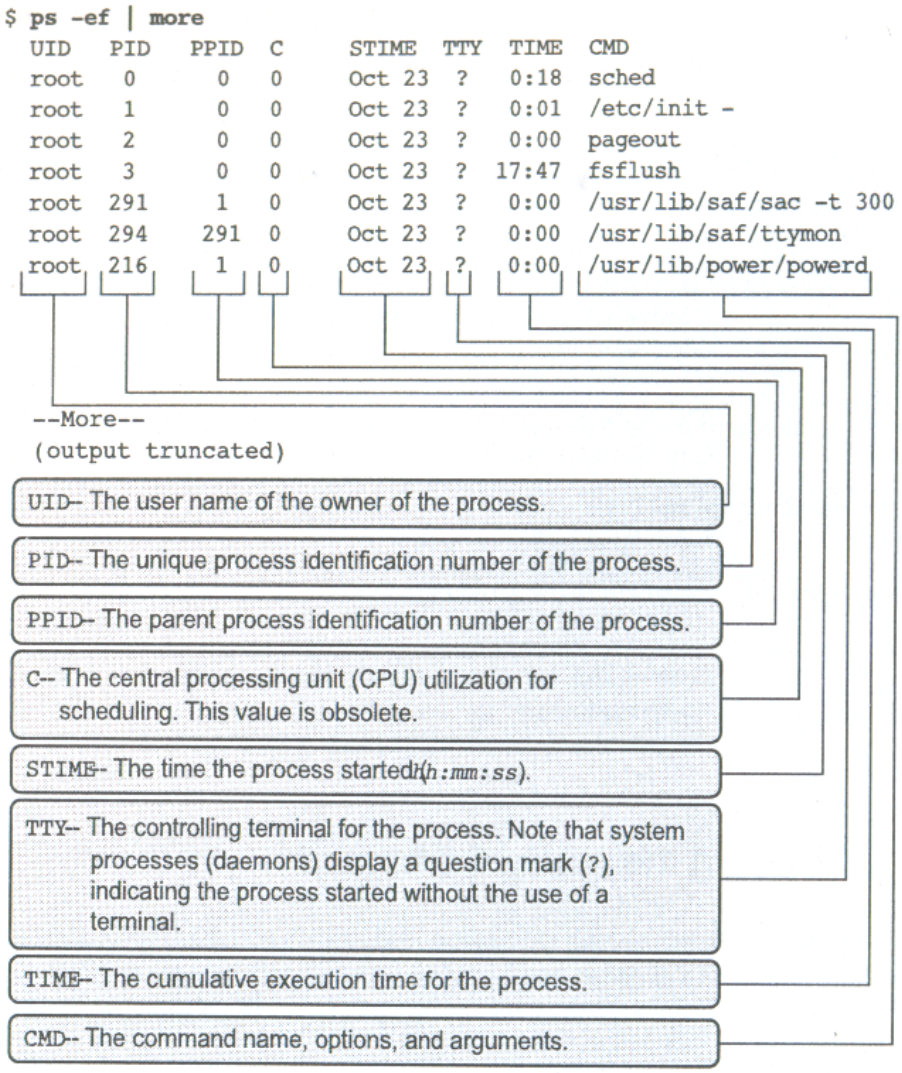
   Зверніть увагу на використання параметру командного рядка "&"

3. Скласти послідовність команд для виконання необхідного варіанту завдання

Довідковий матеріал

UNIX і UNIX-подібні системи – це багатозадачні операційні системи з розділенням часу. Це означає, що в системі одночасно виконується багато процесів. Кожний процес асоційований з певним користувачем, від імені якого цей процес діє. Для того, щоби переглянути список процесів, існує команда `ps`. Ця команда має багато ключів-модифікаторів, які визначають, яку саме інформацію про процеси повинна виводити команда. Слід зазначити, що в різних UNIX-подібних системах значення цих ключів може суттєво відрізнятись. Типові ключі: `-a` виводить інформацію про всі процеси, а не лише про процеси даного користувача, `-l` та `-x` виводять розширену інформацію про процес.

Кожний процес в системі має свій унікальний ідентифікатор – `PID`. За цим ідентифікатором можна звертатись до процесу. Крім того, кожний процес виникає не сам по собі – він має так званий батьківський процес, що характеризується ідентифікатором `PPID` (*parent process ID*). Таким чином утворюється ієрархія процесів, що бере початок від початкового процесу `init`.



The ps -ef Command Output Description

Створення нового процесу у системі UNIX виконується у два етапи. Спочатку системний виклик `fork()` викликає “розщеплення” поточного процесу на два тотожні (різниця буде лише у тому, що процес-нащадок має інші `PID` і `PPID`). Далі виконується системний виклик `exec()`, який замінює контекст поточного процесу іншим контекстом. Розглянемо типовий приклад, коли один процес має запустити на виконання інший – командна оболонка `sh` виконує команду `ls`, для чого утворює новий процес, в якому виконується програма `/usr/bin/ls`. Для цього програміст, який пише код `sh`, повинен передбачити такі кроки:

1. Якщо під час синтаксичного аналізу командного рядка виявлено, що необхідно запустити на виконання деяку команду, то виконується виклик `fork()`.
2. Наступна за `fork()` інструкція перевіряє значення, що повернув виклик. Якщо виклик `fork()` був успішний, то в результаті його виконання утворюється новий процес, тотожній батьківському, і

операційна система буде виконувати їх обидва (в режимі розділення часу). Для процесу-нащадка значення, що повертає `fork()`, дорівнює 0, для батьківського процесу повертається значення PID нащадка.

3. Якщо це процес-нащадок, то виконується виклик `exec()`, якому в якості параметра передається ім'я файлу програми, яку необхідно запустити на виконання (в нашому прикладі – `/usr/bin/ls`). Якщо виклик був успішний, то на цьому виконання коду `sh` припиняється, і на його місце завантажується код `ls`.
4. Якщо це батьківський процес, то його дії залежать від того, які параметри були у командному рядку. Зокрема, якщо `ls` було запущено у пріоритетному режимі[7], то виконання `sh` призупиняється до завершення процесу-нащадка, якщо ж `ls` було запущено у фоновому режимі, то `sh` видає необхідні повідомлення про запуск фонового процесу і продовжує свою роботу, тобто знову приймає і редагує командний рядок.

Процеси можуть взаємодіяти між собою за допомогою так званих сигналів. Існує обмежена кількість сигналів, які мають свої числові ідентифікатори і мнемонічні позначення. Сигнали діють як переривання, тобто вони призупиняють процес, до якого вони направлені, і викликають відповідний обробник сигналу. Деякі із сигналів мають чітко визначене значення і обробляються системним обробником. Інші можуть перехоплюватись процесом, тобто процес може встановлювати для цих сигналів свій обробник. Звичайно, існують певні узгодження щодо призначення певних сигналів, і програмістам слід дотримуватись їх при розробці своїх обробників.

Користувач також може відправити процесу сигнал, для цього існує команда `kill`. Формат команди:

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

`<signal>` – це мнемонічне або числове позначення сигналу (наприклад, `STOP`, `TERM`, `CONT`, `9`), а `<PID>` – ідентифікатор процесу, якому посилають сигнал. Якщо не вказати параметр `<signal>`, то буде відправлено сигнал завершення процесу `TERM` (`15`). Цей сигнал може перехоплюватись процесом, але існує сигнал `KILL` (`9`), який не перехоплюється і безумовно знищує процес (якщо у користувача є достатньо для цього прав). Таким чином можна зупинити будь-який свій процес, якщо над ним втрачене керування (або процес "завис", що у UNIX-подібних системах трапляється нечасто, або користувач не знає, яку команду процес може сприйняти). Для цього слід зайти з іншої консолі

[7] Про пріоритетний і фоновий режими див. далі

(віртуального або фізичного терміналу) і дати команду `kill -9 <PID>`, де `<PID>` можна дізнатись за допомогою попередньої команди `ps`.

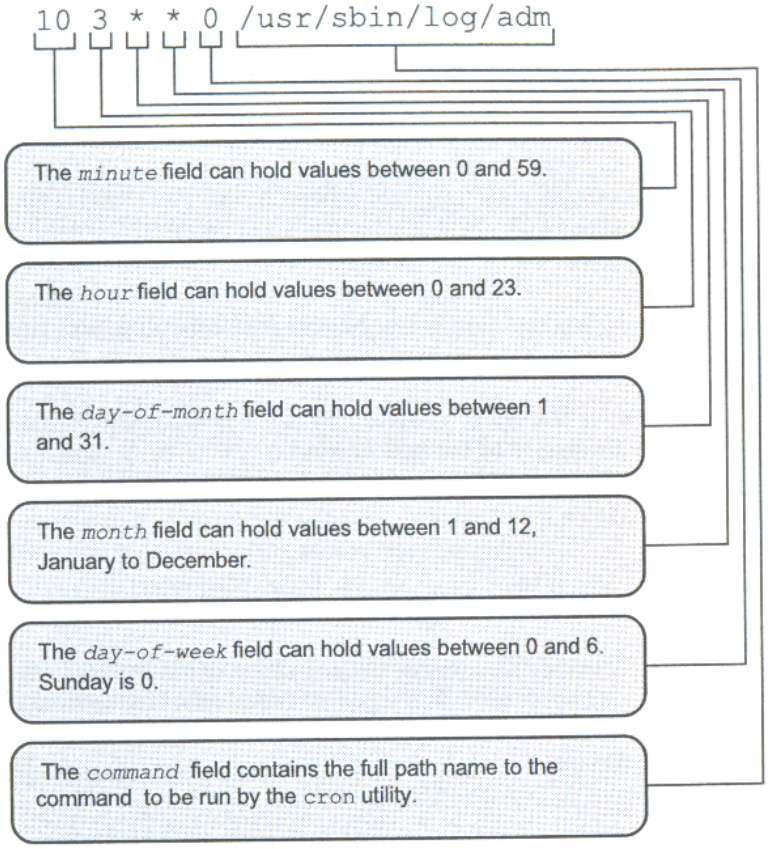
Команда `ps` без аргументів дає список лише процесів даного користувача і (зазвичай) лише пов’язаних з конкретним терміналом. При роботі в графічній оболонці, а також при роботі на персональному комп’ютері, де підтримується певна кількість так званих віртуальних терміналів, користувач може працювати в системі, використовуючи одночасно кілька терміналів (наприклад, на персональних комп’ютерах між ними можна переключатись комбінаціями клавіш Alt+F#, де F# – одна з функціональних клавіш). Однак в часи створення системи UNIX такі можливості не передбачались. Передбачалось, що користувач має доступ лише до одного терміналу. Тому було розроблено систему завдань і фонового виконання процесів, щоби користувач міг одночасно виконувати різні задачі.

Коли користувач дає команду з консолі, в системі запускається процес, або кілька процесів. Якщо в командному рядку утворюється конвеєр (наприклад, `ls –l | wc –l`), то всі процеси (у нашому прикладі `ls` і `wc`) запускаються одночасно. Разом вони складають так зване завдання. Завдання пов’язано з терміналом. Поки воно не завершиться, користувач не має можливості вводити наступну команду. Це так званий пріоритетний (*foreground*) режим виконання завдання. Щоби під час виконання завдання мати можливість запускати інше завдання, перше з них слід запустити в так званому фоновому (*background*) режимі. Для того, щоби запустити завдання в фоновому режимі, слід завершити рядок команди символом “`&`” (після пробілу). При цьому стандартний потік введення за умовчанням назначається порожньому файлу `/dev/null`. Слід врахувати, що завдання у фоновому режимі може намагатись здійснювати виведення на екран, заважаючи при цьому виводу пріоритетного процесу (спробуйте працювати, запустивши у фоновому режимі команду `ping`). Тому слід подбати, щоби фонові завдання здійснювали виведення у файли (див. Роботу №3). Завдання, що було запущено у пріоритетному режимі, можна перевести у фоновий. Для цього необхідно спочатку призупинити виконання завдання (комбінація клавіш `Ctrl+Z`). Далі можна поновити виконання завдання у пріоритетному режимі (команда `fg`) або у фоновому режимі (команда `bg`). Завдання мають свої номери. Переглянути їх можна за допомогою команди `jobs`.

Важливою можливістю є запуск певних завдань за розкладом. Це реалізується за допомогою програми-демона `crond`, який переглядає свої файли `crontab` (окремі для кожного користувача) і запускає завдання згідно розкладу від імені того користувача, в файлі якого це завдання задано. Для зміни розкладу завдань треба відредагувати файл `crontab` і після цього перезапустити `crond`, що може зробити лише root та ті користувачі, яким це

право надано. Останнім рекомендується для цього користуватись командою `crontab`, яка після редагування автоматично перезапускає `crond`.

Користувачу дозволено виконувати команду `crontab` тільки за умови, що його ім'я зустрічається в файлі `cron.allow` і не зустрічається в файлі `cron.deny` (в різних UNIX-подібних системах ці файли можуть знаходитись в різних каталогах, наприклад `/usr/lib/cron/` або `/etc/cron.d/`). Якщо обидва файли відсутні, то тільки root може користуватись командою `crontab`. Якщо `cron.allow` не існує, `cron.deny` існує, але не містить імен, то використовувати команду `crontab` дозволено усім. Файли `cron.allow` і `cron.deny` повинні містити по одному імені в рядку.



First Five Fields in a crontab File

Існує спеціальна команда `at`, яка призначена для одноразового виконання заданої команди в будь-який заданий час. Правила дозволу і заборони користування командою `at` аналогічні `crontab` (файли `at.allow` і `at.deny`). Завдання, що задані командою `at`, виконуються тим же демоном `crond`.

Завдання до виконання

1. Перегляньте список процесів користувача (вас).
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 (щоденне) видалення в домашньому каталозі усіх файлів з розширенням `*.bak` і `*.tmp`.

[8] Увага! Ця команда коректно працює в оболонках `sh` і `bash`, і не працює коректно у оболонках `csh` і `tcsh` (це пов’язано з правилами перенаправлення потоку помилок, див. Роботу №3).

Робота №6. Створення процесів у Linux із застосуванням системних викликів fork() і exec()

Мета

Оволодіння практичними навичками застосування системних викликів у програмах, дослідження механізму створення процесів у UNIX-подібних системах.

Завдання для самостійної підготовки

1. Ознайомитись з документацією системних викликів `fork()` і `exec()`:
   - man pages;
   - книги з числа рекомендованих [1, 5];
   - [6] (електронний ресурс на сайті www.ibm.com).

2. Перевірити, чи встановлений у вашій системі Linux компілятор C/C++ (g++). Якщо ні, встановіть за допомогою менеджера пакетів.

Довідковий матеріал

Тут наведено лише мінімальну інформацію, достатню хіба що для того, щоби зрозуміти, про що йде мова. Решту інформації необхідно здобути з джерел, названих вище.

Створення процесу у UNIX-подібних системах відбувається у два кроки, для чого існують два різних сімейства викликів.

Перший виклик — це `fork()` (фактично — “розгалуження”). Виклик створює тотожню копію існуючого процесу, копіюючи також адресний простір. При цьому “старому” (так званому батьківському) процесу в результаті виклику повертають `PID` “нового” (так званого дочірнього процесу, або процесу-нащадка), а нащадку повертають значення 0. Якщо виклик завершився з помилкою, повертають -1. Дещо детальніше робота цього виклику була описана вище у Роботі №5. Слід пам’ятати, що є безліч деталей — що у нащадка буде те ж саме, що й у батьківського процесу, а що буде іншим, тому слід уважно перевіряти це за документацією. Після виклику `fork()` будуть паралельно працювати обидва процеси, виконуючи один і той самий код, але з різними значеннями, що їх повернув виклик.

Існує виклик `vfork()`, який відрізняється тим, що адресний простір не копіюється (тобто, зокрема, обидва процеси мають доступ до одних і тих самих змінних, а не до різних їх копій), процес-нащадок виконується, а батьківський процес призупиняється до завершення процесу-нащадка.

Другий виклик — `exec()` — представлений сімейством функцій, які розрізняються наборами параметрів, які вони приймають, і тим, як вони передають їх новому процесу. Спільна ідея — замінити поточний процес на інший. Точніше, у тому самому процесі (що зберігає свій `PID`) починає виконуватись новий програмний код, що його завантажують з виконуваного файлу, у загальному випадку за допомогою параметрів виклику змінюючи його оточення і передаючи йому параметри командного рядка.

Згідно стандарту POSIX, функції `exec()` оголошуються у файлі `unistd.h` (у деяких реалізаціях імена функцій можуть починатись з символу підкреслювання, наприклад `_execl()`).

```
int execl(char const *path, char const *arg0, ...);
int execle(char const *path, char const *arg0, ..., char const *envp[]);
int execlp(char const *file, char const *arg0, ...);
int execv(char const *path, char const *argv[]);
int execve(char const *path, char const *argv[], char const *envp[]);
int execvp(char const *file, char const *argv[]);
```

Фактично, назви мають сенс: exec – “виконати”, а далі кожна літера має певне значення:

- e – образу нового процесу у явному вигляді передається масив вказівників на змінні оточення (тобто, функції `execve()` і `execle()` запускають нову програму в новому оточенні, яке задається аргументами виклику, а інші функції запускають нову програму у тому самому отченні, що існувало до виклику);
- l – функції передається список (*list*) аргументів командного рядка (список має закінчуватись нульовим аргументом);
- p – використовує змінну оточення PATH, щоби знайти виконуваний файл названий в аргументі `file` (з одного боку, не треба задавати повний або відносний шлях до файлу, але з іншого боку якщо поточний каталог явно не вказаний у PATH, то файл з поточного каталогу знайдений і запущений не буде);
- v – аргументи командного рядка передаються функції як масив (*vector*) вказівників.

Якщо передається аргумент `path`, то наступний аргумент `arg0` – це і є їм’я файлу, який треба запустити на виконання. У більшості випадків в якості `path` і в якості `arg0` задають один і той самий рядок — ім’я виконуваного файлу зі шляхом до нього.

Завдання до виконання

1. Для початку можна взяти демонстраційну програму, запропоновану *Greg Ippolito* [7][9].

```
01  #include <iostream>
02  #include <string>
03
04  // Required by for routine
05  #include <sys/types.h>
06  #include <unistd.h>
07  #include <stdlib.h>   // Declaration for exit()
08
09  using namespace std;
10
11  int globalVariable = 2;
12
13  Main()
14  {
15      string sIdentifier;
16      int   iStackVariable = 20;
17
18      pid_t pID = fork();
19      if (pID == 0)                // child
20      {
21          // Code only executed by child process
22
23          sIdentifier = "Child Process: ";
24          globalVariable++;
25          iStackVariable++;
26        }
27        else if (pID < 0)            // failed to fork
28        {
29              cerr << "Failed to fork" << endl;
30              exit(1);
31              // Throw exception
32        }
33        else                         // parent
34        {
35          // Code only executed by parent process
36
37          sIdentifier = "Parent Process:";
38        }
39
40        // Code executed by both parent and child.
41
42        cout << sIdentifier;
43        cout << " Global variable: " << globalVariable;
44        cout << " Stack variable: "  << iStackVariable << endl;
```

[9] У цій програмі закладені деякі змінні для використання у наступних програмах. Якщо бажаєте і впевнені у своїх діях, можете програму скоротити або взагалі написати власну із аналогічною функціональністю.

```
45 }
```

2. Скомпілюйте програму (вважаємо, текст збережено у файлі `myforktest.cpp`)

   `g++ -o myforktest myforktest.cpp`

   Увага! Пам’ятайте, що не можна називати власні програми просто `test`! `test` – це вбудована команда shell (з якою ви вже зустрічалися в Роботі №4).

3. Запустіть програму `myforktest`. У якій послідовності виконуються батьківський процес і процес-нащадок? Чи завжди цей порядок дотримується?

4. Додайте затримку у виконання одного або обох з цих процесів (функція `sleep()`, аргумент — затримка у секундах). Чи змінились результати виконання?

5. Додайте цикл, який забезпечить кількаразове повторення дій після виклику `fork()`. Які результати показують процеси (значення глобальної змінної і змінної, що визначена у стеку)? Поясніть.

6. Спробуйте у первинній програмі (без циклу) замість виклику `fork()` здійснити виклик `vfork()`. У чому різниця роботи цих двох викликів? Чи виникає помилка (якщо так, то яка)? У чому причина? Як “змусити” працювати виклик `vfork()`? Які результати тепер показують процеси (значення глобальної змінної і змінної, що визначена у стеку)? Поясніть.

7. Тепер додайте виклик `exec()` у код процесу-нащадка. Для початку використайте простішу функцію `execl()`. Варіант виклику на прикладі утиліти ls:

   `execl("/bin/ls", "/bin/ls", "-a", "-l", (char *) 0);`

   У наведеному прикладі передаються аргументи командного рядка.

8. Проведіть експерименти з викликом різних програм, у тому числі `ps`, `bash`, а також з викликами `execl()` у батьківському процесі. Як запустити фоновий процес-нащадок? Як процес-нащадок дізнається власний `PID`? `PID` батьківського процесу?

9. Усі отримані результати і відповіді на запитання, які були задані вище, оформіть у вигляді протоколу.

Робота №7. Основи роботи з потоками у Linux з використанням бібліотеки pthread

Мета

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

Завдання для самостійної підготовки

1. Ознайомитись з документацією і прикладами використання бібіліотеки `pthread`:
   - man pages;
   - книги з числа рекомендованих, зокрема [1, розд. 2.2], [5, п. 3.8.4];
   - для першого знайомства — POSIX Threads, матеріал з Вікіпедії [8] (російською мовою, бо на поточний момент стаття українською мовою є вкрай неповною);
   - дуже непогана стаття [9];
   - простий приклад застосування pthread [10];
   - велика книга [11];
   - авторитетне першоджерело [12].

2. Якщо не робили попередню роботу, то перевірити, чи встановлений у вашій системі Linux компілятор C/C++ (g++). Якщо ні, встановіть за допомогою менеджера пакетів.

Довідковий матеріал

Тут наведено лише мінімальну інформацію, достатню хіба що для того, щоби зрозуміти, про що йде мова. Решту інформації необхідно здобути з джерел, названих вище.

Спочатку створення потоків у Linux здійснювали за допомогою виклику `clone()`, який багато у чому подібний до `fork()`, але дозволяє призначити, які ресурси є спільними між батьківським процесом і нащадком. В результаті насправді з’являвся не потік, а новий процес.

Для підтримки потоків у ядрі Linux спочатку була розроблена бібліотека LinuxThreads, а пізніше — NPTL. Слід зазначити, що на відміну від Solaris і Windows, де процеси фактично є контейнерами для потоків, потоки у Linux все одно залишаються “полегшеними процесами”, але програмно вони виглядають так (або майже так) як повинні виглядати потоки POSIX.

Програмний інтерфейс роботи з потоками — це бібліотека pthread. У цій роботі ви застосуєте цю бібліотеку для створення і завершення потоків і дослідите, як це працює. Ця бібліотека реалізована для різних операційних систем, у тому числі і для Windows. Тому навички, які будуть отримані вами у ході виконання цієї роботи, можна буде застосовувати для програмування під багатьма різними операційними системами.

Функції, які вам необхідно застосовувати (а отже, необхідно прочитати їх документацію):

`pthread_create()`

`pthread_exit()` (зверніть увагу на особливості застосування цієї функції у початковому потоці, тобто у функції main())

`pthread_join()`

`pthread_cancel()`

А також, можливо, і деякі інші.

Завдання до виконання

1. **Створення потоку.** Напишіть програму, що створює потік. Застосуйте атрибути за умовчанням. Батьківський і дочірній потоки мають роздрукувати по десять рядків тексту.
2. **Очікування потоку.** Модифікуйте програму п. 1 так, щоби батьківський потік здійснював роздрукування після завершення дочірнього (функція `pthread_join()`).
3. **Параметри потоку.** 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. **Примусове завершення потоку.** Дочірній потік має роздруковувати текст на екран. Через дві секунди після створення дочірнього потоку, батіківський потік має перервати його (функція `pthread_cancel()`).
5. **Обробка завершення потоку.** Модифікуйте програму п. 4 так, щоби дочірній потік перед завершенням роздруковував повідомлення про це (`pthread_cleanup_push()`).

Усі отримані результати оформіть у вигляді протоколу. Під час здачі роботи продемонструйте викладачеві роботу розроблених вами програм і їх код.

Робота №8. Засоби синхронізації потоків

Мета

Оволодіння практичними навичками розроблення багатопотокових програм з підтримкою засобів синхронізації

Завдання для самостійної підготовки

1. Ознайомитись з документацією і прикладами використання засобів синхронізації такими як семафори, м'ютекси, умовні змінні:
   - man pages;
   - книги з числа рекомендованих, зокрема [1, розд. 2.3], [5, розд. 5];
   - [11, с. 103-126];
   - [12, розд. 7, 8];
   - корисна стаття [13] (у цій роботі нас цікавлять лише семафори і м'ютекси, але ми до цієї статті ще повернемось);
   - великі книги з програмування в Linux, що орієнтовані на кодерів, містять приклади коду, перекладені російською мовою, тому комусь можуть бути цікавими, зрозумілими, і взагалі дуже корисними [14, 15, 16] (для цієї роботи див. розділи про семафори, м'ютекси, умовні змінні, тощо).
2. Якщо не робили попередні роботи, то перевірити, чи встановлений у вашій системі Linux компілятор C/C++ (g++). Якщо ні, встановіть за допомогою менеджера пакетів.

Довідковий матеріал

Оскільки потоки мають доступ до спільної пам'яті, проблема синхронізації стоїть дуже гостро. Можливі змагання (race condition) і, як наслідок, спотворення результатів (уявіть два потоки, що без синхронізаціїї друкують рядок за рядком деякий вірш), втрати даних (два потоки записують в одну й ту ж саму ділянку пам'яті), взаємні блокування. З іншого боку, оскільки потоки працюють хз спільною пам'яттю, принципово організувати синхронізацію дуже просто.

Тут наведено лише мінімальну інформацію, достатню хіба що для того, щоби зрозуміти, про що йде мова. Решту інформації необхідно здобути з джерел, названих вище.

Для синхронізації можна застосовувати прості об'єкти, такі як семафори, м'ютекси, умовні змінні. Існують також засоби більш високого рівня — монітори. Семафори можуть застосовуватись як для блокування критичних

секцій (бінарні семафори), так і для очікування події. М’ютекси спеціалізовані для блокування критичних секцій, Умовні змінні спеціалізовані для очікування подій.

Зверніть увагу на принципово різні правила застосування семафорів і умовних змінних, щоби уникнути взаємних блокувань.

Завдання до виконання

1. Розминка. Стандартна задача виробник-споживач.

Задача була розглянута на лекції. Також детально розглянута в рекомендованих книжках [1, 5]. Розробіть програму, що демонструє рішення цієї задачі за допомогою семафорів. Для цього напишіть:

- функції виробника і споживача (наприклад, як на лекції, або як у Шеховцові, але так, щоби працювало);
- функції створення і споживання об’єктів (рекомендується “створювати” рядки тексту шляхом зчитування їх з файлу, хоча можливі й інші варіанти за вибором викладача або за вашою фантазією, наприклад розрахунки геш-функцій sha2 з рядків рандомних символів, а “споживати” їх шляхом роздрукування на екрані з додатковою інформацією такою як ідентифікатор потоку і мітка часу, причому і там, і там для моделювання складного характеру реального життя виробників і споживачів можна додавати рандомні затримки);
- функцію main(), що створює потоки-виробники і потоки-споживачі, при цьому треба передбачити введення з клавіатури або як параметри командного рядка кількості записів у буфері, кількості виробників і кількості споживачів для досліджень їх роботи;
- обов’язково передбачити коректне завершення усього цього господарства.

Продемонструвати викладачеві як воно працює (не менше двох виробників і двох споживачів) і код, що ви написали.

2. Продовження розминки. Теж саме, але не на семафорах, а на м’ютексі і умовних змінних

Модифікуйте програму п. 1 так, щоби використовувати м’ютекс і умовну змінну.

3. Продовження розминки для тих, хто шукає пригод. Взаємне блокування

Модифікуйте програму п. 1 так, щоби викликати взаємне блокування. Для цього поміняйте місцями семафори. Переконайтесь у факті взаємного блокування і отримайте задоволення.

4. Індивідуальне завдання

А тепер напишіть програму згідно індивідуального завдання (варіант вказує викладач).

Усі отримані результати оформіть у вигляді протоколу. Під час здачі роботи продемонструйте викладачеві роботу розроблених вами програм і їх код.

Варіант 1 а. Обчислення числа π

Напишіть багатопотокову програму, що обчислює число π за допомогою ряда Ляйбниця:

$$1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\frac{1}{11}+\ldots=\frac{\pi}{4}.$$

Кількість потоків програми і кількість ітерацій мають визначатися параметрами командного рядка. Для передавання часткових сум ряду, що підраховані потоками, можна застосовувати `pthread_exit()` і `pthread_join()`.

Слід врахувати, що на 32-розрядних платформах `sizeof(double)>sizeof(void *)`, тому часткову суму ряду не можна перетворювати у вказівник, а треба виділяти для неї власну память.

Варіант 1 б. Обчислення π аж поки не набридне

Завдання аналогічне Варіанту 1 а. Відмінність полягає в тому, що програма не повинна сама по собі завершуватись. Завершення має відбуватись після натискання Ctrl+C (тобто, після одержання сигналу SIGINT), при чому програма повинна не перериватись, а саме якнайшвидше завершуватись, збираючи часткові суми ряду, після чого виводити отримане наближення числа.

Для розв’язвння поставленго завдання рекмендується встановите обробник SIGINT. Обробник має встановлювати глобальну змінну — прапор. Потоки, що виконують обчислення, мають переглядати значення прапору через певну (доволі велику) кількість ітерацій (проведіть дослідження, як ця кількість впливає на швидкість обчислень). Якщо прапор встановлений, потік має виходити за допомогою pthread_exit.

Як можна мінімізувати помилку, зумовлену тим, що різні потоки до завершення встигли пройти різну кількість ітерацій?

Варіант 2 а. Філософи, що обідають

Розробіть симулятор класичної задачі про філософів, що обідають. П’ять філософів сидять за круглим столом і їдять спагетті. Спагетті їдять за

допомогою двох виделок. Всього виделок п’ять. Кожні двоє філософів, що сидять поруч, користуються однією спільною виделкою.

Кожний філософ незалежно від інших може знаходитись в одному з двох станів — їсть або думає. Філософ думає деякий час (передбачте можливість рандомізувати цей час у певному інтервалі, а також можливість задавати цей інтервал для дослідження), потім він намагається взяти виделки.

У цьому варіанті завдання усі філософи спочатку намагаються взяти ліву виделку, а потім праву. Якщо йому вдалося захопити обидві виделки, він починає їсти. Їсть він також деякий час (як і думає — але співвідношення часів варто змінювати для дослідження), після чого він звільняє обидві виделки і знову починає думати. І так далі, поки у нього не закінчаться спагетті. Якщо одну з виделок взяти неможливо, філософ чекає, поки вона звільниться. Якщо йому протягом певного часу (помітно більшого, ніж час їжи і час роздумів) так і не вдається ухопити дві виделки, він падає в обморок (потік завершується).

Природно моделювати філософів за допомогою потоків, а виделки — за допомогою м’ютексів. Програма повинна синхронно (тобто, у тому ж порядку, як воно і відбувалося) друкувати усі події з мітками часу. Наприклад:

10:31:11.253 Філософ 1 узяв виделку 5 (ліву). Стан виделок XOOOX

10:31:11.255 Філософ 2 узяв виделку 2 (праву). Стан виделок XXOOX

10:31:11.541 Філософ 1 не зміг узяти виделку 1 (праву). Стан виделок XXOOX

10:31:11.883 Філософ 4 узяв виделку 3 (ліву). Стан виделок XXXOX

10:31:11.253 Філософ 2 почав їсти.

10:31:12.117 Філософ 3 не зміг узяти виделку 2 (ліву). Стан виделок XXXOX

10:31:12.733 Філософ 4 узяв виделку 4 (праву). Стан виделок XXXXX

...

10:31:14.125 Філософ 2 закінчив їсти.

...

10:31:11.255 Філософ 2 поклав виделку 1 (ліву). Стан виделок OXXXX

...

Чи спостерігалося взаємне блокування? За яких умов? Як його уникнути?

<u>Варіант 2 б. Філософи, що обідають 2</u>

Все тотожнє варіанту 2 а, за винятком протоколу захоплення виделок. Коли філософ може взяти одну виделку, але не може взяти другу, він має

покласти виделку на стіл і повторити спробу через деякий час. Реалізуйте активне очікування.

Чи часто філософи падають в обморок?

Варіант 2 в. Філософи, що обідають 3

Все тотожнє варіанту 2 б, за винятком типу очікування. Якщо філософ не може взяти виделку (першу або другу), він має покласти на стіл першу виделку, якщо він її вже захопив, і заснути. Реалізувати можна за допомогою додаткових м'ютексів і умовних змінних. Відповідно, коли філософ звільнює виделки, він має сигналізувати про це.

Чи часто філософи падають в обморок? А якщо час його сну додавати до часу "голодування"?

Варіант 2 г. Філософи, що обідають 4

Все тотожнє варіанту 2 а, за винятком протоколу захоплення виделок. Усе рандомно — кожний філософ у рандомні моменти бере (або намагається узяти) і кладе виделки, причому майже незалежно одну від одної (все ж задайте тенденцію захопити дві виделки, бо інакше зголодніє). Коли у нього дві виделки, він їсть.

Чи часто філософи падають в обморок?

Варіант 3. Пором у Парку культури та відпочинку

У Парку культури та відпочинку десантники та морські пехотинці одночасно відмічають свята відповідних родів військ. Повертатись з парку вони мають на поромі, що вміщає 2N вояків. Пором можна відправляти лише тоді, коли він повністю заповнений і на ньому або представники лише одного роду військ, або рівна кількість десантників і морських піхотинців. Якщо сили будуть нерівними, неминуча жорстока бійка. Переправа займає рандомний час. На березі вояки поводять себе культурно, бо там чергує озброєний контингент "Беркуту", що контролює посадку на пором. Вояки підходять з парку поодинці у випадкові моменти часу і стають у чергу.

Змоделювати роботу такої переправи, створюючи для "обслуговування" кожного вояка окремий потік. Програма має синхронно (тобто, у тому ж порядку, як воно і відбувалося) писати у журнал події приходу вояків, посадки їх на пором, відправлення і повернення порому, з мітками часу і звітом, скільки яких вояків у черзі і скільки їх на поромі.

Робота №9. Засоби синхронізації і взаємодії процесів

Мета

Оволодіння практичними навичками використання засобів міжпроцесової взаємодії в Linux

Завдання для самостійної підготовки

1. Ознайомитись з документацією і прикладами використання засобів міжпроцесової взаємодії такими як семафори, м'ютекси, умовні змінні:
   - man pages;
   - книги з числа рекомендованих, зокрема [1, розд. 2.3], [5, розд. 6];
   - стаття [13];
   - книги з програмування в Linux [14, 15, 16].
2. Якщо не робили попередні роботи, то перевірити, чи встановлений у вашій системі Linux компілятор C/C++ (g++). Якщо ні, встановіть за допомогою менеджера пакетів.

Довідковий матеріал

Тут наведено лише мінімальну інформацію, достатню хіба що для того, щоби зрозуміти, про що йде мова. Решту інформації необхідно здобути з джерел, названих вище (та інших).

Міжпроцесна взаємодія (*Inter-process communication, IPC*) — це набір методів і засобів для обміну даними між процесами. Можна говорити також про взаємодію потоків, і деякі засоби (такі, наприклад, як семафор) можуть застосовуватись для синхронізації потоків одного процесу. Але особливість засобів, що будуть розглянуті у цій роботі, полягає у тому, що операційна система надає їх потокам різних процесів, які мають різні адресні простори і взагалі можуть бути запущені на різних комп'ютерах, з'єднаних мережею. Прикладами механізмів IPC є сигнали, сокети, семафори, файли, повідомлення, канали, поділювана пам'ять. Для реалізації цих механізмів передбачені відповідні системні виклики.

Зверніть увагу на такі системні виклики, які вам ймовірно доведеться застосувати (деякі з них ви мабуть вже застосовували у попередніх роботах):

- Створення, завершення процесу, отримання інформації про процес: `fork()`, `exit()`, `getpid()`, `getppid()`.
- Синхронізація процесів: `signal()`, `kill()`, `sleep()`, `alarm()`, `wait()`, `pause()`.

- Створення інформаційного каналу і робота з ним: `pipe()`, `read()`, `write()`.

Для роботи з семафорами підтримують три системних виклики:

- `semget()` для створення і одержання доступу до набору семафорів;
- `semop()` для маніпулювання значеннями семафорів (це той системний виклик, що дозволяє процесам синхронізуватися на основі використання семафорів);
- `semctl()` для виконання різних операцій керуванngя набором семафорів.

Прототипи цих системних викликів описані у файлах:

```
#include <sys/ipc.h>
#include <sys/sem.h>
```

Для обміну повідомленнями між процесами існує механізм черг, що підтримується такими системними викликами:

- `msgget()` для утворення нової черги повідомлень або одержання дескриптора черги, що існує;
- `msgsnd()` для постановки повідомлення у задану чергу повідомлень;
- `msgrcv()` для вибору повідомлення з черги повідомлень;
- `msgctl()` для виконання низки дій керування.

Прототипи цих системних викликів описані у файлах:

```
#include <sys/ipc.h>
#include <sys/msg.h>
```

Для роботи з поділюванною пам’яттю застосовуються системні виклики:

- `shmget()` створює новий сегмент поділюваної пам’яті або знаходить сегмент, що існує, з тим самим ключем;
- `shmat()` підключає сегмент з указаним дескриптором до віртуального адресного простору процесу;
- `shmdt()` відключає від віртуального адресного простору сегмент зі вказаною віртуальною адресою початку, що був раніше до нього підключеним;
- `shmctl()` служить для керування різними параметрами, що пов’язані із сегментом, що існує.

Прототипи цих системних викликів описані у файлах:

```
#include <sys/ipc.h>
```

```
#include <sys/shm.h>
```

Завдання до виконання

1. Ознайомитись із завданням до лабораторної роботи (згідно варіанту, вказаного викладачем).

Варіант 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.

Варіант 2

Два дочірніх процеси виконують деякі цикли робіт, передаючи після закінчення чергового циклу через один і той же сегмент поділюваної пам'яті батьківському процесові чергові чотири рядки деякого віршу, при цьому перший процес передає непарні чотиривірші, а другий — парні. Цикли робіт процесів не збалансовані за часом. Батьківський процес компонує з переданих фрагментів закінчений вірш і виводить його після закінчення роботи обох дочірніх процесів. Розв’язати задачу з використанням апарату семафорів.

Варіант 3

Чотири дочірніх процеси виконують деякі цикли робіт, передаючи після закінчення чергового циклу через один і той же сегмент поділюваної пам'яті батьківському процесові черговий рядок деякого віршу, при цьому перший процес передає 1-й, 5-й, 9-й і т.д. рядки, другий — 2-й, 6-й, 10-й і т.д. рядки, третій — 3-й, 7-й, 11-й і т.д. рядки, четвертий — 4-й, 8-й, 12-й і т.д. рядки. Цикли робіт процесів не збалансовані за часом. Батьківський процес компонує з переданих фрагментів закінчений вірш і виводить його після закінчення роботи всіх дочірніх процесів. Розв’язати задачу з використанням апарату семафорів.

Варіант 4

Програма моделює роботу примітивної СКБД, що зберігає єдину таблицю в оперативній пам'яті. Виконуючи деякі цикли робіт, *K* породжених процесів за допомогою черги повідомлень передають батьківському процесові номер рядка, який потрібно вилучити з таблиці. Батьківський процес виконує зазначену операцію і повертає вміст вилученого рядка.

Варіант 5

Програма моделює роботу примітивної СКБД, що зберігає єдину таблицю в оперативній пам'яті. Виконуючи деякі цикли робіт, *K* породжених процесів за допомогою черги повідомлень передають батьківському процесові номер рядка і вміст, на який потрібно замінити дані, що у ньому зберігаються. Батьківський процес виконує зазначену операцію і повертає попередній вміст рядка, що був змінений.

Варіант 6

Програма моделює роботу примітивної СКБД, що зберігає єдину таблицю в оперативній пам'яті. Виконуючи деякі цикли робіт, K породжених процесів за допомогою черги повідомлень передають батьківському процесові вміст рядка, який потрібно додати до таблиці. Батьківський процес перевіряє, чи немає у таблиці такого рядка, і, якщо немає, додає рядок і повертає кількість рядків у таблиці.

Варіант 7

Чотири дочірніх процеси виконують деякі цикли робіт, передаючи після закінчення чергового циклу через чергу повідомлень батьківському процесові черговий рядок деякого віршу, при цьому перший процес передає 1-й, 5-й, 9-й і т.д. рядки, другий — 2-й, 6-й, 10-й і т.д. рядки, третій — 3-й, 7-й, 11-й і т.д. рядки, четвертий — 4-й, 8-й, 12-й і т.д. рядки. Цикли робіт процесів не збалансовані за часом. Батьківський процес компонує з переданих фрагментів закінчений вірш і виводить його після закінчення роботи всіх дочірніх процесів. Розв’язати задачу з використанням апарату семафорів.

Варіант 8

Батьківський процес поміщає в сегмент поділюваної пам'яті імена програм з попередніх лабораторних робіт, які можуть бути запущені. Виконуючи деякі цикли робіт, породжені процеси випадковим чином обирають імена програм з таблиці сегмента поділюваної пам'яті, запускають ці програми, і продовжують свою роботу. За допомогою апарату семафорів має бути забезпечено, щоб не були одночасно запущені дві програми від одного процесу. В процесі роботи через чергу повідомлень батьківський процес інформується, які програми і від імені кого запущені.

Варіант 9

Батьківський процес поміщає в сегмент поділюваної пам'яті імена програм з попередніх лабораторних робіт, які можуть бути запущені. Виконуючи деякі цикли робіт, породжені процеси випадковим чином обирають імена програм з таблиці сегмента поділюваної пам'яті, запускають ці програми, і продовжують свою роботу. За допомогою апарату семафорів має бути

забезпечено, щоб не були одночасно запущені дві однакові програми. В процесі роботи через чергу повідомлень батьківський процес інформується, які програми і від імені кого запущені.

Варіант 10

Програма моделює роботу монітора обробки повідомлень. Породжені процеси, що володіють різними пріоритетами і виконують деякі цикли робіт, за допомогою черги повідомлень передають батьківському процесові імена програм з попередніх лабораторних робіт, які їм повинні бути запущені. Батьківський процес, обробляючи повідомлення відповідно до їх пріоритетів, стежить, щоб одночасно було запущено не більше трьох програм.

2. Ознайомитись з основними поняттями механізму IPC (див. вище, див. літературу).
3. Розібратись з набором системних викликів, що забезпечують розв’язання завдання.
4. Налагодити і зневадити складену програму, використовуючи інструментарій ОС Linux.
5. Усі отримані результати оформити у вигляді протоколу.
6. Захистити роботу, продемонструвавши викладачеві роботу розробленої вами програми та її код, а також відповівши на контрольні запитання.

Контрольні запитання

1. У чому різниця між двійковим і звичайним семафорами?
2. Чим відрізняються операції `P()` і `V()` від звичайних операцій збільшення і зменшення на одиницю?
3. Для чого служить набір програмних засобів IPC?
4. Для чого введені масові операції над семафорами в ОС Linux?
5. Яке призначення механізму черги повідомлень?
6. Які операції над семафорами існують в ОС Linux?
7. Яке призначення системного виклику `msgget()`?
8. Які умови мають бути виконані для успішної постановки повідомлення в чергу?
9. Як отримати інформацію про власника і права доступу черги повідомлень?
10. Яке призначення системного виклику `shmget()`?

Робота №10. Інтерфейс файлової системи в ОС Linux

Мета

Ознайомитися з реалізацією файлових систем в Linux і основними структурами даних, що використовуються віртуальною файловою системою (VFS). Дослідити механізм доступу до файлів через інтерфейс віртуальної файлової системи в Linux.

Завдання для самостійної підготовки

1. Ознайомитись з документацією VFS для ОС Linux. Звернути увагу на архітектуру системи, структури даних, що використовує ця система.
2. Ознайомитись з правилами і прикладами використання функцій (системних викликів) для роботи з файловою системою (перелік функцій див. нижче у розділі Довідковий матеріал):
   - man pages;
   - книги з числа рекомендованих, зокрема [1, розд. 4.3.7], [5, розд. 13];
   - стаття [17];
   - інші джерела.

Довідковий матеріал

Тут наведено лише мінімальну інформацію, достатню хіба що для того, щоби зрозуміти, про що йде мова. Решту інформації необхідно здобути з джерел, названих вище.

Назвемо системні функції, що забезпечують звернення до існуючих файлів, такі як `open()`, `read()`, `write()`, `lseek()` і `close()`, потім функції створення нових файлів, а саме, `creat()` і `mknod()`, і, нарешті, функції для роботи з індексними дескрипторами (*i-node*) або для пересування по файловій системі: `chdir()`, `chroot()`, `chown()`, `stat()` і `fstat()`.

Більш складні системні функції: `pipe()` і `dup()` — мають важливе значення для реалізації каналів в shell; `mount()` і `umount()` розширюють видиме для користувача дерево файлових систем; `link()` і `unlink()` змінюють ієрархічну структуру файлової системи.

Функції для роботи з файловою системою

Функції для роботи з файловою системою і їх зв'язок з іншими алгоритмами можна звести у такі таблиці:

Повертають дескриптори файлу	Використовують алгоритм namei	Призначають індексні дескриптори	Працюють з атрибутами файлу	Здійснюють введення-виведення з файлу	Працюють зі структурою файлових систем	Здійснюють керування деревами
open creat dup pipe close	open stat creat link chroot unlink chdir mknod chown mount chmod umount	creat mknod link unlink	chown chmod stat	read write lseek	mount umount	chdir chown

Алгоритми роботи з файловою системою на нижньому рівні		
namei	ialloc ifree	alloc free bmap
iget iput		

Алгоритми роботи з буферами				
getblk	brelse	bread	breada	bwrite

Системні функції класифікують на кілька категорій, хоча деякі з функцій присутні більш, ніж в одній категорії:

1. Системні функції, які повертають дескриптори файлів для використання іншими системними функціями.
2. Системні функції, що використовують алгоритм `namei` для аналізу імені шляху пошуку.
3. Системні функції, які призначають і звільняють індексний дескриптор з використанням алгоритмів `ialloc` та `ifree`.
4. Системні функції, які встановлюють або змінюють атрибути файлу.
5. Системні функції, що дозволяють процесу проводити введення-виведення з використанням алгоритмів `alloc`, `free` і алгоритмів виділення буфера.
6. Системні функції, які змінюють структуру файлової системи.
7. Системні функції, що дозволяють процесу змінювати власне уявлення про структуру дерева файлової системи.

Виклик системної функції `open()` (відкрити файл) — це перший крок, який повинен зробити процес, щоб звернутися до даних у файлі.

Для читання з файлу використовується функція `read()`, а для запису в файл `write()`. Звичайне використання системних функцій `read()` і `write()` забезпечує послідовний доступ до файлу, однак процеси можуть використовувати виклик системної функції `lseek()` для зазначення місця у

файлі, де буде проводитися введення-виведення, а також здійснення довільного доступу до файлу.

Процес закриває відкритий файл, коли процесу більше не потрібно звертатися до нього, за допомогою функції `close()`.

Системна функція `open()` дає процесу доступ до файлу, що вже існує. Системна функція `creat()` створює в системі новий файл. Системна функція `mknod()` створює в системі спеціальні файли, в число яких входять іменовані канали, файли пристроїв і каталоги.

Зміна власника або режиму (прав) доступу до файлу є операцією, яка здійснюється над індексним дескриптором, а не над файлом. Це робиться операціями `chown()` і `chmod()`, відповідно.

Системні функції `stat()` і `fstat()` дозволяють процесам запитувати інформацію про статус файлу: тип файлу, власника файлу, права доступу, розмір файлу, кількість зв'язків, номер індексного дскриптора і час доступу до файлу.

Системна функція `dup()` копіює дескриптор файлу в перше вільне місце в таблиці користувальницьких дескрипторів файлу, повертаючи новий дескриптор користувачеві. Вона діє для всіх типів файлів.

Системна функція `mount()` пов'язує файлову систему із зазначеного розділу на диску з існуючою ієрархією файлових систем, а функція `umount()` вилучає файлову систему з ієрархії. Функція `mount()`, таким чином, дає можливість звертатися до даних в дисковому розділі як до файлової системи, а не як до послідовності дискових блоків.

Системна функція `link()` пов'язує файл з новим ім'ям в структурі каталогів файлової системи, створюючи для існуючого індексного дескриптора новий запис у каталозі.

Системна функція `unlink()` видаляє з каталогу точку входу для файлу.

Наявність віртуальної файлової системи дає ядру можливість підтримувати одночасно множину файлових систем, таких як мережні файлові системи або файлові системи з інших операційних систем. Процеси користуються для звернення до файлів звичайними функціями системи, а ядро встановлює відповідність між загальним набором файлових операцій і операціями, специфічними для кожного типу файлової системи.

Структури даних ядра (системні таблиці)

При здійсненні операцій введення-виведення в файл, специфікований призначеним для користувача дескриптором файлу `fd`, ОС Linux ставить у відповідність використовуваному системному виклику послідовність

програмних запитів до апаратури комп'ютера за допомогою цілого ряду пов'язаних наборів даних, структура яких підтримується самою ОС Linux, її файлової системою і системою керування введенням-виведенням.

Основною із згаданих структур можна вважати таблицю дескрипторів файлів. Таблиця дескрипторів файлів являє собою структуру даних, що зберігається в оперативній пам'яті комп'ютера, елементами якої є копії дескрипторів файлів `inode`, по одній на кожен файл ОС Linux, до якого була здійснена спроба доступу. При виконанні операції відкриття файлу в ОС Linux спочатку по повному імені файлу визначається елемент каталогу, де в полі імені міститься ім'я файлу, для якого проводиться операція відкриття файлу. У знайденому елементі каталогу з поля посилання витягується порядковий номер дескриптора файлу (`inode`). Потім дескриптор файлу з відповідним номером копіюється в оперативну пам'ять, в її область, яка називається таблицею дескрипторів файлів (якщо він до цього там був відсутній).

З таблицею дескрипторів файлів тісно пов'язана інша структура даних, яка називається таблицею відкритих файлів. Кожен елемент `file` таблиці відкритих файлів містить інформацію про режим відкриття файлу, специфікований при відкритті файлу, а також інформацію про становище покажчика читання-запису. При кожному відкритті файлу в таблиці відкритих файлів з'являється новий елемент `file`.

Один і той самий файл ОС Linux може бути відкритий декількома не пов'язаними один з одним процесами, при цьому йому буде відповідати один елемент таблиці дескрипторів файлів `inode` і стільки елементів таблиці відкритих файлів `file`, скільки разів цей файл був відкритий. Однак з цього правила є один виняток, який стосується випадку, коли файл, відкритий процесом, потім відкривається процесом-нащадком, породженим за допомогою системного виклику `fork()`. При виникненні такої ситуації операції відкриття файлу, здійсненої процесом-нащадком, буде поставлений у відповідність той з існуючих елементів таблиці відкритих файлів (в тому числі положення покажчика читання-запису), який свого часу був поставлений у відповідність операції відкриття цього файлу, здійсненої процесом-предком.

Третій набір даних називається масивом файлових дескрипторів процесу (`fd`). Кожному процесу в ОС Linux відразу після породження ставиться у відповідність масив файлових дескрипторів процесу. Якщо, в свою чергу, зазначений процес породжує новий процес, наприклад, за допомогою системного виклику `fork()`, то процесу-нащадку ставиться у відповідність масив файлових дескрипторів процесу, який в перший момент функціонування процесу-нащадка є копією масиву файлових дескрипторів процесу-предка.

В результаті кожен елемент масиву файлових дескрипторів процесу (`fd`) містить покажчик місця розташування відповідного елементу таблиці

відкритих файлів (`file`), який у свою чергу містить посилання на елемент таблиці дескрипторів файлу (`inode`). В реалізації VFS у Linux це посилання насправді здійснюється через об’єкти елементів каталогу (`dentry`), впроваджені для кешування часто використовуваних елементів каталогів.

Завдання до виконання

1. Спочатку виконати завдання для самостійної підготовки, тобто опанувати теорію.
2. Ознайомитись із завданням до лабораторної роботи (згідно варіанту, наданого викладачем).

Варіант 1

Процес відкриває *N* файлів, що реально існують на диску або є новостворюваними. Розробити програму, яка демонструвала б динаміку формування таблиці дескрипторів файлів і зміни інформації в її елементах (при зміні інформації в файлах). Наприклад, сценарій програми може бути таким:

- відкриття першого призначеного для користувача файлу;
- відкриття другого призначеного для користувача файлу;
- відкриття третього призначеного для користувача файлу;
- зміна розміру третього файлу до нульової довжини;
- копіювання другого файлу в третій файл.

Після кожного з етапів друкується таблиця дескрипторів файлів для всіх відкритих файлів.

Варіант 2

Процес створив новий файл і перепризначив на нього стандартний потік виведення. Розробити програму, яка демонструвала б динаміку створення таблиць, пов'язаних з цією подією (таблиця відкритих файлів, масив файлових дескрипторів процесу). Наприклад, сценарій програми може бути таким:

- неявне відкриття стандартного файлу введення;
- неявне відкриття стандартного файлу виведення;
- неявне відкриття стандартного файлу виведення помилок;
- відкриття призначеного для користувача файлу;
- закриття стандартного файлу введення (моделювання `close (0)`);
- отримання копії дескриптора призначеного для користувача файлу (моделювання `dup(fd)`, де `fd` — дескриптор призначеного для користувача файлу);

- закриття призначеного для користувача файлу (моделювання `close(fd)`, де `fd` — дескриптор призначеного для користувача файлу).

Після кожного з етапів друкуються таблиця описателей файлів, таблиця файлів, таблиця відкритих файлів процесів.

Варіант 3

Нехай два процеси здійснюють доступ до одного і того ж файлу, але один з них читає файл, а інший пише в нього. Настає момент, коли обидва процеси звертаються до одного й того ж блоку диска. Нехай деяка гіпотетична ОС використовує ту ж механіку керування введенням-виведенням, що і ОС UNIX, але не дозволяє, як в ситуації, описаній вище, звертатися до одного блоку файлу.

Розробити програму, яка демонструє "заморожування" переміщення покажчика читання-запису одного з процесів до тих пір, поки покажчик другого процесу знаходиться в цьому блоці. Показати динаміку створення всіх таблиць, пов'язаних з файлами і процесами, і зміну їх вмісту.

Після кожного з етапів друкуються таблиці відкритих файлів і масиви дескрипторів файлів процесу обома процесами.

Варіант 4

Нехай *N* процесів здійснюють доступ до одного і того ж файлу на диску (але з різними режимами доступу). Розробити програму, яка демонструвала б динаміку формування таблиці відкритих файлів і зміни її елементів (при переміщенні покажчиків читання-запису, наприклад). Наприклад, сценарій програми може бути таким:

- відкриття файлу процесом 0 для читання;
- відкриття файлу процесом 1 для запису;
- відкриття файлу процесом 2 для додавання;
- читання зазначеного числа байт файлу процесом 0;
- запис зазначеного числа байт в файл процесом 1;
- додавання вказаного числа байт в файл процесом 2.

Після кожного з етапів друкуються таблиці файлів всіх процесів.

Варіант 5

Розробити програму, яка демонструвала б роботу ОС Linux при відкритті файлу процесом і читанні-записи в нього. При цьому досить показати тільки динаміку створення таблиць, пов'язаних з цією подією (таблиця дескрипторів файлу, таблиця відкритих файлів, масив файлових дескрипторів процесу). Наприклад, сценарій програми може бути таким:

- неявне відкриття стандартного файлу введення;
- неявне відкриття стандартного файлу виведення;
- неявне відкриття стандартного файлу виведення помилок;
- відкриття першого призначеного для користувача файлу;
- відкриття другого призначеного для користувача файлу;
- записування 20 байт в перший файл;
- зчитування 15 байт з другого файлу;
- записування 45 байт в перший файл.

Після кожного з етапів друкуються таблиця дескрипторів файлів, таблиця відкритих файлів, таблиця відкритих файлів процесів.

Варіант 6

Розробить програму, яка демонструвала б роботу ОС Linux при відкритті файлу процесом. При цьому досить показати тільки динаміку створення таблиць, пов'язаних з цією подією (таблиця дескрипторів файлів, таблиця відкритих файлів, масив файлових дескрипторів процесу). Наприклад, сценарій програми може бути таким:

- неявне відкриття стандартного файлу введення;
- неявне відкриття стандартного файлу виведення;
- неявне відкриття стандартного файлу виведення помилок;
- відкриття першого призначеного для користувача файлу;
- відкриття другого призначеного для користувача файлу;
- відкриття третього призначеного для користувача файлу.

Після кожного з етапів друкуються таблиця дескрипторів файлів, таблиця відкритих файлів, таблиця відкритих файлів процесів.

Варіант 7

Нехай кожен з N процесів здійснює доступ до P_i файлів ($i = 1..N$). Далі нехай $M<N$ процесів породили процеси-нащадки (за допомогою системного виклику `fork ()`) і серед цих нащадків $K<M$ процесів додатково відкрили ще S_j файлів ($j = 1..K$). Розробити програму, яка демонструвала б динаміку формування масивів файлових дескрипторів процесів. Наприклад, сценарій програми може бути таким:

- процес 0 відкриває два файли (загальне число відкритих файлів, включаючи стандартні файли, дорівнює п'яти);

- процес 1 відкриває два файли (загальне число відкритих файлів, включаючи стандартні файли, дорівнює п'яти);
- процес 2 відкриває два файли (загальне число відкритих файлів, включаючи стандартні файли, дорівнює п'яти);
- процес 0 породжує процес 3, який успадковує таблицю відкритих файлів процесу 0;
- процес 1 породжує процес 4, який успадковує таблицю відкритих файлів процесу 1;
- процес 4 додатково відкрив ще два файли.

Після кожного з етапів друкуються таблиці відкритих файлів процесів, що беруть участь в даному етапі.

Варіант 8

Процес створив новий файл і перепризначив на нього стандартний потік введення. Розробити програму, яка демонструвала б динаміку створення таблиць, пов'язаних з цією подією (таблиця дескрипторів файлів, таблиця відкритих файлів, масив файлових дескрипторів процесу). Наприклад, сценарій програми може бути таким:

- неявне відкриття стандартного файлу введення;
- неявне відкриття стандартного файлу виведення;
- неявне відкриття стандартного файлу виведення помилок;
- зчитування зі стандартного файлу введення 5 байт;
- відкриття призначеного для користувача файлу;
- закриття стандартного файлу введення (моделювання `close(0)`);
- отримання копії дескриптора призначеного для користувача файлу (моделювання `dup(fd)`, де `fd` — дескриптор призначеного для користувача файлу);
- закриття призначеного для користувача файлу (моделювання `close(fd)`, де `fd` — дескриптор призначеного для користувача файлу);
- читання зі "стандартного" файлу введення 10 байт.

Після кожного з етапів друкуються таблиця дескрипторів файлів, таблиця відкритих файлів, масив файлових дескрипторів процесу.

Варіант 9

Нехай процес, який відкрив N файлів, перед породженням процесу-нащадка за допомогою системного виклику `fork()` закриває $K<N$ файлів. Процес-нащадок відразу після породження закриває $M<N-K$ файлів і через

деякий час завершується (в цей час процес-предок очікує його завершення). Розробити програму, яка демонструвала б динаміку зміни даних в системі керування введенням-виведенням ОС Linux (таблиці відкритих файлів і масиви файлових дескрипторів процесів). Наприклад, сценарій програми може бути таким:

- відкриття процесом-предком стандартних файлів введення-виведення і чотирьох призначених для користувача файлів для зчитування;
- закриття процесом-предком двох призначених для користувача файлів;
- процес-предок породжує процес, який успадковує таблиці файлів і відкритих файлів процесу-предка;
- завершується процес-нащадок.

Після кожного з етапів друкуються таблиці відкритих файлів і масиви файлових дескрипторів для обох процесів.

Варіант 10

Нехай процес здійснює дії у відповідності з наступним фрагментом програми:

```
main ()
{
    ...
    fd = creat (temporary, mode); /* Відкрити тимчасовий файл */
    ...
    /* Виконання операцій запису-читання */
    ...
    close (fd);
}
```

Розробити програму, яка демонструвала б динаміку зміни даних системи керування введенням-виведенням ОС Linux (таблиця дескрипторів файлів, таблиця відкритих файлів, масив файлових дескрипторів процесу).

3. Для вашого варіанту розробити програму, що моделює роботу системи керування введенням-виведенням ОС Linux з ведення структур (таблиць), які відстежують операції введення-виведення в системі. Після кожного кроку роботи програми щодо операцій роботи з файлами, вона повинна роздруковувати у вигляді таблиць поточну інформацію стану таблиць відкритих файлів, файлів і дескрипторів файлів (не слід намагатись друкувати усю таблицю `inode` вашої файлової системи!).

   ***Вказівка*:** користуйтесь `stat()/fstat()`. Інформацію, що отримана зі структури `stat`, доповнену ім'ям файлу, і слід в лабораторних роботах трактувати як таблиці дескрипторів файлів. У тих завданнях, де потрібно відстежувати динаміку створення і модифікації таблиць файлів і таблиць відкритих файлів процесу, ці таблиці повинні програмно моделюватися

при виникненні обставин, вказаних в завданнях лабораторної роботи. Ніяких дій по створенню процесів в програмах виконувати не потрібно.

4. Налагодити і зневадити складену програму, використовуючи інструментарій ОС Linux.
5. Усі отримані результати оформити у вигляді протоколу.
6. Захистити лабораторну роботу, продемонструвавши викладачеві роботу розробленої вами програми та її код, а також відповівши на контрольні запитання.

Контрольні запитання

1. Яка структура дескрипторів файлів, таблиці відкритих файлів, таблиці відкритих файлів процесу?
2. Яким є ланцюжок відповідності дескриптора файлу, відкритого процесом, і файлом на диску?
3. Опишіть функціональну структуру операції введення-виведення (пули, асоціація їх з драйверами, способи передачі інформації і т.д.).
4. Яким чином здійснюється підтримка пристроїв введення-виведення в ОС Linux?
5. Яка структура таблиці відкритих файлів і масиву файлових дескрипторів процесу після відкриття файлу?
6. Яка структура таблиці відкритих файлів і масиву файлових дескрипторів процесу після закриття файлу?
7. Яка структура таблиці відкритих файлів і масиву файлових дескрипторів процесу після створення каналу?
8. Яка структура таблиці відкритих файлів і масиву файлових дескрипторів процесу після створення нового процесу?

Література та посилання

1. Таненбаум Э., Бос Х. Современные операционные системы. 4-е изд. — СПб.: Питер, 2015. — 1120 с.
2. Osamu Aoki. Debian Reference. URL: https://www.debian.org/doc/manuals/debian-reference/debian-reference.en.pdf
3. Памятка по Debian. URL: https://www.debian.org/doc/manuals/refcard/refcard
4. Steve Parker. Shell Scripting Tutorial. URL: https://www.shellscript.sh/
5. Шеховцов В. А. Операційні системи. — К.: Видавнича група BHV, 2005. — 576 с.
6. Sean Walberg. Delve into UNIX process creation. URL: https://www.ibm.com/developerworks/aix/library/au-unixprocess.html

   переклад російською, URL: https://www.ibm.com/developerworks/ru/library/au-unixprocess/index.html
7. Greg Ippolito. Програма для демонстрації викликів fork() і exec(). URL: http://www.yolinux.com/TUTORIALS/ForkExecProcesses.html
8. POSIX Threads. Материал из Википедии — свободной энциклопедии. URL: https://ru.wikipedia.org/wiki/POSIX_Threads
9. Микаел Григорян. Pthreads: Потоки в русле POSIX. URL: https://habr.com/ru/post/326138/
10. Пример многопоточности в C/C++ используя библиотеку pthread. URL: https://nelex.in.ua/content/%D0%BF%D1%80%D0%B8%D0%BC%D0%B5%D1%80-%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE%D0%BF%D0%BE%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D1%81%D1%82%D0%B8-%D0%B2-cc-%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D1%83%D1%8F-%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D1%83-pthread
11. Bil Lewis, Daniel J. Berg. PThreads Primer: A Guide to Multithreaded Programming. — SunSoft Press, A Prentice Hall Title, 1996. — 361 p. — URL: https://www8.cs.umu.se/kurser/TDBC64/VT03/pthreads/pthread-primer.pdf
12. Blaise Barney. POSIX Threads Programming. — Lawrence Livermore National Laboratory. URL: https://computing.llnl.gov/tutorials/pthreads/
13. Сергей Балабанов. Знакомство с межпроцессным взаимодействием на Linux. URL: https://habr.com/ru/post/122108/

14. Нейл Мэтью (Neil Matthew), Ричард Стоунс (Richard Stones). Основы программирования в Linux. URL: http://wm-help.net/lib/b/book/1696396857/
15. Sven Goldt, Sven van der Meer, Skott Burkett, Matt Welsh. Руководство программиста для Linux /Пер.: Алексей Паутов. URL: http://citforum.ck.ua/operating_systems/linux_pg/index.shtml або http://www.codenet.ru/progr/cpp/7/
16. Базовый курс Linux. URL: https://www.opennet.ru/docs/RUS/linux_base/
17. М. Джонс. Анатомия файловой системы Linux. URL: https://www.ibm.com/developerworks/ru/library/l-linux-filesystem/