ЗМІСТ

Ці методичні вказівки призначені для надання допомоги студентам 4-го курсу заочного факультету Одеського державного екологічного університету, які навчаються за напрямком «Комп'ютерні науки», при самостійному вивченні розділів курсу «Операційні системи» і при виконанні контрольної роботи з цього курсу.

Опис навчальної дисципліни

<table>
<tr><td rowspan="2">Найменування показників</td><td rowspan="2">Галузь знань, напрям підготовки, освітньо-кваліфікаційний рівень</td><td>Характеристика навчальної дисципліни</td></tr>
<tr><td>заочна форма навчання</td></tr>
<tr><td rowspan="2">Кількість кредитів – 4</td><td>Галузь знань
0501 Інформатика та обчислювальна техніка
(шифр і назва)</td><td rowspan="2">Нормативна</td></tr>
<tr><td>Напрям підготовки
050101 Комп'ютерні науки
(шифр і назва)</td></tr>
<tr><td>Модулів – 2</td><td rowspan="5">Спеціальність (професійне спрямування):
6.05010101 Інформаційні управляючи системи та технології
(шифр і назва)</td><td>Рік підготовки:</td></tr>
<tr><td>Змістових модулів – 4</td><td>4-й</td></tr>
<tr><td rowspan="2">Індивідуальне завдання
Контрольна робота
(назва)</td><td>Лекції</td></tr>
<tr><td>4 години</td></tr>
<tr><td rowspan="2">Загальна кількість годин – 144</td><td>Практичні, семінарські</td></tr>
<tr><td>– годин</td></tr>
<tr><td rowspan="6"></td><td rowspan="6">Освітньо-кваліфікаційний рівень:
бакалавр</td><td>Лабораторні</td></tr>
<tr><td>8 годин</td></tr>
<tr><td>Самостійна робота</td></tr>
<tr><td>132 години</td></tr>
<tr><td>Індивідуальні завдання</td></tr>
<tr><td>36 годин (за рахунок часу самостійної роботи)</td></tr>
<tr><td>іспит</td></tr>
</table>

Примітка.

Співвідношення кількості годин аудиторних занять до самостійної і індивідуальної роботи для заочної форми навчання становить:– 0.09

1. Мета та завдання навчальної дисципліни

Мета: придбання студентами основних відомостей з архітектури сучасних операційних систем (ОС), принципів функціонування окремих складових систем та взаємозв'язків між складовими.

Завдання: вивчення питань дисципліни згідно зі стандартом дисципліни й робочою програмою курсу шляхом самостійного ознайомлення з рекомендованою літературою (див. перелік наприкінці методичних вказівок), а також, закріплення теоретичного матеріалу під час виконання завдань контрольної роботи і лабораторних робіт, які охоплюють найбільш важливі теми дисципліни.

За результатами вивчення навчальної дисципліни студент повинен

знати: архітектуру сучасних операційних систем загального призначення, принципи керування оперативною пам'яттю, процесами і потоками, взаємодію потоків в ОС, структуру файлових систем та засоби захисту інформації;

вміти: засобами мови програмування C/C++ в ОС Windows та Linux керувати розподілом оперативної пам'яті, синхронізувати потоки та здійснювати обмін інформацією між ними, працювати з файловими системами та здійснювати захист операційних систем.

2. Програма навчальної дисципліни

Змістовий модуль 1. 3.03.01 Архітектура операційних систем.

Тема 1. 3.03.01.01 Основні концепції, еволюція, різновиди операційних систем.

Тема 2. 3.03.01.02 Архітектура та ресурси операційних систем.

Змістовий модуль 2. 3.03.02 Оперативна пам'ять, потоки та процеси.

Тема 1. 3.03.02.01 Планування та керування процесами і потоками.

Тема 2. 3.03.02.02 Багатозадачність, взаємодія потоків, міжпроцесова взаємодія.

Тема 3. 3.03.02.03 Керування оперативною пам'яттю.

Тема 4. 3.03.02.04 Організація пам'яті у захищеному режимі, керування розподілом пам'яті.

Змістовий модуль 3. 3.03.03 Файлова система.

Тема 1. 3.03.03.01 Логічна та фізична організація файлових систем.

Тема 2. 3.03.03.02 Реалізація файлових систем.

Тема 3. 3.03.03.03 Виконувані файли.

Тема 4. 3.03.03.04 Керування пристроями введення-виведення.

Змістовий модуль 4. 3.03.04 Мережеві, багатопроцесорні операційні системи та захист інформації.

Тема 1. 3.03.04.01 Мережні засоби операційних систем.

Тема 2. 3.03.04.02 Взаємодія з користувачем в операційних системах.

Тема 3. 3.03.04.03 Захист інформації в операційних системах.

Тема 4. 3.03.04.04 Завантаження та адміністрування операційних систем.

Тема 5. 3.03.04.05 Багатопроцесорні та розподілені системи.

3. Структура навчальної дисципліни

Назви змістових модулів і тем	Кількість годин					
	Заочна форма					
	усього	у тому числі				
		л.	п.	лаб.	інд.	с.р.
1	2	3	4	5	6	7
Модуль 1						
Змістовий модуль 1. 3.03.01 Архітектура операційних систем						
Тема 1. 3.03.01.01 Основні концепції, еволюція, різновиди операційних систем		0,5				1
Тема 2. 3.03.01.02 Архітектура та ресурси операційних систем		0,5				1
Разом за змістовим моду-	3	1	0	0	0	2

1	2	3	4	5	6	7
лем 1						
Змістовий модуль 2. 3.03.02 Оперативна пам’ять, потоки та процеси						
Тема 1. 3.03.02.01 Планування та керування процесами і потоками						6
Тема 2. 3.03.02.02 Багатозадачність, взаємодія потоків, міжпроцесова взаємодія		0,5		4		12
Тема 3. 3.03.02.03 Керування оперативною пам’яттю		0,5				12
Тема 4. 3.03.02.04 Організація пам’яті у захищеному режимі, керування розподілом пам’яті		0,5				6
Разом за змістовим модулем 2	41,5	1,5	0	4	0	36
Змістовий модуль 3. 3.03.03 Файлова система						
Тема 1. 3.03.03.01 Логічна та фізична організація файлових систем						4
Тема 2. 3.03.03.02 Реалізація файлових систем		0,5		4		12
Тема 3. 3.03.03.03 Виконувані файли						6
Тема 4. 3.03.03.04 Керування пристроями введення-виведення		0,5				6

1	2	3	4	5	6	7
Разом за змістовим модулем 3	33	1	0	4	0	28
Змістовий модуль 4. 3.03.04 Мережеві, багатопроцесорні операційні системи та захист інформації						
Тема 1. 3.03.04.01 Мережні засоби операційних систем						6
Тема 2. 3.03.04.02 Взаємодія з користувачем в операційних системах						6
Тема 3. 3.03.04.03 Захист інформації в операційних системах						6
Тема 4. 3.03.04.04 Завантаження та адміністрування операційних систем						6
Тема 5. 3.03.04.05 Багатопроцесорні та розподілені системи		0,5				6
Разом за змістовим модулем 4	30,5	0,5	0	0	0	30
Усього годин	108	4	0	8	0	96
Модуль 2						
ІНДЗ (контрольна робота №1)						36
Усього годин	0	0	0	0	0	36

4. Теми лабораторних занять

№ з/п	Назва теми	Кількість годин
1	Створення файлової системи ОС Windows	4
2	Потоки та їх синхронізація у ОС Windows	4
	Разом	8

5. Самостійна робота

№ з/п	Назва теми	Кількість годин
1	2	3
1	Основні концепції, еволюція, різновиди операційних систем	1
2	Архітектура та ресурси операційних систем	1
3	Планування та керування процесами і потоками	6
4	Багатозадачність, взаємодія потоків, міжпроцесова взаємодія	12
5	Керування оперативною пам’яттю	12
6	Організація пам’яті у захищеному режимі, керування розподілом пам’яті	6
7	Логічна та фізична організація файлових систем	4
8	Реалізація файлових систем	12
9	Виконувані файли	6
10	Керування пристроями введення-виведення	6
11	Мережні засоби операційних систем	6
12	Взаємодія з користувачем в операційних системах	6
13	Захист інформації в операційних системах	6
14	Завантаження та адміністрування операційних систем	6
15	Багатопроцесорні та розподілені системи	6
17	Підготовка до іспиту	36
	Разом	132

6. Індивідуальні завдання

Як індивідуальне завдання студенти заочного факультету виконують контрольну роботу №1.

Контрольна робота складається з трьох частин які охоплюють основні теми курсу і мають найбільшу значущість для розуміння архітектури сучасних ОС загального застосування. Ці теми пов'язані з питаннями керування віртуальною пам'яттю, файловою системою ОС і створенням багатопотокових застосувань з синхронізацією потоків і взаємодією між ними.

Для виконання контрольної роботи №1 відводиться 36 годин за рахунок загального часу відведеного на самостійну роботу (по 12 годин на кожну тему).

7. Методи навчання

Навчання ведеться за допомогою читання лекцій, виконання контрольної роботи, проведення лабораторних робіт та за рахунок самостійної роботи студентів.

8. Методи контролю

Оцінювання якості і терміну виконання контрольної роботи №1, контроль за самостійністю та терміном виконання лабораторних робіт під час екзаменаційної сесії. Наприкінці екзаменаційної сесії – проведення письмового іспиту.

9. Розподіл балів, які отримують студенти

<table>
<tr><td colspan="5">Поточне тестування та самостійна робота</td><td>Підсумковий тест (екзамен)</td><td>Сума</td></tr>
<tr><td>КР№1(1)</td><td>КР№1(2)</td><td>КР№1(3)</td><td>ЛР№1</td><td>ЛР№2</td><td rowspan="2">100</td><td rowspan="2">600</td></tr>
<tr><td>100</td><td>100</td><td>100</td><td>100</td><td>100</td></tr>
</table>

КР№1(1), КР№1(2), КР№1(3) – розділи контрольної роботи №1 – перший,

другий, третій відповідно.

ЛР№1, ЛР№2 – лабораторні роботи.

10. Шкала оцінювання: національна та ECTS

Сума балів за всі види навчальної діяльності (у відсотках)	Оцінка ECTS	Оцінка за національною шкалою
		для екзамену, курсового проекту, контрольної роботи
90 – 100	**A**	відмінно
82 – 89	**B**	добре
74 – 81	**C**	
64 – 73	**D**	задовільно
60 – 63	**E**	
35 – 59	**FX**	незадовільно з можливістю повторного складання
0 – 34	**F**	незадовільно з обов'язковим повторним вивченням дисципліни

11. Методичне забезпечення

1. Рльщиков В.Б. Операційні системи / Конспект лекцій. – Одеса: ОДЕКУ, 2015. – 150с.

2. Рольщиков В.Б. Методичні вказівки до виконання лабораторних робіт з курсу «Операційні системи» /Частина перша – Одеса: ОДЕКУ, 2013, – 89 с.

3. Рольщиков В.Б. Методичні вказівки до виконання лабораторних робіт з курсу «Операційні системи» /Частина друга – Одеса: ОДЕКУ, 2014, – 156 с.

4. Рольщиков В.Б. Методичні вказівки для самостійної роботи студентів і виконання контрольної роботи №1 з дисципліни «Операційні системи» для студентів заочного відділення – Одеса: ОДЕКУ, 2015, – 69 с.

12. Рекомендована література

Базова

1. Шеховцов В.А. Операційні системи / Підручник для студентів вищих навчальних закладів. – К.: Видавнича група BHV, 2005. – 576с.

2. Рльщиков В.Б. Операційні системи / Конспект лекцій. – Одеса: ОДЕКУ, 2015. – 150с.

Допоміжна

3. Таненбаум Э. Современные операционные системы. 2-е изд. Классика CS. – СПб.: Питер, 2002 – 1040с.

4. Таненбаум Э., Вудхалл А. Операционные системы: разработка и реализация. Классика CS. – СПб.: Питер, 2006 – 576с.

5. Распределенные системы. Принципы и парадигмы / Э.Таненбаум, М. ван Стеен. – СПб.: Питер, 2003. – 877с.

13. Інформаційні ресурси

1. http://computers.plib.ru/os/Teoria_OS/menu.html
2. http://citforum.ru/operating_systems/rtos/
3. http://chekalov.sumdu.edu.ua/os.09/index.htm

Завдання до контрольної роботи

1. Розділ перший

Розробити найпростіший програмний емулятор менеджера віртуальної сторінкової пам’яті комп’ютера з однорівневою таблицею сторінок.

Розроблена програма повинна забезпечувати:

- можливість вводу з клавіатури або з файла (найбільш бажаний варіант) віртуальної адреси в межах від 0 до максимальної за завданням адреси без одиниці;
- перетворення адреси у формат сторінка/зсув;

- відображення сторінки, що містить адресу, у відповідний сторінковий кадр (з обов'язковим урахуванням того, що сторінка вже може знаходитися в пам'яті);
- запис сторінки, що заміщається, на своє місце у файлі (якщо в цьому є необхідність);
- вивід результатів роботи в таблицю, приблизний вигляд якої показаний нижче.

Приблизний вигляд представлення результатів роботи програми

Віртуальна адреса	Сторінка	Сторінковий кадр	Зсув
25789	3	0	1213
417512	50	1	7912
17555	2	2	1171
213718	26	3	726
3206	0	4	3206

Таблиця результатів роботи програми може містити додаткові поля на розсуд розроблювача програми (наприклад, ознаку зміни сторінки).

Таблиця варіантів завдання

Варіант	1	2	3	4	5	6	7	8	9	10	11	12
Загальний віртуальний простір (адрес)	2^{22}	2^{24}	2^{23}	2^{21}	2^{20}	2^{20}	2^{23}	2^{22}	2^{21}	2^{24}	2^{24}	2^{23}
Розмір сторінкового кадру (подвійних слів)	2^{10}	2^{9}	2^{12}	2^{11}	2^{14}	2^{13}	2^{10}	2^{9}	2^{12}	2^{11}	2^{14}	2^{13}
Кількість сторінкових кадрів фізичної пам'яті	8	10	12	14	8	10	12	14	8	10	12	14
Алгоритм заміщення сторінок	LRU	FIFO	LRU	FIFO	LRU	FIFO	LRU	FIFO	LRU	FIFO	LRU	FIFO

Позначення LRU та FIFO означають застосування алгоритму заміщення сторінок Least Recently Used або First In First Out відповідно.

Заповнення сторінкових кадрів спочатку виробляється на вимогу, а потім по одному з алгоритмів заміщення сторінок відповідно до варіанта завдання.

Ознаку зміни сторінки встановлювати, використовуючи генератор псевдовипадкових чисел.

2. Розділ другий

Використовуючи системні виклики ОС Linux, розробити програму, яка виконує такі дії:

- у домашньому каталозі створює дерево каталогів і файлів відповідно до номера варіанта;
- с консолі приймає й записує у файл ReadMe.txt, розташований у каталозі FILES, номер розділу контрольної роботи, тему роботи, групу, прізвище, ім'я та по батькові автора контрольної роботи;
- в одному з каталогів за завданням, у файл int.bin записує 1000 випадкових цілих чисел, визначає і друкує на консоль розмір файла;
- аналогічно, у файл double.bin записує 1000 випадкових чисел; типу double, визначає і друкує на консоль розмір файла;
- у файл text.txt програмно копіює довільний текст, наприклад, ту ж саму програму мовою С;
- у файл symbols.chr заносить 1000 випадкових байтів у діапазоні від 0x20 до 0xfe;
- програмно робить перевизначення виводу зі стандартного пристрою виводу у файл на диску і виконує програмний обхід дерева каталогів.

Варіант 1

```
FILES/
|   ReadMe.txt
├──Overheads/
|       Ses00
|       int.bin
└──Demo/
    |   Extra.bin
    |   text.txt
    ├──PTR01/
    |       double.bin
    |       symbols.chr
    └──CHAR/
            file.c
            long.asc
```

Варіант 2

```
FILES/
|   ReadMe.txt
├──facts/
|   |   long.asc
|   └──Unicode/
|       |   double.bin
|       └──Icu/
|           text.txt
|           icud.dat
├──Mappings/
|       int.bin
└──Adobe/
        HKSCS.txt
        symbols.chr
```

Варіант 3

```
FILES/
|   ReadMe.txt
├──TechInfo/
|   |   double.bin
|   └──Lang/
|           af.txt
|           text.txt
└──ABBYY/
    └──Reader/
        |   int.bin
        |   Zlib.dll
        └──Support/
              long.asc
              symbols.chr
```

Варіант 4

```
FILES/
|   ReadMe.txt
├──System/
|       int.bin
├──Mappings/
|       long.asc
└──Adobe/
    |   HKSCS.txt
    |   text.txt
    └──Unicode/
        |   double.bin
        └──Icu/
            symbols.chr
            icud.dat
```

Варіант 5

```
FILES/
|   ReadMe.txt
├──PlugIns/
|       File.diz
└──BY/
    ├──Fine/
    |       long.asc
    |       Ablib.dll
    ├──Lingvo/
    |       double.bin
    ├──Reg/
    |       text.txt
    |       int.bin
    |       symbols.chr
    └──Src/
            Src.rar
```

Варіант 6

```
FILES/
|   ReadMe.txt
|   double.bin
├──Panel/
|   |   long.asc
|   └──Lingvo/
|       |   int.bin
|       |   Abbrev.lsd
|       ├──Dic/
|       |   |   text.txt
|       |   └──Index/
|       └──Support
└──Adobe/
        symbols.chr
        Index.dat
```

Варіант 7

```
FILES/
|    ReadMe.txt
|---Acronis/
|    |---Sandal/
|    |    |    symbols.chr
|    |    |    text.txt
|    |    |---Common/
|    |              int.bin
|    |              link.dll
|    |---TrueHome/
|         |    license.txt
|         |    long.asc
|         |---Common/
|                   double.bin
|---Image/
```

Варіант 8

```
FILES/
|    ReadMe.txt
|---BVRDE/
|    |    symbols.chr
|    |---Lex/
|    |         int.bin
|    |---Solutions/
|    |---Sounds/
|    |         long.asc
|    |---Templates/
|         |    text.txt
|         |---Files/
|         |    double.bin
|         |---Wizard/
|                   makefile
|---Sym/
```

Варіант 9

```
FILES/
|    ReadMe.txt
|---Comp/
|    |    text.txt
|    |    manifest
|    |---Comm/
|         |---Code/
|         |         double.bin
|         |---Docu/
|         |    |    int.bin
|         |    |    V24.odc
|         |    |---ru/
|         |              long.asc
|         |              ver.odc
|         |---Sym/
|                   symbols.chr
|---Index/
```

Варіант 10

```
FILES/
|    ReadMe.txt
|---CaliPo/
|    |    libre.exe
|    |---Calibre/
|         |    recomp.exe
|         |    long.asc
|         |---plugins/
|         |    |---imagefor/
|         |              text.txt
|         |---resour/
|              |    symbols.chr
|              |---content/
|                   |    int.bin
|                   |    box.png
|                   |---read/
|---fonts/
         double.bin
```

Варіант 11

```
FILES/
|    ReadMe.txt
|---Ascii/
|    |    long.asc
|    |---Form/
|         |---Code/
|         |         double.bin
|         |         symbols.chr
|         |---Docu/
|         |         text.txt
|         |---Sym/
|                   int.bin
|---Index/
|---Ole/
         Integers.bin
```

Варіант 12

```
FILES/
|    ReadMe.txt
|---Eclipse/
|    |    symbols.chr
|    |    facts.xml
|    |---PlugIns/
|         |    double.bin
|         |---AltHistory/
|         |         Alt.dll
|         |         long.asc
|         |---BackGroundCopy/
|         |    |    File_ID.diz
|         |    |---Reg/
|         |    |         text.txt
|         |    |---Src/
|         |              int.bin
|         |---System/
|                   Dest.on
|---Temp/
```

3. Розділ третій

Використовуючи системні виклики створення, синхронізації і завершення потоків ОС Linux розробити програму яка виконує дії, відповідно до варіанту завдання.

1) Три паралельних потоки обробляють цілочисельний масив M[100] (спочатку масив містить нулі). Перший потік пише в масив на місце нулів позитивні числа. Другий потік пише в масив негативні числа. Третій потік читає з масиву відмінні від нуля числа. Причому, читати інформацію з масиву він може лише тоді, коли хоча б один потік записав в масив число. Для контролю роботи кожний раз при читанні інформації третій потік виводить весь масив на екран. В кожний момент часу лише один потік може працювати з масивом.

2) Два паралельних потоки P1 і P2 обробляють масив M[100] натуральних випадкових чисел. P1 вилучає з масиву поруч розташовані парні числа, P2 – поруч розташовані непарні числа. Кожний раз при вилученні потоки друкують масив і елементи, що вилучаються.

3) Два паралельних потоки P1 і P2 обробляють масив натуральних випадкових чисел M[100]. P1 вилучає з масиву найбільші числа, P2 – найменші. Кожний раз при вилученні потоки друкують масиві і елементи, що вилучаються.

4) Три паралельних потоки P1, P2 і P3 обробляють три цілочисельних масиви випадкових чисел M1[100], M2[100], M3[100]. P1 вилучає з масивів M1 і M3 найменші числа. P2 вилучає з масивів M1 і M2 найбільші числа. P3 вилучає з масивів M2 і M3 число, що зустрічається в обох масивах. Кожний раз при вилученні потоки друкують масив і елементи, що вилучаються.

5) Три паралельних потоки P1, P2 і P3 обробляють цілочисельний масив M[100], що містить спочатку нулі. P1 пише на місце нулів підряд натуральні числа від 1 до 100 по одному за кожний сеанс, P2 пише на місце нулів підряд негативні цілі числа від -100 до -1 по одному за кожний сеанс, P3 читає з масиву числа, замінюючи їх нулями, якщо масив заповнений.

6) Чотири паралельних потоки P1, P2, P3 і P4 обробляють чотири цілочисельних масиви M1[100], M2[100], M3[100], M4[100], що містять спочатку нулі. P1 вставляє в масиви M4 і M1 натуральні парні числа від 1 до 100 по одному за кожний сеанс. P2 вставляє в масиви M1 і M2 натуральні непарні числа від 1 до 100 по одному за кожний сеанс. P3 вставляє в масиви M2 і M3 негативні парні числа від -1 до -100 по одному за кожний сеанс. P4 вставляє в масиви M3 і M4 негативні непарні числа від -1 до -100 по одному за кожний сеанс. Потоки закінчують роботу, коли в масиві немає нульових елементів.

7) Два паралельних потоки P1 і P2 обробляють цілочисельний масив M[100], що містить спочатку нулі. P1 записує в масив числа від 1 до 100 (до 5 чисел за сеанс), якщо в масиві є нулі. P2 читає з масиву числа, замінюючи їх нулями (читати можна, якщо в масиві немає нулів).

8) Два паралельних потоки P1 і P2 обробляють цілочисельну матрицю випадкових чисел M[20,20]. P1 замінює нулями стовпчик з максимальним елементом. P2 замінює нулями рядок з мінімальним елементом.

9) Два паралельних потоки P1 і P2 обробляють цілочисельну матрицю випадкових чисел M[20,20]. P1 замінює нулями всіх сусідів максимального елементу. P2 замінює нулями всіх сусідів мінімального елементу.

10) Три паралельних потоки P1, P2 і P3 обробляють цілочисельний масив M[100] випадкових чисел. Потік P1 копіює до свого власного масиву парні числа по три числа за один раз. Потік P2 теж саме робить з числами котрі кратні трьом. Останній потік після кожної операції виводить на термінал всі три масиви.

11) Два паралельних потоки P1 і P2 обробляють цілочисельний масив M[100], що містить спочатку нулі. P1 пише на місце нулів підряд натуральні числа від 1 до 100 по одному за кожний сеанс, P2 замінює ці числа їхнім квадратом по одному за кожний сеанс. Після операції кожен потік друкує свій ідентифікатор і зміненій їм масив.

12) Два паралельних потоки P1 і P2 обробляють цілочисельну матрицю M[20,20], котра містить нулі. Спочатку обидва потоки по черзі, по п'ять чисел

за один сеанс, замінюють нулі випадковими числами. По заповненню матриці, потік P1 замінює своїм ідентифікатором стовпчик з максимальною кількістю парних елементів, P2 замінює своїм ідентифікатором рядок з максимальною кількістю непарних елементів. У перехресті стовпчика і рядка записується сума ідентифікаторів потоків.

Загальні методичні вказівки до виконання контрольної роботи

Вибір варіантів розділів контрольної роботи виконується за формулою:

(трьохзначне число) mod 12+1

трьохзначне число у формулі задається наступним чином:

– для першої частини роботи – це останні три цифри залікової книжки;

– для другої частини – нове число створюється циклічним зсувом цифр у вихідному числі на один десятковий розряд вліво;

– для третьої частини – повторюються дії з попереднього пункту.

Наприклад, номер залікової книжки 12285, тоді номер варіанту для першого завдання буде 285 mod 12+1=10, відповідно для другого завдання вийде 852 mod 12+1=1 і, нарешті, для третього буде 528 mod 12+1 теж дорівнює 1.

До кожній частині вказується відповідна література і діапазон сторінок з якими необхідно ознайомитись до безпосереднього виконання завдання.

Крім того нижче до кожного розділу контрольної роботи надані деякі теоретичні відомості які можуть бути корисними при виконанні завданій.

Програми пишуться мовою програмування С або С++.

1. Перша частина

Література до виконання завдання: [1] – стор. 184 – 196, [3] – стор. 232 – 283, [4] – стор. 353 – 382, а також [2]. Крім того, нижче наведені деякі відомості з принципів роботи менеджерів віртуальної пам’яті.

Метою частини є з’ясування роботи менеджера віртуальної пам’яті шля-

хом програмного моделювання.

Програма повинна бути написана з використанням об'єктно-орієнтованої технології, з обов'язковим здійсненням моделювання й представленням відповідних UML діаграм. Мова програмування C++, використання віконного інтерфейсу в програмі не є обов'язковим, можна застосовувати термінальний режим.

Віртуальний адресний простір моделюється за допомогою звичайного бінарного файлу відповідної довжини (вказане у варіанті завдання), що містить чотирьохбайтні цілі числа без знака, старше слово яких вказує номер сторінки, а молодше – задає зсув усередині сторінки, наприклад, чотири послідовних байти зі значеннями у бінарному вигляді – 0x013D0999 задають номер сторінки 317 і зсув в ній 2457. Файл може бути створений окремим програмним модулем. Для запису і читання з файла краще за все використати об'єднання (union), яке містить або одне чотирьохбайтне число, або структуру з двох двобайтних слів.

2. Друга частина

Література до виконання завдання: [1] стор. 253 – 335, [3] стор. 14 – 65, [4] стор. 424 – 502 а також [2]. Крім того, нижче наведені деякі відомості з потрібного набору системних вик ліків для створення файлової системи, і надані кілька прикладів їх застосування.

Метою цієї частини є вивчення будови файлової системи ОС Linux, одержання вміння за допомогою системних викликів Linux створювати дерево каталогів, включати в нього файли різних типів, використовувати стандартні пристрої вводу-виводу, встановлювати права доступу до каталогів та файлів в них.

Програма повинна розроблятись мовою програмування C в середовищі операційної системи Linux. Використання віконного інтерфейсу в програмі є необов'язковим, застосування термінального режиму є кращим.

Частина завдання, що стосується обходу дерева каталогів може бути виконана за допомогою приклада наведеного у відповідній теоретичній частині методичних вказівок.

3. Третя частина

Література до виконання завдання: [1] стор. 70 – 75, 82 – 85, 110 – 148, [3] стор. 106 – 149, [4] стор. 68 – 174, а також [2]. Крім того, нижче наведені деякі відомості з потрібного набору системних вик ліків для створення файлової системи, і надані кілька прикладів їх застосування.

Метою цієї частини є вивчення створення потоків в ОС Linux, одержання студентами вміння за допомогою системних викликів Linux створювати потоки, які паралельно працюють у адресному просторі процесу і синхронізуються один з однім за допомогою різних механізмів синхронізації, які притаманні цієї ОС.

Програма повинна розроблятись мовою програмування С в середовищі операційної системи Linux. Використання віконного інтерфейсу в програмі є необов'язковим, застосування термінального режиму є кращим.

Для початкового заповнення масивів випадкових цілих чисел в програмі *необхідно використовувати* файл int.bin, який був створений в одному з каталогів згідно до варіанта першого розділу контрольної роботи і містить у себе 1000 випадкових цілих чисел.

Засоби синхронізації потоків ***обираються студентом самостійно*** відповідно до семантики варіанта завдання.

Основні теоретичні відомості і деякі приклади застосування системних викликів, які необхідні при виконанні контрольної роботи

1. Теоретичні відомості до першої частини

В основі віртуальної пам'яті лежить ідея про поділ понять «адресний простір» і «адреси пам'яті». Для приклада можна розглянути уявлюваний комп'ютер, команди якого мають 16-бітне поле адреси, а ОЗП складає лише 4096 слів. Кількість слів пам'яті, що адресуються, залежить тільки від числа бітів адреси й ніяк не пов'язане з числом реально доступних слів, тому програма, що працює на такому комп'ютері, мала б можливість звертатися до адресного

простору в 65536 слів пам’яті, однак у дійсності комп’ютер має набагато менше слів у пам’яті.

Раніш існувала жорстка відмінність між тими адресами, які менше 4096, і тими, які дорівнюють або більші за 4096. Перші – корисний адресний простір, а адреси вищі за 4095 – даремні, оскільки вони не відповідають реальним адресам пам’яті. Таким чином, між адресним простором і адресами пам’яті малась на увазі взаємооднозначна відповідність.

Ідея віртуальної пам’яті є в тім, що в будь-який момент часу можна одержати прямий доступ до 4096 слів пам’яті, але це не означає, що вони неодмінно повинні відповідати адресам пам’яті від 0 до 4095. Наприклад, можна вважати, що при звертанні до адреси 4096 слід використовувати слово пам’яті з адресою 0, при звертанні до адреси 4097 – слово з адресою 1, нарешті, при звертанні до адреси 8191 – слово з адресою 4095 і т.д. Інакше кажучи, створюється ***відображення адресного простору в дійсні адреси пам’яті***, таке відображення схематично представлене на рис. 1.

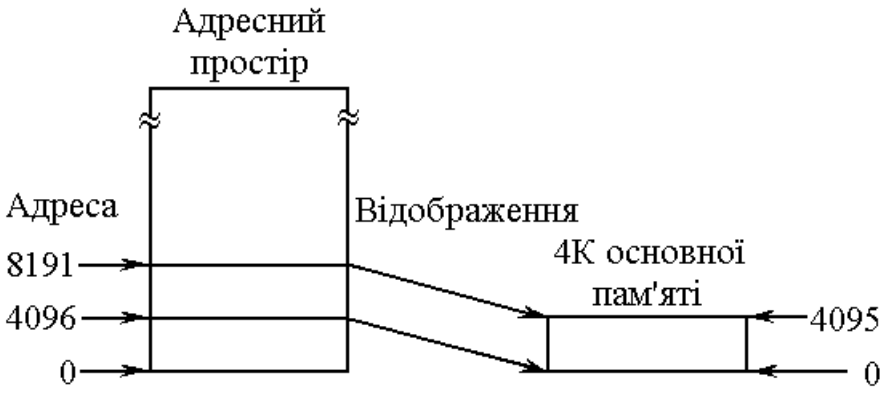

Рисунок 1 – Відображення віртуальних адрес від 4096 до 8191
в адреси основної пам’яті від 0 до 4096.

У випадку коли програма зробить перехід в одну з адрес, наприклад, від 8192 до 12287 у машині з віртуальною пам’яттю виконується наступна послідовність дій:

1. слова від 4096 до 8191 розміщуються на диску;

2. слова від 8192 до 12287 завантажуються в ОЗП;
3. змінюється відображення адрес (адреси від 8192 до 12287 відповідають коміркам пам’яті від 0 до 4095);
4. виконання програми продовжується так, начебто нічого не сталося.

У машині ж без віртуальної пам’яті відбудеться помилка, і виконання програми зупиниться.

Технологія автоматичного накладення називається ***сторінковою організацією пам’яті***, а частини, які зчитуються з диска, називаються ***сторінками***.

Можливі й інші, більш складні способи відображення адрес із адресного простору в реальні адреси пам’яті. Адреси, до яких програма може звертатися, називають ***віртуальним адресним простором***, а реальні адреси пам’яті в апаратному забезпеченні – ***фізичним адресним простором***. Для віртуальної пам’яті потрібен диск на якому є досить місця для зберігання якщо не повного віртуального адресного простору то, принаймні, тієї його частини, що використовується в цей момент. При зміні копії повинен змінюватися й оригінал на диску.

Програми можуть завантажувати слова з віртуального адресного простору або записувати слова у віртуальний адресний простір, незважаючи на те, що, насправді, фізичної пам’яті не вистачає. Програміст може писати програми, навіть не усвідомлюючи, що віртуальна пам’ять існує. Просто створюється таке враження, що обсяг пам’яті даного комп’ютера досить великий.

Необхідно підкреслити те, що сторінкова організація пам’яті є повністю прозорою і тільки створює ілюзію великої лінійної основної пам’яті такого ж розміру, що й адресний простір. У дійсності основна пам’ять може бути менша (або більша), ніж віртуальний адресний простір. З програми ні як не можна визначити те, що пам’ять великого розміру просто моделюється за допомогою сторінкової організації. При звертанні до будь-якої адреси завжди з’являються необхідні дані або потрібна команда.

Ілюзію, що всі віртуальні адреси підтримуються реальною пам’яттю, створює операційна система. Така ілюзія створюється наступним чином.

Віртуальний адресний простір розбивається на низку сторінок рівного розміру, зазвичай, від 512 байт до 64 КіБ, хоча іноді зустрічається й сторінки розміром в 4 МіБ. Розмір сторінки завжди дорівнює ступеню двійки. Фізичний

адресний простір теж розбивається на частині точно такого ж розміру, як і сторінки. Ці частини основної пам'яті називаються ***сторінковими кадрами***.

Для правильного відображення віртуальних адрес на фізичні, створюється так звана ***таблиця сторінок***. Кожен комп'ютер з віртуальною пам'яттю містить спеціальний пристрій для відображення віртуальних адрес на фізичні – ***контролер управління пам'яттю*** (MMU – Memory Management Unit). Він може розташовуватися як безпосередньо на мікросхемі процесора, так і на окремій мікросхемі. Для з'ясування того, як будується таблиця віртуальних сторінок і як працює контролер пам'яті, можна розглянути уявлюваний контролер управління пам'яттю, що, наприклад, відображає 32-бітну віртуальну адресу в 15-бітну фізичну адресу. Це означає, що йому потрібно мати 32-бітний вхідний регістр адреси й 15-бітний вихідний регістр.

Рис. 2 ілюструє принцип роботи такого контролера управління пам'яттю. Коли в контролер надходить 32-бітна віртуальна адреса, він поділяє цю адресу на дві частини – 20-бітний номер віртуальної сторінки і 12-бітне зміщення усередині цієї сторінки (передбачається, що сторінки у прикладі мають розмір 4 КіБ). Номер віртуальної сторінки використовується як індекс у таблиці сторінок для знаходження потрібної сторінки. У прикладі номер віртуальної сторінки дорівнює 3, тому з таблиці вибирається третій елемент.

З наведених даних випливає, що менеджер повинен управляти 2^{20} (приблизно мільйоном) віртуальних сторінок при наявності всього 8 сторінкових кадрів. Очевидно, що не всі віртуальні сторінки можуть одночасно знаходитися в пам'яті. Контролер управління пам'яттю спочатку перевіряє біт присутності потрібної сторінки в пам'яті в момент звернення до неї. У прикладі цей біт дорівнює 1, тобто сторінка знаходиться в пам'яті.

Далі з обраного елемента таблиці береться значення сторінкового кадру, в якому розташована сторінка (у прикладі – 6), і воно копіюється в старші три біти 15-бітного вихідного регістра. Необхідна наявність саме трьох бітів тому, що у фізичній пам'яті є 8 сторінкових кадрів. Паралельно з цією операцією молодші 12 бітів віртуальної адреси (поле зміщення всередині сторінки) копіюються в молодші 12 бітів вихідного регістра. Потім отримана 15-бітна адреса виставляється на шину адреси.

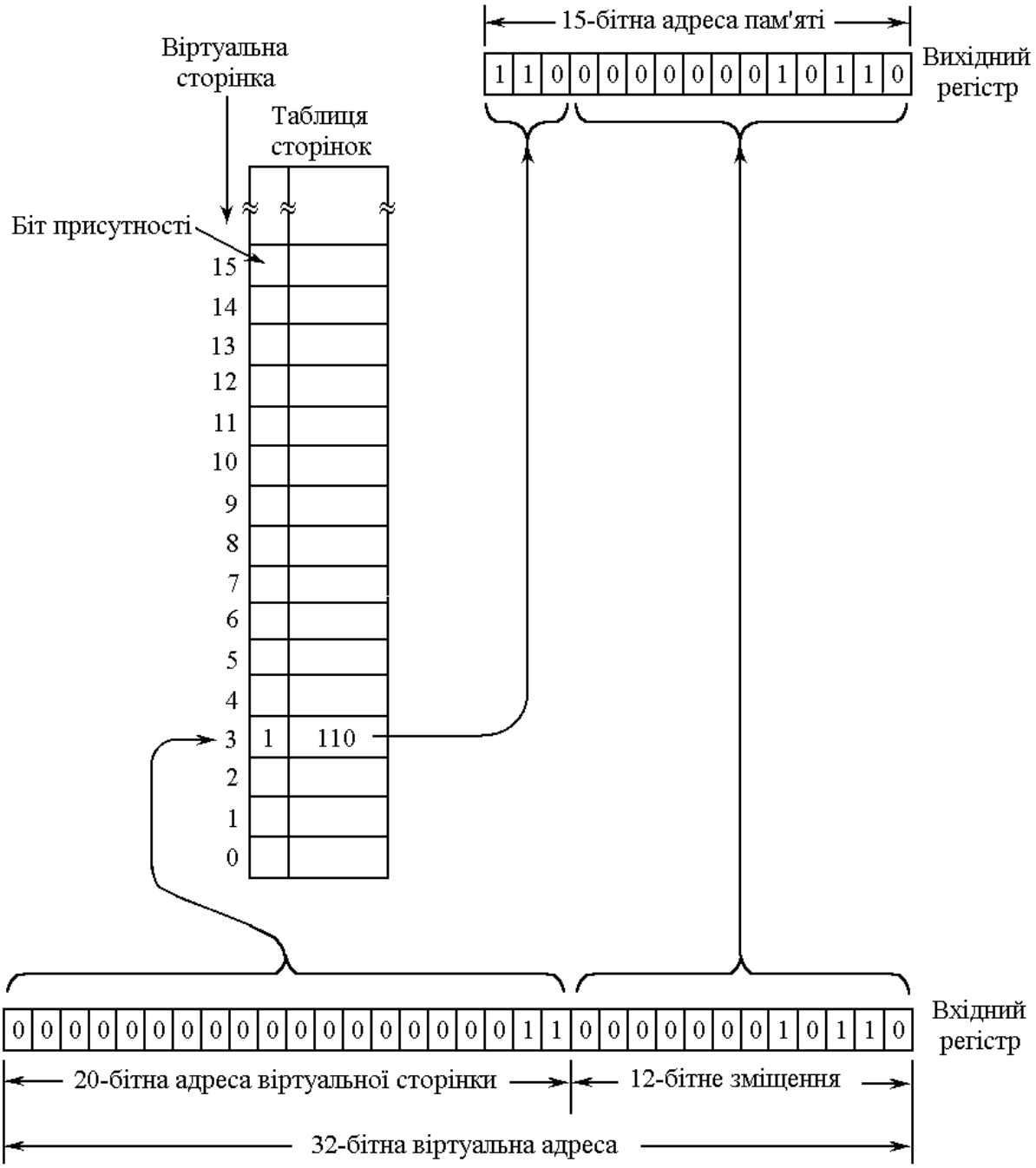

Рисунок 2 – Формування адреси основної пам’яті з адреси віртуальної пам’яті

Таким чином, відображення віртуальних сторінок у фізичні сторінкові кадри може бути зовсім довільним. Нижче на рис. 3 показане одне з таких можливих відображень. Віртуальна сторінка 0 знаходиться в сторінковому кадрі 1. Віртуальна сторінка 1 – у сторінковому кадрі 0. Сторінки 2 немає в основній пам’яті. 3-я сторінка міститься в сторінковому кадрі 2. Віртуальної сторінки 4 немає в пам’яті. Сторінка 5 поміщається в кадрі 6 і т.д.

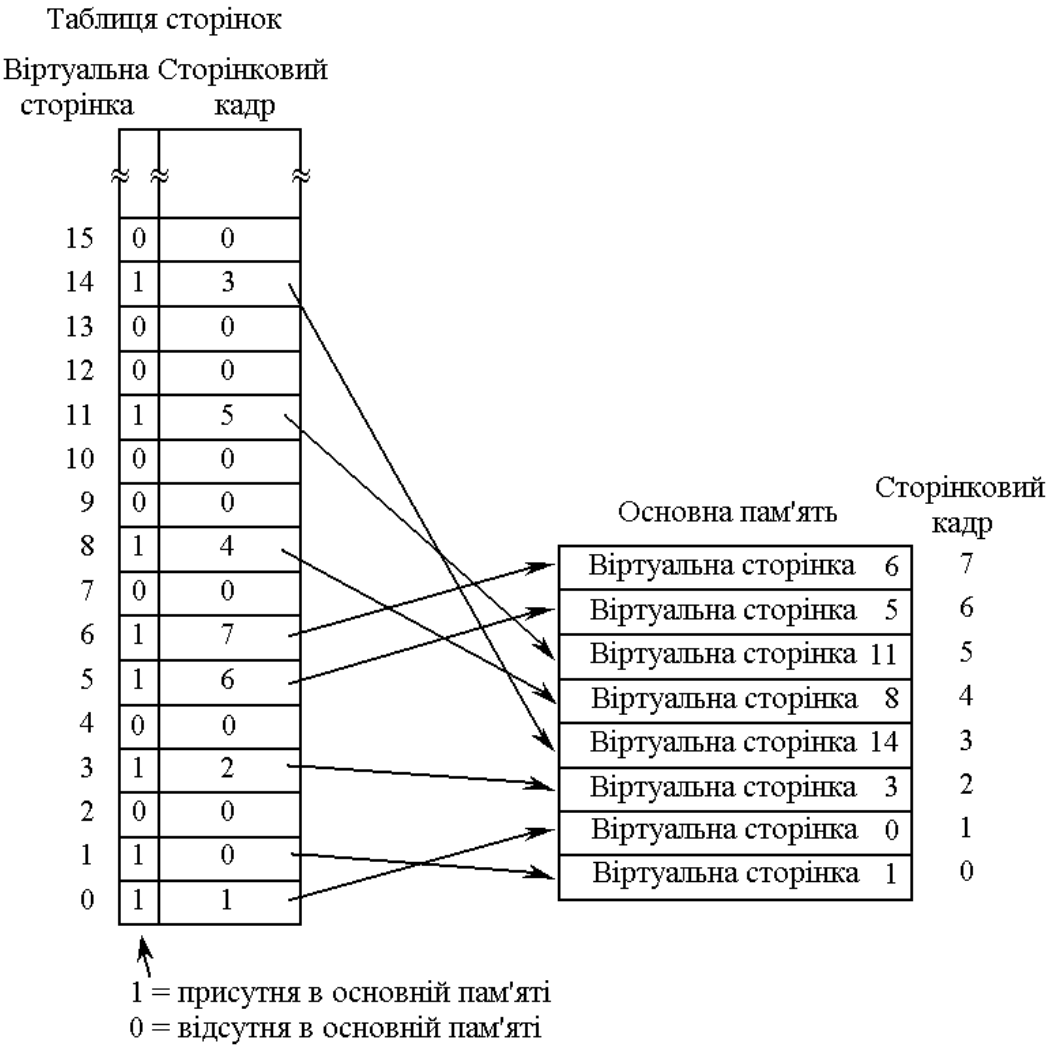

Рисунок 3 – Можливе відображення перших 16 віртуальних сторінок в основну пам’ять, що має 8 сторінкових кадрів

При звертанні до адреси, що знаходиться у сторінці якої немає в основній пам’яті, відбувається помилка через відсутність сторінки. У випадку такої помилки операційна система зчитує потрібну сторінку з диска, уводить у таблицю сторінок нову адресу фізичної пам’яті, а потім повторює команду, що викликала помилку.

На машині з віртуальною пам’яттю можна запустити програму навіть у тому випадку, коли жодній частини програми немає в основній пам’яті. Просто таблиця сторінок повинна вказувати, що абсолютно всі віртуальні сторінки перебувають у допоміжній пам’яті. Якщо центральний процесор намагається викликати першу команду, він відразу одержує помилку через відсутність сторін-

ки, у результаті чого сторінка, що містить першу команду, завантажується у пам'ять й вноситься до таблиці сторінок. Після цього починається виконання першої команди. Якщо перша команда містить дві адреси й обидві ці адреси знаходяться на різних сторінках, причому обидві сторінки не є сторінкою, у якій перебуває команда, то відбудеться ще дві помилки через відсутність сторінки й ще дві сторінки будуть перенесені в основну пам'ять до завершення команди. Наступна команда може викликати ще кілька помилок і т.д.

Такий метод роботи з віртуальною пам'яттю називається «викликом сторінок на вимогу». При виклику сторінок на вимогу сторінки переносяться в основну пам'ять тільки тоді, коли в цьому виникає потреба, але не заздалегідь.

Цілком очевидно, що використання алгоритму виклику сторінок на вимогу, має сенс тільки на самому початку при запуску програми. Коли програма проробить якийсь час, потрібні сторінки вже будуть зібрані в основній пам'яті.

Фахівцями з операційних систем було помічено, що більшість звертань до пам'яті зазвичай належать до невеликого числа сторінок. Тобто в кожний момент часу існує набір сторінок, які використовувалися при деякій кількості останніх звернень. Такий набір сторінок називають «робочою множиною». Робоча множина зазвичай змінюється дуже повільно.

В ідеалі набір сторінок (робоча множина), які постійно використовуються програмою**,** можна було б зберігати в пам'яті, щоб скоротити кількість помилок через відсутність сторінок. Однак навіть у цьому випадку програма може звернутися до сторінки, якої немає в основній пам'яті, і контролеру віртуальної пам'яті буде необхідно викликати її з диска. Отже в цьому разі, у пам'яті потрібно звільнити для неї місце, а для цього на диск потрібно відправити якунебудь іншу сторінку.

Вибирати таку сторінку просто навмання не можна, і, вочевидь, потрібний алгоритм для визначення, яку саме сторінку необхідно усунути з пам'яті. Більшість операційних систем намагаються завбачувати, які зі сторінок у пам'яті є найменш корисними в тому розумінні, що їхня відсутність не вплине сильно на хід програми. Іншими словами, замість того щоб усувати сторінку,

що незабаром знадобиться, операційна система намагається вибрати таку сторінку, котра не буде потрібна довгий час.

Існує безліч алгоритмів, що здійснюють вибір непотрібних сторінок, але у контрольній роботі використовуються тільки два з них, які здійснюються найбільш просто.

Цілком логічно для вибору сторінки, що усувається, використовувати алгоритм FIFO (First In First Out – першим прийшов – першим пішов). По цьому алгоритму усувається та сторінка, що завантажувалася раніш за всіх, незалежно від того, коли в останнє відбувалося звертання до цієї сторінки.

Для реалізації алгоритму з кожним сторінковим кадром зв'язується лічильник звертань. Від початку всі лічильники встановлюються у 0.

Після кожної помилки через відсутність сторінок, лічильники всіх сторінок, що перебувають у пам'яті, збільшуються на 1, а лічильник викликаної сторінки приймає значення 0. Коли потрібно вибрати сторінку для усунення, вибирається сторінка із самим великим значенням лічильника. Оскільки сторінка не завантажувалася у пам'ять дуже давно, існує велика ймовірність того, що вона не знадобиться найближчим часом.

Можна застосовувати й інший алгоритм – усувається та сторінка, котра використовувалася найбільш давно, оскільки ймовірність того, що вона буде в поточній робочій множині, дуже мала. Цей алгоритм називається LRU (Least Recently Used – алгоритм усунення елементів, що використовувалися дуже давно). Реалізація цього алгоритму, а точніше сказати його різновиду, що називається NFU (Not Frequently Used – сторінка, що рідко використовувалася) виконується за допомогою лічильника аналогічного лічильнику в алгоритмі FIFO. Але тепер скидання лічильника у 0 виконується не при завантаженні сторінки у пам'ять, а при звертанні до неї. Зрозуміло, що й зараз максимальне значення лічильника буде мати сторінка, звертання до якої виробляється рідше за все.

Необхідно розуміти, що у випадку коли розмір робочої множини більш, ніж число доступних сторінкових кадрів, жоден з існуючих алгоритмів не дає гарних результатів, і помилки через відсутність сторінок виникають доволі часто, має місце так звана «пробуксовка» (thrashing). Очевидно, що пробуксовка є

дуже небажаним явищем. І навпаки, навіть якщо програма використовує великий віртуальний адресний простір, але має невелику робочу множину, що змінюється повільно і вміщується в основну пам'ять, нічого страшного не відбувається. Це твердження має силу, навіть якщо програма використовує в сто разів більше слів віртуальної пам'яті ніж їх є у фізичній основній пам'яті.

Крім вибору сторінки, яку можна вивантажити з пам'яті, менеджер пам'яті повинен вирішити ще одну задачу: чи треба зберігати на диску сторінку, котра вивантажується? Дійсно, якщо сторінка, яку потрібно усунути, не змінювалася з тих пір, як її зчитали (наприклад, це цілком імовірно, якщо сторінка містить програму, а не дані), то необов'язково записувати її назад на диск, оскільки точна копія там вже існує. У противному випадку, якщо сторінка змінювалася, то копія на диску вже їй не відповідає, і її потрібно оновити.

У багатьох контролерах управління пам'яттю, урахування зміни сторінки здійснюється додаванням ще одного біта стану для кожної сторінки, котрий дорівнює 0 при завантаженні сторінки й набуває значення 1, коли мікропрограма або апаратне забезпечення змінюють цю сторінку. По цьому бітові операційна система визначає, змінювалася дана сторінка або ні й чи потрібно її перезаписувати на диск або ні.

2. Теоретичні відомості до другої частини роботи

Файли і каталоги в Linux

У середовищі Linux, як і у UNIX, файли – особливо важливі поняття, оскільки вони забезпечують простий і узгоджений взаємозв'язок зі службами операційної системи й пристроями. Автор однієї з книг по програмуванню в Linux жартує – «В ОС Linux *файл – це все що завгодно.* Ну, або майже все!». Дійсно, в ОС Linux майже все представлено у вигляді файлів або може бути доступно за допомогою спеціальних файлів. Досить сказати, що навіть процеси в системі організовані у вигляді окремої частини файлової системи – каталогу procs. І основна ідея зберігається навіть, незважаючи на те, що в силу необхідності існують невеликі відмінності від традиційних файлів.

Це означає, що в основному програми можуть обробляти дискові файли, послідовні порти, принтери й інші пристрої точно так само, як вони використовують файли. При цьому найчастіше застосовуються п'ять базових функцій: open, close, read, write і ioctl. Каталоги – також спеціальний тип файлів. У сучасних версіях UNIX, включаючи Linux, навіть суперкористувач не робить безпосередній запис до них. Звичайно всі користувачі для читання каталогів застосовують інтерфейс opendir/readdir, і немає потреби знати подробиці реалізації каталогів у системі.

Крім умісту файли характеризуються ім'ям і набором властивостей, або «адміністративної інформації» – дата створення/модифікації файла й права доступу до нього. Властивості зберігаються у ***файловому індексі*** (inode), спеціальному блоці даних файлової системи, що також містить відомості про довжину файла й місце зберігання файла на диску. При роботі система використовує номер файлового індексу.

Каталог – це файл, що містить номери індексів і імена інших файлів. Кожний елемент каталогу – посилання на файловий індекс; при видаленні ім'я файла, віддаляється тільки посилання на нього. У системі можна створити посилання на той самий файл у різних каталогах. Тобто коли видаляється файл, видаляється елемент каталогу для цього файла, і кількість посилань на файл зменшується на одиницю. Дані файла можуть бути усе ще доступні завдяки іншим посиланням на цей же файл. Коли число посилань на файл стає рівним нулю, індекс файла й блоки даних, на які він посилається, більше не використовуються й позначаються як вільні.

Файли (точніше кажучи, записи про них) поміщаються в каталоги, які можуть містити підкаталоги. Так формується ієрархія файлової системи. Звичайно користувач зберігає файли у вихідному (/home/<ім'я_користувача>) каталозі з підкаталогами для зберігання електронної пошти, ділових листів, службових програм і т.і. Вихідні каталоги користувачів – це, як правило, підкаталоги каталогу більш високого рівня, що створюється спеціально для цієї мети, у нашім випадку це каталог /home, що у свою чергу є підкаталогом корене-

вого каталогу /, розташованого на верхньому рівні ієрархії й такого, що утримує всі системні файли й підкаталоги.

До файлів і пристроїв можна звертатися й управляти ними, застосовуючи невеликий набір функцій. Ці функції, відомі як ***системні виклики***, безпосередньо надаються системою UNIX (і Linux) і слугують інтерфейсом самої операційної системи.

Файли й пристрої використовуються однаково: вони можуть відкриватися, читатися, на них можна записувати і їх можна закривати. Наприклад, один і той виклик open, використовуваний для доступу до звичайного файла, застосовується для звертання до користувальницького термінала, принтеру або до стрічкового накопичувача.

До основних функцій низького рівня або системних викликів, що використовуються для звертання до драйверів пристроїв, належать наступні:

- open – відкриває файл або пристрій;
- read – читає з відкритого файла або пристрою;
- write – пише у файл або пристрій;
- close – закриває файл або пристрій;
- ioctl – передає керуючу інформацію.

Системний виклик ioctl застосовується для апаратно-залежного керування (як альтернатива стандартного вводу/виводу), тому він у кожного пристрою свій, і розгляд цього системного виклику не входить до завдання даної лабораторної роботи.

Довідкову інформацію із системних викликів можна одержати в розділі 2 інтерактивного довідкового керівництва (для цього слугує системна команда man 2 <ім'я_виклику>). Прототипи функцій зі списком параметрів, що використовуються в системних викликах і типом значення, котре повертається функцією, а також пов'язані з ними директиви #define з визначенням констант представлені у файлах include.

Бібліотечні функції

Використання системних викликів низького рівня безпосередньо для вводу й виводу може бути не дуже ефективним. Проблема полягає в тім, що, по-перше, при виконанні системних викликів ОС змушена переключатися з виконання користувальницького програмного коду на власний код ядра й потім вертатися до виконання програми. Тому бажано змушувати кожний такий виклик виконувати максимально можливий обсяг роботи, наприклад, зчитувати й записувати за один раз великі обсяги даних, а не одиночні символи. І, по-друге, у устаткування є обмеження, що накладаються на розмір блоку даних, які можуть бути зчитані або записані в будь-який конкретний момент часу й за цим треба стежити.

Для формування інтерфейсу високого рівня для пристроїв і дискових файлів дистрибутив Linux (і UNIX) надає низку стандартних бібліотек. Вони являють собою колекції функцій, які можна включати до програми. Гарним прикладом слугує стандартна бібліотека вводу/виводу, що забезпечує буферізований вивід. Але у програмах можна ефективно записувати блоки даних різних розмірів, застосовуючи бібліотечні функції, які виконують системні виклики низького рівня, постачаючи їх повними блоками. Це істотно знижує витрати системних викликів.

Бібліотечні функції, як правило, описуються в розділі 3 інтерактивного довідкового керівництва (man 3 <ім'я_функції>) й мають стандартний файл директиви include, наприклад, stdio.h для стандартної бібліотеки вводу/виводу.

На рис. 4 наведена умовна схема системи Linux, на якій показане як розташовані різні функції роботи з файлами відносно користувача, драйверів пристроїв, ядра системи й устаткування.

Доступ низького рівня до файлів

У лабораторній роботі ми будемо користуватися тільки системними викликами без звертання до бібліотечних функцій.

У кожної програми, що виконується, і називається ***процесом***, є ряд пов'я-

заних з нею ***дескрипторів файлів***. Дескриптор – ціле (small integer) число, що використовується для звертання до відкритих файлів і пристроїв. Кількість дескрипторів залежить від конфігурації системи. Коли програма запускається, у неї звичайно вже відкриті три дескриптори. До них належать наступні:

– 0 (STDIN_FILENO) – стандартний пристрій вводу;
– 1 (STDOUT_FILENO) – стандартний вивід;
– 2 (STDERR_FILENO) – стандартний потік помилок.

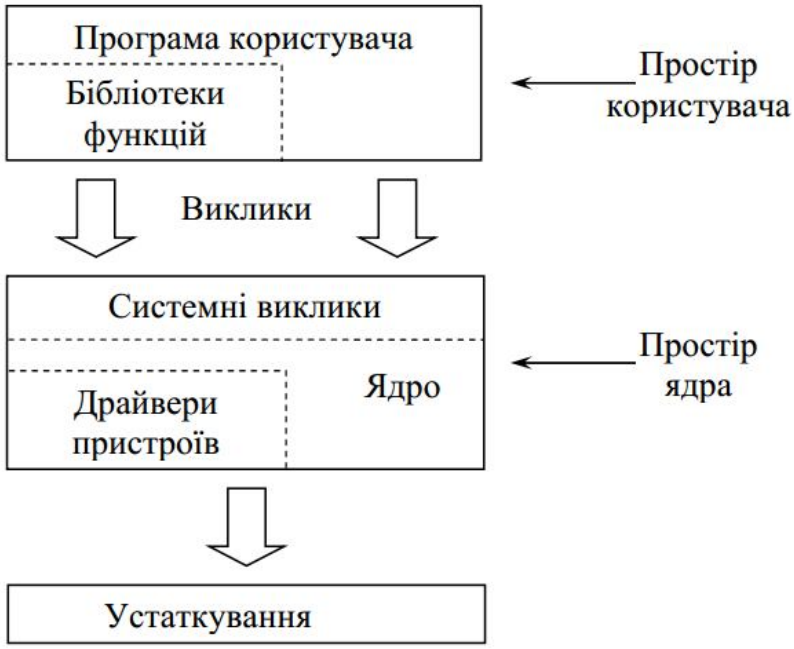

Рисунок 4 – Схематичне представлення використання системних викликів і бібліотечних функцій в ОС Linux

Дескриптори файлів, котрі відкриті автоматично, уже дозволяють створювати прості програми за допомогою звернення до виклику write. З файлами й пристроями можна зв'язати інші дескриптори файлів, використовуючи системний виклик open.

write

Системний виклик write призначений для запису з buf перших nbytes байтів у файл, асоційований з дескриптором fildes. Він повертає кількість реально записаних байтів, що може бути менше nbytes, якщо в дескрипторі файла

знайдена помилка або дескриптор файла, котрий розташований на більш низькому рівні драйвера пристрою, є чутливим до розміру блоку. Якщо функція повертає 0, це означає, що нічого не записане; якщо вона повертає -1, у системному виклику write виникла помилка, котра описується в глобальній змінній errno.

Далі наведений синтаксичний запис цього системного виклику.

```
#include <unistd.h>
size_t write(int fildes, const void *buf, size_t nbytes);
```

Отриманих відомостей про функцію write і стандартні дескриптори цілком достатньо щоб написати першу програму, simple_write.c

```
#include <unistd.h>
#include <stdlib.h>
int main() {
  if ((write (1, "Here is some data\n", 18)) != 18)
      write (2, "A write error has occurred on file
             descriptor 1\n", 46);
  exit(0); }
```

Ця програма просто поміщає повідомлення в стандартний вивід. Коли вона завершується, всі відкриті дескриптори файлів автоматично закриваються, і їх не потрібно закривати явно. Але у випадку буферізованого виводу це не так.

Виклик write може повідомити про те, що при його завершенні записане менше байтів, чим запитувалося. Це не обов'язково помилка. У програмах для виявлення помилок необхідно перевіряти змінну errno і, при необхідності, ще раз викликати write для запису даних, що залишилися.

read

Системний виклик read зчитує до nbytes байтів даних з файла, асоційованого з дескриптором файла fildes, і поміщає їх до області даних buf. Він по-

вертає кількість дійсно прочитаних байтів, котра може бути менше необхідної кількості. Повернення 0 означає досягнення кінця файла, повернення -1 – помилку при виклику.

```
#include <unistd.h>
size_t read (int fildes, void *buf, size_t nbytes);
```

Програма simple_read.c копіює перші 128 байтів зі стандартного вводу на стандартний вивід, або всі данні, що були введені, якщо їх кількість менше за 128 байтів.

```
#include <unistd.h>
#include <stdlib.h>
int main() {
  char buffer[128];
  int nread;
  nread = read(0, buffer, 128);
  if (nread == -1)
    write(2, "A read error has occurred\n", 26);
  if ((write(l, buffer, nread)) != nread)
    write(2, "A write error has occurred\n", 27);
  exit(0); }
```

open, creat

Для створення дескриптора нового файла використовується системний виклик open.

```
#include <fcntl.h>
#include <sys/types.h>
#inelude <sys/stat.h>
int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

При виклику open стандарт POSIX не вимагає підключення файлів sys/types.h і sys/stat.h, але в деяких системах UNIX вони можуть знадобитися.

Виклик open встановлює шлях до файла або пристрою. Якщо встанов-

лення минуло успішно, вертається дескриптор файла, що може застосовуватися в системних викликах read, write і ін. Дескриптор файла унікальний і не використовується спільно іншими процесами, які можуть у цей момент виконуватися. Якщо файл відкритий одночасно у двох програмах, вони підтримують окремі і різні дескриптори файла. Якщо вони обидві пишуть у файл, то дані однієї програми можуть бути записані поверх даних іншої.

Ім'я файла або пристрою, що відкривається, передається як параметр path; параметр oflags застосовується для зазначення дій, що вживаються при відкритті файла.

Параметр oflags задається як комбінація обов'язкового режиму доступу до файла й інших необов'язкових режимів. Список можливих значень параметра oflags наведений у табл. 1

Таблиця 1 – Константи параметра oflags у системному виклику open

Константи	Призначення параметра
	Обов'язкові режими
O_RDONLY	Відкриття тільки для читання
O_WRONLY	Відкриття тільки для запису
O_RDWR	Відкриття для читання й запису
	Необов'язкові режими
O_APPEND	Поміщає записувані дані в кінець файла
O_TRUNC	Задає нульову довжину файла, відкидаючи існуючий вміст
O_CREAT	При необхідності створює файл із правами доступу, заданими в параметрі mode
O_EXCL	Застосовується з режимом O_CREAT і гарантує, що викликаюча програма створить файл

Виклик open – ***атомарний***, тобто він виконується тільки цілком за один виклик функції. Це запобігає створенню файла з тим самим дескриптором двома програмами одночасно.

Всі можливі значення параметра oflags описані на сторінці інтерактивного довідкового керівництва, котра присвячена open (команда man 2 open).

У випадку успішного завершення виклик open повертає новий дескриптор файла – позитивне ціле число або -1 у випадку невдачі. У випадку невдачі

open також встановлює глобальну змінну errno, щоб показати причину невдачі. У нового дескриптора файла завжди найменший невикористаний номер дескриптора, іноді це дуже корисно. Наприклад, якщо програма закриває свій стандартний вивід, а потім знову викликає open, то повторно використовується дескриптор з номером 1 і стандартний вивід перенаправляється в інший файл або на інший пристрій.

Стандарт POSIX визначає також системний виклик creat, але він застосовується рідко. Виклик не тільки створює файл, але й відкриває його. Виклик є еквівалентний виклику open з параметром oflags = O_CREAT | O_WRONLY | O_TRUNC.

Кількість файлів, одночасно відкритих у будь-якій програмі, що виконується, обмежена значенням константи OPEN_MAX у файлі limits.h і змінюється від системи до системи, але стандарт POSIX вимагає, щоб воно було не менше за 16. В ОС Linux це граничне значення можна змінювати під час виконання програми й OPEN_MAX вже не є константою, а її початкове значення дорівнює 256.

close

Системний виклик close застосовується для розривання зв'язку файлового дескриптора fildes з його файлом. Дескриптор файла після цього може використовуватися повторно. При успішному завершенні виклик повертає 0 і -1 при помилці.

```
#include <unistd.h>
int close (int fildes);
```

Коли файл створюється із застосуванням прапора O_CREAT, то в системному виклику open необхідно використовувати форму з трьома параметрами. Третій параметр mode формується із прапорів, визначених у заголовному файлі sys/stat.h (див. табл. 2) і з'єднаних порозрядною операцією OR. Наприклад, виклик

```
open ("myfile", O_CREAT, S_IRUSR | S_IXOTH);
```

у результаті призведе до створення файла з ім'ям myfile і тільки із правами на читання для власника й на виконання для інших. Задані таким чином права встановлюються при створенні файла. Крім того, на права доступу до створюваного файла оказує вплив маска користувача, що задається командою командної оболонки umask. При цьому значення параметра mode у виклику open поєднується по AND з інвертованою маскою користувача вже на етапі виконання програми. Тобто надання запитаних прав, залежить від значення umask (системної змінної, в котрій міститься маска для прав доступу до файла) під час виконання.

Таблиця 2 – Прапори параметра mode для установки прав доступу

Константи	Призначення параметра
S_IRUSR	Право на читання власником
S_IWUSR	Право на запис власником
S_IXUSR	Право на виконання власником
S_IRGRP	Право на читання членами групи власника
S_IWGRP	Право на запис членами групи власника
S_IXGRP	Право на виконання членами групи власника
S_IROTH	Право на читання іншими користувачами
S_IWOTH	Право на запис іншими користувачами
S_IXOTH	Право на виконання іншими користувачами

Інші системні виклики для керування файлами

Існує ряд інших системних викликів, що оперують дескрипторами файлів низького рівня. Вони дозволяють програмі контролювати використання файла, повертаючи інформацію про його стан.

lseek

Системний виклик lseek задає покажчик поточної позиції читання/запису файла, тобто він застосовується для зазначення місця, з якого буде відбуватися

наступне зчитування або на яке буде провадитися наступний запис. Можна задати покажчик на абсолютну позицію у файлі або позицію відносно до поточного положення покажчика або відносно до кінця файла.

```
#include <unistd.h>
#include <sys/types.h>
off_t lseek(int fildes, off_t offset, int whence);
```

Параметр offset застосовується для зазначення позиції, а параметр whence визначає спосіб застосування offset. Можливі значення параметра наведені в табл. 3.

Таблиця 3 – Значення параметра whence у системному виклику lseek

Константи	Призначення параметра
SEEK_SET	Абсолютна позиція у файлі
SEEK_CUR	Позиція відносно поточної
SEEK_END	Позиція відносно кінця файла

При вдалому завершенні виклик lseek повертає величину зсуву відносно початку файла або -1 у противному випадку. Тип даних off_t залежить від реалізації системи й визначається у файлі sys/types.h.

fstat, stat і lstat

Ці системні виклики призначені для повернення інформації про стан файла, асоційованого з відкритим дескриптором файла. Інформація записується в структуру buf.

Синтаксичний запис викликів має вигляд:

```
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
int fstat(int fildes, struct stat *buf);
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

Функції stat і lstat працюють із файлами відкритими не по файловому дескриптору, а по символічному посиланню й повертають інформацію про стан такого файла. Виклик lstat повертає дані про саме посилання, а виклик stat – про файл, на який посилання вказує.

Елементи викликуваної структури stat можуть бути різними в різних UNIX-подібних системах. У табл. Таблиця **4** наведені такі поля структури, що є обов'язковими для всіх операційних систем.

Таблиця 4 – Обов'язкові поля структури stat

Поле	Призначення поля
st_mode	Права доступу до файла й відомості про тип файла
st_ino	Індекс, асоційований з файлом
st_dev	Пристрій, на якому розміщений файл
st_uid	Ідентифікатор (user identity) власника файла
st_gid	Ідентифікатор групи (group identity) власника файла
st_atime	Час останнього звертання
st_ctime	Час останньої зміни прав доступу, власника, групи або об'єм
st_mtime	Час останньої модифікації вмісту
st_nlink	Кількість твердих посилань на файл

У прапорів st_mode, що повертаються в структурі stat, у заголовному файлі sys/stat.h є ряд асоційованих макросів (див. табл. Таблиця **5**). У ці макроси включені імена прапорів для прав доступу й типів файлів і деякі маски, що допомагають перевіряти специфічні типи й права.

Прапори для прав доступу визначаються тими ж константами, що й при створенні файла.

Макроси для визначення типу файла допомагають визначити типи файлів, тобто належним чином порівнюють установлені прапори режиму файла з відповідним прапором, типу пристрою.

Наприклад, для перевірки того, що файл не є каталогом і в нього є права на виконання тільки для власника й більше ніяких інших прав, можна скористатися наступним тестом:

Таблиця 5 – Макроси прапорів поля st_mode структури stat

Мнемонічне ім'я	Призначення прапорця, маски або макроса
	Прапори для типів файла:
S_IFBLK	Блоковий пристрій
S_IFDIR	Каталог
S_IFCHR	Символьний пристрій
S_IFIFO	FIFO (іменований канал)
S_IFREG	Звичайний файл
S_IFLNK	Символічне посилання
	Інші прапори режимів файла
S_ISUID	Елемент одержує setUID при виконанні
S_ISGUID	Елемент одержує setGID при виконанні
	Маски, що інтерпретують прапори поля st_mode
S_IFMT	Тип файла
S_IRWXU	Права користувача на читання/запис/виконання
S_IRWXG	Права групи на читання/запис/виконання
S_IRWXO	Права інших на читання/запис/виконання
	Макроси для визначення типу файла
S_ISBLK	Перевірка для блокового файла
S_ISCHR	Перевірка для символьного файла
S_ISDIR	Перевірка для каталогу
S_ISFIFO	Перевірка для FIFO
S_ISREG	Перевірка для звичайного файла
S_ISLNK	Перевірка для символічного посилання

```
struct stat statbuf;
mode_t modes;
stat("filename", &statbuf);
modes = statbuf.st_mode;
if (!S_ISDIR(modes) && (modes & S_IRWXU) = S_IXUSR)
        ...
```

dup, dup2

Ці системні виклики дозволяють дублювати дескриптор файла, надаючи два або кілька різних дескрипторів, які звертаються до того самого файла. Ця можливість може застосовуватися для читання й запису в різні частини файла (наприклад, при закачуванні файла з Інтернету в кілька потоків).

Синтаксичний запис для викликів такий:

```
#include <unistd.h>
int dup(int tildes);
int dup2(int fildes, int fildes2);
```

Також ці виклики корисні, коли декілька процесів, взаємодіють через іменовані канали.

Приклад програми копіювання файла за допомогою системних викликів

Системні виклики дозволяють копіювати файл блоками доволі великого розміру, що дозволяє домогтися високої швидкості копіювання. Найпростіша програма copy_block.c, що працює із блоками 1 КіБ може мати такий вигляд:

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
int main(){
  char block[1024];
  int in, out;
  int nread;
  in = open("file.in". O_RDONLY);
  out = open("file.out". O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
  while((nread = read(in, block, sizeof(block))) > 0)
    write(out, block, nread);
  exit(0);}
```

У прикладі не передбачені перевірки результатів виконання системних викликів, які в справжній програмі обов'язково повинні виконуватися.

Ведення файлів і каталогів

На додаток роботи з файлами, системні виклики разом з функціями зі стандартної бібліотеки вводу/виводу С дозволяють забезпечити повний контроль над створенням і веденням файлів і каталогів у файловій системі.

chmod

Права доступу до файла або каталогу після їхнього завдання при відкритті файла можна змінити за допомогою системного виклику chmod.

Синтаксичний запис виклику має вигляд:

```
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
```

Права доступу до файла, ім'я якого задається параметром path, змінюються у відповідності зі значенням параметра mode як у системному виклику open (див. вище). Якщо програма не має достатніх повноважень, то тільки власник файла й суперкористувач системи мають можливість змінювати права доступу до файла.

chown

Крім зміни прав доступу до файла суперкористувач може змінити і його власника. Досягається це використанням системного виклику chown з таким прототипом:

```
#include <sys/types.h>
#include <unistd.h>
int chown(const char *path, uid_t owner, gid_t group);
```

Числові значення ідентифікаторів (ID) нового користувача й групи можна одержати із системних викликів getuid і getgid. Власник і група файла змінюються, тільки якщо задані відповідні повноваження.

unlink, link і *symlink*

Ці системні виклики використовуються для включення файла в який-небудь каталог, його видалення або створення символічного посилання на файл. Їхні прототипи указані нижче:

```
#include <unistd.h>
int unlink(const char *path);
int link(const char *path1, const char *path2);
int symlink(const char *path1, const char *path2);
```

Системний виклик unlink застосовується для видалення файла. Точніше сказати при використанні цього виклику видаляється тільки запис про файл у каталозі й зменшується на одиницю лічильник посилань на файл. Виклик повертає значення 0, якщо видалення посилання пройшло успішно, і -1 у випадку помилки. Для виконання виклику необхідно мати права на запис і виконання в каталозі, що зберігає посилання.

Якщо лічильник стає рівним нулю й файл не є відкритим у жодному процесі, він віддаляється повністю. У дійсності елемент каталогу завжди видаляється негайно, а місце, зайняте вмістом файла, не очищується доти, поки останній із процесів, що володіють файлом, не закриє його.

Навпаки системний виклик link створює нове жорстке посилання на існуючий файл path1 і приєднує його до каталогу path2. Аналогічно символічні посилання створюються за допомогою системного виклику symlink. Символічні посилання на файл не збільшують значення лічильника посилань і не заважають видаленню файла.

mkdir і *rmdir*

Системні виклики mkdir і rmdir призначені для створення й видалення каталогів.

Системний виклик mkdir із прототипом

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(const char *path, mode_t mode);
```

використовується для створення каталогів. При його виклику формується новий каталог з ім'ям, зазначеним у параметрі path. Права доступу до каталогу

передаються в параметрі mode аналогічно такому ж параметру системного виклику open і так само залежать від змінної umask.

```
#include <unistd.h>
int rmdir(const char *path);
```

Системний виклик rmdir, із указаним прототипом, видаляє каталоги за умови, що вони порожні. Значення, що повертаються, 0 і -1 є стандартними.

chdir і *getcwd*

Програма може змінювати поточний каталог аналогічно тому, як користувач виконує зміну каталогу у файловій системі, застосовуючи команду оболонки cd.

```
#include <unistd.h>
int chdir(const char *path);
```

Якщо в програмі необхідно визначити поточний робочий каталог, то це можна зробити шляхом системного виклику getcwd.

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

Функція getcwd записує ім'я поточного каталогу в заданий буфер buf. Вона повертає NULL, якщо ім'я каталогу перевищить розмір буфера (помилка ERANGE), котрій задається у параметрі size. У випадку успішного завершення вона повертає buf. Функція getcwd може також повернути значення NULL, якщо під час виконання програми каталог вже був видаленим (EINVAL) або змінилися його права доступу (EACCESS).

Перегляд каталогів

Досить часто в програмах виникає задача знаходження потрібного файла

у файловій системі. Для цього можна, наприклад, відкрити будь-який каталог як звичайний файл і безпосередньо зчитати його елементи, але різні структури і реалізації файлових систем зробили цей підхід таким, що не може бути перенесеним з машини на машину. Тому був розроблений стандартний комплект бібліотечних функцій, що істотно спрощує перегляд каталогів.

Функції роботи з каталогами оголошені в заголовному файлі dirent.h. У них використовується структура DIR як основа обробки каталогів. Покажчик на цю структуру, називаний ***потоком каталогу*** (DIR*), діє багато в чому так само, як діє потік файла (FILE*) при роботі зі звичайним файлом, за допомогою стандартної бібліотеки вводу/виводу. Оскільки досить не бажано безпосередньо змінювати поля в структурі DIR, елементи каталогу вертаються в структурі dirent, що також оголошена у файлі dirent.h.

При роботі з каталогами найчастіше використовуються наступні функції:

- opendir, closedir;
- readdir;
- telldir;
- seekdir;
- closedir.

opendir

Функція opendir відкриває каталог і формує його потік. Якщо вона завершується успішно, то повертає покажчик на структуру DIR, котра використовується для читання елементів каталогу. У противному випадку вертається порожній покажчик (NULL). Оголошення функції наступне:

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

readdir

Успішний виклик функції readdir із прототипом

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

повертає покажчик на структуру dirent, що містить наступний елемент каталогу. Переданий у функцію параметр dirp указує на структуру відкритого каталогу – DIR. Наприкінці каталогу readdir повертає NULL. Це ж значення вертається й при виникненні помилки, але стандарт POSIX у цьому випадку вимагає встановлення нового значення змінної errno.

У полі d_ino типу ino_t у структурі dirent утримується такий реквізит елемента каталогу, як індекс файла (inod), а в полі d_name[ ] типу char – ім'я файла. Інші реквізити файла в каталозі визначаються стандартним образом за допомогою системного виклику stat.

Перегляд каталогу за допомогою функції readdir не гарантує формування списку всіх файлів (і підкаталогів) у каталозі, якщо в цей час виконуються інші процеси, що створюють і видаляють файли в каталозі.

telldir

Функція telldir повертає значення, що реєструє поточну позицію в потоці каталогу. Прототип її наступний:

```
#include <sys/types.h>
#include <dirent.h>
long int telldir(DIR *dirp);
```

seekdir

Функція seekdir встановлює покажчик на елемент каталогу в потоці каталогу, котрий задається в параметрі dirp. Значення параметра loc, що застосовується для установки позиції, необхідно попередньо одержати з виклику функції telldir.

```
#inelude <sys/types.h>
#inelude <dirent.h>
void seekdir (DIR *dirp, long int loc);
```

closedir

Нарешті, функція closedir закриває потік каталогу й звільняє ресурси, що йому виділені. Як і завжди, вона повертає 0 у випадку успіху й -1 при наявності помилки.

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dirp);
```

Для демонстрації використання описаних функцій нижче наведена програма printdir.c. Вона створює простий перелік умісту каталогу. Кожний файл представляється окремим рядком. Ім'я кожного підкаталогу завершується символом косої риси, а файли, що втримуються в підкаталозі, виводяться з відступом шириною в чотири пробіли.

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>
#include <stdlib.h>
void printdir(char *dir, int depth) {
  DIR *dp;
  struct dirent *entry;
  struct stat statbuf;
  if ((dp = opendir(dir)) == NULL) {
    fprintf(stderr, "cannot open directory: %s\n", dir);
    return; }
  chdir(dir);
  while((entry = readdir(dp)) != NULL) {
    lstat(entry->d_name, &statbuf);
    if (S_ISDIR(statbuf.st_mode)) {
/* Знаходить каталог, але ігнорує каталоги з ім'ям . і .. */
      if (strcmp(".",entry->d_name) == 0
          || strcmp("..",entry->d_name) == 0) continue;
      printf("%*s%s/\n", depth, " ", entry->d_name);
/* Рекурсивний виклик з новим відступом */
```

```
      printdir(entry->d_name, depth+4);}
    else printf("%*s%s\n", depth, " ", entry->d_name); }
  chdir("..");
  closedir(dp); }

int main() {
/* Огляд каталогу /home */
  printf("Directory scan of /home:\n");
  printdir("/home", 0);
  printf("done.\n");
  exit(0); }
```

Обробка помилок

Як вказувалося вище, багато системних викликів і функції вказують причину збою операції, яка щойно виконувалася, встановлюючи значення зовнішньої змінної errno. Ця змінна використовується як стандартний спосіб оповіщення про проблеми, що виниклі. Програми повинні перевіряти значення змінної errno відразу ж після виникнення проблеми у функції.

Імена констант і варіанти помилок перераховані в заголовному файлі errno.h і наведені в табл. 6.

Таблиця 6 – Символічні імена констант помилок і їхня розшифровка

Мнемонічне ім'я	Зміст помилки
EPERM	Operation not permitted (Операція не дозволена)
ENOENT	No such file or directory (Немає такого файла або каталогу)
EINTR	Interrupted system call (Перерваний системний виклик)
EIO	I/O Error (Помилка вводу/виводу)
EBUSY	Device or resource busy (Пристрій або ресурс зайнятий)
EEXIST	File exists (Файл існує)
EINVAL	Invalid argument (Невірний аргумент)
EMFILE	Too many open files (Занадто багато відкритих файлів)
ENODEV	No such device (Немає такого пристрою)
EISDIR	Is a directory (Це каталог)
ENOTDIR	Isn’t a directory (Це не каталог)

У бібліотеках мови С є дві дуже корисних функцій strerror і perror, що

оперують зі змінною errno і повідомляють про помилки у разі їх виникнення при виконанні різних системних викликів та бібліотечних функцій.

Перша функція strerror, що має наступний синтаксичний запис

```
#include <string.h>
char *strerror(int errnum);
```

перетворює номер помилки в рядок, що описує тип виниклої помилки. Вона може бути корисна для реєстрації умов, що викликають помилку.

Прототип функції perror оголошується у заголовному файлі stdio.h і має вигляд:

```
#include <stdio.h>
void perror(const char *s);
```

Функція перетворює поточну помилку зі змінної errno, у рядок і виводить його до стандартного потоку помилок. Строковому повідомленню про помилку передує повідомлення, котре задається в рядку s (якщо покажчик не дорівнює NULL), за яким ідуть двокрапка й пробіл.

3. Теоретичні відомості до третьої частини роботи

Потоки в ОС Linux

При створенні нового потоку, він одержує власний стек (й, відповідно, локальні змінні), але використовує разом з батьківським процесом глобальні змінні, файлові дескриптори, оброблювачі сигналів і поточний каталог.

Стандарт POSIX 1003.1C стандартизував процедури роботи з потоками, і в даний час вони реалізовані практично у всіх ОС сімейства Linux. Багатоядерні процесори вже є звичайними для персональних комп'ютерів, і в більшості машин є низькорівнева апаратна підтримка, котра дозволяє їм виконувати кілька потоків одночасно. Раніше при наявності одноядерних процесорів одночасне виконання потоків було лише просто ефективною ілюзією.

Існує ціла низка бібліотечних викликів, пов'язаних з потоками, більшість ідентифікаторів цих викликів починається із префікса pthread. Для застосування цих бібліотечних викликів у програмі необхідно включити заголовний файл pthread.h, а компонування програми робити з використанням опції -lpthread. У деяких ОС система програмування вимагає також визначити спеціальний макрос _REENTRANT (це повинне робитись до того, як будуть об'явлені будь-які директиви #include), а так само визначення _POSIX_C_SOURCE, але у більшості випадків в цьому немає необхідності. Таки вимоги обумовлені тим, що при роботі із процесами необхідно використовувати, так звані, реентерабельні функції бібліотеки С. Реентерабельний програмний код може викликатися кілька разів або з різних потоків, або викликами, котрі якимсь чином вкладені один до одного, й при цьому працювати коректно. Отже, реентерабельна частина програмного коду, як правило, повинна застосовувати локальні змінні таким чином, щоб кожний з будь-яких викликів коду одержував власну унікальну копію даних.

Створення потоку

Для успішної роботи з потоками конче необхідно розглянути функції, що виконують керування ними: створення, завершення й таке інше.

pthread_create

Новий потік створюється функцією pthread_create.

```
#include <pthread.h>
int pthread_create(pthread_t * thread,
                   pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

Прототип функції більш складний, ніж у виклику fork, але застосування функції є досить простим. Перший аргумент – покажчик на змінну типу pthread_t. В області пам'яті на яку вказує цей покажчик, при створенні потоку міститься ***ідентифікатор потоку***, призначений для посилань на потік. Насту-

пний аргумент задає атрибути потоку. Як що немає потреби в особливих атрибутах, можна просто передати в цьому аргументі null. У третьому формальному аргументі виклику – void *(*start_routine)(void *) потокові передається адреса функції, що як параметр повинна приймати не типізований покажчик void, і повертати теж покажчик на void. Цю функцію потік повинен почати виконувати після створення. Таким чином можна передати єдиний аргумент будь-якого типу й повернути покажчик на будь-який тип. У четвертому, останньому параметрі вказуються аргументи, які потрібно передати цій функції, якщо вони їй потрібні.

Функція створення потоку, у випадку успішного завершення, повертає значення, яке, як прийнято, дорівнює 0 і, у противному випадку, вертається номер помилки.

pthread_exit

При завершенні потік повинен викликати функцію pthread_exit – декотрий аналог функції exit для процесу. Ця функція завершує потік, котрий її викликав, і повертає покажчик на об'єкт. Функція pthread_exit оголошується в такий спосіб:

```
#include <pthread.h>
void pthread_exit(void *retval);
```

Ніколи не можна використовувати функцію з поверненням покажчика на локальну змінну тому, що така змінна перестає існувати коли потік завершується при виникненні серйозної помилки.

pthread_join

Функція pthread_join – для потоків є еквівалентом функції wait, котру застосовують процеси для очікування своїх дочірніх процесів. Вона оголошується так:

```
#include <pthread.h>
int pthread_join(pthread_t th, void** thread_return);
```

Перший параметр – це ідентифікатор потоку, який відслідковується. Ідентифікатор, як відомо, вертається через покажчик – перший параметр у функції pthread_create при створенні потоку. Другий аргумент – покажчик на покажчик, що вказує на значення, котре повертається з потоку. Функція повертає нуль у випадку успішного завершення й код помилки у разі аварійної ситуації.

Нижче наведена проста програма, що створює один додатковий потік, демонструє спільне використання змінних обома потоками (вже існуючим у процесі й щойно створеним) й змушує новий потік повернути результат вихідному потоку.

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

void *thread_function(void *arg);
char message[] = "Hello World";

int main() {
  int res;
  pthread_t a_thread;
  void *thread_result;
  res = pthread_create(&a_thread, NULL,
                       thread_function,
                       (void *)message);
  if (res ! = 0) {
    perror("Thread creation failed");
    exit(EXIT_FAILURE);
  }
  printf("Waiting for thread to finish...\n");
  res = pthread_join(a_thread, &thread_result);
  if (res != 0) {
    perror("Thread join-failed");
    exit(EXIT_FAILURE);
  }
  printf("Thread-joined, it returned %s\n",
```

```
            (char *)thread_result);
    printf("Message is now %s\n", message);
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    printf("thread_function is running. Argument was
              %s\n", (char *)arg);
    sleep(3);
    strcpy(message, "Bye!");
    pthread_exit("Thank you for the CPU time");
}
```

На початку цієї програми оголошується прототип функції, котра буде викликана потоком після його створення. Цей прототип повністю відповідає оголошенню третього параметра в прототипі функції pthread_create (приймає покажчик на void, а повертає також покажчик на void).

У функції main оголошується декілька змінних і потім здійснюється виклик функції pthread_create, щоб почати виконання нового потоку.

Після нормального створення потоку в процесі тепер виконуються два потоки. Вихідний потік (main) триває й виконує код, розташований слідом за функцією pthread_create, а новий потік починає виконання функції, котра має назву thread_function (звісно, що ім'я цієї функції може бути цілком довільним).

Вихідний потік викликає функцію pthread_join, що перш ніж повернути керування, очікує завершення потоку з ідентифікатором a_thread, після чого виводиться значення, що повертається з потоку, а також вміст змінної message, котра була змінена у потоці, наприкінці програми робота процесу й двох його потоків припиняється.

Нижче наведений ще один приклад програми з потоками, але тут уже перевіряється одночасне виконання двох потоків. Логіка роботи програми заснована на спільному використанні змінних обома потоками в процесі (це не стосується локальних змінних у функції потоку).

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

void *thread_function(void *arg);
int run_now = 1;
char message[] = "Hello World";

int main() {
  int res;
  pthread_t a_thread;
  void *thread_result;
  int print_count1 = 0;
  res = pthread_create(&a_thread, NULL,
                thread_function, (void *)message);
  if (res != 0) {
      perror("Thread creation failed");
      exit(EXIT_FAILURE);
  }
  while(print_count1++ < 20) {
    if (run_now == 1) {
      printf("1");
      run_now = 2;
    } else {
      sleep(1);
    }
  }
  printf("\nWaiting for thread to finish...\n");
  res = pthread_join(a_thread, &thread_result);
  if (res != 0) {
    perror("Thread join failed");
    exit(EXIT_FAILURE);
  }
  printf("Thread joined\n");
  exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
  int print_count2 = 0;
  while(print_count2++ < 20) {
    if (run_now == 2) {
      printf("2");
      run_now = 1;
    } else {
```

```
        sleep(1);
      }
    }
    sleep(3);
  }
```

Програма являє собою незначно змінений приклад першої програми. У ній два потоки (один main, а другий у функції потоку), працюючи паралельно, змінюють значення змінної run_now, тим самим змушуючи її приймати значення, котре дорівнює 1 у першому потоці й 2 у другому.

Причому кожний з потоків, перш ніж змінити значення цієї змінної перевіряє його, і якщо воно не було змінено в паралельному потоці, то процес, котрий провадить перевірку, засипає на 3 секунди. Таким чином, здійснюється синхронізація роботи потоків. Цей прийом називається ***циклом активного*** або ***діяльного очікування*** (busy wait).

Синхронізація потоків

На жаль, такий процес синхронізації потоків є досить неефективним через цілу низку обставин. Тому на практиці для синхронізації потоків застосовують ряд функцій, спеціально розроблених для цих цілей.

До таких методів синхронізації належать використання ***семафорів***, котрі виступають у ролі сторожів, що охороняють фрагменти коду, а також ***м'ютексів*** (mutex – скорочення від mutual exclusions – взаємні виключення) – об'єктів, котрі у дечому схожі на семафори і діють як спеціальні пристрої взаємного виключення для захисту фрагментів програмного коду. Ці методи досить схожі, і один може бути описаний у термінах іншого. Але вибір того чи іншого методу синхронізації визначається семантикою розв'язуваного завдання. В одних випадках більш ефективним є використання м'ютексів, наприклад, керування доступом до деякої області спільно використовуваної пам'яті, до якої у кожний окремий момент часу може звертатися тільки один потік. Але у задачах керування доступом до низки ідентичних об'єктів у цілому, наприклад, надання потокові одного USB-порту з набору, що включає п'ять доступних портів, більше

підходить семафор.

Для семафорів існують два набори інтерфейсних функцій: один з них, в основному, застосовується для синхронізації процесів, другий набір включений до стандарту POSIX Real-time Extensions і застосовується для потоків. Тому надалі описаним є тільки другий набір і він повинен використовуватися при програмуванні синхронізації потоків.

Як це було показано в лекціях, семафор – це змінна особливого типу, що може змінюватися потоками з позитивним або негативним приростом, але звертання до змінної у відповідний момент завжди атомарно навіть у багатопоточних програмах. Це означає, що якщо два потоки (або декілька їх) у програмі намагаються змінити значення семафора, система гарантує, що всі операції будуть насправді виконуватися по черзі одна за іншою. У випадку звичайних змінних результат конфліктних операцій різних потоків в одній програмі є абсолютно довільним.

Найчастіше семафори використовуються для захисту фрагмента програмного коду, так щоб тільки один потік виконання міг змінити його в будь-який конкретний момент часу. Для вирішення цього завдання досить щоб семафор приймав тільки значення 0 і 1 (***двійковий*** або ***бінарний семафор***). Більше узагальнений тип семафора – ***семафор з лічильником*** застосовується тоді коли потрібно обмеженому числу потоків дозволити виконувати заданий фрагмент коду. На практиці у програмуванні склалась ситуація, що семафори з лічильником є менш популярними ніж бінарні.

Згідно зі стандартом POSIX, імена функцій для роботи з семафорами починаються із префікса sem_. Для роботи з потоками застосовують чотири базові функції.

sem_init

Семафор створюється за допомогою функції sem_init, що оголошується в такий спосіб.

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared,
             unsigned int value);
```

Ця функція ініціалізує об'єкт-семафор, на який указує параметр-покажчик sem, задає варіант його спільного використання pshared і привласнює йому початкове цілочисельне значення value. Параметр pshared управляє типом семафора. Якщо pshared дорівнює 0, семафор є локальним відносно поточного процесу. У противному випадку семафор може бути спільно використаний різними процесами. Не всі версії ОС Linux підтримують спільне використання семафорів, і передача ненульового значення параметру pshared призводить до аварійного завершення виклику.

sem_wait, *sem_post*

Наступна пара функцій управляє значенням семафора й оголошується вони в такий спосіб.

```
#include <semaphore.h>
int sem_wait(sem_t* sem);
int sem_post(sem_t* sem);
```

Обидві ці функції приймають покажчик на об'єкт-семафор, котрий був створений викликом функції sem_init.

Функція sem_post атомарно збільшує значення семафора на 1 (відповідає дії підняття семафора – up).

Функція sem_wait атомарно зменшує значення семафора на одиницю (дія опускання – down), але завжди чекає доти, поки спочатку лічильник семафора не одержить ненульове значення. Таким чином, якщо викликається sem_wait для семафора зі значенням 2, то потік продовжить своє виконання, а семафор буде зменшений до 1. Якщо sem_wait викликається для семафора зі значенням 0, функція буде чекати доти, поки який-небудь інший потік не збільшить значення, і воно стане ненульовим. Якщо на закритому семафорі очі-

кують два або більше потоків, то тільки один з них одержить можливість продовжитися при відкритті семафора, інші залишаться в стані очікування.

sem_destroy

Остання з базових функцій для роботи з семафорами – sem_destroy. Вона оголошується в такий спосіб:

```
#include <semaphore.h>
int sem_destroy(sem_t* sem);
```

і очищає семафор, коли закінчується робота з ним.

Якщо робиться спроба знищити семафор, на якому чекає хоча б один потік, то вертається помилка. У випадку ж успішного завершення, всі вище вказані функції, як завжди, повертають значення 0.

Приклад наведений нижче демонструє застосування семафорів для синхронізації потоків.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
sem_t bin_sem;
#define WORK_SIZE 1024
char work_area[WORK_SIZE];

int main() {
  int res;
  pthread_t a_thread;
  void *thread result;
  res = sem_init(&bin_sem, 0, 0);
  if (res != 0) {
    perror("Semaphore initialization failed");
    exit(EXIT_FAILURE);
  }
```

```
  res = pthread_create(&a_thread, NULL,
                       thread_function, NULL);
  if (res != 0) {
    perror("Thread creation failed");
    exit(EXIT_FAILURE);
  }
  printf("Input some text. Enter 'end' to finish\n");
  while (strncmp("end", work_area, 3) != 0) {
    fgets(work_area, WORK_SIZE, stdin);
    sem_post(&bin_sem);
  }
  printf("\nWaiting for thread to finish...\n");
  res = pthread_join(a_thread, &thread_result);
  if (res != 0) {
    perror("Thread join failed");
    exit(EXIT_FAILURE);
  }
  printf("Thread joined\n");
  sem_destroy(&bin_sem);
  exit(EXIT_SUCCESS);
}

void *thread function(void *arg) {
  sem_wait(&bin_sem);
  while(strncmp("end", work area, 3) != 0) {
    printf("You input %d characters\n",
           strlen(work_area)-1);
    sem_wait(&bin_sem);
  }
  pthread_exit(NULL);
}
```

У прикладі оголошується масив для спільного використання і семафор, котрий перед створенням нового потоку, одержує значення 0 (закритий).

У функції main, після запуску нового потоку, у робочу область, поки семафор закритий, читається деякий текст із клавіатури й потім за допомогою sem_post відкривається семафор. Це дозволяє другому потоку порахувати символи перед тим, як перший потік почне знову зчитувати ввід із клавіатури. Як видно з вихідного коду, потоки спільно використовують той самий масив work_area.

У прикладі пропущені деякі перевірки помилок, але в робочому програмному коді *обов'язково завжди перевіряти й виявляти помилки*, що повертаються із системних викликів.

У програмах подібних до розглянутої, через недотримання часових умов роботи декількох потоків можливо не передбачене нарощування семафора кілька разів. У цьому випадку один з потоків одержує можливість виконати свій код кілька разів, зменшуючи значення семафора поки воно не стане нульовим. Очевидно, що таке явище порушує логіку роботи програми, і повинне бути ліквідованим. Зробити це можна шляхом застосування додаткового семафора таким чином, щоб той потік, який багаторазово відкриває семафор, на другому семафорі очікував закінчення коду іншого процесу (приклад розглянутий у курсі лекцій).

Але в таких випадках набагато ефективніше застосування не семафорів, а м'ютексів.

М'ютекси дають можливість програмістам «замикати» поділюваний об'єкт так, що тільки один потік може звернутися до нього.

_mutex_init, *lock*, *unlock*, *destoy*

Базові функції, необхідні для використання м'ютексів, дуже схожі на функції семафорів. Вони оголошуються в такий спосіб:

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t* mutex,
            const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t* mutex);
int pthread_mutex_unlock(pthread mutex_t* mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Ну і як завжди, у випадку успішного завершення функцій вертається 0 і код помилки у випадку аварійного завершення, але ці *функції не встановлюють змінну* errno і потрібно використовувати тільки код повернення.

Як і функції семафорів, функції м'ютексів приймають покажчик на попе-

редньо створений об'єкт типу pthread_mutex_t. Додатковий параметр атрибутів у функції pthread_mutex_init дозволяє задати атрибути м'ютекса, що призначені для керування його поведінкою.

При деяких атрибутах поведінки (у тому числі й при атрибутах за замовчуванням) програма може потрапити в тупикову ситуацію. Це трапляється коли, наприклад, здійснюється спроба виклику функції pthread_mutex_lock для раніше вже заблокованого м'ютекса. Зміна атрибутів м'ютекса дозволяє уникнути такої ситуації й змушує м'ютекс або перевіряти наявність такої ситуації й повертати помилку або діяти рекурсивно, дозволяючи множинні блокування тим самим потоком, але, при цьому, у подальшому очікується така ж кількість розблокувань. Якщо зміни атрибутів м'ютекса не передбачається, то у другому параметрі потрібно передавати значення NULL, і використовувати поведінку за замовчуванням.

Нижче наведений приклад програми, що аналогічна попередньої, але використовує м'ютекс для забезпечення доступу до поділюваних змінних тільки одному потоку в кожній з моментів часу. Для простоти в програмі знову пропущені дії по перевірці помилок при поверненнях з заблокованого й відкритого м'ютекса. Але слід ще раз підкреслити, що в реальних програмах *таке спрощення не припустиме*.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
pthread_mutex_t work_mutex;
#define WORK_SIZE 1024
char work_area[WORK_SIZE];
int time_to_exit = 0;

int main() {
  int res;
```

```
  pthread_t a_thread;
  void *thread_result;
  res = pthread_mutex_init(&work_mutex, NULL);
  if (res != 0) {
    perror("Mutex initialization failed");
    exit(EXIT_FAILURE);
  }
  res = pthread_create(&a_thread, NULL,
                       thread_function, NULL);
  if (res != 0) {
    perror("Thread creation failed");
    exit(EXIT_FAILURE);
  }
  pthread_mutex_lock(&work_mutex);
  printf("Input some text. Enter 'end' to finish\n");
  while (!time_to_exit) {
    fgets (work_area, WORK_SIZE, stdin);
    pthread_mutex_unlock(&work_mutex);
    while(1) {
      pthread_mutex_lock(&work_mutex);
      if (work_area[0] != '\0') {
        pthread_mutex_unlock(&work_mutex);
        sleep(1);
      } else {
        break;
      }
    }
  }
  pthread_mutex_unlock(&work_mutex);
  printf("\nWaiting for thread to finish...\n");
  res = pthread_join(a_thread, &thread_result);
  if (res ! = 0) {
    perror("Thread join failed");
    exit(EXIT_FAILURE);
  }
  printf("Thread joined\n");
  pthread_mutex_destroy(&work_mutex);
  exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
  sleep(1);
  pthread_mutex_lock(&work_mutex);
  while(strncmp("end", work_area, 3) ! = 0) {
    printf("You input %d characters\n",
```

```
            strlen(work_area)-1);
      work_area[0] = '\0';
      pthread_mutex_unlock(&work_mutex);
      sleep(1);
      pthread_mutex_lock(&work_mutex);
      while (work_area[0] == '\0') {
        pthread_mutex_unlock(&work_mutex);
        sleep(1);
        pthread_mutex_lock(&work_mutex);
      }
    }
    time_to_exit = 1;
    work_area[0] = '\0';
    pthread_mutex_unlock(&work_mutex);
    pthread_exit(0);
}
```

У програмі, як і раніше, оголошується поділюваний масив робочої області, а також додаткова змінна time_to_exit. Тут же оголошується, а в потоці main створюється об'єкт м'ьютекса, після чого запускається новий потік. Перший потік робить читання даних у буфер доти, поки другий потік не встановить прапор завершення роботи time_to_exit. Другий потік підраховує кількість уведених до буферу символів, після чого очищає його вміст. Потік працює, поки в буфері не з'являться символи «end» після цього встановлює прапор time_to_exit. Під час роботи обидва процеси поперемінно за допомогою м'ютекса захоплюють у виняткове користування спільно поділюваний буфер. При цьому кожний з них намагається заблокувати м'ютекс і, коли це вдається, перевіряє, чи закінчив паралельно працюючий потік своє завдання. Якщо ні, м'ютекс відкривається, виконується очікування на якийсь час. Після установки прапора виконується приєднання другого потоку, видалення об'єкта м'ютекса й процес завершується.

Скасування потоку

Іноді логіка виконання програми вимагає, щоб якийсь потік завершувався на вимогу з іншого потоку. Для цього призначені кілька функцій.

pthread_cancel

Функція для створення запиту на завершення потоку.

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

Функція досить проста. Їй на вході передається ідентифікатор потоку, що підлягає завершенню. Функція стандартним образом повідомляє про вдале виконання або про помилку.

Але потік, що одержав повідомлення про закриття, може по-різному реагувати на цей сигнал.

pthread_setcancelstate

При запуску потоку він може за допомогою спеціальної функції pthread_setcancelstate із прототипом:

```
#include <pthread.h>
int pthread_setcancelstate(int state, int *oldstate);
```

установити власну реакцію на скасування. Параметр state у виклику функції може приймати два значення:

- PHTREAD_CANCEL_ENABLE – дозволяє потоку одержувати й реагувати на запити на скасування;
- PTHREAD_CANCEL_DISABLE – змушує ігнорувати запити на завершення.

Через покажчик oldstate функція повертає попередній стан. Якщо його значення не представляє інтересу, то можна передати параметр NULL.

Коли дозволена обробка запитів на скасування, то знову є дві альтернативні можливості обробки запиту.

pthread_setcanceltype

Після встановлення власної реакції потоку на скасування, тип реакції потоку тепер установлюється функцією pthread_setcanceltype і залежить від параметра type, переданого до неї. Цей параметр так само може приймати два значення:

- PTHREAD_CANCEL_ASYNCHRONOUS – запит на скасування обробляється негайно;
- PTHREAD_CANCEL_DEFERRED, скасування потоку відкладається доти, поки він не звернеться до якій-небудь із наступних функцій: pthread_join, pthread_cond_wait, pthread_cond_timedwait, pthread_testcancel, sem_wait, sigwait, у деяких системах можливе використання sleep.

Функція pthread_setcanceltype має наступний прототип:

```
#include <pthread.h>
int pthread_setcanceltype(int type, int *oldtype);
```

На закінчення опису роботи з потоками в ОС Linux, наведений приклад створення декількох (у прикладі шістьох) потоків у тому самому процесі з наступним видаленням їх у послідовності, що відрізняється від порядку створення (видалення робиться у зворотному порядку).

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#define NUM_THREADS 6

void *thread_function(void *arg);

int main() {
  int res;
  pthread_t a_thread[NUM_THREADS];
  void *thread_result;
  int lots_of_threads;
```

```
  for (lots_of_threads = 0; lots_of_threads <
       NUM_THREADS; lots_of_threads++) {
    res = pthread_create(&(a_thread[lots_of_thread]),
                         NULL, thread_function,
                         (void*)lots_of_threads);
    if (res != 0) {
      perror("Thread creation failed");
      exit(EXIT_FAILURE);
    }
  }
  printf("Waiting for threads to finish...\n");
  for (lots_of_threads = NUM_THREADS - 1;
       lots_of_threads >= 0; lots_of_threads-i) {
    res = pthread_join(a_thread[lots_of_threads],
                       &thread_result);
    if (res == 0) {
      printf("Picked up a thread\n");
    } else {
      perror("pthread_join failed");
    }
  }
  printf("All done\n");
  exit(EXIT_SUCCESS);
}

void* thread_function(void* arg) {
  int my_number = (int)arg;
  int rand_num;
  printf("thread_function is running. Argument was %d\n",
         my_number);
  rand_num = 1+(int)(9.0*rand()/(RAND_MAX+1.0));
  sleep(rand_num);
  printf("Bye from %d\n", my_number);
  pthread_exit(NULL);
}
```