МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
імені І.І. МЕЧНИКОВА

Інститут математики, економіки і механіки
Кафедра математичного забезпечення комп'ютерних систем

Н.Ф. Трубіна

МЕТОДИЧНІ ВКАЗІВКИ

до самостійної роботи студентів
та виконання контрольної роботи з дисципліни

«Системне програмування»

для студентів 3 курсу заочної форми навчання
напрям підготовки – комп'ютерна інженерія

Одеський національний
університет
імені І.І. Мечникова
2011

МЕТОДИЧНІ ВКАЗІВКИ до самостійної роботи студентів та виконанню контрольної роботи з дисципліни «Системне програмування» для студентів 3 курсу заочної форми навчання, напрям підготовки – комп’ютерна інженерія.

***Автор-укладач**:*

Н.Ф. Трубіна, старший викладач кафедри
математичного забезпечення комп'ютерних систем

***Рецензенти**:*

Т.І. Петрушина, к.ф-м.н., завідувач кафедрою математичного забезпечення комп'ютерних систем

Ю.Н. Крапівний, к.ф-м.н., доцент кафедри математичного забезпечення комп'ютерних систем

Рекомендовано до видання Вченою Радою
Інституту математики, економіки і механіки
ОНУ імені І.І. Мечникова.
Протокол № 1 від 9 жовтня 2009 року.

ЗМІСТ

ПЕРЕДМОВА

Методичні вказівки до самостійної роботи студента та виконанню контрольної роботи з дисципліни «Системне програмування» призначені для студентів 3 курсу напряму підготовки – комп'ютерна інженерія.

Мета методичних вказівок – поглиблене вивчення та закріплення як теоретичних знань, так і практичних навичок в створенні та використанні компонент системного програмного забезпечення.

Для підготовки дисципліни використана велика низка новітніх наукових та навчально-методичних робіт як зарубіжних так і вітчизняних фахівців у галузі системного програмування, тощо.

В методичних вказівках розглядаються питання, які відповідають навчальній програмі дисципліни.

Методичні вказівки містять основні відомості про навчальну дисципліну, короткий огляд теоретичного матеріалу, необхідного для виконання контрольної роботи, приклади розв'язання завдань та завдання до контрольної роботи. У додатках наведені прототипи системних викликів та коди помилок, що можуть виникати при їх виконанні.

1 ЗАГАЛЬНІ ВІДОМОСТІ ПРО ДИСЦИПЛІНУ

1.1 Мета дисципліни, її місце в навчальному процесі

Навчальна дисципліна „Системне програмування” належить до циклу підготовки бакалаврів з напряму «Комп’ютерна інженерія » 6.0904.

Курс “Системне програмування” орієнтований на вивчення основних принципів створення системних програм. Мета курсу — отримання як теоретичних знань, так и практичних навичок в створенні та використанні компонент системного програмного забезпечення, що повинне забезпечити вміння оперувати з простими і складними структурами даних,користуватися стандартними системними засобами вводу та виводу, працювати з дисковими файлами, розробляти драйвери, управляти розподілом пам’яті, файлами, процесами, здійснювати захист інформації від несанкціонованого доступу.

Загальний обсяг навчального часу, що припадає на вивчення дисципліни, дорівнює 122 години, з них: лекцій – 10, лабораторних занять – 6, самостійної роботи - 106.

Дисципліна являє собою вступ до тематики створення програмного забезпечення. У дисципліні висвітлюються питання, зв'язані з загальною архітектурою системи й описом її різних компонентів як блоків єдиної конструкції. Лабораторні роботи виконуються в середовищі операційної системи UNIX, яка застосовується сьогодні практично в усіх сферах інформаційних систем.

Студенти вивчать організацію файлової системи ОС UNIX і інформаційні структури й алгоритми, що використовуються файловою системою. Велика увага приділяється управлінню процесами, подаються загальні поняття системного програмування, поняття системного виклику як базового інтерфейсу з ядром операційної системи, обробку сигналів, принципи та методи побудови обробляючих програм в операційних системах, принципи визначення ефективно-

сті використання ЕОМ та побудови комфортного середовища для розробника системного програмного забезпечення.

Курс „Системне програмування” є першим в логічній послідовності дисциплін, що присвячені програмному забезпеченню сучасних комп’ютерних систем, йому передує вивчення таких нормативних дисциплін як „Архітектура ЕОМ”, „Програмування”.

1.2 Зміст дисципліни

1. Засоби та задачі системного програмування. Створення та використання бібліотек. Поняття системної програми. Типи системних програм. Підходи до створення системних програм

2. Середовище програмування UNIX. Системні виклики та бібліотечні підпрограми. Створення и виконання програми в середовищі UNIX. Взаємодія з командним рядком. Коди завершення програм. Обробка помилок.

3. Формати файлів, що виконуються. Створення та використання бібліотек.

4. Файлова підсистема. Робота с файлами. Примітиви доступу до файлів в системі UNIX. Системні виклики *open*, *close*, *read*, *write*, *lseek*. Дописування даних в кінець файла.

5. Файли в багатокористувачевому середовищі. Користувачі и права доступу. Права доступу и режими файлів. Додаткові права доступу. Маска створення файла і системний виклик *umask*. Зміна прав доступу (виклик *chmod*). Зміна володаря за допомогою виклику chown. Файли с декількома іменами. Знищення файла. Отримання інформації про файл.

6. Каталоги. Реалізація каталогів. Права доступу до каталогів. Використання каталогів при програмуванні. Створення та знищення каталогів.

7. Відкриття та закриття каталогів. Читання каталогів. Поточний робочий каталог. Зміна робочого каталогу за допомогою виклику chdir. Виявлення імені поточного робочого каталогу. Оббіг дерева каталогів

8. Пристрої. Драйвери пристроїв. Файловий інтерфейс. Файли блокових і символьних пристроїв. Апаратні пристрої. Спеціальні пристрої. Пристрої /dev/null,

/dev/zero, /dev/full/. Пристрої генерування випадкових чисел. Пристрої зворотного зв'язку.

9. Основи управління процесами. Структури данних процесу. Принципи управління пам’яттю. Створення процесів. Системний виклик fork. Запуск нових програм за допомогою виклику exec. Сімейство викликів exec. Сумісне використання викликів exec і fork.

10. Синхронізація процесів. Системний виклик wait. Очікування завершення конкретного нащадка: виклик waitpid. Зомбі-процеси и передчасне завершення програми.

11. Атрибути процесу. Ідентифікатор процесу. Системні виклики для отримання інформації про процес. Групи процесів и ідентифікатори групи процесів. Зміна групи процесу. Сеанси и ідентифікатор сеансу. Пріоритети процесів: виклик nice.

12. Сигнали и системні виклики. Процедури sigsetjmp и siglongjmp. Блокування сигналів. Посилка сигналів. Посилка сигналів другим процесом: виклик kill. Посилка сигналів самому процесу. Системний виклик pause.

13. Синхронізація процесів. Зомбі-процеси і передчасне завершення програми. Асинхронне видалення дочірніх процесів.

14. Межпроцесна взаємодія за допомогою програмних каналів. Канали. Канали на рівні команд. Використання каналів в програмі. Розмір каналу. Закриття каналів. Запис та читання без блокування. Іменовані канали (FIFO). Програмування за допомогою каналів FIFO.

15. Додаткові методи межпроцесної взаємодії. Блокування записів за допомогою виклику fcntl. Семафори.

16. Принципи апаратури введення-виведення. Програмні рівні введення-виведення. Дослідження введення-виведення.

1.3 Контролюючі заходи

Вивчення дисципліни „Системне програмування” для студентів 6-го курсу заочної форми навчання складається з лекційних та лабораторних занять, самостійної роботи студента по засвоєнню теоретичного курсу, виконання контрольної роботи.

Контроль самостійної роботи студента заочної форми навчання здійснюється шляхом перевірки контрольних робіт, опитів на лабораторних заняттях та на підсумковому іспиті.

2 ТЕОРЕТИЧНІ МАТЕРІАЛИ

2.1 Програмний інтерфейс UNIX

2.1.1 Системні виклики

Всі версії ОС UNIX надають строго обмежений набір входів у ядро операційної системи, через які прикладні програми мають можливість користуватися базовими послугами, що надаються ОС UNIX. Ці точки входу одержали назву ***системних викликів*** (system calls). Системний виклик, таким чином, визначає функцію, виконувану ядром ОС від імені процесу, і є інтерфейсом найнижчого рівня взаємодії прикладних програм з ядром.

Системні виклики документовані в розділі 2 електронного довідника. (man 2 kill). У середовищі програмування UNIX вони визначаються як функції С, незалежно від фактичної реалізації виклику функції ядра ОС. В UNIX використаний підхід, при якому кожен системний виклик має функцію (функції) з тим же ім'ям, що зберігається в бібліотеці мови С. Ці функції виконують необхідне перетворення аргументів і викликають необхідну процедуру ядра. У цьому випадку бібліотечний код виконує тільки роль оболонки, у той час як фактичні команди, розташовані в ядрі ОС.

Крім системних викликів програмістові пропонується великий набір функцій загального призначення. Вони не є точками входу в ядро ОС, хоча в ході виконання багато хто з них (але не все) виконують системні виклики. Ці функції зберігаються в системній бібліотеці С і поряд із системними викликами становлять середовище програмування ОС UNIX (документовані в розділі 2 електронного довідника).

Прототипи необхідних функцій наведені у додатку А.

2.1.2 Основні бібліотеки UNIX й їхні заголовні файли

Використання системних функцій звичайно вимагає включення в текст програми файлів заголовків, що містять визначення функцій. Більшість системних файлів заголовків розташовані в каталогах */usr/include* або */usr/include/sys*.

Файли заголовків включаються в програму за допомогою директиви *#include*. При цьому, якщо ім'я файлу укладене в кутові дужки, це означає, що пошук файлу буде здійснюватися в загальноприйнятих каталогах зберігання файлів заголовків. Якщо ж ім'я файлу заголовка укладено в лапки, то використається явно зазначене або відносне ім'я файлу.

Середовище програмування UNIX визначається декількома стандартами й може незначно розрізнятися для різних версій системи. Зокрема стандарти ANSI З, POSIX.1 й XPG4 визначають назви й призначення файлів заголовків, наведених у таблиці 1.

Таблиця 1 - Стандартні файли заголовків

Файл заголовка	**Призначення**
1	2
<assert.h>	Містить прототип функції assert, що використовується для діагностики
<cpio.h>	Містить визначення, використовувані для файлових архівів *cpio*
<ctype.h>	Містить визначення символьних типів, а також прототипи функцій класів символів
<dirent.h>	Містить визначення структур даних каталогу, а також функцій для роботи з каталогами
<errno.h>	Містить визначення кодів помилок
<fcntl.h>	Містить прототипи системних викликів *fcntl*, *open*, *creat*, а також визначення констант і структур даних, необхідних при роботі з файлами
<float.h>	Містить визначення констант, необхідних для операцій з дійсними числами
<ftw.h>	Містить прототипи функцій, використовуваних для сканування дерева файлової системи, а також визначення необхідних констант
<grp.h>	Містить прототипи функцій і визначення структур даних, використову-

	ваних для роботи із групами користувачів
<langinfo.h>	Містить визначення язикових констант дня тижня, назва місяця, а також прототип функції *langinfo*()
<limits.h>	Містить визначення констант, що визначають значення обмежень для даної реалізації
<locale.h>	Містить визначення констант, які використовуються для створення середовища користувача, що залежить від язикових і культурних традицій

Продовження таблиці 1

1	2
<math.h>	Містить визначення математичних констант
<nl_types.h>	Містить визначення для каталогів повідомлень, а також прототипи функцій *catopen* й *catclose*
<pwd.h>	Містить визначення структури файлу паролів, а також прототипи функцій для роботи з ним
<regex.h>	Містить визначення констант і структур даних, що використовуються у регулярних виразах, та прототипи функцій
<search.h>	Містить визначення констант і структур даних, а також прототипи функцій, необхідних для пошуку
<setjmp.h>	Містить прототипи функцій переходу, а також визначення пов'язаних з ними структур даних
<signal.h>	Містить визначення констант і прототипи функцій, необхідних для роботи із сигналами
<stdarg.h>	Містить визначення, необхідні для підтримки списків аргументів змінної довжини
<stddef.h>	Містить стандартні визначення
<stdio.h>	Містить визначення стандартної бібліотеки введення-виведення
<stdlib.h>	Містить визначення стандартної бібліотеки
<string.h>	Містить прототипи функцій роботи з рядками
<tar.h>	Містить визначення, використовувані для файлових архівів
<termios.h>	Містить визначення констант, структур даних і прототипи функцій для термінального введення-виведення
<time.h>	Містить визначення типів, констант і прототипи функцій для роботи із часом і датою, а також визначення, що ставляться до дат

<ulimit.h>	Містить визначення констант і прототип системного виклику *ulimit*() для керування обмеженнями, що накладають на процес
<unistd.h>	Містить визначення системних символьних констант, а також прототипи більшості системних викликів
<utime.h>	Містить визначення констант і прототип системного виклику utime для роботи з тимчасовими характеристиками файлу

Продовження таблиці 1

1	2
<sys/ipc.h>	Містить визначення, що ставляться до системи взаємодії між процесами (IPC)
<sys/msg.h>	Містить визначення, що ставляться до системи IPC (повідомлення)
<sys/resource.h>	Містить визначення констант і прототипи системних викликів для управління розмірами ресурсів, доступних процесу
<sys/sem.h>	Містить визначення, що ставляться до системи IPC (семафори)
<sys/shm.h>	Містить визначення, що ставляться до системи IPC(поділювана пам'ять)
<sys/stat.h>	Містить визначення структур даних і прототипи системних викликів, необхідних для одержання інформації про файли
<sys/times.h>	Містить визначення структур даних і прототип системного виклику *times*(), службовця для одержання статистики виконання процесу
<sys/types.h>	Містить визначення примітивів системних даних
<sys/utsname.h>	Містить визначення прототип системного виклику *uname*(), використовуваного для одержання імен системи
<sys/wait.h>	Містить визначення констант і прототипи системних викликів, використовуваних для синхронізації родинних процесів

2.1.3 Помилки системних викликів

Використовуючи системні виклики для доступу до ресурсів, здійснення операцій введення-виведення або інших цілей, потрібно розуміти не тільки те, що саме відбувається при успішному завершенні виклику, але також при яких обставинах він може завершитися не успішно. Збої системних викликів відбуваються в найрізноманітніших ситуаціях.

- У системі можуть скінчитися ресурси (або ж програма може вичерпати ліміт ресурсів, накладений на неї системою). Наприклад, програма може запросити занадто багато пам'яті, записати надто великий обсяг даних на диск або відкрити надмірну кількість файлів одночасно.
- Операційна система блокує деякі системні виклики, коли програма намагається виконати операцію при відсутності належних привілеїв. Наприклад, програма може спробувати здійснити запис у доступний тільки для читання файл, звернутися до пам'яті іншого процесу або знищити програму іншого користувача.
- Аргументи системного виклику можуть виявитися невірними або через помилково введені користувачем дані, або через помилку самої програми. Наприклад, програма може передати системному виклику неправильну адресу пам'яті або недійсний дескриптор файлу. Інший варіант помилки - спроба відкрити каталог замість звичайного файлу або передати ім'я файлу системному виклику, що очікує ім'я каталогу.
- Системний виклик може аварійно завершитися з причин, що не залежать від самої програми. Найчастіше це відбувається при доступі до апаратних пристроїв. Пристрій може працювати некоректно або не підтримувати необхідну операцію, або в пристрой просто не вставлений диск.
- Виконання системного виклику іноді переривається зовнішніми подіями, наприклад, при отриманні сигналу. Це не обов'язково означає помилку, але відповідальність за перепуск системного виклику покладає на програму.

У добре написаній програмі, що часто звертається до системних викликів, більша частина коду присвячена виявленню й обробці помилок, а не рішенню основного завдання.

Більшість системних викликів у випадку збою повертає значення -1. (Іноді використаються інші угоди. Наприклад, функція *malloc*() при виникненні помилки повертає нульовий покажчик.) Зазвичай цієї інформації досить для того,

щоб вирішити, чи варто продовжувати виконання програми. Але для більш спеціалізованої обробки помилок необхідні додаткові відомості.

Практично всі системні виклики зберігають у спеціальної змінної *errno* розширену інформацію про помилку, що відбулася. У цю змінну записується число, що ідентифікує помилку. Оскільки всі системні виклики працюють із однією й тією ж змінною, необхідно відразу ж після завершення функції скопіювати значення змінної в інше місце. Змінна *errno* модифікується після кожного системного виклику.

З метою забезпечення безпечної роботи потоків змінна *errno* реалізована у вигляді макросу, але до неї можна звертатися як до глобальної змінної.

Коди помилок є цілими числами. Можливі значення задаються макроконстантами препроцесора, які, за існуючою згодою, записуються прописними буквами й починаються з літери "E", наприклад EACCES й EINVAL. При роботі зі значеннями змінної errno варто завжди використовувати макроконстанти, а не дійсні числові значення. Всі ці константи визначені у файлі <errno.h>. Розшифровка кодів помилок наведена у додатку Б.

У стандартній бібліотеці мови С є зручна функція *strerror*(), що повертає строковий еквівалент коду помилки. Ці рядки можна включати в повідомлення про помилки. Оголошення функції перебуває у файлі <string.h> та виглядає наступним чином:

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

Є також функція perror() (оголошена у файлі <stdio.h>), що записує повідомлення про помилку безпосередньо в потік stderr. Перед власне повідомленням варто розміщати строковий префікс, що містить ім'я функції або модуля, що стали причиною збою.

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

У наступному фрагменті програми робиться спроба відкрити файл. Якщо це не виходить, виводиться повідомлення про помилку й програма завершує свою роботу.

```
fd  =  open("inputfile.txt",   O_RDONLY);
if   (fd == -1)
{
   /* Відкрити файл не вдалося.
      Виведення повідомлення про помилку
      та завершення роботи. */
   fprintf (stderr, "error opening file: %s\n",
            strerror(errno));
   exit (1);
}
```

Залежно від особливостей програми й використовуваного системного виклику конкретні дії, що вживають у випадку помилки, можуть бути різними: висновок повідомлення про помилку, скасування операції, аварійне завершення програми, повторне виконання й навіть ігнорування помилки. Проте, важливо включити в програму код, що обробляє всі можливі варіанти помилок.

2.1.4 Питання для самоперевірки

1. Що таке системний виклик? Чим він відрізняється від функцій стандартної бібліотеки?
2. Яким чином можна отримати інформацію про системний виклик?
3. Як включаються в програму файли заголовків?
4. Де зберігаються файли заголовків?
5. В яких ситуаціях виникають помилки системних викликів?
6. Яким чином системний виклик інформує про помилки?
7. Для чого використовується змінна errno?
8. Які функції використовуються для полегшення виведення повідомлень про помилки?

2.2 Взаємодія програми з середовищем виконання

При запуску будь-яка програма UNIX отримує від процесу, що її викликає, два набори даних: аргументи командного рядка і змінні середовища оточення. У програмах на мові С обидва набори представлено у вигляді масивів покажчиків, причому останній покажчик в кожному з масивів має значення NULL. Крім того, програма отримує лічильник, що містить кількість елементів в масиві аргументів. Крім того, при завершенні будь-яка програма повинна повідомити про причину свого завершення.

2.2.1 Список аргументів

Для запуску програми досить ввести її ім'я в командному рядку. Додаткові інформаційні елементи, передані програмі, також задаються в командному рядку і відділяються від імені програми і один від одного пропусками. Такі елементи називаються аргументами командного рядка. (Аргумент, що містить пропуск, повинен екрануватися.)

Коли програма запускається з командного рядка, список аргументів охоплює весь вміст рядка, включаючи ім'я програми і будь-які присутні аргументи. Наприклад, викликається програма ls, що відображує вміст кореневого каталогу і розміри відповідних файлів:

```
%ls  -s   /
```

В даному випадку список аргументів програми ls складається з трьох елементів. Перший — це ім'я самої програми, вказане в командному рядку, а саме ls. Другий і третій елементи — аргументи командного рядка -s та /.

Виконання програм на мовах С і С++ починається з функції main(). Функція main() дістає доступ до списку аргументів за допомогою своїх параметрів. Традиційно ця функція визначається таким чином:

```
int main(int argc, char *argv[]);
```

Параметр argc містить кількість переданих програмі параметрів, включаючи ім'я програми. Параметр *argv* - це масив покажчиків на рядки, що містять параметри. Розмір масиву рівний *argc*. Через *argv*[0] адресується ім'я програми *argv*[1] вказує на ім'я програми і так далі до *argv*[*argc*-1].

Якщо програма не передбачає прийом аргументів, то argc і argv можуть бути опущені.

Робота з аргументами командного рядка зводиться до перегляду параметрів argc і argv. Наведемо приклад програми, яка виводить ім'я програми, кількість переданих параметрів і значення цих параметрів.

```
#include  <stdio.h>
int main(int argc, char*  argv[])
{
  printf("The name of this program is '%s'.\n", argv[0]);
  printf("This program was invoked with %d arguments.\n",
         argc-1);
  /*  Чи є хоч один аргумент?    */
  if (argc  >  1)
  {
    int   i ;
    printf ("The arguments are:\n"); . •
    for (i = 1; i < argc; ++i)
      printf ("  %s\n", argv[i]);
  }
  return  0;
```

2.2.2 Змінні оточення

ОС UNIX надає кожній виконуваній програмі середовище виконання або, іншими словами, оточення програми. Під середовищем мається на увазі сукупність пар змінна-значення. Ці змінні називаються змінними оточення. Імена змінних ото-

чення і їх значення є рядками. За існуючою угодою змінні оточення записуються прописними буквами.

Наведемо деякі змінні оточення:

USER — містить ім'я поточного користувача;

HOME — містить шлях до початкового каталогу поточного користувача;

SHELL — містить шлях до командного інтерпретатору, яким користується поточний користувач;

PS1 — підказка першого рівня, говорить користувачу, що він може вводити нову команду;

PS2 — підказка другого рівня, говорить користувачу, що він введення команди не закінчене;

PATH — містить розділений двокрапками список каталогів, які операційна система переглядає в пошуку викликаної програми.

Стандарт POSIX.1, щоб дістати доступ до списку змінних оточення, в оголошенні функції main визначає ще один аргумент envp:

```
int main(int argc, char *argv[], char *envp[]);
```

Стандарт ANSI C визначає лише два перші аргументи. Тому рекомендується здійснювати доступ до змінних оточення через глобальну змінну environ:

```
extern char **environ;
```

Наведемо приклад програми, що виводить всі змінні оточення програми.

```
#include  <stdio.h>
extern char **environ;

int main(int argc, char*  argv[])
{
  char** var;
  for (var = environ; *var != NULL; ++var)
       printf ("%s\n", *var);
  return 0;
}
```

Для здобуття і установки конкретних значень змінних оточення використовуються функції getenv() і putenv(). Для установки і скидання значень змінних оточення призначені функції setenv() і unsetenv() відповідно. Зазвичай при запуску програма отримує копію середовища своєї батьківської програми (інтерпретатора команд, якщо вона була запущена користувачем). Таким чином, програми, запущені з командного рядка, можуть досліджувати середовище інтерпретатора команд.

2.2.3 Коди завершення програми

Коли програма завершує роботу, вона повідомляє операційну систему про свій стан, посилаючи їй код завершення, який є 16-разрядним цілим числом. За існуючою угодою нульовий код свідчить про успішне завершення, а ненульовий вказує на наявність помилки. Деякі програми повертають різні ненульові коди, позначаючи різні ситуації.

У більшості інтерпретаторів команд код завершення останньої виконаної програми міститься в спеціальній змінній $?.

Програма, написана на мові C або C++, вказує код повернення в операторі return у функції main або виконує виклик функції *exit*(). Процес може бути завершений і по незалежним від нього обставинам, наприклад, унаслідок отримання сигналу. В цьому випадку функція *exit*() буде викликана ядром від імені процесу.

Наявність коду повернення дозволяє програмам взаємодіяти між собою.

2.2.4 Правила формування і засоби розбору командних рядків

Розглянемо правила формування командних рядків.

У загальному випадку командний рядок складається з:

- імені службової програми (утиліти);
- опцій;
- аргументів опцій;
- операндів команди.

Розробники службових програм повинні керуватися наступними правилами:

- Ім'я утиліти складається не менше ніж з двох і не більше ніж з дев'яти малих латинських букв або цифр.
- Ім'я опції – це один буквено-цифровий символ. Опціям передує знак мінус. Після одного мінуса можуть розташовуватися декілька опцій без аргументів.
- Опції відокремлені від своїх аргументів.
- У опцій немає необов'язкових аргументів.
- Якщо у опції декілька аргументів, вони представляються одним словом і відокремлюються один від одного комами або екранованими пропусками.
- Всі опції розташовуються в командному рядку перед операндами.
- Спеціальний елемент командного рядка --, який не є ні опцією, ні операндом, позначає кінець опцій. Всі подальші слова трактуються як операнди, навіть якщо вони починаються із знаку мінус.
- Порядок різних опцій в командному рядку не має значення. Якщо повторюється одна опція з аргументами, останні повинні інтерпретуватися в порядку, вказаному в командному рядку.
- Порядок інтерпретації операндів може залежати від утиліти.
- Якщо операнд задає файл для читання чи запису, то знак мінус на його місці використовується тільки для позначення стандартного введення (або стандартного виведення виводу, якщо з контексту ясно, що специфікується вихідний файл).

2.2.5 Питання для самоперевірки

1. Яким чином в командному рядку задаються аргументи програми?
2. Яким чином функція main() дістає доступ до списку аргументів?
3. Що таке змінна оточення? В якому форматі вони зберігаються?
4. Яким чином програма дістає доступ до змінних оточення?
5. Як програма повертає код завершення програми?

2.3 Компілятори GCC

2.3.1 Робота з компілятором

GCC - це вільно доступний компілятор для мов C, C ++, Ada 95, Java, Objective-C, Fortran і Chill. Його версії існують для різних реалізацій ОС UNIX (а також VMS, OS/2 та інших систем), і дозволяють генерувати код для безлічі процесорів.

GCC можна використовувати для компіляції програм в об'єктні модулі і для компонування отриманих модулів в єдину програму. Компілятор здатний аналізувати імена файлів, що передаються йому в якості аргументів, і визначати, які дії необхідно виконати. Файли з іменами типу name.c розглядаються, як файли на мові C, а файли виду name.o вважаються об'єктним (тобто внутрішнім машинним) представленням.

Щоб відкомпілювати програму, що знаходиться у файлі F.cc, і створити об'єктний файл F.o, досить виконати команду:

```
gcc -c <compile-options> F.cc
```

Тут рядок compile-options вказує можливі додаткові опції компіляції.

Щоб скомпонувати один або декілька об'єктних файлів - F1.o, F2.o, ... - в єдиний виконуваний файл F, можна використати команду:

```
gcc -o F <link-options> F1.o F2.o . -lg++ <other-
libraries>
```

Тут рядок link-options означає можливі додаткові опції компонування, а рядок other-libraries - підключення при компонуванні додаткових бібліотек, що розділяються.

Можна поєднати два етапи обробки - компіляцію і компонування - в один спільний етап за допомогою команди:

```
gcc -o F <compile-and-link-options> F1.cc ... -lg++
<other-libraries>
```

Після компонування буде створений виконуваний файл F, який можна запустити за допомогою команди

```
./F <arguments>,
```

де рядок arguments визначає аргументи командного рядка при запуску програми.

В процесі компонування дуже часто доводиться використовувати бібліотеки. Бібліотекою називають набір об'єктних файлів, згрупованих в єдиний файл і проіндексованих. Коли команда компонування виявляє деяку бібліотеку в списку об'єктних файлів для компонування, вона перевіряє, чи містять вже скомпоновані об'єктні файли виклики для функцій, визначених в одному з файлів бібліотек. Якщо такі функції знайдені, відповідні виклики зв'язуються з кодом об'єктного файлу з бібліотеки.

Бібліотеки зазвичай визначаються через аргументи вигляду -llibrary-name. Зокрема -lg++ означає бібліотеку стандартних функцій C++, а -lm визначає бібліотеку різних математичних функцій (sin, cos, arctan, sqrt, і так далі). Бібліотеки мають бути перераховані після файлів, що містять виклики до відповідних функцій.

Серед безлічі опцій компіляції і компонування найчастіше вживаються наступні:

```
-c
```

Лише компіляція. З вихідних файлів програми створюються об'єктні файли у вигляді name.o. Компонування не проводиться.

```
-Dname=value
```

Визначити ім'я name в компільованій програмі, як значення value. Ефект такий же, як наявність рядка #define name value на початку програми. Частина `=value' може бути опущена, в цьому випадку значення за умовчанням дорівнює 1.

```
-o file-name
```

Використовувати file-name як ім'я для створюємого gcc.

`-llibrary-name`

Використовувати при компонуванні вказану бібліотеку.

`-g`

Помістити в об'єктний або виконуваний файл інформацію для відладника gdb. Опція має бути вказана і для компіляції, і для компонування.

`-pg`

Помістити в об'єктний або виконуваний файл інструкції профілізації для генерації інформації, що буде використана утилітою gprof. Опція має бути вказана і для компіляції, і для компонування. Профілізація - це процес вимірювання тривалості виконання окремих ділянок програми. Якщо вказана ця опція, програма при запуску генерує файл статистики. Програма gprof на основі цього файлу створює розшифровку, що показує час, витрачений на виконання кожної функції.

`-Wall`

Виведення повідомлень про всі попередження або помилки, що виникають під час трансляції програми.

2.3.2 Питання для самоперевірки

1. Які мови підтримує компілятор GCC?
2. Яким чином треба написати команду компіляції, щоб отримати об’єктний файл програми?
3. Як скомпонувати декілька об’єктних файлів?
4. Як об’єднати етапи компіляції і компонування програми?
5. Що таке бібліотека?
6. Як підключити до компонування задану бібліотеку?

2.4 Введення-виведення низького рівня

У операційній системі UNIX все введення і виведення здійснюється за допомогою читання файлів або їх запису, тому що всі периферійні пристрої, включаючи навіть термінал користувача, є файлами певної файлової системи. Це означає, що один однорідний інтерфейс управляє всіма зв'язками між програмою і периферійними пристроями.

У найбільш загальному випадку перед читанням із файлу або записом у файл необхідно повідомити систему про свій намір; цей процес називається відкриттям файлу. Система з'ясовує, чи маєте ви право поступати таким чином, і якщо все гаразд, повертає в програму невелике позитивне ціле число, зване дескриптором файлу. Всякий раз, коли цей файл використовується для введення або виведення, для ідентифікації файлу вживається дескриптор файлу, а не його ім'я. Вся інформація про відкритий файл міститься в системі; програма користувача звертається до файлу тільки через дескриптор файлу.

Для зручності виконання звичайних операцій введення і виведення за допомогою терміналу користувача існують спеціальні угоди. Коли інтерпретатор команд проганяє програму, він відкриває три файли, звані стандартним введенням, стандартним виведенням і стандартним виведенням помилок, які мають відповідно числа 0, 1 і 2 як дескриптори цих файлів. У нормальному стані всі вони пов'язані з терміналом, так що якщо програма читає з дескриптором файлу 0 і пише з дескрипторами файлів 1 і 2, то вона може здійснювати введення і вивід за допомогою терміналу, не піклуючись про відкриття відповідних файлів. У текстах програм, замість номерів набагато зручніше використовувати константи `STDIN_FILENO`, `STDOUT_FILENO` і `STDERR_FILENO`.

Будь-який із стандартних дескрипторів може відповідати звичайному файлу, каналу, іменованому каналу FIFO, пристрою і навіть сокету. Це відмінний спосіб забезпечити незалежність від типу пристрою введення і виведення, проте це можливо далеко не завжди. Наприклад, екранний редактор не зможе працювати, якщо пристрій стандартного виведення не є термінальним пристроєм.

2.4.1 Основні поняття

Файловий дескриптор (file descriptor)

Це ціле знакове число. Кожному відкритому файлу операційна система призначає файловий дескриптор. Файлові дескриптори, відповідні реально відкритим файлам, більше або дорівнюють нулю. Кожен відкритий файл має свій унікальний дескриптор. Поняття дескриптора прив'язане до процесу. Дочірній процес успадковує файлові дескриптори свого батьківського процесу.

Прапори відкриття файлу

Кожен файл в програмі відкривається з якоюсь метою (для читання, для запису і т. д.). Використання прапорів відкриття файлу дозволяє явно прописати свої наміри під час відкриття файлу, щоб потім захистити програміста, що працює над програмою, від нецільового використання файлового дескриптора. Прапори відкриття файлів визначають локальні права доступу до файлу в межах процесів, що працюють з дескриптором. Для формування набору прапорів відкриття файлів використовується звичайний цілочисельний тип даних.

Режим доступу до файлу

На відміну від прапорів відкриття, режими доступу визначають глобальні права доступу до файлу в межах файлової системи (читання, запис, виконання, власники, користувачі, групи і інш.). Для установки режиму запису файлів використовується тип mode_t.

Позиція у файлі

Використання позицій дозволяє діставати довільний доступ до даних у файлі. Для роботи з позиціями використовується тип off_t.

2.4.2 Системні виклики. read і write

Найнижчий рівень введення-виведення в системі UNIX не передбачає ні якої-небудь буферизації, ні якого-небудь іншого сервісу; він по суті є безпосереднім входом в операційну систему. Все введення і виведення здійснюється двома функціями: *read* і *write*. Першим аргументом обох функцій є дескриптор файлу. Другим аргументом є буфер у вашій програмі, звідки або куди повинні поступати дані. Третій аргумент – це число тих байтів, що підлягають пересилці. Звернення до цих функцій мають вигляд:

```
n_read=read(fd,buf,n);
n_written=write(fd,buf,n);
```

При кожному зверненні повертається лічильник байтів, що вказує фактичне число переданих байтів. При читанні повернене число байтів може виявитися менше, ніж запитане число. Повернене нульове число байтів означає кінець файлу, а "-1" вказує на наявність якої-небудь помилки. При запису повернене значення дорівнює числу фактично записаних байтів; неспівпадіння цього числа з числом байтів, яке передбачалося записати, зазвичай свідчить про помилку.

Кількість байтів, що підлягають читанню або запису, може бути довільним. Двома найпоширенішими величинами є "1", яка означає передачу одного символу за звернення (тобто без використання буфера), і "512", яка відповідає фізичному розміру блоку на багатьох периферійних пристроях. Цей останній розмір буде найбільш ефективним, але навіть введення або виведення по одному символу за звернення не буде надзвичайно дорогим.

Для приклада наведемо просту програму для копіювання стандартного файлу введення у стандартний файл виведення. В системі UNIX ця програма зможе копіювати що завгодно куди завгодно, тому що є можливість перенаправити як стандартне введення, так і стандартне виведення до будь-якого файлу чи пристрою.

```
#define bufsize 512 /*best size for pdp-11 unix*/
main() /*copy input to output*/
{
char buf[bufsize];
int n;
while((n=read(0,buf,bufsize))>0)
write(1,buf,n);
}
```

Якщо розмір файлу не буде кратний *bufsize*, то при передостанньому зверненні до *read* буде повернено менше число байтів, які потім записуються за допомогою *write*; при наступному після цього зверненні до read буде повернений нуль.

2.4.3 Відкриття, створення, закриття і знищення файлів

Крім випадку, коли за умовчанням визначені стандартні файли введення, виведення і помилок, потрібно явно відкривати файли, щоб потім читати з них або писати в них. Для цієї мети існують два системних виклики: open і creat.

```
#include  <sys/stat.h>
#include  <fcntl.h>
int open(
  const char *path, /* ім'я файлу */
  int flags, /* прапори */
  mode_t perms /*права доступу (для нового файлу)*/
 );
```

Аргумент path є символьним рядком, відповідним зовнішньому імені файлу. Проте аргумент, що визначає режим доступу, відмінний: rwmode рівне:

- `O_RDONLY` - для читання,

- O_WRONLY - для запису,
- O_RDWR - для читання і запису.

Якщо відбувається якась помилка, функція open повертає "-1"; інакше вона повертає дійсний дескриптор файлу.

Спроба відкрити файл, який не існує, є помилкою. Проте виклик open створить його, за умови, що аргумент *flags* містить прапор O_CREAT. Прапор O_CREAT зазвичай доповнюється прапором O_WRONLY або O_RDWR. Крім того, в такому разі необхідно визначити права доступу до файлу, наприклад:

```
fd = open("/home/student/newfile", O_RDWR | O_CREAT,
PERM_FILE)
```

Інколи виникає необхідність видаляти вміст файлу при відкритті. Робиться це за допомогою передачі прапора O_TRUNC:

```
fd = open("/home/student/newfile",
          O_WRONLY | O_CREAT | O_TRUNC, PERM_FILE)
```

Комбінація прапорів O_WRONLY | O_CREAT | O_TRUNC настільки звична («створити файл або очистити його і відкрити на запис»), що для цієї мети існує спеціальний системний виклик *creat*.

```
#include  <sys/stat.h>
#include <fcntl.h>
int creat(
   const char *path, /* повне ім'я файла */
   mode_t perms /* права доступу */
);
```

Звернення

```
fd=creat(name,pmode);
```

повертає дескриптор файлу, якщо виявилося можливим створити файл з ім'ям *name*, і "-1" інакше. Якщо файл з таким ім'ям вже існує, функція *creat* скоротить його до нульової довжини; створення файлу, який вже існує, не є помилкою.

Існує обмеження на кількість файлів, які програма може мати відкритими одночасно. Відповідно до цього будь-яка програма, що збирається працювати з багатьма файлами, має бути підготовлена до повторного використання дескрипторів файлів. Процедура *close* перериває зв'язок між дескриптором файлу і відкритим файлом і звільняє дескриптор файлу для використання з деяким іншим файлом. Завершення виконання програми за допомогою функції *exit*() або в результаті повернення із провідної програми приводить до закриття всіх відкритих файлів.

Функція знищення *unlink*(*filename*) видаляє з системи файлів файл з ім'ям *filename.*

2.4.4 Довільний доступ - lseek

Зазвичай при роботі з файлами введення і виведення здійснюється послідовно: при кожному зверненні до функцій *read* і *write* читання або запис починаються з позиції, безпосередньо наступної за попередньою обробленою. Але при необхідності файл може читатися або записуватися в будь-якому довільному порядку. Звернення до системи за допомогою функції *lseek* дозволяє пересуватися по файлу, не проводячи фактичного читання або запису. В результаті звернення

```
lseek(fd,offset,origin);
```

поточна позиція у файлі з дескриптором fd пересувається на позицію offset (зміщення), яка відлічується від місця, що вказується аргументом origin (початок відліку). Наступне читання або запис тепер починатимуться з цієї позиції. Аргумент offset має тип off_t; fd і origin мають тип int. Аргумент origin може набувати значень SEEK_SET, SEEK_CUR, SEEK_END, вказуючи на те, що ве-

личина offset повинна відлічуватися відповідно від початку файлу, від поточної позиції або від кінця файлу. Наприклад, щоб доповнити файл, слід перед записом знайти його кінець:

```
lseek(fd,0L,2);
```

щоб повернутися до початку ("перемотати назад"), можна написати:

```
lseek(fd,0L,0);
```

Функція lseek дозволяє поводитися з файлами приблизно так само, як з великими масивами, правда ціною повільнішого доступу. наступна проста функція, наприклад, прочитує будь-яку кількість байтів, починаючи з довільного місця у файлі.

```
get(fd,pos,buf,n) /*read n bytes from position pos*/
int fd, n;
long pos;
char *buf;
{
  lseek(fd,pos,0); /*get to pos*/
  return(read(fd,buf,n));
}
```

2.4.5 Питання для самоперевірки

1. Що таке відкриття файлу? Для чого воно використовується?
2. Поясніть поняття дескриптору файла.
3. Перечисліть стандартні потоки даних? Які вони мають дескриптори?
4. Що таке прапори відкриття файлу?
5. Що таке режим доступу до файлу? Який тип даних використовується для зберігання режиму доступу?

6. Які права доступу до файлу використовуються с ОС UNIX?
7. Що повертають системні виклики читання і запису?
8. Як визначається кінець файлу?
9. Для чого використовується функція close?
10. Яким чином моделюється довільний доступ до даних?

2.5 Доступ до каталогів

2.5.1 Функції для роботи з каталогами

В ОС UNIX каталоги є файлами спеціального формату. Блоки даних каталогів містять множину пар — ім’я файла, що міститься у каталозі, и числове значення його індексного дескриптора.

Робота з каталогами, по суті, нічим не відрізняється від роботи з будь-яким іншим файлом. Перед початком роботи з ним його слід відкрити зверненням до стандартної функції *opendir*(), що створює в програмі дескриптор каталогу, що використовується як посилання на відкритий каталог при виконанні необхідних операцій:

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

Функція *opendir* служить для відкриття потоку інформації для каталогу з ім'ям *name*. Тип даних DIR є деякою структурою даних, що описує такий потік. Функція *opendir* готує грунт для роботи інших функцій, що виконують операції над каталогом, і позиціонує потік на першому записі каталогу. При успішному завершенні функція повертає покажчик на відкритий потік каталогу, який надалі передаватиметься як параметр всім іншим функціям, що працюють з ним. При невдалому завершенні повертається значення NULL.

Для читання файлів, що містяться в каталозі, використовується функція *readdir*:

```
#include <dirent.h>
struct dirent *readdir(DIR *dir);
```

Параметр dir представляє вказівник на структуру, яка описує потік каталогу, що повернений функцією *opendir*.

Тип даних *struct dirent* є деякою структурою даних, що описує один запис в каталозі. Поля цього запису варіюються від однієї файлової системи до іншої, але одне з полів, яке власне нас і цікавитиме, завжди присутнє. Це поле *d_name*[ ] невизначеної довжини, що не перевищує значення NAME_MAX+1, яке містить символьне ім'я файлу. Дані, що повертаються функцією *readdir*, перепису-ються при черговому виклику цієї функції для того ж самого потоку каталогу.

При успішному завершенні функція повертає покажчик на структуру, що містить черговий запис каталогу. При невдалому завершенні або досягши кінця директорії повертається значення NULL.

Відкритий в програмі каталог закривається функцією closedir( ) з єдиним параметром — дескриптором каталогу:

```
#include <dirent.h>
int closedir(DIR *dir);
```

При успішному завершенні функція повертає значення 0, при невдалому завершенні – значення -1.

Інколи після читання частини вмісту каталогу виникає необхідність знову повернутися до його початку. Функцією rewinddir( ) поточна позиція в каталозі встановлюється на початок, що дозволяє здійснювати повторне читання файлів каталогу, не закриваючи його. Єдиним параметром цієї функції є дескриптор відкритого каталогу.

```
#include <dirent.h>
void rewinddir(DIR *dir);
```

2.5.2 Питання для самоперевірки

1. Що таке каталог? Які дані зберігаються в запису каталогу?
2. Яка функція використовується для відкриття каталогу?
3. За допомогою якої структури можна отримати запис каталогу?
4. Як повернутися до початку каталогу?

2.6 Управління процесами

Основним ресурсом комп'ютера є його процесор (або процесори). У кожен момент часу один процесор може виконувати лише один процес. Організація планування процесів так, щоб за рахунок їх перемикання створювалася ілюзія одночасної роботи декількох процесів - одне з основних завдань будь-якої розрахованої на багато користувачів і багатозадачної операційної системи.

У ОС UNIX основним засобом організації і одиницею багатозадачності є процес - програма, що ідентифікується унікальним чином, яка потребує отримання доступу до ресурсів комп'ютера. Операційна система маніпулює образом процесу, який є програмним кодом, а також розділами даних процесу, що визначають середовище виконання.

2.6.1 Атрибути процесу

Процес в UNIX має низку атрибутів, що дозволяють операційній системі управляти його роботою.

Розглянемо найважніші атрибути:

Ідентифікатор процесу (PID)

Кожен процес має унікальний ідентифікатор PID, що дозволяє ядру системи розрізняти процеси. Коли створюється новий процес, ядро присвоює йому наступний вільний (тобто не асоційований ні з яким процесом) ідентифікатор. Присвоєння ідентифікатора зазвичай відбувається по зростанню, тобто ідентифікатор нового процесу більший, ніж ідентифікатор процесу, створеного перед ним. Якщо іденти-

фікатор досягає максимального значення, наступний процес отримає мінімальний вільний PID і цикл повторюється. Коли процес завершує роботу, ядро звільняє ідентифікатор, що використовувався ним.

Ідентифікатор батьківського процесу (PPID)

Ідентифікатор процесу, що породив даний процес. Всі процеси в системі, крім системних процесів і процесу init, що є прародителем всіх інших процесів, породжені одним з існуючих або існуючих раніше процесів.

Поправка пріоритету (NI)

Відносний пріоритет процесу, що враховується планувальником при визначенні черговості запуску. Фактичний же розподіл процесорних ресурсів визначається пріоритетом виконання (атрибут PRI), залежним від декількох факторів, зокрема від заданого відносного пріоритету. Відносний пріоритет не змінюється системою на всьому протязі життя процесу (хоча може бути змінений користувачем або адміністратором) на відміну від пріоритету виконання, динамічно змінного планувальником.

Термінальна лінія (TTY)

Термінал або псевдотермінал, пов'язаний з процесом. З цим терміналом за умовчанням пов'язані стандартні потоки: вхідний, вихідний і потік повідомлень про помилки. Потоки (програмні канали) є стандартним засобом взаємодії між процесами в ОС UNIX.

Реальний (UID) і ефективний (EUID) ідентифікатори користувача

Реальним ідентифікатором користувача даного процесу є ідентифікатор користувача, що запустив процес. Ефективний ідентифікатор служить для визначення прав доступу процесу до системних ресурсів (в першу чергу до ресурсів файлової системи). Зазвичай реальний і ефективний ідентифікатори збігаються, тобто процес має в системі ті ж права, що і користувач, що запустив його. Проте існує можливість задати процесу ширші права, ніж права користувача, шляхом установки біта SUID, коли ефективному ідентифікатору присвоюється значення ідентифікатора власника виконуваного файлу (наприклад, користувача root).

Реальний (GID) і ефективний (EGID) ідентифікатори групи

Реальний ідентифікатор групи дорівнює ідентифікатору основної або поточної групи користувача, що запустив процес. Ефективний ідентифікатор служить для визначення прав доступу до системних ресурсів від імені групи. Зазвичай ефективний ідентифікатор групи збігається з реальним. Але якщо для виконуваного файлу встановлений біт SGID, такий файл виконується з| ефективним ідентифікатором групи-власника.

2.6.2 Створення процесу

Кожному процесу при створенні присвоюється свій ідентифікатор процесу (ID), який є унікальним серед всіх ідентифікаторів процесів. Час життя процесу закінчується в мить, коли повідомлення про припинення роботи дочірнього процесу передається батьківському процесу. У цей момент всі ресурси, що асоціюються з процесом, звільняються.

Зазвичай процеси створюються за допомогою функції *fork*. Дочірній процес, створений за допомогою fork, є копією батьківського процесу за тим винятком, що має власний ідентифікатор.

Після створення дочірнього процесу обидва процеси продовжують виконуватися в нормальному режимі. Якщо існує необхідність дочекатися закінчення виконання дочірнього процесу, а лише за цим продовжити виконувати потік батьківського процесу, то для цього можна використовувати функції *wait* і *waitpid*.

Знов створений дочірній процес виконує ту ж саму програму, що і батьківський процес, починаючи з тієї точки, де повернула управління функція *fork*.

Для ілюстрації сказаного давайте розглянемо наступну програму:

```
/* приклад створення нового процесу з однаковою
   роботою процесів нащадка і батька */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
```

```
int main()
{
   pid_t pid, ppid;
   int a = 0;
   (void)fork();
   /*При успішному створенні нового процесу з цього мі-
сця
     починають працювати два процеси: старий  і новий.
     Перед виконанням наступного оператору
     значення змінної a в обох процесах дорівнює 0*/
   a = a+1;
   /*Дізнаємося ідентифікатори поточного і
     батьківського процесів (у кожному з процесів!!!)*/
    pid = getpid();
    ppid = getppid();
   /*Друкуємо значення PID, PPID і обчислене
     значення змінної a (у кожному з процесів!!!) */
    printf("My pid = %d, my ppid = %d,
      result = %d\n", (int)pid, (int)ppid, a);
    return 0;
}
```

Значення, що повернув системний виклик *fork*, може використовуватися для того, щоб визначити, виконується програма в батьківському або дочірньому процесі.

Загальна схема організації різної роботи процесу-дитини і процесу-батька виглядає так:

```
pid = fork();
if(pid == -1){
    ...
    /* помилка */
```

```
    ...
} else if (pid == 0){
    ...
    /* дитина */
    ...
} else {
    ...
    /* батько */
    ...
}
```

Деякі атрибути дочірнього процесу відрізняються від атрибутів батьківського процесу:

- Дочірній процес має власний ідентифікатор;
- Дочірній процес одержує власні копії відкритих файлових дескрипторів батьківського процесу. Проте зміна атрибутів дескрипторів дочірнього процесу ніяк не впливає на батьківський і навпаки;
- Процесорний час дочірнього процесу у момент створення встановлюється рівним нулю;
- Дочірній процес не успадковує захвати файлів батьківського процесу;
- Дочірній процес не успадковує аварійних сигналів, установленнях батьківським процесом;

Виконання однієї і тієї ж програми декількома процесами рідко є корисним. Проте, дочірній процес може виконувати іншу програму, якщо він створений функцією exec.

2.6.3 Виконання файлів

Тут описується застосування сімейства функцій exec, призначених для виконання файлу як зображення процесу. Ці функції можуть використовуватися для того, що використовувати в процесі яку-небудь програму вже після того, як він був створений.

Всі функції відрізняються використанням параметрів, проте насправді вони виконують одне і те ж. Прототипи функцій визначені в заголовному файлі <unistd.h>.

```
int execv (const char *FILENAME, char *const ARGV[])
```

Функція виконує файл, заданий рядком FILENAME як нове зображення процесу. Аргумент ARGV є масивом рядків, що закінчуються нульовим покажчиком, які використовуються як аргумент argv функції main.

Останнім елементом цього масиву має бути нульовий покажчик. Перший елемент масиву – ім'я файлу програми. Середовище нового зображення процесу береться із змінної environ поточного зображення процесу.

```
int execl (const char *FILENAME, const char *ARG0,
...)
```

Ідентична execv, однак всі аргументи програми передаються у функцію індивідуально замість задавання масиву.

```
int execve (const char *FILENAME, char *const ARGV[],
char *const ENV[])
```

Ідентична execv за винятком того, що процесу передаються змінні середовища в аргументі ENV.

```
int execle (const char *FILENAME, const char *ARG0,
char *const ENV[], ...)
```

Ідентична попередній функції за винятком того, що аргументи програми передаються індивідуально.

```
int execvp (const char *FILENAME, char *const ARGV[])
```

Ідентична execv з тією різницею, що намагається знайти команду з таким ім'ям в каталогах, вказаних в змінній середовища PATH, якщо FILENAME не містить жодного символу слеша.

```
int execlp (const char *FILENAME, const char *ARG0, ...)
```

Ідентична execl за винятком того, що виконує пошук команди аналогічний попередній функції.

2.6.4 Завершення процесу

Існує два способи коректного завершення процесу в програмах, написаних на мові С. Перший спосіб ми використовували до цих пір: процес коректно завершувався після досягнення кінця функції *main*() або при виконанні оператора return у функції main(); другий спосіб застосовується при необхідності завершити процес в якому-небудь іншому місці програми. Для цього використовується функція *exit*() із стандартної бібліотеки функцій для мови С. При виконанні цієї функції відбувається скидання всіх частково заповнених буферів введення-виведення із закриттям відповідних потоків, після чого ініціюється системний виклик припинення роботи процесу і переведення його в стан завершення виконання.

Повернення із функції в поточний процес не відбувається і функція нічого не повертає.

Значення параметра функції *exi*t() – коди завершення процесу – передається ядру операційної системи і може бути потім отримане процесом-породжувачем процесу, що завершився. Насправді досягши кінця функції *main*() також неявно викликається ця функція зі значенням параметра 0.

```
#include <stdlib.h>
void exit(int status);
```

Значення параметра status – коди завершення процесу – передається ядру операційної системи і може бути потім отримане процесом-породжувачем процес, що завершився. При цьому використовуються тільки молодші 8 біт пара-

метра, так що для коди завершення допустимі значення від 0 до 255. За угодою, код завершення 0 означає безпомилкове завершення процесу.

Якщо процес завершує свою роботу раніше, ніж його батько, і батько явно не вказав, що він не хоче одержувати інформацію про статус завершення породженого процесу, то процес, що завершився, не зникає з системи остаточно, а залишається в системі до завершення процесу-батька, або до того моменту, коли батько отримає цю інформацію. Такі процеси в операційній системі UNIX прийнято називати процесами-зомбі (zombie, defunct).

2.6.5 Питання для самоперевірки

1. Що таке процес? Які атрибути процесса є найважнішими?
2. Які атрибути дочірнього процесу відрізняються від атрибутів батьківського процесу?
3. В чому різниця між реальним та ефективним ідентифікаторами користувача?
4. Як процес може довідатися, є він дочірнім чи батьківським?
5. Яким чином процес може змінити програму, що виконується?
6. Яким чином процес завершує свою роботу?
7. Які процеси прийнято називати процесами-зомбі?

2.7 Сигнали

Сигнали в UNIX, Unix-подібних і інших POSIX- сумісних операційних системах є одним із способів взаємодії між процесами. Фактично, сигнал — це асинхронне повідомлення процесу про яку-небудь подію.

Головна відзнака сигналів від інших засобів взаємодії між процесами полягає в тому, що їх обробка програмою зазвичай відбувається відразу ж після надходження сигналу (або не відбувається взагалі), незалежно від того, що програма робить в даний момент. Сигнал перериває нормальний порядок виконання команд в програмі і передає управління спеціальної функції – обробникові сигналу. Якщо обробка сигналу не приводить до завершення процесу, то після виходу із функції-обробника

виконання процесу поновлюється з тієї точки, в якій воно було перерване. У програм також є можливість припинити обробку сигналів, що надходять, тимчасово, на період виконання якої-небудь важливої операції. У традиційній термінології призупинення отримання певних сигналів називається блокуванням. Якщо для сигналу, що поступив, було встановлено блокування, сигнал буде переданий програмі, як тільки вона розблоковує даний тип сигналів. Цим блокування відрізняється від ігнорування сигналу, при якому сигнали відповідного типу ніколи не передаються програмі. Слід пам'ятати, що не всі сигнали можуть бути проігноровані. Наприклад, при отриманні програмою сигналу примусового завершення SIGKILL система нічого не повідомляє програму, а просто припиняє її роботу. Таким чином, перевага сигналів перед іншими засобами взаємодії з програмою полягає в тому, що посилати програмі сигнали можна у будь-який момент її роботи, не чекаючи настання якихось особливих умов. Джерелом сигналів може бути як сама операційна система, так і інші програми користувача.

Сигнал може бути відправлений процесу або ядром, або іншим процесом за допомогою системного виклику kill.

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t, int sig);
```

Сигнал можна надіслати будь-якому спорідненому процесу, у тому числі і собі. Процес, що володіє привілеями суперкористувача, може надіслати сигнал будь-якому процесу.

2.7.1 Типи сигналів

У стандарті SUS2002 визначено 28 різних сигналів, але більшість реалізацій доповнюють цей список своїми сигналами. Крім того, існують додаткові сигнали, які є частиною розширень реального часу POSIX. Для зручності розділимо всі сигнали SUS на групи. У наступному списку символи в дужках описують дію за умовчанням, прийняту для сигналу (розшифровка значень символів наводиться нижче). Для сигналів, що мають виключно штучне походження,

цей факт відмічений особливо. Будь-який природний сигнал може бути породжений штучно.

1. **Виявлені помилки**

SIGBUS — спроба доступу до невизначеної частини пам'яті (A)

SIGFPE — помилка арифметичної операції (A)

SIGILL — некоректна команда (A)

SIGPIPE — запис в канал, з якого ніхто не читає (T)

SIGSEGV — неприпустиме звернення до сегменту пам'яті (A)

SIGSYS — некоректне звернення до системного виклику (A)

SIGXCPU — вичерпаний ліміт процесорного часу (A)

SIGXFSZ — перевищене обмеження на розмір файлу (A)

2. **Сгенеровані користувачем або застосуванням**

SIGABRT — звернення до системного виклику abort (A)

SIGHUP — виявлений обрив зв'язку з терміналом або завершення термінального процесу (T)

SIGINT — переривання роботи (з клавіатури) (T)

SIGKILL — "ліквідувати", може мати тільки штучне походження (T)

SIGQUIT — завершити (з клавіатури) (A)

SIGTERM — завершити, може мати тільки штучне походження (T)

SIGUSR1 — сигнал користувача № 1, має тільки штучне походження (T)

SIGUSR2 — сигнал користувача №2, має тільки штучне походження (T)

3. **Управління завданнями**

SIGCHLD — дочірній процес завершив або припинив роботу (I)

SIGCONT — продовжити виконання (з клавіатури) (C)

SIGSTOP — припинити роботу, може мати тільки штучне походження (C)

SIGTSTP — сигнал з терміналу, припинити роботу (з клавіатури) (S)

SIGTTIN — спроба читання з фонового процесу (S)

SIGTTOU — спроба запису з фонового процесу (S)

4. Події таймеру

SIGALRM — витік час таймеру (T)

SIGVTALRM — витік час віртуального таймеру (T)

SIGPROF — витік час профілюючого таймеру (T)

5. Інші події

SIGPOLL — сталася очікувана подія (T)

SIGTRAP — пастка трасувальника — точка останову (A)

SIGURG — доступні позачергові дані в сокеті (I)

Символи в дужках мають наступні значення:

- I — сигнал ігнорується (від англ. «Ignore»)
- T — приводить до завершення процесу (від англійського «Terminate»)
- A — те ж саме, що і T, але при цьому виконуються додаткові дії, визначувані системою, наприклад, створення файлу з дампом пам'яті процесу (від англійського «Abort»)
- S — зупинка (від англійського «Stop»)
- C — продовження після зупинки (від англ. «Continue»)

Сигнали виявлення помилок, що мають природне походження, є результатом помилок в програмі. Для сигналів SIGBUS, SIGSEGV, SIGFPE і SIGILL точна причина помилки стандартами не обмовляється, але, як правило, всі вони виявляються на рівні апаратури. Крім того, ці чотири сигнали підкоряються визначеним правилам, в разі їх природного походження:

- якщо для цих сигналів, за допомогою sigaction, встановлена дія SIG_IGN, то реакція застосування на них — не визначена (залежить від системи);
- реакція застосування, в разі нормального повернення з функції-обробника сигналу — не визначена (залежить від системи);
- результат блокування сигналу не визначений.

2.7.2 Обробка сигналів

При отриманні сигналу процес може виконати одну з трьох дій:

- виконати дію за умовчанням. Звичайна дія за умовчанням полягає в припиненні виконання процесу. Для деяких сигналів дія за умовчанням полягає в ігноруванні сигналу. Для інших сигналів дія за умовчанням полягає в зупинці процесу;
- ігнорувати сигнал і продовжувати виконання. У великих програмах несподівано виникаючі сигнали можуть привести до проблем;
- виконати визначену користувачем дію.

Одним із способів зміни поведінки процесу при отриманні сигналу в операційній системі UNIX є використання системного виклику signal().

```
#include <signal.h>
void (*signal (int sig,
      void (*handler) (int)))(int);
```

Цей системний виклик має два параметри: один з них задає номер сигналу, реакцію процесу на який потрібно змінити, а другою визначає, як саме ми збираємося її змінювати. Другий параметр може мати три види значень. Перше – SIG_DFL – константа, що повідомляє, що потрібно відновити обробку сигналу за умовчанням. Друге – SIG_IGN – константа, що означає, що потрібно ігнорувати даний сигнал. Третє – адреса функції, що приймає аргумент типу int. Ця функція виконуватиметься при отриманні сигналу signo, а саме значення signo буде передано як аргумент функції, що викликається. Управління буде передано функції, як тільки процес отримає сигнал, яка б ділянка програми при цьому не виконувалася. Після повернення із функції управління буде знову передано процесу і продовжиться з точки, в якій виконання процесу було перерване.

Використання функції signal має на увазі семантику застарілих або ненадійних сигналів. Процес при цьому має дуже слабкі можливості управління сигналами. По-перше, процес не може заблокувати сигнал, тобто відкласти отри-

мання сигналу на період виконання критичної ділянки коду. По-друге, кожного разу при отриманні сигналу, його диспозиція встановлюється на дію за умовчанням. Дана функція і відповідна нею семантика збережені для підтримки старих версій застосувань.

Стандарт POSIX. 1 визначив новий набір функцій управління сигналами, заснований на інтерфейсі 4.2BSD UNIX і позбавлений розглянутих вище недоліків.

Модель сигналів, запропонована POSIX, заснована на понятті набору сигналів (signal set), що описується змінною типу sigset_t. Кожен біт цієї змінної відповідає за один сигнал. Набори сигналів є одним з основних параметрів, які передаються системним викликам, що працюють з сигналами. Вони задають список сигналів, які необхідно передати системному виклику.

Вибрати певні сигнали можна, почавши або з повного набору сигналів і видаливши непотрібні сигнали, або з пустого набору, включивши в нього потрібні. Ініціалізація пустого і повного набору сигналів виконується за допомогою процедур sigemptyset і sigfillset, відповідно. Після ініціалізації з наборами сигналів можна оперувати за допомогою процедур sigaddset і sigdelset, що відповідно додають і видаляють вказані сигнали.

Після визначення списку сигналів можна задати певний метод обробки сигналу за допомогою процедури sigaction:

```
#include <signal.h>
int sigaction (
   int signo,
   const struct sigaction *act,
   sigaction *oact
);
```

Перший параметр signo задає окремий сигнал, для якого потрібно визначити дію. Щоб ця дія виконувалася, процедура sigaction має бути викликана до

отримання сигналу типу signo. Значення змінної signo може бути будь-яке з раніше визначених імен сигналів, за винятком SIGSTOP і SIGKILL, які призначені лише для зупинки або завершення процесу і не можуть оброблятися по-іншому.

Другий параметр, act, визначає обробника сигналу signo. Третій параметр, oact, якщо не рівний NULL, вказує на структуру, куди буде поміщено опис старого методу обробки сигналу. Розглянемо структуру sigaction, визначену у файлі <signal.h>:

```
struct sigaction {
  void (*sa_handler) (int); /*Функція обробника*/
  sigset_t sa_mask, /*Сигнали, які блокуються
                      під час обробки сигналу*/
  int sa_flags; /*Прапори, що впливають на поведінку
                сигналу*/
  void (*sa_sigaction) (int, siginfo_t *, void *);
 /*Покажчик на обробник сигналів*/
};
```

Перше поле, sa_handler, задає обробник сигналу signo. Це поле може мати три види значень. Ця функція виконуватиметься при отриманні сигналу signo, а саме значення signo буде передано як аргумент функції, що викликається. Управління буде передано функції, як тільки процес отримає сигнал, яка б ділянка програми при цьому не виконувалася. Після повернення із функції управління буде знову передано процесу і продовжиться з місця, в якому виконання процесу було перерване.

Друге поле, sa_mask, демонструє перше практичне використання набору сигналів. Сигнали, задані в цьому полі, блокуватимуться під час виконання функції, заданої полем sa_handler. Це не означає, що ці сигнали ігноруватимуться,

просто їх обробка буде відкладена до завершення функції. При вході у функцію перехоплений сигнал також буде неявно доданий до поточної маски сигналів.

Поле sa_flags може використовуватися для зміни характеру реакції на сигнал signo.

В більшості випадків, якщо процесу посилається сигнал під час виконання ним системного виклику, то обробка сигналу відкладається до завершення виклику. Але деякі системні виклики поводяться по-іншому, і їх виконання можна перервати за допомогою сигналу. Це відноситься до викликів введення/виведення (read, write, open, і так далі), викликів wait або pause. У всіх випадках, якщо процес перехоплює виклик, то перерваний системний виклик повертає значення -1 і поміщає в змінну errno значення EINTR.

Процедура sigaction дозволяє автоматично повторювати перерваний таким чином системний виклик. Це досягається установкою значення SA_RESTART в полі sa_flags структури struct sigaction. Якщо встановлений цей прапор, то системний виклик буде виконаний знову, і значення змінної errno не буде встановлено.

Сигнали UNIX зазвичай не можуть накопичуватися. У будь-який момент часу тільки один сигнал кожного типу може чекати обробки даним процесом, хоча декілька сигналів різних типів можуть чекати обробки одночасно. Фактично те, що сигнали не можуть накопичуватися, означає, що вони не можуть використовуватися як повністю надійний метод взаємодії між процесами, оскільки процес не може бути впевнений, що посланий ним сигнал не буде «втрачений».

Для демонстрації роботи обробника сигналів наведемо невелику програму.

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

void term_handler(int i) {
  printf ("Terminating\n");
```

```
  exit(EXIT_SUCCESS);
}

int main(int argc, char ** argv) {
  struct sigaction sa;
  sigset_t newset;
  sigemptyset(&newset);
  sigaddset(&newset, SIGHUP);
  sigprocmask(SIG_BLOCK, &newset, 0);
  sa.sa_handler = term_handler;
  sigaction(SIGTERM, &sa, 0);
  printf("My pid is %i\n", getpid());
  printf("Waiting...\n");
  while(1)
    sleep(1);
  return EXIT_FAILURE;
}
```

Ця програма робить дві речі: обробляє сигнал SIGTERM (при отриманні цього сигналу програма виводить діагностичне повідомлення і завершує свою роботу) і блокує сигнал SIGHUP, так що цей сигнал не може завершити її роботу. У тексті програми ми насамперед визначаємо функцію-обробник сигналу SIGTERM term_handler(). Функції-обробники сигналів – це звичайні функції С, вони мають доступ до всіх глобально видимих змінних і функцій. Проте, оскільки невідомо, в який момент виконання програми буде викликано функцію-обробник, треба проявляти особливу обережність при зверненні до глобальних структур даних з цієї функції. Єдиним параметром нашого варіанту функції-обробника сигналу (у Unix-системах існує і інший варіант) є змінна типу int, в якій передається номер сигналу, що викликав обробник. В даному разі цей номер не потрібний, оскільки відомо, що тільки один сигнал – SIGTERM, може викликати нашу функцію, проте, в принципі,

ніщо не заважає використовувати одну функцію для обробки декількох різних сигналів. Функція-обробник не повертає ніякого значення, що цілком логічно, тому що вона викликається не нашою програмою, а деяким системним компонентом.

2.7.3 Питання для самоперевірки

1. Що таке сигнал? Чим сигнал відрізняється від інших способів взаємодії процесів?
2. Яким чином процес може відправити сигнал іншому процесу?
3. На які групи розподіляються сигнали?
4. Які сигнали мають виключно штучне походження?
5. Яким чином процес може відреагувати на надходження сигналу?
6. Чому використання функції signal вважається недоцільним?
7. Які переваги має реєстрація сигналів за допомогою функції sigaction по відношенню до функції signal?

3 ПРИКЛАДИ РОЗВ'ЯЗАННЯ ЗАВДАНЬ

Приклад 1. Використовуючи системний виклик stat, написати, що визначає тип файлу: звичайний файл, каталог, пристрій, FIFO-файл, сокет. Ім'я файлу задавати у вигляді аргументу командного рядка. Перевірити, чи не виникають помилки при системних викликах.

Розв'язання

Для того щоб розв'язати дане завдання потрібно використовувати системний виклик stat, оскільки серед типів файлів у завданні не вказане символічне посилання. Це єдиний системний виклик, що використовується в даній програмі, тому вона містить єдину перевірку.

Для визначення імені файлу напишемо функцію, яка приймає ім'я файлу як параметр, а повертає одиницю, якщо вдалося розпізнати та вивести тип файлу та нуль у протилежному випадку. Заголовок цієї функції виглядає так:

```
int typeOf( char *name).
```

Головна програма лише перевіряє правильність завдання аргументів при запуску та викликає функцію `typeOf`.

Текст програми приведено нижче.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/stat.h>

int typeOf(  char *name)
```

```
{
   int type;
   struct stat st;
   if(stat( name, &st ) < 0 )
   {
      fprintf(stderr,"%s не існує\n", name );
      return 0;
   }
   printf("Файл має %d імен\n", st.st_nlink);
   switch(type = (st.st_mode & S_IFMT))
   {
    case S_IFREG:
      printf("Звичайний файл розміром %ld байт\n",
            st.st_size );           break;
    case S_IFDIR:
      printf( "Каталог\n" );      break;
    case S_IFCHR:   /* байтоорієнтоване  */
    case S_IFBLK:   /* блочноорієнтоване */
     printf( "Пристрій\n" );   break;
    case S_IFIFO:
      printf( "FIFO-файл\n" );    break;
    default:
      printf( "Інший тип\n" );   break;
   }       return type;
}

/* Головна програма */
int main(int argc, char*  argv[])
{
   if (argc  !=  1)
```

```
    {
      printf ("Usage: %s <filename>\n", argv[0]);
      return 1;
    }
    return  typeOf(argv[1])-1;
}
```

Приклад 2. Написати програму, яка виводить имена файлів поточного каталогу.

Розв'язання

В цій програмі нам знадобяться дві змінних:

- дескриптор каталогу dp типу DIR;
- змінна ep типу struct dirent для зчитування наступного запису в каталозі.

Програма спочатку відкриває каталог за допомогою функції opendir, потім послідовно читає записи каталогу за допомогою функції readdir та виводить імена файлів. Потім каталог закривається.

```
#include <stddef.h>
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
int  main (void)
{
  DIR *dp;
  struct dirent *ep;
  dp = opendir ('.');
  if (dp != NULL)
  {
      while (ep = readdir (dp))
           puts (ep->d_name);
           (void) closedir (dp);
 }
  else
      puts ('Couldn't open the directory.');
  return 0;
}
```

Приклад 3. Напишіть програму, що при одержанні сигналу SIGUSR1 породжує новий процес, а при одержанні сигналу SIGUSR2 виводить повідомлення про кількість породжених процесів, і про те, скільки з них продовжують працювати. Породжений процес повинен бути припинений на випадкову кількість секунд, після чого завершений. У даному завданні потрібно виконати рішення двома способами:

- з використанням системного виклику signal;
- з використанням системного виклику sigaction.

Розв'язання

Рішення з використанням системного виклику signal.

```
#include <signal.h>
#include <stdio.h>
static int kProc=0,kActProc=0;
static void handler1(int signo)
//оброблювач сигналу SIGUSR1
{
signal(SIGUSR1, SIG_IGN);
if (fork()==0)
{
    sleep(random(10);
    exit(0);
}
else
{
   kProc++; kActProc++;
}
```

```
}
static void handler2(int signo)
// оброблювач сигналу SIGUSR2
{
signal(SIGUSR2, handler2);
printf("Породжено %d процесів, працює %d процесів\n",
        kProc, kActProc);
}

static void handler3(int signo)
// оброблювач сигналу SIGCHLD
{
signal(SIGCHLD, handler2);
kActProc-i;
wait(0);
}

int main()
{
   // реєстрація оброблювача сигналу SIGUSR1
   signal(SIGUSR1, handler1);
   // реєстрація оброблювача сигналу SIGUSR2
   signal(SIGUSR2, handler2);
   // реєстрація оброблювача сигналу SIGCHLD
    signal(SIGCHLD, handler3);

   //нескінченний цикл для демонстрації роботи
   while (1)
      pause(); //зупинка до одержання сигналу
```

```
}
```

У даному рішенні змінена диспозиція сигналів SIGUSR1 й SIGUSR2. Крім того змінена диспозиція сигналу SIGCHLD, що дозволяє відслідковувати завершення породжених процесів. Після установки диспозиція сигналів процес запускає нескінченний цикл, у процесі якого викликається функція pause.

Помітимо, що кожного разу при одержанні сигналів ми змушені відновлювати необхідну диспозицію, у противному випадку одержання сигналу цього типу викликало б завершення програми (дія за замовчуванням).

Рішення с використанням системного виклику sigaction.

```
#include <signal.h>
#include <stdio.h>
static int kProc=0,kActProc=0;
static void handler (int signo)
// оброблювач сигналу
{
   if (signo==SIGUSR1)
   {  // обробка сигналу SIGUSR1
      if (fork()==0)
{
    sleep(random(10);
    exit(0);
}
else
{
   kProc++; kActProc++;
}
```

```
      }
      else if(signo==SIGUSR2)
      {  // обробка сигналу SIGUSR2
         printf("Породжено %d процесів, працює %d
процесів\n",
           kProc, kActProc);

      }
      else
      if(signo==SIGCHLD)
      {
        kActProc-i;
   wait(0);

      }
   }

   int main()
   {
      struct sigaction set;
      set.sa_handler=handler;
      set.sa_mask=0;
      sigemptyset(&set);
      sigaddset(&set, SIGUSR1);
      sigaddset(&set, SIGUSR2);
      sigaddset(&set, SIGCHLD);

      // реєстрація оброблювача сигналу SIGUSR1
      sigaction(SIGUSR1, &set, NULL);
      // реєстрація оброблювача сигналу SIGUSR2
```

```
    sigaction(SIGUSR2, &set, NULL);
    // реєстрація оброблювача сигналу SIGCHLD
    sigaction(SIGCHLD, &set, NULL);

    // нескінченний цикл для демонстрації роботи
    while (1)
       pause();
}
```

У даному рішенні використовується тільки один оброблювач. Це зручно тому, що при виклику оброблювача диспозиція сигналу не змінюється.

У головній програмі спочатку структура set заповнюється інформацією, необхідної для керування сигналами. Для рішення нашого завдання всі поля, крім поля sa_handler, заповнюються нулями, а в це поле записується адреса оброблювача handler.

Після встановлення диспозиції сигналів процес запускає нескінченний цикл, у процесі якого викликається функція pause.

4 ВАРІАНТИ ЗАВДАНЬ

В процесі вивчення курсу студенти виконують контрольну роботу. Контрольна робота містить 5 завдань. В кожному завданні студент повинен обрати варіант згідно рекомендацій, які наведені нижче разом з завданнями. По кожному завданню студент складає звіт, який містить:

1. Розрахунок номеру варіанту
2. Текст завдання
3. Розв’язання задачі з коментарями
4. Висновки

4.1 Завдання 1

Номер варіанта дорівнює останній цифрі номера залікової книжки, збільшеної на 1.

1. Написати програму, що копіює байти з одного файлу в інший у зворотному порядку. Імена файлів задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах.

2. Написати програму, що додає вміст одного файлу в кінець іншого. Імена файлів задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах.

3. Написати програму, що додає вміст одного файлу в кінець іншого у зворотному порядку. Імена файлів задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах.

4. Написати програму, що підраховує кількість рядків у файлі, ім'я якого задається у вигляді аргументу командного рядка. Перевірити, чи не виникають помилки при системних викликах.

5. Написати програму, що дописує в кінець файлу його вміст у зворотному

порядку. Ім'я файлу задавати у вигляді аргументу командного рядка. Перевірити, чи не виникають помилки при системних викликах.

6. Написати програму порівняння двох файлів, що буде друкувати номер першого з рядків, що розрізняються, і позицію символу, у якому вони розрізняються. Імена файлів задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах.

7. Написати програму, що дописує вміст файлу в його кінець. Ім'я файлу задавати у вигляді аргументу командного рядка. Перевірити, чи не виникають помилки при системних викликах.

8. Написати програму, що копіює вміст одного файлу в інший, видаляючи всі входження даного символу. Імена файлів і символ, що видаляється, задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах.

9. Написати програму, що копіює вміст одного файлу в інший, заміняючи при цьому всі багаторазові входження пробілів знаком табуляції. Імена файлів задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах.

10. Написати програму, що змінює порядок проходження байтів у файлі на протилежний. Ім'я файлу задавати у вигляді аргументу командного рядка. Перевірити, чи не виникають помилки при системних викликах.

4.2 Завдання 2

Номер варіанта дорівнює залишку від ділення номера залікової книжки на 12, збільшеному на 1.

1. Написати програму, що створює каталог із заданим ім'ям. Ім'я каталогу задавати у вигляді аргументу командного рядка. Перевірити, чи не виникають помилки при системних викликах.

2. Написати програму, що видаляє каталог із заданим ім'ям. Ім'я каталогу задавати у вигляді аргументу командного рядка. Перевірити, чи не виникають помилки при системних викликах.

3. Написати програму, що створює кілька каталогів. Імена каталогів задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах.

4. Написати програму, що видаляє кілька каталогів. Імена каталогів задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах.

5. Написати програму для перейменування файлу. Старі й нові імена файлів задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах. Вказівка: використайте системні виклики link() і unlink().

6. Написати програму для створення символічного посилання. Імена цільового файлу й посилання задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах.

7. Написати програму, що змінює час останнього доступу до файлу на поточне. Ім'я файлу задавати у вигляді аргументу командного рядка. У випадку, якщо файл не існує, то його необхідно створити. Перевірити, чи не виникають помилки при системних викликах.

8. Написати програму для створення твердого посилання. Імена цільового файлу й посилання задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах.

9. Написати програму для перейменування файлу. Старі й нові імена файлу задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах.

10. Написати програму для виведення шляхового імені поточного каталогу.

11. Написати програму, що додає право читання файлу всім категоріям користувачів для декількох файлів. Імена файлів задавати у вигляді аргументів командного рядка. Перевірити, чи не виникають помилки при системних викликах.

12. Написати програму, що виводить дату й час останньої зміни файлу.

Ім'я файлу задавати у вигляді аргументу командного рядка. Перевірити, чи не виникають помилки при системних викликах.

4.3 Завдання 3

Номер варіанта дорівнює залишку від ділення номера залікової книжки на 15, збільшеному на 1.

1. Написати функцію, що визначає, чи існує заданий файл. Аргументом є шляхове ім'я файлу. Функція повертає значення 0, якщо файл існує, та значення -1 у протилежному випадку.
2. Написати функцію, що визначає, чи є заданий файл каталогом. Аргументом є шляхове ім'я файлу. Функція повертає значення 0, якщо є, та значення -1 у протилежному випадку.
3. Написати функцію, що повертає останній байт файлу. Аргументом функції є файловий дескриптор.
4. Написати функцію, що додає право на читання файлу всім категоріям користувачів. Аргументом є шляхове ім'я файлу. Функція повертає значення 0, якщо операція вдалася, та значення -1 у протилежному випадку
5. Написати функцію, що створює файл нульової довжини з наступними правами доступу: читання й запис для власника, читання для групи й інших користувачів. Аргументом функції є шляхове ім'я файлу. Функція повертає значення 0, якщо вдалося створити файл, та значення -1 у протилежному випадку.
6. Написати функцію, що визначає, чи є заданий файл символічним посиланням. Аргументом є шляхове ім'я файлу. Функція повертає значення 0, якщо є, та значення -1 у протилежному випадку.
7. Написати функцію, що визначає, чи є право чи читати заданий файл. Аргументом є шляхове ім'я файлу. Функція повертає значення 0, якщо таке право є, та значення -1 у протилежному випадку.
8. Написати функцію, що повертає довжину файлу. Аргументом функції є файловий дескриптор.

9. Написати функцію, що визначає, чи встановлений для цього файл біт SUID. Аргументом є шляхове ім'я файлу. Функція повертає значення 0, якщо це так, та значення -1 у протилежному випадку.

10. Написати функцію, що визначає чи досягнутий кінець файлу. Аргументом функції є файловий дескриптор. Функція повертає значення 0, якщо це так, та значення -1 у протилежному випадку.

11. Написати функцію, що повертає довжину файлу. Аргументом функції є шляхове ім'я файлу.

12. Написати функцію, що повертає вміст символічного посилання. Аргументом функції є шляхове ім'я посилання. Функція повертає порожній покажчик у випадку невдачі.

13. Написати функцію, що визначає власника файлу. Аргументом функції є шляхове ім'я файлу. Функція повертає ідентифікатор власника у випадку успіху та значення -1 у протилежному випадку.

14. Написати функцію, що визначає, чи є символічне посилання висячим. Аргументом функції є шляхове ім'я посилання. Функція повертає 1 у випадку, якщо це так, та значення 0 у протилежному випадку.

15. Написати функцію, що створює файл заданої довжини з наступними правами доступу: читання й запис для власника, читання для групи й інших користувачів. Аргументами функції є шляхове ім'я файлу і його довжина. Функція повертає значення 0, якщо вдалося створити файл, та значення -1 у протилежному випадку.

4.4 Завдання 4

Номер варіанта дорівнює залишку від ділення номера залікової книжки на 11, збільшеному на 1.

1. Написати програму, що видаляє всі висячі посилання в заданому каталозі й всіх його підкаталогах. Ім'я каталогу задавати у вигляді аргументу командного рядка.

2. Написати програму, що підраховує кількість звичайних файлів, що належать заданому користувачу у заданому каталозі та всіх його підкаталогів». Ім'я користувача та каталогу задавати у вигляді аргументів командного рядка.

3. Написати програму, що підраховує сумарний обсяг звичайних файлів у заданому каталозі й всіх його підкаталогах. Ім'я каталогу задавати у вигляді аргументу командного рядка.

4. Написати програму, що підраховує сумарну кількість файлів кожного типу в заданому каталозі й всіх його підкаталогах. Ім'я каталогу задавати у вигляді аргументу командного рядка.

5. Написати програму, що видаляє всі звичайні файли нульової довжини в заданому каталозі й всіх його підкаталогах. Ім'я каталогу задавати у вигляді аргументу командного рядка.

6. Написати програму, що виводить уміст заданого каталогу й всіх його підкаталогів. Ім'я каталогу задавати у вигляді аргументу командного рядка.

7. Написати програму, що видаляє із заданого каталогу й всіх його підкаталогів файли із суфіксом «.o». Ім'я каталогу задавати у вигляді аргументу командного рядка.

8. Написати програму, що видаляє заданий не порожній каталог. Ім'я каталогу задавати у вигляді аргументу командного рядка.

9. Написати програму, що підраховує сумарну кількість символічних посилань у заданому каталозі й всіх його підкаталогах. Ім'я каталогу задавати у вигляді аргументу командного рядка.

10. Написати програму, що видаляє всі звичайні файли, імена яких не починаються із крапки, у заданому каталозі й всіх його підкаталогах. Ім'я каталогу задавати у вигляді аргументу командного рядка.

11. Написати програму, що видаляє із заданого каталогу й всіх його підкаталогів файли з ім'ям «core». Ім'я каталогу задавати у вигляді аргументу командного рядка.

4.5 Завдання 5

У даному завданні потрібно виконати рішення двома способами:

1. з використанням системного виклику signal;
2. з використанням системного виклику sigaction.

Номер варіанта дорівнює залишку від ділення номера залікової книжки на 9, збільшеному на 1.

1. Напишіть програму, яка при одержанні сигналу SIGUSR1 породжує новий процес і виводить його ідентифікатор, а по завершенні породженого процесу виводить повідомлення про цю подію. Породжений процес повинен бути припинений на випадкову кількість секунд, після чого завершений.

2. Напишіть програму, яка при одержанні сигналу SIGUSR1 реєструє оброблювач сигналу SIGINT, що скасовує завершення програми по цьому сигналу, породжує новий процес, а при одержанні сигналу SIGUSR2 скасовує дія цього оброблювача.

3. Напишіть програму, яка після одержання сигналу SIGUSR1 ігнорує сигнали SIGUSR1 й SIGINT, а при одержанні сигналу SIGUSR2 відновлює обробку цих сигналів за замовчуванням.

4. Напишіть програму, яка при одержанні сигналу SIGUSR1 породжує новий процес, що повинен містити нескінченний цикл, а при одержанні сигналу SIGUSR2 припиняє роботу цього процесу. Забезпечте неможливість одночасної роботи двох породжених процесів.

5. Напишіть програму, яка при одержанні сигналу SIGUSR1 породжує два нових процеси, після чого одержання сигналу SIGUSR1 повинне приводити до закінчення одного з них, а одержання сигналу SIGUSR2 припиняти роботу другого й відновлювати обробку сигналу SIGUSR1. Породжені процеси повинні очікувати одержання сигналу.

6. Напишіть програму, яка після одержання сигналу SIGUSR1 ігнорує сигнали SIGUSR1 й SIGINT, а при одержанні сигналу SIGUSR2 відновлює об-

робку цих сигналів за замовчуванням і видає повідомлення про кількість отриманих за цей час сигналів SIGUSR1 й SIGINT.

7. Напишіть програму, яка при одержанні сигналу SIGUSR1 виводить повідомлення про кількість оброблених сигналів SIGINT, при одержанні сигналу SIGINT ігнорує сигнал SIGUSR1, а при одержанні сигналу SIGUSR2 ігнорує сигнал SIGINT і відновлює обробку сигналу SIGUSR1. Якщо сигнал SIGINT ігнорується, то він не вважається обробленим.

8. Напишіть програму, яка при одержанні сигналу SIGUSR1 породжує новий процес і передає йому сигнал SIGSLEEP, а при одержанні сигналу SIGUSR2 будить цей процес. Породжений процес повинен містити системний виклик PAUSE. Забезпечте неможливість одночасної роботи двох породжених процесів.

9. Напишіть програму, яка при одержанні сигналу SIGUSR1 породжує два нових процеси, після чого одержання сигналу SIGUSR1 повинне приводити до закінчення одного з них, а одержання сигналу SIGUSR2 припиняти роботу другого. Породжені процеси повинні містити нескінченний цикл. Процес, що породжує, після завершення роботи нащадків закінчує свою роботу.

5 ОРГАНІЗАЦІЯ КОНТРОЛЮ ЗНАНЬ ТА ВМІНЬ СТУДЕНТІВ

5.1 Система контролю знань та вмінь студентів

Контроль знань та вмінь студентів, що навчаються за заочною формою, здійснюється за допомогою системи контролюючих заходів. Вони складаються з заходів поточного та підсумкового контролю.

Поточний контроль здійснюється на протязі всього навчального року (семестру) та включає заходи самостійної роботи студента під час вивчення навчальної дисципліни поза межами університету та роботи студента на лабораторних заняттях у період заліково-екзаменаційної сесії.

Підсумковий контроль здійснюється під час заліково-екзаменаційної сесії та має на меті встановлення рівня знань та вмінь, які опанував студент після вивчення навчальної дисципліни. Форма підсумкового контролю – залік, іспит – встановлюється навчальним планом дисципліни.

При вивченні дисципліни „Системне програмування” використовується накопичувальна система оцінювання. Форма підсумкового контролю –іспит. Максимальна сума, що може одержати студент – 100 (30 – контрольна робота, 20- робота студента при проведенні лабораторних занять та 50 на іспиті). Головними рисами накопичувальної системи є:

- Підсумкова оцінка знань студента складається як арифметична сума оцінки. Яку накопив студент, виконуючи заходи поточного контролю, та оцінки, яку отримав студент на підсумковому контролі (іспит).

- Для отримання відмітки:

„задовільно” – студент повинен мати накопичену суму балів поточного контролю не менше 30 балів та отримати за екзаменаційну контрольну роботу не менше 30 балів (тобто загалом не менше 60% від максимально можливої оцінки);

„добре” - студент повинен мати накопичену суму балів поточного контролю не менше 37 балів та отримати за екзаменаційну контрольну роботу не менше 38 балів (тобто загалом не менше 75% від максимально можливої оцінки);

„відмінно” - студент повинен мати накопичену суму балів поточного контролю не менше 45 балів та отримати за екзаменаційну контрольну роботу не менше 45 балів (тобто загалом не менше 90% від максимально можливої оцінки);

Якщо студент має накопичену суму балів поточного контролю менше 35 балів, він не допускається до заходів підсумкового контролю.

5.2 Форми контролю знань та вмінь студентів

А) Оцінка самостійної роботи студента до екзаменаційно-залікової сесії здійснюється з оцінки виконання контрольної роботи. Максимальний бал, що може отримати студент за кожне завдання – 6 балів, всього за контрольну роботу – 30 балів.

Контрольна робота зараховується, якщо студент отримав сумарну оцінку не менше 18 (тобто не менше 60% від максимальної суми 30 балів). Студент, який отримав за виконання контрольної роботи сумарну оцінку меншу за 18 балів не допускається до підсумкового контролю.

Б) Оцінка роботи студента при проведенні лабораторних занять по дисципліні під час екзаменаційно-залікової сесії.

Загальна максимально оцінка за цей вид поточного контролю складає 20 балів. Ця сума складається з:

- оцінки відвідування студентом лабораторних занять (максимально 5 балів);
- оцінки знань студента під час усного опитування перед початком виконання лабораторних робіт (максимально 7 балів);
- оцінки знань та вмінь студента при складанні алгоритму програми (максимально 8 балів);

ЛІТЕРАТУРА

Основна

1. Рочкинд М. Программирование для UNIX. \Пер. с англ., 2-е изд. Перераб. и доп. – М. Издательско-торговый дом „Русская редакция”; СПб. БХВ – Санкт-Петербург, 2005 – 704 с.
2. Робачевский А.М. Операционная система UNIX. – СПб.: БХВ – Санкт-Петербург, 1999 - 528 с., ил.
3. К. Хэвиленд, Д. Грэй, Б. Салама. Системное программирование в UNIX. \Пер. с англ. – М., ДМК Пресс, 2000 - 364 с.
4. М. Митчелл, Д.Оулдем, А.Саммюэл. Программирование для LINUX. Профессиональный подход. \Пер. с англ. – М.: Издательский дом «Вильямс», 2002 - 288 стр., с ил

Додаткова

1. Т. Чан. Системное программирование на C++ для Unix. \Пер. с англ.– К.: BHV-Киев · 1999 - 592 с.
2. Моли Б. Unix/Linux: теория и практика программирования. \Пер. с англ.,– М.:КУДИЦ-ОБРАЗ, 2004 - 576 с.
3. Таненбаум Э. Современные операционные системы. \Пер. с англ., 2-е изд. – СПб.: Питер, 2007 - 1040 с.
4. В.Г.Олифер, Н.А.Олифер. Сетевые операционные системы. – СПб.: Питер, 2001 - 528 с.
5. Таненбаум Э. Архитектура компьютера. 5-е изд. \Пер. с англ.– СПб.: Питер, 2009 - 848 с.
6. Керниган В., Ритчи Д. Язык программирования Си. \Пер. с англ., 3-е изд., испр. - СПб.: "Невский Диалект", 2001. - 352 с.: ил.
7. Глас Г., Эйблс К. Unix для программистов и пользователей. \Пер. с англ.– СПб.: БХВ-Петербург, 2004, 848 с.

8. Белломо М. Unix: наглядный курс освоения операционной системы. - Киев: Диалектика, 2001. - 336 с.: ил.
9. Гриффитс А. GCC. Настольная книга пользователей, программистов и системных администраторов. - Киев:"ДиаСофт", 2004. - 624 с.
10. Белломо М. Unix: наглядный курс освоения операционной системы. - Киев: Диалектика, 2001. - 336 с.: ил.

ДОДАТОК А. СПИСОК ФУНКЦІЙ ПО КАТЕГОРІЯХ

Доступ до змінних оточення

getenv - повертає значення змінної оточення

```
#include <stdlib.h>
 char *getenv(
      const char *var //ім'я змінної
 );
  // повертає значення змінної
  //або NULL, якщо така не знайдена
```

putenv - змінює або додає змінні в середовище оточення

```
#include <stdlib.h>
 int putenv(
     const char *string //рядок у вигляді «ім'я=значення»
 );
  // повертає 0 у випадку успіху,
  // ненульове значення у випадку помилки
```

setenv - змінює або додає змінні в середовище оточення

```
#include <stdlib.h>
int setenv(
 const char *var, // ім’я змінної
 const char *val, //значення
 )
// повертає 0 у випадку успіху, -1 у випадку помилки
```

unsetenv - видаляє змінну середовища оточення

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

```
int unsetenv(
     const char *var //видаляє змінну
 )
     //повертає 0 у випадку успіху, -1 у випадку помилки
```

Файлові атрибути

access - визначає доступність файлу

```
#include <unistd.h>
 int access(
     const char *path,//шляхове ім'я файлу
     int what       //перевіряють права
 )
 //повертає 0 у випадку успіху, -1 у випадку помилки
```

chmod - змінює права доступу до файлу

```
#include <sys/stat.h>
 int chmod(
     const char *path,//шляхове ім'я файлу
     mode_t uid    //нові права доступу
 )
 //повертає 0 у випадку успіху, -1 у випадку помилки
```

chown - змінює власника й групу файлу по його імені

```
#include <unistd.h>
 int chown(
     const char *path,//шляхове ім'я файлу
     uid_t uid,         //новий ідентифікатор власника
     gid_t gid          // новий ідентифікатор групи
 )
 //повертає 0 у випадку успіху, -1 у випадку помилки
```

fchmod - змінює права доступу до файлу по дескрипторі

```
#include <sys/stat.h>
 int fchmod(
      int fd,            //дескриптор
      mode_t uid    //нові права доступу
 )
 //повертає 0 у випадку успіху, -1 у випадку помилки
```

fchown - змінює власника й групу файлу по дескрипторі

```
#include <unistd.h>
int fchown(
      int fd,       //дескриптор
      uid_t uid,    //новий ідентифікатор власника
      gid_t gid     // новий ідентифікатор групи
 )
 //повертає 0 у випадку успіху, -1 у випадку помилки
```

fstat - повертає відомості про стан файлу по дескрипторі

```
#include <sys/stat.h>
 int fstat(
      int fd,           //дескриптор
  struct stat *buf // метадані файлу
 );
//повертає 0 у випадку успіху або -1 у випадку помилки
```

lchown - змінює власника й групу символічного посилання по її імені

```
#include <unistd.h>
int lchown(
```

```
     const char *path,//шляхове ім'я файлу
     uid_t uid,    //новий ідентифікатор власника
     gid_t gid     // новий ідентифікатор групи
 )
//повертає 0 у випадку успіху, -1 у випадку помилки
```

lstat - повертає відомості про стан файлу по його імені без слідування символічним посиланням

```
#include <sys/stat.h>
 int lstat(
     const char *path, //шляхове ім'я файлу
   struct stat *buf    //запитувана інформація
 );
//повертає 0 у випадку успіху або -1 у випадку помилки
```

stat - повертає відомості про стан файлу по його імені

```
#include <sys/stat.h>
 int stat(
     const char *path, //шляхове ім'я файлу
struct stat *buf   //запитувана інформація
 );
//повертає 0 у випадку успіху або -1 у випадку помилки
```

Файлове введення-виведення

close - закриває дескриптор файлу

```
#include <unistd.h>
 int close(
```

```
     int fd;           //дескриптор
 )
//повертає 0 у випадку успіху, -1 у випадку помилки
```

creat - створює новий або очищає існуючий файл і відкриває його на запис

```
#include <sys/stat.h>
#include <fcntl.h>
 int creat(
     const char *path, //шляхове ім'я файлу
     mode_t perms      //права доступу
 );
//повертає дескриптор файлу або -1 у випадку помилки
```

dup - дублює дескриптор файлу

```
#include <unistd.h>
 int dup(
     int fd   //дескриптор
 )
//повертає новий дескриптор файлу, -1 у випадку помилки
```

dup2 - дублює дескриптор файлу

```
#include <unistd.h>
 int dup2(
     int fd1,      //дублируемый дескриптор
     int fd2,      //використовуваний дескриптор
 )
 //повертає 0 у випадку успіху, -1 у випадку помилки
```

fcntl - виконує керуючі операції над відкритим файлом

```
#include <unistd.h>
#include < fcntl.h>
 int fcntl(
      int fd,  //дескриптор
      int op,  //операція
      . . .         //необов'язкові аргументи,
                //залежать від виду операції
 )
//Повертає значення, що, залежить від вигляду операції,
  // у випадку помилки повертається значення -1
```

ftruncate - змінює розмір файлу, заданого файловим дескриптором

```
 #include <unistd.h>
 int truncate(
      int fd,           //дескриптор
      off_t length      //новій раз мер файлу
 );
//повертає 0 у випадку успіху, -1 у випадку помилки
```

lseek - установлює й повертає поточну позицію у файлі

```
#include <unistd.h>
 int lseek(
      int fd,           //дескриптор
      off_t pos,
      int whence
 );
//повертає нову позицію у файлі, -1 у випадку помилки
```

open - відкриває або створює файл

```
#include <sys/stat.h>
#include <fcntl.h>
 int open(
      const char *path,      //шляхове ім'я файлу
      int flags,             //прапори
      mode_t perms       //права доступу
 );
//повертає дескриптор файлу або -1 у випадку помилки
```

read - виконує читання з файлового дескриптора

```
#include <unistd.h>
 ssize_t read(
      int fd,            //дескриптор
      const void *buf,   //адреса буфера для даних
      size_t nbytes      //кількість байтів для читання
 );
   // повертає кількість прочитаних байтів,
   // -1 у випадку помилки
```

readv - читання зі злиттям

```
#include <sys/iov.h>
 ssize_t readv(
      int fd,                //дескриптор
      const struct iovec *iov,//масив буферів для даних
      size_t nbytes              //кількість буферів
 );
   // повертає кількість прочитаних байтів,
   // -1 у випадку помилки
```

truncate - змінює розмір файлу, заданого ім'ям

```
#include <unistd.h>
int truncate(
     const char *path,      //шляхове ім'я файлу
     off_t length      //новій раз мер файлу
);
//повертає 0 у випадку успіху, -1 у випадку помилки
```

write - виконує запис у файловий дескриптор

```
#include <unistd.h>
ssize_t write(
     int fd,           //дескриптор
     const void *buf,  //адреса буфера з даними
     size_t nbytes          //кількість байтів для запису
);
  // повертає кількість записаних байтів,
  // -1 у випадку помилки
```

writew - запис зі злиттям

```
#include <sys/iov.h>
ssize_t writev(
     int fd,                //дескриптор
     const struct iovec *iov,//масив буферів з даними
     size_t nbytes              //кількість буферів
);
  // повертає кількість записаних байтів,
  // -1 у випадку помилки
```

Каталоги

closedir - закриває каталог

```
#include <dirent.h>
int closedir(
     DIR *dirp // покажчик на DIR,
);
//повертає 0 у випадку успіху  -1 у випадку помилки
```

opendir - відкриває каталог

```
#include <dirent.h>
DIR *opendir(
     const char *path //ім'я каталогу
);
  // повертає покажчик на DIR у випадку успіху,
  // -1 у випадку помилки
```

readdir - читає запис із каталогу

```
#include <dirent.h>
struct dirent *readdir(
     DIR *dirp // покажчик на DIR
);
// повертає покажчик на структуру у випадку успіху,
// -1 у випадку помилки
```

rewinddir - перехід у початок каталогу

```
#include <dirent.h>
void rewinddir(
     DIR *dirp // покажчик на DIR
);
```

seekdir - перехід до необхідного місця розташування

```
#include <dirent.h>
void seekdir(
     DIR *dirp // покажчик на DIR
     Long loc // місце розташування
);
```

telldir - одержує поточну позицію в каталозі

```
#include <dirent.h>
long telldir(
     DIR *dirp // покажчик на DIR
);
// повертає поточну позицію в каталозі
```

Управління каталогами

link - створює тверде посилання

```
#include <unistd.h>
int link(
     const char *oldpath //старе шляхове ім'я файлу
     const char *newpath //нове ім'я файлу
);
//повертає 0 у випадку успіху, -1 у випадку помилки
```

mkdir - створює каталог

```
#include <sys/stat.h>
int mkdir(
     const char *path, //шляхове ім'я каталогу
mode_t perms       //права доступу
);
//повертає 0 у випадку успіху, -1 у випадку помилки
```

readlink - читає вміст файлу символічного посилання

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

```
ssize_t readlink(
     const char *path, //шляхове ім'я afqkf
     char *buf,    //адреса буфера для читання
     size_t bufsize // розмір буфера
)
//повертає 0 у випадку успіху, -1 у випадку помилки
```

rename - перейменовує файл

```
#include <stdio.h>
int link(
     const char *oldpath //старе шляхове ім'я файлу
     const char *newpath //нове ім'я файлу
);
//повертає 0 у випадку успіху, -1 у випадку помилки
```

rmdir - видаляє каталог

```
#include <unistd.h>
int rmdir(
     const char *path  //шляхове ім'я каталогу
);
//повертає 0 у випадку успіху, -1 у випадку помилки
```

unlink - видаляє запис у каталозі

```
#include <unistd.h>
int unlink(
     const char *path //шляхове ім'я файлу
);
//повертає 0 у випадку успіху, -1 у випадку помилки
```

Атрибути процесів

chdir — робить заданий каталог поточним

```
#include <unistd.h>
int chdir(
     const char *path // ім'я каталогу
);
//повертає 0 у випадку успіху, -1 у випадку помилки
```

chroot — змінює кореневий каталог

```
#include <unistd.h>
int chroot(
     const char *path // шлях до каталогу
);
//повертає 0 у випадку успіху, -1 у випадку помилки
```

fchdir — робить поточним каталог по дескриптору

```
#include <unistd.h>
int fchdir(
     int fd  // дескриптор файла
);
//повертає 0 у випадку успіху, -1 у випадку помилки
```

getcwd — повертає повне ім'я поточного каталога

```
#include <unistd.h>
char *getcwd(
     char *buf, //повертаємо повне ім'я поточного
каталога
     size_t bufsize //розмір буферу
     );
//повертає 0 у випадку успіху, -1 у випадку помилки
```

nice — змінює значення параметра nice

```
#include <unistd.h>
 int nice(
      int incr  // приріст
 );
 //повертає 0 у випадку успіху, -1 у випадку помилки
```

Управління процесами

_Exit — завершує процес без звернення до коду збирання сміття

```
#include <stdlib.h>
 void _Exit(
      int status  // код завершення
 );
   // в програму управління вже не повертається
```

_exit — завершує процес без звернення до коду збирання сміття

```
#include <unistd.h>
 void _exit(
      int status  // код завершення
 );
   // в програму управління вже не повертається
```

exit — завершує процес із зверненням до коду збирання сміття

```
#include <stdlib.h>
 void exit(
      int status  // код завершення
 );
   // в програму управління вже не повертається
```

abort — посилає сигнал SIGABRT

```
#include <stdlib.h>
 void abort(void);
  // управління не повертається
```

atexit — реєструє функцію, що викликається при завершенні роботи процесу

```
#include <stdlib.h>
int atexit(
void(*fcn)(void) //викликаєма функція
);
 //повертає 0 у випадку успіху та ненульове значення
//при  помилці
```

execl — запускає програму, вхідні аргументи передаються у вигляді списку

```
#include <unistd.h>
 int execl(
      const char *path, //повний шлях до програми
const char *arg0, //перший аргумент (arg[0]-ім'я
//файлу)
const char *arg1, //другий аргумент(якщо необхідно)
...,               //інші аргументи(якщо необхідні)
NULL          //пустий вказівник,завершуючий список
 );
  // повертає -1 у випадку помилки
```

execle — запускає програму, аргументи передаються у вигляді списку, також передається середовище оточення

```
#include <unistd.h>
 int execle(
      const char *path, //повний шлях до файлу з програмою
```

```
    const char *arg0, //перший аргумент (ім'я файлу)
    const char *arg1, //другий аргумент(якщо необхідно)
    ...,          //інші аргументи(якщо необхідні)
    NULL,  //пустий вказівник,завершуючий список
аргументів
    char *const env[]  //масив сформованого середовища
оточення
);
 // повертає -1 у випадку помилки
```

execlp — запускає програму, аргументи передаються у вигляді списку, пошук файлу ведеться з використанням змінної PATH

```
#include <unistd.h>
int execlp(
    const char *file, // ім'я файлу з програмою
    const char *arg0, //перший аргумент (ім'я файлу)
    const char *arg1, //другий аргумент(якщо необхідно)
    ...,  //інші аргументи(якщо необхідні)
    NULL  //пустий вказівник,завершуючий список
);
 // повертає -1 у випадку помилки
```

execv — запускає програму, аргументи передаються у вигляді масиву

```
#include <unistd.h>
int execv(
    const char *path, //повний шлях до файлу з програмою
    char *const argv[] //масив аргументів
);
 // повертає -1 у випадку помилки
```

execve — запускає програму, аргументи передаються у вигляді масиву, так само передається середовище оточення

```
#include <unistd.h>
 int execve(
      const char *path, //повний шлях до файлу з програмою
      char *const argv[] //масив аргументів
      char *const envv[] //массив сформованного середовища
 оточення
);
  // повертає -1 у випадку помилки
```

execvp — запускає програму, аргументи передаються у вигляді масиву, пошук файлу ведеться з використанням змінної PATH

```
#include <unistd.h>
 int execvp(
      const char *file, // ім'я файлу з програмою
      char *const argv[] //масив аргументів
 );
  // повертає -1 у випадку помилки
```

fork — створює новий процес

```
#include <unistd.h>
pid_t fork(void);
// повертає ідентифікатор дочірнього процесу або 0 у
випадку успіху
// та -1 у випадку помилки
```

system — запускає команду

```
#include <stdlib.h>
int system(
```

```
    const char *command //команда
);  //повертає код завершення програми або -1 у випадку
помилки
```

wait — очікує завершення дочірнього процесу

```
#include <sys/wait.h>
pid_t   wait(
      int *statusp  //вказівник на статус або NULL
);  // повертає ідентифікатор процесу або -1 у випадку
помилки
```

waitpid — очікує зміни стану дочірнього процесу

```
 #include <sys/wait.h>
pid_t   waitpid(
      pid_t pid,  //ідентифікатор процесу або групи
процесів
      int *statusp,  //вказівник на статус або NULL
      int options   //прапори
);
// у випадку успіху повертає ідентифікатор процесу або 0
та
// -1 у випадку помилки
```

Атрибути процесів

getegid — повертає діючий ідентифікатор групи

```
#include <unistd.h>
gid_t getegid(void);
// повертає ідентифікатор групи(коди помилок не
передбачені
```

geteuid — повертає діючий ідентифікатор користувача

```
#include <unistd.h>
uid_t geteuid(void);
// повертає ідентифікатор користувача(коди помилок не
передбачені
```

getgid — повертає реальний ідентифікатор групи

```
#include <unistd.h>
gid_t getgid(void);
// повертає ідентифікатор групи(коди помилок не
передбачені
```

getuid—возвращает реальний ідентифікатор користувача

```
#include <unistd.h>
uid_t getuid(void);
// повертає ідентифікатор користувача(коди помилок не
передбачені
```

setegid — встановлює діючий ідентифікатор групи, що діє

```
#include <unistd.h>
int setegid(
gid_t gid   // діючий ідентифікатор групи
);// повертає 0 у випадку успіху та -1 у випадку помилки
```

seteuid — встановлює діючий ідентифікатор користувача

```
#include <unistd.h>
int seteuid(
uid_t uid  // діючий ідентифікатор користувача
);// повертає 0 у випадку успіху та -1 у випадку помилки
```

setgid — встановлює реальний ідентифікатор групи

```
#include <unistd.h>
int setgid(
```

```
gid_t gid   // реальний та збережений ідентифікатор групи
);// повертає 0 у випадку успіху та -1 у випадку помилки
```

setuid — встановлює реальний ідентифікатор користувача

```
#include <unistd.h>
int setuid(
uid_t uid  // реальний та збережений ідентифікатор користувача
);// повертає 0 у випадку успіху та -1 у випадку помилки
```

umask — встановлює і повертає маску прав доступу для новостворюваних файлів

```
#include <sys/stat.h>
mode_t umask(
     mode_t cmask  // нова маска
);// повертає попереднє значення маски
```

Ресурси процесів

getpgid — отримує ідентифікатор групи процесів

```
#include <unistd.h>
pid_t getpgid(
     pid_t pid  // ідентифікатор процесу або 0
);
//повертає ідентифікатор групи процесів або -1 у випадку помилки
```

getpid — повертає ідентифікатор процесу

```
#include <unistd.h>
pid_t getpid(void);
// повертає ідентифікатор процесу
```

getppid — повертає ідентифікатор батьківського процесу

```
#include <unistd.h>
pid_t getppid(void);
// повертає ідентифікатор батьківського процесу
```

getsid — отримує ідентифікатор сеансу

```
#include <unistd.h>
pid_t getsid(
pid_t sid  // ідентифікатор процесу або
// 0(відповідає викликаючому процесу)
);
// повертає ідентифікатор сеансу або -1 у випадку помилки
```

setpgid — задає або створює групу процесів

```
#include <unistd.h>
int setpgid(
     pid_t pid, // ID процесу або 0
     pid_t pgid // ідентифікатор групи процесів
); // повертає 0 при успіху  або -1 у випадку помилки
```

setsid — створює сеанс і групу процесів

```
#include <unistd.h>
pid_t setsid(void);
// повертає ідентифікатор групи процесів або -1 у
випадку помилки
```

Сигнали

kill — посилає сигнал процесу

```
#include <signal.h>
int kill(
```

```
        pid_t pid, //ідентифікатор процесу ID або групи
процесів
        int signum //сигнал
    ); // повертає 0 у випадку успіху або -1 у випадку
помилки
```

killpg — посилає сигнал групі процесів

```
    #include <signal.h>
    int killpg(
        pid_t pgrp, //ідентифікатор групи процесів
        int signum //сигнал
    ); // повертає 0 у випадку успіху або -1 у випадку
помилки
```

raise — посилає сигнал тому ж потоку, з якого був викликаний

```
    #include <signal.h>
    int raise(
        int signum //сигнал
    );// повертає 0 у випадку успіху
    // або ненульове значення у випадку помилки
```

sigaction — встановлює реакцію застосування на сигнал

```
    #include <signal.h>
    int sigaction(
        int signum //номер сигналу
        const struct sigaction *act //нова дія
        struct sigaction *oact //стара дія
    );
    // повертає 0 у випадку успіху або -1 у випадку помилки
```

sighold — блокує сигнал

```
 #include <signal.h>
int sighold(
      int signum //номер сигналу
); // повертає 0 у випадку успіху
//або ненульове значення у випадку помилки
```

sigismember — перевіряє наявність сигналу в наборі

```
#include <signal.h>
int sigismember(
      const sigset_t *set //набір сигналів
      int signum //номер сигналу
);
// повертає 1 якщо сигнал додано до набору, 0 – якщо не
// додано до набору, -1 - у випадку помилки
```

signal — встановлює реакцію застосування на сигнал

```
#include <signal.h>
void (*signal(
      int signum //номер сигналу
      void(*act)(int)  //дія
))(int);
// повертає стару дію або SIG_ERR у випадку помилки
```

sigpause — змінює маску і чекає доставки сигналу

```
#include <signal.h>
int sigpause(
      int signum //номер сигналу
); // завжди повертає -1
```

sigpending — повертає набір сигналів, чекаючих обробки

```
#include <signal.h>
int sigpending(
      sigset_t *set //повертаємий набір сигналів
); // повертає 0 у випадку успіху або ненульове значення
у
// випадку помилки
```

sigset — встановлює реакцію додатку на сигнал

```
#include <signal.h>
void (*sigset(
      int signum //номер сигналу
      void(*act)(int)  //дія
))(int);
// повертає стару дію або SIG_ERR у випадку помилки
```

sigsuspend — змінює маску сигналів і чекає доставки сигналу

```
#include <signal.h>
int sigsuspend(
      const sigset_t *sigmask  //тимчасова маска сигналів
);
// завжди повертає -1 як признак помилки
```

Маски сигналів

sigaddset — додає сигнал до набору

```
#include <signal.h>
int sigaddset(
      sigset_t *set, //набір сигналів
      int signum //номер сигналу

);// повертає 0 у випадку успіху або -1 у випадку помилки
```

sigdelset — видаляє сигнал з набору

```
#include <signal.h>
int sigdelset(
      sigset_t *set, //набір сигналів
      int signum //номер сигналу

);// повертає 0 у випадку успіху або -1 у випадку помилки
```

sigemptyset — інізіалізує пустий набір сигналів

```
#include <signal.h>
int sigemptyset(
      sigset_t *set //набір сигналів
 );// повертає 0 у випадку успіху або -1 у випадку
помилки
```

sigfillset — інізіалізує повний набір сигналов

```
#include <signal.h>
int sigfillset(
      sigset_t *set //набір сигналів
 );// повертає 0 у випадку успіху або -1 у випадку
помилки
```

sigprocmask — змінює маску сигналів для потоку (тільки якщо процес має єдиний потік)

```
#include <signal.h>
int sigprocmask(
      int how, //визначає порядок установки нової маски
const sigset_t *set //нова маска
      sigset_t *oset  // стара маска
 );// повертає 0 у випадку успіху або -1 у випадку
помилки
```

ДОДАТОК Б. КОДИ ПОМИЛОК СИСТЕМНИХ ВИКЛИКІВ

Код помилки і повідомлення	Опис
EEXIST File exists	Ім'я існуючого файлу використане в недопустимому контексті
EFAULT Bad address	Апаратна помилка при спробі використання системою аргументу функції
EFBIG File too large	Розмір файлу перевищив встановлене обмеження RLIMIT_FSIZE, або максимально допустимий розмір для даної файлової системи
EINPROGRESS Operation now in progress	Спроба тривалої операції для об'єкту, що не блокується
EINTR Interrupted system call	Здобуття асинхронного сигналу під час обробки системного виклику. Якщо виконання процесу буде продовжено після обробки сигналу, перерваний системний виклик завершиться з цією помилкою
EINVAL Invalid argument	Передача невірного аргументу системному виклику
EIO I/O error	Помилка введення-виведення фізичного пристрою
EISDIR Is a directory	Спроба операції, недопустимої для каталогу
ELOOP Number of symbolic links encountered during path name traversal exceeds MAXSYMLINKS	При спробі трансляції імені файлу було виявлено недопустимо велике число символічних посилань, що перевищує значення MAXSYMLINKS
EMFILE Too many open files	Число відкритих файлів для процесу перевищило максимальне значення OPEN_MAX
ENAMETOOLONG File name is too long	Довжина повного імені файлу перевищила максимальне значення PATH_MAX
ENFILE File table overflow	Переповнювання файлової таблиці
ENODEV No such device	Спроба недопустимої операції для пристрою
ENOENT No such file or directory	Файл з вказаним ім'ям не існує, або відсутній каталог, вказаний в повному імені файлу
ENOEXEC Exec format error	Спроба запуску на виконання файлу, який має право на виконання, але не є файлом допустимого виконуємого формату
ENOMEM No enough space	При спробі запуску программі або розміщення пам'яті розмір запрошуваної пам'яті перевищив максимально припустимий в системі
ENOMSG No message of desired type	Спроба отримання повідомлення визначеного типу, якого немає у черзі

Код помилки і повідомлення	Опис
ENOSPC No space left on device	Спроба запису у файл або створення нового каталогу при відсутності вільного місця на пристрої (в файловій системі)
ENOSR Out of stream resources	Відсутність черг чи головних модулів при спробі відкриття пристрою типу STREAMS. Цей стан є тимчасовим. Після звільнення відповідних ресурсів іншими процесам операція може пройти успішно
ENOSTR Not a stream device	Спроба вжити операцію, яка визначена для пристроїв типу STREAMS для пристроя іншого типу
ENOTDIR Not a directory	В операції, що очікує в якості аргументу ім’я каталогу, було вказане ім’я файлу іншого типу
ENOTTY Inappropriate ioctl for device	Спроба системного виклику ioctl для пристрою, який не є символьним
EPERM Not owner	Спроба модифікації файла способом, що дозволений тільки володарю і суперкористувачу і заборонений іншим користувачам. Спроба операції, що дозволена тільки суперкористувачу.
EPIPE Broken pipe	Спроба запису в канал, для якого немає процесу, що приймає дані. В цій ситуації процесу зазвичай посилайться сигнал. Помилка повертається лише при ігноруванні сигналу.
EROFS Read-only file system	Спроба модифікації файлу чи каталогу для пристрою, що змонтований тільки для читання
ESRCH No such process	Процесс з вказаним PID не існує

Навчальне видання

Трубіна Наталія Федорівна

Методичні вказівки до самостійної роботи студентів та виконання контрольної роботи з дисципліни «Системне програмування »

Видано в авторській редакції

Підп. до друку 20.12.2010. Формат 60х84/8.
Гарн. Таймс. Тираж 40 прим.

Редакційно-видавничий Центр
Одеського національного університету
імені І.І. Мечникова,
65082, м. Одеса, вул. Єлісаветинська, 12, Україна
Тел.: (048) 723 28 39