МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

Конспект лекцій з дисципліни «Програмування в UNIX»

для студентів 5 курсу денної форми навчання
напрям підготовки – комп'ютерні науки

Затверджено на засіданні
кафедри Інформатики
Протокол №___від_____
Зав. кафедри
_____

Одеса, 2009

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

Конспект лекцій з дисципліни «Програмування в UNIX»

для студентів 5 курсу денної форми навчання
напрям підготовки – комп'ютерні науки

Одеса, 2009

Конспект лекцій з дисципліни «Програмування в UNIX»
для студентів 5 курсу денної форми навчання.
Напрям підготовки – комп’ютерні науки

Укладач:
ТРУБІНА Н.Ф. старший викладач кафедри інформатики,

ЗМІСТ

1 ОСНОВНІ ПОНЯТТЯ

1.1 Середовище програмування UNIX. Системні виклики та бібліотечні функції

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 крапками входу в ядро ОС, хоча в ході виконання багато з них (але не всі) виконують системні виклики. Ці функції зберігаються в системній бібліотеці С і поряд із системними викликами становлять середовище програмування ОС UNIX (документовані в розділі 3 електронного довідника).

1.2 Створення и виконання програми в середовищі UNIX

Процедура створення більшості програмних застосувань приведена на рис.1.1.

Першою фазою є фаза компіляції, коли файли з текстами програм, обробляються компілятором. Параметри компіляції задаються або за допомогою файла в форматі утілити *make*, або безпосереднім завданням необхідних опцій компілятора у командному рядку. В результаті компілятор створює набір проміжних об’єктних файлів. Традиційно імена проміжних об’єктних файлів мають суфікс „*.o*”.

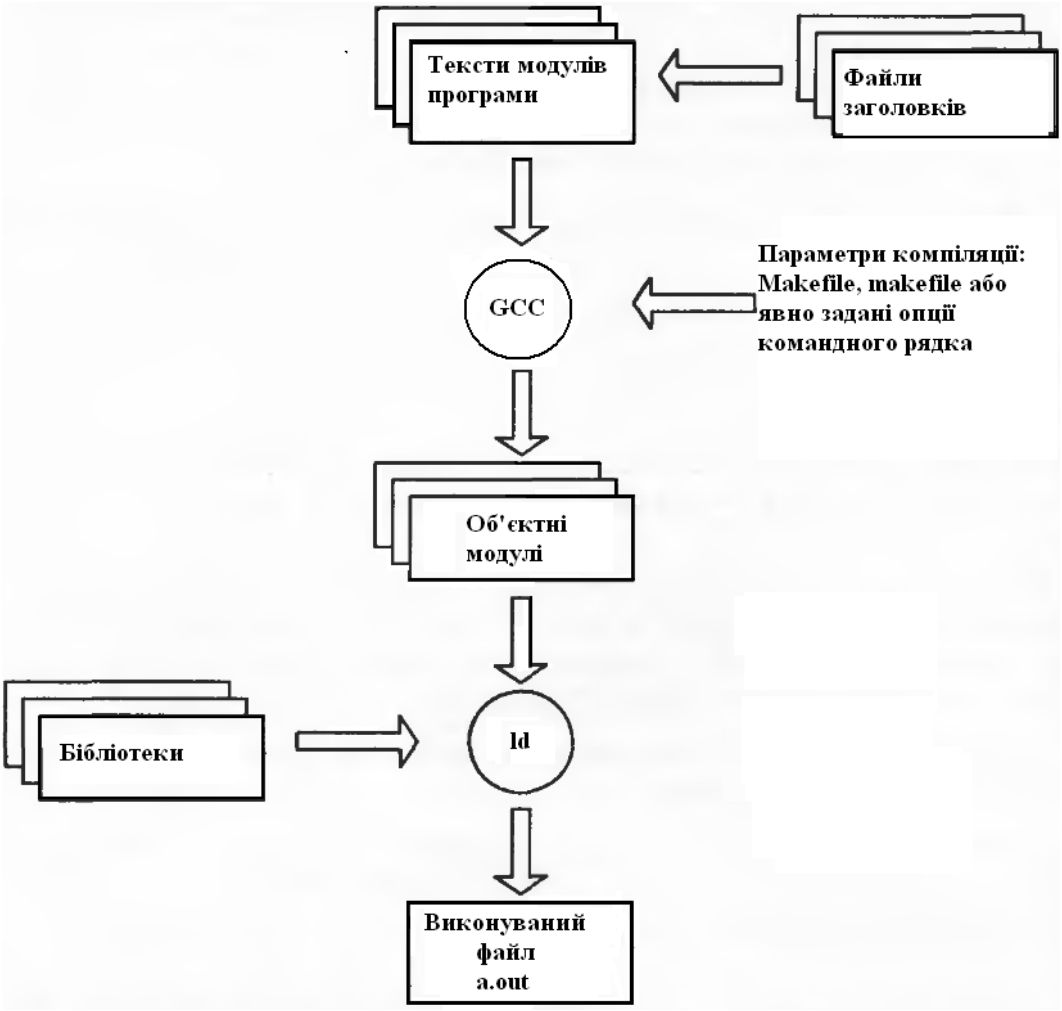

Рисунок 1.1 –Схема компіляції програми

Далі об’єктні файли за допомогою редактору зв’язків з’єднуються між собою та різними бібліотеками, у тому числі стандартною бібліотекою і бібліотеками, що вказані користувачем в якості параметрів. При цьому редактор зв’язків може виконуватися у двох режимах: статичному і динамічному, що задається відповідними опціями. В статичному режимі компонуються об’єктні файли і статичні бібліотеки (їх імена мають суфікс „*.a*”) і створюється єдиний файл, що містить увесь необхідний для виконання код. У другому випадку редактор зв’язків по можливості підключає бібліотеки, що використовуються сумісно, (їх імена мають суфікс „*.so*”) і створюється єдиний файл, к якому в процесі запуску програми на виконання будуть приєднані всі об’єкти сумісного використання.

На сьогодні в UNIX- системах у більшості випадків використовується компілятор GCC, 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 С, а файли виду *name.o* вважаються об'єктним представленням.

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

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

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 С++, а -lm визначає бібліотеку різних математичних функцій (*sin, cos, arctan, sqrt* тощо). Бібліотеки мають бути перераховані після файлів, що містять виклики до відповідних функцій.

1.3 Основні бібліотеки UNIX і їх файли заголовків

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 вироблятися 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 визначаються 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***>	Містять 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***>	Містять прототипи функцій, використовуваних для сканування дерева файлової системи, а також визначення необхідних констант
<***grp.h***>	Містять прототипи функцій і визначення структур даних, використовуваних для роботи із групами користувачів
<***langinfo.h***>	Містить визначення язикових констант дня тижня, назва місяця, а також прототип функції *langinfo*()

Продовження таблиці 1

1	2
<***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
<***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***>	Містять 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***>	Містять 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
<***sys/ipc.h***>	Містить визначення, що ставляться до системи взаємодії

	між процесами (IPC)
<***sys/msg.h***>	Містить визначення, що ставляться до системи IPC (повідомлення)

Продовження таблиці 1

1	2
<***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

1.4 Взаємодія програми з середовищем виконання

При запуску будь-яка програма UNIX отримує від процесу, що викликає, два набори даних: аргументи командного рядка і змінні середовища оточення. У програмах на мові С обидва набору 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 кожна програма повинна повернути код завершення. Нульове значення означає нормальне завершення, а ненульове значення свідчить або про аварійне завершення, або про те що програма не достигла своєї мети.

1.4.1 Список аргументів

Для запуску програми досить ввести її ім'я в командному рядку. Додаткові інформаційні елементи, що передаються 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 і /.

Виконання програм на мовах С та 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 ймення 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, що 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;
}
```

1.4.2 Змінні оточення

ОС UNIX надає кожній виконуваній програмі середовище виконання або, кажучи іншими словами, оточення програми. Під середовищем мається на увазі сукупність пар змінна-значення. Ці змінні називаються змінними оточення. Імена змінних оточення і їх значення є рядками. За існуючою угодою змінні оточення записуються прописними буквами.

Наведемо приклади змінних оточення:

- USER — містить реєстраційне ім'я поточного користувача;
- HOME — містить дорогу до початкового каталогу поточного користувача;
- MAILBOX — розташування поштової скриньки,
- 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 для того, 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, яка 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*().

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

int putenv(
const char *string //рядок у вигляді «ім'я=значення»
)
  // повертає 0 у випадку успіху,
  // ненульове значення у випадку помилки
```

Для установки і скидання значень змінних оточення призначені функції *setenv*() і *unsetenv*() відповідно.

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

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

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.

1.4.3 Коди завершення програми

Коли програма завершує роботу, вона повідомляє операційну систему про свій стан, посилаючи їй код завершення, який є 16-розрядним цілим числом. За існуючою угодою нульовий код свідчить про успішне завершення, а ненульовою 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 операторові *return* у функції main або виконує виклик функції *exit*(). Процес може бути завершений і по незалежних від нього обставинах, наприклад унаслідок отримання сигналу. В цьому випадку функція *exit*() буде викликана ядром від імені процесу.

Системний виклик *exit*() виглядає таким чином:

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

Аргумент *status* повертається батьківському процесу і є кодом повернення програми.

Наявність коду завершення дозволяє програмам взаємодіяти один з одним.

Окрім передачі коди повернення, функція *exit*() проводить ряд додаткових дій, зокрема виводить дані, що містяться в буферах , і закриває потоки введення-виведення.

Завдання може зареєструвати до 32 обробників виходу (*exit handler*), - функції, які викликаються після виклику *exit*(), але до остаточного завершення процесу. Викликаються ці обробники за принципом LIFO, причому лише при добровільному завершенні процесу.

Обробники реєструються за допомогою функції *atexit*().

Приклад:

```
#include <stdio.h>
#include <stdlib.h>
void handler1()
{
  printf("handler1\n");
}
void handler2()
{
  printf("handler2\n");
}
void handler3()
```

```
{
  printf("handler2\n");
}
int main()
{
  atexit(&handler1);
  atexit(&handler2);
  atexit(&handler3);
  return(0);
}
```

1.5 Помилки системних викликів

Використовуючи системні виклики для доступу до ресурсів, здійснення операцій введення-виводу або інших цілей, потрібно розуміти не лише те, що саме відбувається при успішному завершенні виклику, але також за яких обставин він може завершитися неуспіх. Збої системних викликів відбуваються в самих різних ситуаціях.

- У системі можуть закінчитися ресурси (або ж програма може вичерпати ліміт ресурсів, накладений на неї системою). Наприклад, програма може запитати надто багато пам'яті, записати занадто великий об'єм даних на диск або відкрити надмірну кількість файлів одночасно.
- Операційна система блокує деякі системні виклики, коли програма намагається виконати операцію за відсутності належних привілеїв. Наприклад, програма може спробувати здійснити запис в доступний лише для читання файл, звернутися до пам'яті іншого процесу або знищити програму іншого користувача.
- Аргументи системного виклику можуть виявитися неправильними або унаслідок помилковий введених користувачем даних, або із-за помилки самої програми. Наприклад, програма може передати системному виклику неправильну адресу пам'яті або невірний дескриптор файлу. Інший варіант помилки — спроба відкрити каталог замість звичайного файлу або передати ім'я файлу системному виклику, який очікує на ім'я каталогу.
- Системний виклик може аварійно завершитися по причинах, не залежних від самої програми. Найчастіше це відбувається при доступі до апаратних пристроїв. Пристрій може працювати некоректно або не підтримувати необхідну операцію, або в дисковод просто не вставлений диск.
- Виконання системного виклику інколи уривається зовнішніми подіями, до яких відноситься, наприклад, здобуття сигналу. Це не

обов'язково означає помилку, але відповідальність за повторний запуск системного виклику покладається на програму.

У добре написаній програмі, що часто звертається до системних викликів, велика частина коди присвячена виявленню і обробці помилок, а не рішенню основної задачі.

1.6 Коди помилок системних викликів

Більшість системних викликів повертають значення. Якщо системний виклик хоче повідомити про помилку, то він повертає значення, яке неможливо сплутати з коректними даними, як правило, це число -1. (В деяких випадках використовуються інші угоди. Наприклад, функція *malloc*() при виникненні помилки повертає нульовий покажчик.) Зазвичай цієї інформації вистачає для того, щоб вирішити, чи слід продовжувати звичне виконання програми. Але для більш спеціалізованої обробки помилок необхідні додаткові відомості.

Більшість системних викликів (приблизно 80%) зберігають в спеціальній змінній *errno* розширену інформацію про помилку, що сталася. У цю змінну записується число, що ідентифікує виниклу ситуацію. Оскільки всі системні виклики працюють з однією і тією ж змінною, необхідно відразу ж після завершення функції скопіювати значення змінної в інше місце. Змінна *errno* модифікується після кожного системного виклику.

В цілях 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 угодою, записуються прописними буквами і починаються з літери "E", наприклад EACCES і EINVAL. При роботі значеннями змінної errno слід завжди використовувати макроконстанти, а. не реальні числові значення. Всі ці константи визначені у файлі <*errno.h*>.

У бібліотеці UNIX є зручна функція *strerror*(), що повертає строковий еквівалент коди помилки. Ці рядки можна включати в повідомлення про помилки.

Оголошення функції знаходиться у файлі <*string.h*>.

```
#include <string.h>
char *strerror(
   int errnum ..// код помилки
);
// Повертає повідомлення
```

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

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

У наступному фрагменту програми робиться спроба відкрити файл. Якщо це не виходить, виводиться повідомлення про помилку і програма завершує свою роботу. Відмітимо, що в разі успіху операції функція *open*() повертає дескриптор відкритого файлу, інакше — -1.

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

Залежно від особливостей програми і використаного системного виклику конкретні дії, що робляться в разі помилки, можуть бути різними: виведення повідомлення про помилку, відміна операції, аварійне завершення програми, повторні тортури і навіть ігнорування помилки. Проте, поважно включити в програму код, що оброблює всі можливі варіанти помилок.

Коди помилок системних викликів, системні повідомлення та короткий опис наведені у таблиці 2.

Таблиця 2. - Коди помилок системних викликів

Код помилки і повідомлення	Опис
1	2
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

Продовження таблиці 2

1	2
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 не має у черзі
ENOSPC	Спроба запису у файл або створення нового

No space left on device	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

Продовження таблиці 2

1	2
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

1.7 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?
9. Яким чином в командному рядку задаються аргументи програми?
10. Яким чином функція *main*() дістає доступ до списку аргументів?
11. Що таке змінна оточення? В якому форматі вони зберігаються?
12. Яким чином програма дістає доступ до змінних оточення?
13. Як додати змінну оточення?
14. Як програма повертає код завершення програми?
15. Що таке обробник виходу? Як його зареєструвати?

2 ЗНАЙОМСТВО З ФАЙЛОВОЮ СИСТЕМОЮ

Операційна система виконує два основні завдання: маніпулювання даними і їх зберігання. Більшість програм в основному маніпулюють даними, але, кінець кінцем, вони де-небудь зберігаються. У системі UNIX таким місцем зберігання є файлова система. Більш того, в UNIX всі пристрої, з якими працює операційна система, також представлені у вигляді спеціальних файлів у файловій системі.

Файл — це іменована область зовнішньої пам'яті, в яку можна записувати і з якої можна прочитувати дані

Основні цілі використання файлу.

- Довготривале і надійне зберігання інформації.
- Довга тривалість досягається за рахунок використання пристроїв, що запам'ятовують, не залежних від живлення, а висока надійність визначається засобами захисту доступу до файлів і загальною організацією програмної коди ОС, при якій збої апаратури найчастіше не руйнують інформацію, що зберігається у файлах.
- Спільне використання інформації.
- Файли забезпечують природний і легкий спосіб розділення інформації між додатками і користувачами за рахунок наявності зрозумілого людині символьного імені і постійності інформації, що зберігається, і розташування файлу. Файл може бути створений одним користувачем, а потім використовуватися зовсім іншим користувачем, при цьому творець файлу або адміністратор можуть визначити права доступу до нього інших користувачів.
- Файлова система — це частина операційної системи, що включає:
- сукупність всіх файлів на диску;
- набори структур даних, використовуваних для управління файлами, такі, наприклад, як каталоги файлів, дескриптори файлів, таблиці розподілу вільного і зайнятого простору на диску;
- комплекс системних програмних засобів, що реалізовують різні операції над файлами, такі як створення, знищення, читання, запис, іменування і пошук файлів.

Файлова система контролює права доступу до файлів, виконує операції створення і видалення файлів, а також виконує запис/читання даних файлу. Оскільки більшість прикладних функцій виконуються через інтерфейс файлової системи, отже, права доступу до файлів визначають привілеї користувача в системі.

Файлова система забезпечує направлення запитів, адресованих периферійним пристроям, відповідним модулям підсистеми введення-виведення.

Файлова система UNIX забезпечує уніфікований інтерфейс доступу до даних, розташованих на різних носіях, і до периферійних пристроїв.

2.1 Орієнтація і навігація у файловій системі

Ієрархічна структура файлової системи UNIX спрощує орієнтацію в ній. Кожен каталог, починаючи з кореневого (/), у свою чергу, містить файли і інші каталоги (підкаталоги). Кожен каталог містить також посилання на батьківський каталог (для кореневого каталогу батьківським є він сам), представлене каталогом з ім'ям дві крапки (..) і посилання на самого себе, представлене каталогом з ім'ям крапка (.).

2.2 Імена файлів в ОС UNIX

У ОС UNIX підтримується три способи вказівки імен файлів:

Коротке ім'я. Ім'я, що не містить спеціальних метасимволів коса риска (/), є коротким ім'ям файлу. По короткому імені можна послатися на файли поточного каталогу. Наприклад, команда *ls -l .profile* вимагає отримати повну інформацію про файл *.profile* у поточному каталозі.

У ієрархічних файлових системах різним файлам дозволено мати однакові прості символьні імена за умови, що вони належать різним каталогам.

Відносне ім'я. Ім'я, що не починається з символу косої риски (/), але що включає такі символи. Воно посилається на файл відносно поточного каталогу. При цьому для Посилання на файл або каталог в якомусь іншому каталозі використовується метасимвол косої риски (/). Наприклад, команда

```
ls -l ../.profile
```

вимагає отримати повну інформацію про файл *.profile* у батьківському каталозі поточного каталогу, а команда

```
cat doc/text.txt
```

вимагає видати вміст файл *text.txt* у підкаталозі *doc* поточного каталогу.

Повне(абсолютне) ім'я. Ім'я, що починається з символу косої риски (/). Воно посилається на файл відносно кореневого каталогу. Це ім'я ще

називають абсолютним, оскільки воно, на відміну від попередніх способів завдання імені, посилається на один і той же файл незалежно від поточного каталогу. Наприклад, команда

```
ls -l /home/user01/.profile
```

вимагає отримати повну інформацію про файл *.profile* у каталозі */home/user01* незалежно від того, в якому каталозі виконується.

Між файлом і його повним ім'ям є взаємно однозначна відповідність

Інші символи, окрім косої риски, не мають в іменах файлів UNIX особливого значення. Зокрема, немає системного поняття розширення файлу. Імена файлів чутливі до регістра.

У ОС UNIX немає теоретичних обмежень на кількість вкладених каталогів. Проте, в кожній реалізації є практичні обмеження на максимальну довжину імені файлу, яке вказується в командах (як і на довжину командного рядка в цілому).

2.3 Типи файлів

У UNIX існує декілька типів файлів, що розрізняються по функціональному призначенню і діям операційної системи при виконанні тих або інших операцій над ними.

Звичайний файл

Звичайний файл *(regular file*) є найбільш загальним типом файлів, що містить дані в деякому форматі. Для операційної системи такі файли є просто послідовністю байтів. До цих файлів відносяться текстові файли, двійкові дані і виконувані програми. У довгому лістингу ознакою звичайного файлу є дефіс (-) в першій позиції першого стовпця:

```
-rw-rw-r--   1 root    sys        8296 Фев 23 15:39 ps_data
```

Каталог

За допомогою каталогів (*directory*) формується логічне дерево файлової системи. Каталог - це файл, що містить імена файлів, що знаходяться в нім, а також покажчики на додаткову інформацію - метадані, що дозволяють операційній системі виробляти дії з цими файлами. Каталоги визначають положення файлу в дереві файлової системи. Будь-який процес, що має право на читання каталогу, може прочитати його вміст, але лише ядро має право на запис даних каталогу.

У довгому лістингу ознакою каталогу є символ *d* в першій позиції першого стовпця:

```
drwxr-xr-x   2 informix informix     115 Фев 24 13:05 txt
```

Спеціальний файл пристрою

Забезпечує доступ до фізичних пристроїв. У UNIX розрізняють символьні (*character special device*) і блокові (*block special device*) файли пристроїв. Доступ до пристроїв здійснюється шляхом відкриття, читання і запису в спеціальний файл пристрою.

Символьні файли пристроїв використовуються для обміну даними з пристроєм без буферизації. Блокові файли пристроїв дозволяють виробляти обмін даними у вигляді пакетів фіксованої довжини - блоків.

У довгому лістингу ознакою спеціального символьного і блокового пристроїв є символи *c* і *b* в першій позиції першого стовпця, відповідно.

FIFO - іменований канал

Цей файл використовується для зв'язку між процесами за принципом черги. Іменовані канали вперше з'явилися в UNIX System V, але більшість сучасних систем підтримують цей механізм.

У довгому лістингу ознакою іменованого каналу є символ *p* в першій позиції першого стовпця:

Посилання

Каталог містить імена файлів і покажчики на їх метадані. Така архітектура дозволяє одному файлу мати декілька імен у файловій системі. Імена жорстко пов'язані з метаданими і, відповідно, з даними файлу, тоді як сам файл існує незалежно від того, як його називають у файловій системі.

Стандарт POSIX вимагає реалізувати підтримку двох типів посилань - жорстких і символічних.

Жорстким посиланням вважається елемент каталогу, що вказує безпосередньо на деякий індексний дескриптор. Жорсткі посилання дуже ефективні, але у них існують певні обмеження, оскільки вони можуть створюватися лише в межах однієї фізичної файлової системи. Коли створюється такий зв'язок, файл повинен вже існувати.

Кількість жорстких посилань файлу (а також кількість файлів в каталозі, якщо файл є каталогом) відображується в другому полі довгого лістингу:

Символічне посилання (*symbolic link*) - це спеціальний файл, який містить дорогу до іншого файлу. Вказівка на те, що даний елемент каталогу є символічним Посиланням, знаходиться в індексному дескрипторі. Тому звичайні команди доступу до файлу замість здобуття

даних з фізичного файлу, беруть їх з файлу, ім'я якого приведене в посиланні. Цей шлях може вказувати на що завгодно: це може бути каталог, він може навіть знаходитися в іншій фізичній файловій системі, більш того, вказаного файлу може і зовсім не бути.

Сокет

Сокети дозволяють представити у вигляді файлу в логічній файловій системі мережеве з'єднання. У довгому лістингу ознакою сокету є символ *s* в першій позиції першого стовпця.

2.4 Файлові дескриптори і системна таблиця файлів

Кожен процес в UNIX володіє набором файлових дескрипторів, які нумеруються від 0 до N, де N — максимально можливе число одночасно відкритих файлових дескрипторів. Число N залежить від версії ОС UNIX і її конфігурації. У будь-якому випадку це число не буває менше 16, а частенько — багато більше. Аби набути фактичного значення числа N, можна викликати функцію *sysconf* з аргументом _SC_OPEN_MAX, наприклад, так:

```
printf("_SC_OPEN_MAX = Xld\n", sysconf(_SC_OPEN_MAX));
```

У операційній системі Linux 2.4 ми отримаємо число 1024, в FREEBSD — 957, в Solaris — 256.

2.5 Стандартні дескриптори

Згідно прийнятим угодам, перші три файлові дескриптори передаються процесу вже відкритими. Дескриптор з номером 0 відповідає пристрою стандартного введення, дескриптор з номером 1 — пристрою стандартного виводу, а з номером 2 — пристрою стандартного виведення повідомлень про помилки. Ці три дескриптори зазвичай використовуються для управління терміналом. У текстах програм, замість номерів набагато зручніше використовувати константи *STDIN_FILENO, STDOUT_FILENO* та *STDERR_FILENO*.

Утиліти UNIX повинні виконувати читання вхідних даних з STDIN_FILENO і записувати результати в *STDOUT_FILENO*, завдяки цьому командна оболонка може об'єднувати їх в конвеєри. Для видачі важливих повідомлень слід використовувати пристрій *STDERR_FILENO*, оскільки будь-який запис в *STDOUT_FILENO* може бути переправлений у

файл або переданий далі по конвеєру іншій утиліті, що є звичайною практикою при роботі з командною оболонкою.

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

До моменту запуску процесу, три стандартні дескриптори вже відкрито і готові до використання у викликах *read*() і *write*(). Процес може відкривати інші дескриптори, які використовуються для роботи з файлами, каналами і тому подібне. Батьківський процес може «заповідати» нащадкові не лише ці три стандартні дескриптори, але і ті, що відкрив сам.

2.6 Використання файлових дескрипторів

ОС UNIX використовує файлові дескриптори для роботи з чим завгодно, що поводиться подібно до файлів, якщо допустимі операції читання і запису. Файлові дескриптори не використовуються в механізмах взаємодії між процесами, таких як черги повідомлень, до яких непридатні виклики *read*() і *write*() (для цих цілей призначені спеціальні виклики).

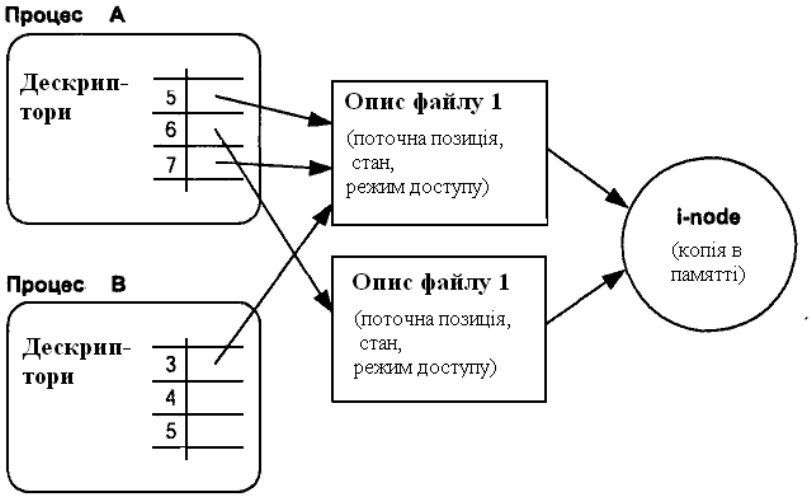

Рисунок 2.1 – Файлові дескриптори, описи відкритих файлів і індексні вузли

Існує не так багато способів відкрити новий файловий дескриптор. Хоча ми з вами доки не готові розглянути їх у всіх подробицях, буде доречно хоч би перелічити їх:

- *open*() — використовується в більшості випадків, коли можна вказати шлях;
- *pipe*() — створює і відкриває неіменований канал;

- *socket*(), *accept*() і *connect*() — використовуються для створення мережевих з'єднань.

Мова програмування С не передбачає окремого типа для опису дескрипторів (як, наприклад, для ідентифікаторів процесів), тому ми просто використовуємо звичайний тип *int*.

2.7 Системні виклики open і creat

Системний виклик *open*() відкриває існуючий файл (звичайний, спеціальний або іменований канал) або створює новий, але в даному випадку це може бути лише звичайний 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
```

Спершу поговоримо про відкриття існуючого файлу, ім'я якого задається аргументом *path*. Якщо через аргумент *flags* передається значення O_RDONLY, то файл лише для читання, O_WRONLY — лише для запису, O_RDWR — як для запису, так і для читання. В разі відкриття існуючого файлу, аргумент *perms* ігнорується і в текстах програм, як правило, взагалі опускається. Таким чином, функція *open*() викликається всього з двома аргументами, наприклад:

```
fd=open("/home/student/oldfile",    O_RDONLY)
```

Виклик open може завершитися з помилкою з багатьох причин. В більшості випадків у нас є можливість повідомити користувача про конкретну проблему. Невірна вказівка дорозі до файлу (ENOENT) вимагає іншого рішення, чим відсутність прав доступу (EACCESS).

Виклик open привласнює файловому дескриптору найменший з доступних номерів, але, як правило, вас це число не цікавить. Проте, знання цієї обставини може деколи послужити непогану службу. Наприклад, коли необхідно переправити один із стандартних дескрипторів — 0, 1 або 2, потрібно просто закрити цей дескриптор і відкрити файл, який отримає номер тільки що закритого дескриптора.

Якщо файл з вказаним ім'ям не існує, 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, цього разу необхідно визначити права доступу до файлу, наприклад:

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

Остаточний набір прав доступу до файлу, що створюється, формується як результат логічного множення аргументу *perms*, переданого в системний виклик, на логічне доповнення маски прав доступу. Маска зазвичай встановлюється під час реєстрації користувача в системі (командою *umask*) або системним викликом *umask*(). Фраза «логічне множення на логічне доповнення» означає: якщо деякий біт маски встановлений, то відповідний йому біт в остаточному наборі прав доступу скидається. Так, маска 002 приведе до того, що біт S_IWOTH (право на запис для всіх інших) буде скинутий, навіть якщо в системний виклик open буде переданий прапор S_IWOTH.

Що станеться, якщо ви створити файл з прапором O_WRONLY або O_RDWR, але при цьому не дати йому права на запис? Оскільки файл був тільки що створений, він поки що доступний для запису. Проте, коли наступного разу ви спробуєте відкрити його, всі накладені вами обмеження наберуть чинності.

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)
```

Використання прапору O_TRUNC наводить до знищення всіх даних у файлі, тому його допустимо вказувати лише при відкритті існуючого файлу і за умови, що процес має право на запис у файл, що відкривається. З тієї ж причини, прапор O_TRUNC не може використовуватися спільно з прапором O_RDONLY. Аби створити новий файл (з прапором O_CREAT), процес повинен мати право на запис в каталог, оскільки Посилання на файл буде записано у файл каталогу. При відкритті існуючого файлу права доступу до каталогу не мають значення. В цьому випадку враховуються лише права доступу до файлу.

Слід також відмітити, що процес повинен володіти правом на пошук (виконання) в проміжних каталогах. Прапор O_TRUNC не повинен використовуватися без O_CREAT, якщо передбачається створення нового файлу. Сам по собі він означає наступне: «якщо файл існує — видалити з нього всі дані, якщо файл не існує — завершитися з ознакою помилки».

С іншої сторони, комбінація опцій O_WRONLY | O_CREAT | O_TRUNC настільки звичайна («створити файл або очистити його і відкрити на запис»), що для цієї мети існує спеціальний системний виклик:

```
#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
```

При зверненні до системного виклику open для відкриття існуючого файлу, можна вказувати лише перший і другий аргументи (*path* і *flags*). Системний виклик creat використовує лише перший і третій аргументи.

Знов створеному файлу призначаються ідентифікатор користувача-власника і ідентифікатор групи-власника, які скорочено ми далі називатимемо як «власник» і «група». Правила призначення власника і групи виглядають таким чином.

- Як власник файлу призначається ефективний ідентифікатор користувача процесу.
- Як група встановлюється або ідентифікатор групи каталогу, в якому створюється файл, або ефективний ідентифікатор групи процесу.

В програми немає можливості вибрати, яка з двох груп буде привласнена файлу, але воно може взнати це за допомогою системного виклику *stat*(). Аби примусово призначити файлу потрібну групу, можна скористатися системним викликом *chown*().

Існує ще один прапор — O_EXCL, який у поєднанні з O_CREAT викликає помилку при спробі відкрити існуючий файл. Системний виклик open, використовуваний без прапора O_CREAT, означає: «відкрити файл, якщо він існує, інакше — завершитися з помилкою». З комбінацією прапорів O_CREAT | O_EXCL — з точністю до навпаки: «створити новий файл, якщо він не існує, інакше — завершитися з помилкою».

У таблиці 3 перераховані всі прапори системного виклику open, визначені в SUS3.

Таблиця 3 - Прапори системного виклику open

Прапор	Опис
0_RDONLY	Відкрити лише для читання
O_WRONLY	Відкрити лише для запису
O_RDWR	Відкрити для читання і для запису
O_APPEND	Відкрити для доповнення в кінець файлу
O_CREAT	Створити новий, якщо не існує
O_DSYNC	Встановити режим синхронного введення-виводу
O_EXCL	Не відкривати, якщо існує.

O_NOCTTY	Не робити пристрій терміналом, що управляє
O_NONBLOCK	Неблокуючий режим роботи з іменованими каналами і спеціальними файлами
O_RSYNC	Встановити режим синхронного введення-виводу
O_SYNC	Встановити режим синхронного введення-виводу
O_TRUNC	Видалити вміст файлу — усікти його розмір до нуля

2.8 Запис у файл. Системний виклик *write*

Системний виклик *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
```

Системний виклик *write*() записує *nbytes* байт з буфера *buf* у відкритий файл, який представлений дескриптором *fd*. Запис починається з поточної позиції у файлі, а після її закінчення поточна позиція зміщується на число записаних байт. У програму, що викликає, повертається число байт, яке було записано або -1,если виникла помилка. Нагадаю: якщо файл був відкритий з прапором O_APPEND, то безпосередньо перед записом поточна позиція автоматично буде переміщатися в кінець файлу.

Виклик *write*() може також використовуватися для запису в канали, в спеціальні файли або в сокети, але в цих випадках він має деякі особливості. Найважливіша відмінність: у подібних вживаннях виклик *write*() може блокуватися, це означає, що запис припиняється до настання деякої події, наприклад звільнення місця для запису чергової порції даних. Якщо виклик *write*() заблокований, він може бути перерваний сигналом, що поступив; в цьому випадку процесу, що викликає, повертається -1 і код помилки EINTR в змінній *errno*.

Простота виклику *write*() брехлива. На перший погляд здається, що він спочатку записує дані, а потім повертає управління, але нескладні розрахунки показують, що це неможливо — повернення в програму відбувається дуже швидко. Дійсно, виклик *write*() не виконує запис на диск. Він просто переписує дані в буферний кеш ядра і закінчує свою роботу.

Коли все йде добре, ми отримуємо фантастичний виграш! Сенс той же, неначебто ми фактично виконали запис на диск, лише набагато швидше. Проте, якщо виникне збій в роботі дискового пристрою або ядро «звалиться» з якої-небудь причини, ми виявимо, що на диску немає даних, які були «записані».

На додаток до невизначеності з часом фактичному запису даних на диск, техніка відкладеного запису має ще дві проблеми. Перша — процес, що ініціював запис, не може бути проінформований про помилки, що виникли під час запису на диск. Насправді, буфери файлової системи не належать жодному з процесів — якщо декілька процесів намагаються записувати дані в одну і ту ж ділянку одного і того ж файлу, то їх дані попадуть в один і той же буфер. Звичайно, можна було б передбачити механізм передачі сигналу «*Write error*» всім процесам, які писали в цей буфер, але що тоді повинен робити процес з даними, які були записані після нього. І як повідомити процеси, які вже закінчили роботу?

Друга проблема полягає в тому, що фізичний порядок запису буферів не піддається регулюванню. Порядок часто має значення. Наприклад, при зміні структури зв'язного списку у файлі, переважно спочатку записати у файл новий елемент списку, а потім відновити посилання на нього, ніж навпаки, тому що елемент, на який немає жодного посилання, викликає значно менше проблем, чим посилання, які вказують в нікуди. Навіть якщо серія системних викликів *write*() слідує в певному порядку, це не гарантує той же порядок запису буферів на диск.

З іншого боку, не варто переоцінювати значущість цих проблем. Враховуючи надійність сучасних комп'ютерів і реалізацій UNIX, можна сміливо сподіватися, на те, що аварійні ситуації в ядрі зустрічатимуться украй рідко. Більшість користувачів вважають за краще мати виграш за часом, який дає буферний кеш, і ніколи не знати про те, що ядро «обманює» їх.

2.9 Читання із файлу. Системний виклик *read*

Системний виклик *read*() виконує читання з файлового дескриптору.

```
#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
```

Системний виклик *read*() є протилежністю викличу *write*(). Він прочитує *nbytes* байт з файлу, представленого дескриптором *fd*, і розміщує їх в буфері *buf*. Читання починається з поточної позиції у файлі після закінчення і поточна позиція зміщується на кількість прочитаних байт. У програму, що викликає, повертається число прочитаних байт, 0 (після досягнення кінця файлу), або -1, як ознака помилки.

На відміну від *write*(), виклик *read*() не має таких широких можливостей в «обмані» Він не може спочатку передати дані процесу, що викликає, а потім прочитати їх з диска. Якщо в кеші немає необхідних даних (що можливо попали туди в результаті інших операцій введення-виводу), то процес вимушений чекати, поки ядро прочитає їх з диска. Інколи ядро прочитує з диска дані, що не лише зажадалися, а відразу декілька сусідніх блоків, намагаючись в такий спосіб передбачити потреби програми і підвищити швидкість читання даних з файлу. Якщо система працює з невеликим навантаженням, дані можуть зберігатися в буферах достатньо довго і часто виконується читання сусідніх блоків даних, то такий прийом випереджаючого читання дає істотний виграш.

2.10 Системний виклик *close*

Системний виклик *close*() закриває дескриптор файлу.

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

Найважливіше, що потрібно знати про системний виклик *close*(), це те, що він практично нічого не робить. Він не виштовхує на диск вміст буферів ядра - він просто звільняє дескриптор файлу для повторного використання. Коли закривається останній дескриптор, що посилається на запис в таблиці файлів, цей запис також може бути видалений. У свою чергу, коли віддаляється останній запис в таблиці файлів, що посилається на копію індексного вузла в пам'яті, вона також може бути видалена. І ще один момент: якщо лічильник посилань даного індексного вузла виявився рівним 0, то з диска віддаляється індексний вузол і всі пов'язані з ним дані.

Оскільки виклик *close*() не наводить до виштовхування даних з кеша на диск і зовсім ніяк не прискорює цей процес, немає жодної необхідності закривати файл, аби прочитати з нього тільки що записані дані. Ядро гарантує здобуття записаних вами даних. Іншими словами, наявність

механізму буферізації в ядрі ніяк не позначається на семантиці *read*(), *write*(), *lseek*() і будь-якого іншого системного виклику.

Фактично, закривати дескриптори файлів зовсім необов'язково, оскільки вони автоматично утилізують після закінчення процесу. Проте всі структури ядра краще звільняти своєчасно, а в текст програми додавати коментар, який повідомлятиме про закінчення роботи з файлом. Це допоможе вам під час відладки і запобіжить використанню дескриптора файлу помилково.

2.11 Системний виклик *lseek*

Системний виклик *lseek*() використовується для переустановлення поточної позиції у файлі. Він не виконує жодних операцій введення-виводу і не віддає команд контролеру диска.

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

Аргумент *whence* може приймати одне з наступних значень:

SEEK_SET	Аргумент *pos* містить зсув від початку файлу (абсолютна позиція у файлі).
SEEK_CUR	Аргумент *pos* містить зсув від поточної позиції у файлі. Може бути позитивним числом, нулем і негативним числом. Вказавши в аргументі *pos* значення 0 — ми отримаємо поточну позицію у файлі.
SEEK_END	Аргумент *pos* містить зсув від кінця файлу. Може бути позитивним числом, нулем і негативним числом. Вказавши в аргументі *pos* значення 0 — ми встановимо поточну позицію в кінець файлу

Результатом роботи виклику може бути будь-яке ненегативне число, що навіть перевищує розмір файлу. Якщо нова поточна позиція виявилася за межами файлу, то найближча операція запису вставить «недостатній» шматок в кінець файлу і заповнить його байтами із значенням 0. Виклик *read*(), з поточною позицією встановленою в кінець файлу або за його межами, поверне ознаку кінця файлу — 0. Спроба читання з інтервалу, який заповнив виклик *write*() при вставці «недостатнього» шматка,

увінчається успіхом і в програму, що викликає, буде повернено буфер, заповнений нулями, як того і слід було чекати.

Коли виробляється запис за межами файлу, більшість реалізацій ОС UNIX не зберігають вставлені таким чином порожні блоки. Це дає можливість, наприклад, на диску об'ємом в 3000000 блоків зберігати файли з сумарною довжиною більше 3000000 блоків. Така особливість може породити серйозні проблеми при відновленні файлів з резервних копій, тому що при створенні резервної копії на пристрій зберігання буде переданий повний об'єм файлів, тобто більше 3000000 блоків!

2.12 Системні виклики *pread* і *pwrite*

Системний виклик *pread*() виконує читання з файлу, починаючи із заданої позиції.

```
#include <unistd.h>
ssize_t pread(
   int fd, // дескриптор файлу
   void *buf, //адреса буфера для даних, що приймаються
size_t nbytes,  // об'єм даних, що приймаються
off_t offset     // позиція у файлі */
);
  // повертає кількість прочитаних байтів,
  // -1 у випадку помилки
  //(код помилки - в змінній errno)
```

Системний виклик *pwrite*() виконує запис у файл, в задану позицію.

```
#include <unistd.h>
ssize_t  pwrite(
 int fd,/* дескриптор файлу */
 const void *buf, //адреса буфера з даними для запису/
 size_t nbytes,  // кількість даних для запису
 off_t offset   // позиція у файлі
)
// Повертає кількість записаних байт
// або -1 в разі помилки
//(код помилки - в змінній errno)
```

Системні виклики *pread*() і *pwrite*() по своїй функціональності практично повністю збігаються з комбінацією виклику *lseek*() і подальшим викликом *read*() або *write*() відповідно, за виключенням:

- вони не користуються поточною позицією у файлі, оскільки позиція для запису або читання передається явно, через аргумент *offset*;
- вони не змінюють поточну позицію у файлі. Фактично, поточна позиція повністю ігнорується цими викликами.

Для файлів, відкритих з прапором O_APPEND, поведінка *pwrite*() повністю збігається з поведінкою виклику *write*() (аргумент *offset* ігнорується).

Зручність вживання одного виклику замість двох незаперечно, але, що важливіше, викликам *pread*() і *pwrite*() невластиві проблеми, пов'язані із зміною поточній позиції у файлі між *lseek*() і подальшими викликами *read*() або *write*() з іншого процесу або потоку. Це може статися при використанні одного дескриптора декількома потоками додатку або при використанні дублікатів дескрипторів декількома процесами. То ж відноситься і до файлів, відкритих з прапором O_APPEND. Оскільки ні *pread*(), ні *pwrite*() не використовують покажчик поточної позиції у файлі, їм невластиві помилки пов'язані з невірною установкою позиції.

2.13 Питання для самоперевірки

1. Що таке файлова система?
2. Які способи вказівки імен файлів підтримуються в ОС UNIX?
3. Які типи файлів існують в ОС UNIX?
4. Для чого використовуються спеціальні файли пристроїв?
5. Чим відрізняється жорстке посилання від символьного?
6. Чому жорстке посилання не вважається окремим типом файлу?
7. Що таке файловий дескриптор? Для чого вони використовуються?
8. Які файлові дескриптори вважаються стандартними?
9. Яким чином можна відкрити звичайний файл?
10. Чим відрізняється системний виклик *open*() від системного виклику *creat*()?
11. Що таке прапори відкриття файлу?
12. Чи має смисл комбінація прапорів O_WRONLY | O_EXCL | O_TRUNC при відкритті файлу? Чому?
13. Чи можна в програмі кілька разів відкрити один і той же файл?
14. Як змінюється поточна позиція в результаті запису в файл (читання з файлу)?
15. Яких складнощів при запису додає використання системою буферного кешу?
16. Що станеться, якщо перейти в кінець файлу і спробувати читати дані?

17. Що служить ознакою кінця файлу при читанні?
18. Що таке позіціюнування?
19. Чи можна збільшити довжину файлу, не проводячи запис в нього? Якщо так, то, яким чином, якщо немає, то чому?
20. Які переваги, на вашу думку, дає використання системних викликів *pread*() і *pwrite*()?

3 ФАЙЛИ В СЕРЕДОВИЩІ З БАГАТЬМА КОРИСТУВАЧАМИ

3.1 Користувачі и права доступу

Перш ніж почати роботу в UNIX, необхідно стати користувачем системи.

Користувач, з точки зору системи, не обов'язково людина. Користувач – це об'єкт, який володіє певними правами, може запускати на виконання програми і володіти файлами.

Як користувачі можуть виступати, наприклад, видалені комп'ютери або групи користувачів з однаковими правами і функціями. Такі користувачі називаються псевдокористувачами. Більшість користувачів, як правило, є реальними людьми, які зареєстровані в системі і запускають ті або інші програми.

Кожен користувач має своє унікальне ім'я (*login*), але система розрізняє користувачів по ідентифікатору, відповідному імені. Він називається ідентифікатором користувача (UID) .Кожний користувач є членом однієї або декількох груп.

Група – це користувачі, що мають схожі завдання. Кожна група має своє унікальне ім'я. Система ж розрізняє групи по ідентифікатору групи (GID).

UID і GID визначають, якими правами володіє користувач в системі.

Привласнення унікального ідентифікатора користувача виконується при закладі системним адміністратором нового реєстраційного імені. Значення ідентифікатора користувача і групи - не просто числа, які ідентифікують користувача, - вони визначають власників файлів і процесів

У системі існують один користувач, що володіє необмеженими можливостями. Це суперкористувач (*root*). Його UID=0 GID=0.

Права доступу визначаються окремо для володаря файлу, групи володаря та всіх останніх користувачів.

3.2 Права доступу и режими файлів. Додаткові права доступу

Кожен файл в ОС UNIX містить набір прав доступу, по якому визначається, як користувач взаємодіє з даним файлом. Цей набір зберігається в індексному дескрипторі даного файлу у вигляді цілого

значення, з якого зазвичай використовується 12 бітів. Причому кожен біт використовується як перемикач, вирішуючи (значення 1) або забороняючи (значення 0) той або інший доступ.

Три перших біта встановлюють різні види поведінки при виконанні. Що залишилися дев'ять діляться на три групи по три, визначаючи права доступу для власника, групи і останніх користувачів. Кожна група задає права на читання, запис і виконання.

Біт читання для всіх типів файлів має одне і те ж значення: він дозволяє читати вміст файлу (отримувати лістинг каталогу командою ls).

Біт запису також має одне і те ж значення: він дозволяє редагувати цей файл. Для каталогу – це можливість міняти його вміст, тобто створювати і видаляти файли.

Якщо для деякого файлу встановлений біт виконання, то файл може виконуватися як команда. В разі установки цього біта для каталогу, цей каталог можна зробити поточним (перейти в нього командою *cd*).

Встановлений біт зміни ідентифікатора користувача SUID означає, що доступний користувачеві на виконання файл виконуватиметься з правами (з ефективним ідентифікатором) власника, а не користувача, що викликав файл (як це зазвичай відбувається).

Встановлений біт зміни ідентифікатора групи SGID означає, що доступний користувачеві на виконання файл виконуватиметься з правами (з ефективним ідентифікатором) групи-власника, а не користувача, що викликав файл (як це зазвичай відбувається).

Прикладом може бути утиліта *passwd*, що дозволяє користувачеві міняти свій пароль. Для зміни пароля потрібно редагувати вміст файлу */etc/passwd* і */etc/shadow*. Надати права всім змінювати вміст цих файлів неможливо. Установка SUID для утиліти *passwd* дозволяє вирішити цю проблему. Власником файлу */usr/bin/passwd*, в якому зберігається утиліта є суперкористувач. Хто б не запустив утиліту на виконання, на час роботи даної програми отримує права суперкористувача, а значить, може виробляти зміни в системних файлах.

Якщо біт SGID встановлений для файлу, не доступного для виконання, він означає обов'язкове блокування, тобто незмінність прав доступу на читання і запис доки файл відкритий певною програмою.

Встановлений клейкий біт (*sticky*) для звичайних файлів раніше (за часів PDP-11) означав необхідність зберегти образ програми (тобто код і дані) в пам'яті після виконання (для прискорення повторного завантаження). Зараз при установці звичайним користувачем він скидається.

Установка клейкого біта для каталогу означає, що файл в цьому каталозі може бути видалений або перейменований лише в наступних випадках:

- користувачем-власником файлу;
- користувачем-власником каталогу;

- якщо файл доступний користувачеві на запис; користувачем *root*.

Прикладом може служити каталог */tmp*, який відкритий на запис для всіх користувачів, але, в якому небажано видаляти чужі тимчасові файли.

Стандартом POSIX1988 замість вісімкових чисел для опису прав доступу були рекомендовані до вживання спеціальні ідентифікатори. Принцип їх іменування слідує шаблону `S_Ipwww`, де p визначає режим доступу (R, w або X), а www — кому видається право на цей режим доступу (USR, GRP або OTH).

Наприклад, для попереднього файлу замість вісімкового числа 755 можна записати:

```
S_IRUSR|S_IWUSR|S_IXUSR|S_IRGRP|S_IXGRP|S_IROTH|S_IXOTH
```

Існують окремі ідентифікатори для USR, GRP і OTH, які описують повні права доступу. Іменування цих ідентифікаторів слідує формі `S_IRWXw`. Тут символ w визначає, кому видається повне право доступу до файлу (від англ., «*whom*» — «кому») — U, G або O. Таким образом, попередній приклад може бути записаний в наступному вигляді:

```
S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH|S_IXOTH
```

Використання ідентифікаторів дає розробникам ОС свободу вибору порядку дотримання бітів, що описують права доступу до файлу.

3.3 Маска створення файла і системний виклик *umask*

Ми вже згадували про маску, що накладається на набір прав доступу при створенні файлу. Накладення маски виконує системний виклик *umask*(), який рідко використовується де-небудь ще, окрім як в команді *umask*.

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

Оскільки кожен процес має маску і будь-яка комбінація з 9 біт вважається допустимим, *umask*() не передбачає вихід помилково. Він завжди повертає колишнє значення маски. Аби взнати поточне значення маски не змінюючи її, потрібно виконати два послідовні виклики *umask*(). Перший — аби набути поточного значення маски (передавши у виклик будь-яке значення нової маски), другої, — аби відновити її в первинний стан.

3.4 Зміна прав доступу

Системний виклик *chmod*() змінює режими доступу до існуючого файлу будь-якого типа. Він не може використовуватися для зміни типа файлу — йому доступні лише прапори S_ISUID, S_ISGID, S_ISVTX і біти прав доступу.

```
#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
```

Системний виклик *fchmod*() дуже схожий на *chmod*(), але як аргумент приймає не ім'я, а дескриптор файлу.

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

Змінити режими доступу може лише процес, що має права суперкористувача, або ідентичний файлу ідентифікатор користувача, що діє. Недостатньо мати права на запис у файл. Крім того, аби встановити прапор S.ISGID, процес повинен мати ідентичний файлу ідентифікатор групи, що діє (окрім суперкористувача).

Якщо ви не збираєтеся міняти режими доступу цілком, то спочатку потрібно отримати існуючі режими викликом *stat*(), потім встановити або скинути необхідні біти і потім викликати *chmod()* для зміни режимів.

Як правило, виклик *chmod*() не використовується в застосуваннях, тому що режими доступу задаються під час створення файлу. З іншого боку, користувачі досить часто користуються командою *chmod*.

3.5 Зміна володаря файлу

Системний виклик *chown*() змінює ідентифікатори користувача і групи файлу. Ця операція доступна лише власникові файлу (процесу з ідентичним файлу ідентифікатором користувача, що діє) і суперкористувачеві. Якщо один з аргументів має значення -1, то відповідний йому ідентифікатор залишається без змін.

```
#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
```

Системний виклик *fchown*() ідентичний *chown*(), лише приймає як аргумент не ім'я файлу, а дескриптор.

```
#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
```

Системний виклик *lchown*() впливає на саме символічне посилання, а не на об'єкт посилання.

```
#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
```

Якщо процес, який зробив системний виклик, не володіє правами суперкористувача, то дані виклики скидають біти set-user-ID і set-group-ID для запобігання сповна очевидній формі атак:

```
$ cp /bin/sh mysh
```
[отримати персональну копію командного інтерпретатора]

```
$ chmod 4700 mysh
```
[включити біт set-user-ID]

```
$ chown  root mysh
```
[зробити суперкористувача власником]

```
$ my sh
```
[отримати права суперкористувача]

Якщо ви хочете мати персональний командний інтерпретатор з правами суперкористувача, вам потрібно змінити порядок виклику команд *chmod* і *chown*. Проте, якщо ви не суперкористувач, то ви не зможете виконати команду *chmod*. Таким чином лазівка закрита.

Якщо в системі є макрос _P0SIX_CHOWN_RESTRICTED, правила гри можуть трохи відрізнятися. У цьому випадку правому зміни власника

файлу володіє лише суперкористувач, але власник файлу може змінити групу файлу на групу процесу, що діє, або одну з додаткових груп. Іншими словами: власник не може «віддати» свій файл, саме більше — змінити групу файлу на одну з асоційованих з процесом.

3.6 Питання для самоперевірки

1. Хто може бути користувачем системи UNIX?
2. Які ідентифікатори користувача і групи має суперкористувач?
3. Які категорії прав доступу до файлів розрізняють в ОС UNIX?
4. Для чого використовуються додаткові права доступу?
5. Як змінити маску, що накладається на набір прав доступу при створенні файлу?
6. Хто може змінити права доступу до файлу?
7. Хто може змінити володаря файлу?

4 ФАЙЛІ З ДЕКІЛЬКОМА ІМЕНАМИ. МЕТАДАНІ ФАЙЛІВ

4.1 Файлі з декількома іменами

Запис у файлі каталогу, що містить ім'я і номер індексного вузла, називається жорстким посиланням. Існує ще один тип посилань — символічні посилання. Посилання дають можливість файлам мати декілька імен.

4.1.1 Створення жорсткого посилання

Жорстке посилання з'являється при створенні файл будь-якого типа, включаючи каталоги. Можна створити додаткові жорсткі Посилання на файли, що не є каталогами, за допомогою системного виклику *link*():

```
#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
```

Перший аргумент (*oldpath*) має бути ім'ям існуючого жорсткого Посилання — вона представляє номер використовуваного індексного вузла. Другий аргумент (*newpath*) задає ім'я нового жорсткого посилання. Старе і нове посилання абсолютно рівноправні, оскільки UNIX не розрізняє первинні і вторинні посилання. Процес, що створює жорстке посилання, повинний мати право на запис в каталог, де воно буде розміщене. Імені посилання, представленого в іншому аргументі, не повинне існувати — системний виклик *link*() не уміє змінювати існуючі посилання. В разі споживи існуюче посилання має бути спочатку видалене викликом *unlink*().

4.1.2 Створення символічних посилань

На відміну від жорсткого посилання, символічне посилання — просто маленький файл, який зберігає рядок повного імені об'єкту в текстовому вигляді. Символічні посилання можуть посилатися на інші символічні посилання. Зазвичай, коли системному виклику *open*() передається символічне посилання такого роду, ядро проходить по ланцюжку до тихий пір, поки не зустріне об'єкт, який не є символічним посиланням. З жорсткими посиланнями цього не відбувається, тому що сморід прямо посилаються на індексний вузол, який не може бути іншим жорстким посиланням. Іншими словами, якщо дорога до об'єкту визначається жорстким посиланням, він може розглядатися буквально. Якщо символічним посиланням, то фактична дорога до об'єкту залежить від вмісту проміжних символічних посилань.

Створюються символічні посилання командою *ln* з ключем *-s*, проте існує системний виклик *symlink*(), який виконує ту ж роботові.

```
#include <unistd.h>
int symlink(
  const char *oldpath, /* можливе старе повне ім'я »/
  const char *newpath /* нове повне ім'я */
)
//повертає 0 у випадку успіху -1 у випадку помилки
```

Головнім чином, *symlink*() працює аналогічно *link*(). В обох випадках створюється нове жорстке посилання з ім'ям *newpath*. Проте в разі символічного посилання, це нове жорстке посилання вказує на файл символічного посилання, який містить рядок *oldpath*.

В коментарях до системного виклику використано уточнення — «можливе». Зроблено це тому, що *symlink* взагалі ніяк не перевіряє коректність аргументу *oldpath*. Така особливість символічних посилань закладена в них навмисно — сморід можуть посилатися на неіснуючі об'єкти, до яких можна зарахувати об'єкти, що знаходяться у відмонтованих файлових системах. З іншого боку, подібна незалежність від об'єкту посилання має свої недоліки — ядро нічого не робить з символічним посиланням, коли видаляється об'єкт посилання. Під словом «видаляється» мається на увазі видалення жорсткого посилання.

Прочитати зміст символічного посилання можна за допомогою системного виклику *readlink*() .

```
#include <unistd.h>
 ssize_t readlink(
      const char *path,  //шляхове ім'я файлу
      char *buf,     //адреса буфера для читання
      size_t bufsize // розмір буфера
```

```
);
//повертає кількість прочитаних бійтів у випадку
//успіху, NULL у випадку помилки
// (код помилки -в змінній errno)
```

Системний виклик *readlink*() повертає повне ім'я об'єкту посилання без символу «0» наприкінці, що в UNIX служить маркером кінця рядка. Тому як третій аргумент слід передавати число, на 1 менше фактичного розміру буфера, а після здобуття заповненого буфера необхідно додати в нього символ «\0»:

4.2 Знищення файла

Системний виклик *unlink*() видаляє з каталогу посилання на файл і зменшує на 1 лічильник посилань в індексному вузлі. Якщо лічильник посилань дорівнював 0, файлова система видалить файл, після цього дисковий простір і сам індексний вузол стануть доступними для повторного використання. Процес повинен мати право на запис в каталог, з якого видаляється посилання.

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

За допомогою цього виклику можна видалити посилання на файл будь-якого типа, включаючи звичайні файли, сокети. іменовані канали, спеціальні файли і так далі Але видалити посилання на каталог може лише суперкористувач, а в деяких системах навіть йому це буде не під силу. В будь-якому разі, для видалення посилання на каталог слід використовувати системний виклик *rmdir*(), а не *unlink*().

Якщо лічильник заслань в індексному вузлі досяг значення 0, але в системі є процеси, що тримають файл відкритими, файлова система відкладає момент звільнення дискового простору до тих пір, поки останній процес не закриє файл, що видаляється, аби уникнути обвалення працюючих процесів. Ця особливість часто використовується при роботі з тимчасовими файлами, які потрібні лише на час роботи програми, приблизно таким чином:

```
fd =open("temp",0_RDWR|0_CREAT|0_TRUNC|0_EXCL, 0);
unlink("temp");
```

Такий підхід має дві переваги:

- Перша — якщо процес завершує роботу з якої-небудь причини, файл буде видалений автоматично. Завдяки цьому відпадає необхідність реєструвати функцію виходу з програми викликом *atexit*(), аби забезпечити видалення непотрібних тимчасових файлів.
- Друга — оскільки посилання на файл видаляється викликом *unlink*() негайно, зменшується вірогідність обвалення іншого процесу із-за використання прапора 0_EXCL у виклику *open*() і випадкового збігу імені тимчасового файлу. Проте, такий збіг все ще може статися, якщо виклик *open*() другого процесу потрапляє в проміжок між викликами *open*() і *unlink*() першого процесу.

4.3 Отримання інформації о файлі

У цьому розділі ми розглянемо способи здобуття відомостей про файл, таких як власник, час останньої зміни і способи їх правильного відображення.

Індексний вузол містить повний набір метаданих: про файл — все, окрім імені, яке насправді ніяк не пов'язано з індексним вузлом і даними, на які він вказує. Процедура читання даних з індексного вузла прямо з файлу пристрою є дуже низькорівневою. На щастя, для здобуття інформації з індексних вузлів, існують три стандартних системних виклики — *stat*(), *lstat*() і *fstat*().

Системний виклик *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
```

Системний виклик *lstat*() - повертає відомості про стан файлу по його імені без раз іменування символічних посилань

```
#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
```

Виклик *lstat*() аналогічний виклику *stat*() за одним виключенням — якщо як ім'я файлу було передано символічне посилання, то буде

повернена інформація про саме посилання, а не про об'єкт, який воно уособлює.

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

Виклик *fstat*() приймає дескриптор відкритого файлу і відшукує відповідний індексний вузол в таблиці активних вузлів ядра.

Всі три виклики повертають одні і ті ж метадані, через структуру типа *stat*(), в першому аргументі.

Нижче наводиться визначення структури *stat*(). Проте в деяких реалізаціях поля структури можуть бути переставлені місцями, можуть бути додані нові поля

```
struct stat {
dev_t st_dev;  //ідентифікатор пристрою файлової системи
ino_t st_ino;  // номер індексного вузла
mode_t st_mode;     // режим доступу
nlink_t st_nlink;   // кількість жорстких посилань
uid_t st_uid;  // ідентифікатор користувача
gid_t st_gid;  // ідентифікатор групи
dev_t st_rdev;      // ідентифікатор пристрою
                    // (для спеціального файлу)
off_t st_size;      // розмір в байтах
tirne_t st_atime;   /• час останнього звернення
time_t st_mtlrae;   // час останньої зміни
time_t st_ctime; //час останньої зміни індексного вузла
blksize_t st_blksize;    // оптимальний розмір блоку
                         // для операцій введення-виводу
blkcnt_t st_blocks;      // кількість блоків по 512 байти
```

Деякі зауваження по цій структурі.

- Ідентифікатор пристрою (типа *dev_t*) — це число, унікальне для кожної змонтованої файлової системи, навіть для NFS-томів. Таким чином, комбінації *st_dev* і *st_ino* однозначно ідентифікують будь-який індексний вузол в системі.— власне пристрій. Як правило, молодший номер пристрою — це самий молодший байт.
- Поле *st_dev* — це пристрій, на якому знаходиться індексний вузол. Поле *st_rdev* використовується лише для спеціальних файлів пристроїв і є пристроєм, який спеціальний файл представляє.

- Поле *st_size* інтерпретується залежно від типа файлу і реалізації. Для звичайних файлів, каталогів і символічних посилань — це об'єм даних на диску (для символічних посилань — довжина дороги). Для об'єктів в пам'яті, що розділяється, — об'єм займаної пам'яті. Для каналів — кількість даних в каналі.
- Час останнього звернення (*st_atime*) змінюється при читанні файлу.
- Час останньої зміни (*st_mtime*) оновлюється при записі даних у файл і при додаванні або видаленні жорстких посилань для каталогів.
- Час останньої зміни індексного вузла (*st_ctime*) інколи називається як «час останньої зміни статусу». Оновлюється при записі даних у файл або при явній зміні інформації в індексному вузлі (наприклад, при зміні власника файлу або при зміні лічильника посилань), але не оновлюється при читанні даних з файлу.
- Поле *st_blksize* призначене для зміни розміру блоку файлу. В більшості випадків, ймовірно, має те ж призначення, що і аналогічне поле в суперблоці.
- Якщо файл має «діри», отримані як результат установки поточної позиції у файлі за його межі і подальшому записі яких-небудь даних, значення *st_blocks* * 512 може виявитися менше значення *st_size*.
- Системний виклик *fstat*() особливо зажадався в ситуаціях, коли потрібно отримати відомості про елемент, що не має імені у файловій системі, наприклад, про неіменований канал або сокет. Для цих об'єктів поля *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime* і *st_mtime* містять актуальні значення, але що стосується інших полів — все залежить від реалізації. Проте, для іменованих і неіменованих каналів поле *st_size*, як правило, містить кількість непрочитаних байт в каналі.

Поле *st_mode* містить біти, що визначають типа файлу (звичайний файл, каталог, сокет і ін.), права доступу і ряд інших характеристик. Для більшої мобільності додатків, при аналізі цього поля рекомендується користуватися спеціальними макросами. Спершу розглянемо макроси, за допомогою яких можна визначити типа файлу.

```
S_IFMT       /* всі біти, які визначає тип файлу */
S_IFBLK     /* спеціальний файл блокового пристрою */
S_IFCHR     /* спеціальний файл символьного пристрою */
S_IFDIR     /* каталог */
S_IFIFO     /* іменований або неіменований канал */*
S_IFLNK     /« символічне Посилання */
S_IFREG     /* звичайний файл »/
```

```
S_IFSOCK /* сокет */
```

Макрос S_IFMT — це маска всіх бітів, які відносяться до типа файлу в полі *st_mode*. Інші комбінації бітів типа файлу є не маскою, а значенням, тому наступний варіант перевірки — чи є файл сокетом, виконаний неправильно:

```
if ((buf.st_mode & S_IFSOCK) == S_IFSOCK) /* невірно */
```

Подібного роду перевірки повинні виконуватися так:

```
if  ((buf.st_mode & S_IFMT)  == S_IFSOCK)
```

А ще краще використовувати для цих цілей спеціальні макроси, які повертають 0 в разі невідповідності і ненульове значення, — в разі відповідності заданому типові:

```
S_ISBLK(mode) //є спеціальним файлом блокового пристрою
S_ISCHR(mode) //є спеціальним файлом символьного пристрою
S_ISDIR(mode) //є каталогом
S_ISFIFO(mode) //є іменованим або неіменованим каналом
S_ISLNK(mode)  //є символічним посиланням
S_ISREG(roode) //є звичайним файлом
S_ISSOCK(mode) //є сокетом
```

Перевірка — чи є файл сокетом, за допомогою макросу може бути виконана так:

```
if (S_ISSOCK(buf.st_mode))
```

Приклад. 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.

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;
```

```
}
```

4.4 Питання для самоперевірки

1. Яким чином можна дата файлу декілька імен?
2. Як створити жорстке посилання? При яких умовах це можливе?
3. Які обмеження на використання жорстких посилань?
4. Чи можна отримати висяче жорстке посилання?
5. Як Ви думаєте, чому системний виклик *symlink*() не перевіряє коректність путнього імені цільового файлу?
6. Як зберігається символічне посилання?
7. Як прочитати зміст символічного посилання?
8. Чи можна створити циклічне символічне заслання?
9. У якому випадку при видаленні посилання звільнюється файловий простір?
10. Як ви думаєте, чому системний виклик *unlink*() не слід символічному засланню?
11. Що відбувається в випадку, коли видаляється файл, що відкрито якимсь процесом?
12. Що таке метадані файлів? Яким чином їх можна прочитати?
13. Які метадані файлу дозволяє взнати функція *stat*()?
14. Чому для отримання метаданих система має три системних виклики?
15. Як визначити типа файлу?
16. Чому переважніше узнавати довжину відкритого файлу за допомогою системного виклику *fstat*(), а не за допомогою системного виклику *lseek*()?

5 РОБОТА З КАТАЛОГАМИ

5.1 Каталоги. Реалізація каталогів

Нагадаємо, що каталоги є файлами, з яких будується ієрархічна структура файлової системи.

Насправді каталоги UNIX – не більше ніж файли. У багатьох аспектах система поводиться з ними точно так, як і із звичайними файлами. Вони мають власника, групу, розмір і пов'язані з ними права доступу.

У ОС UNIX атрибути файлів розміщуються в спеціальних таблицях, а каталог містять лише посилання на ці таблиці. Такий підхід реалізований, наприклад, у файловій системі ufs ОС UNIX. У цій файловій системі структура каталогу дуже проста. Запис про кожен файл містить коротке символьне ім'я файлу і покажчик на індексний дескриптор файлу, так називається таблиця, в якій зосереджені значення атрибутів файлу.

У кожному каталозі завжди присутні два імена файлів: крапка (.) і подвійна крапка (..). Крапка є стандартною для системи UNIX способом позначення поточного робочого каталогу.

Фактично імена «крапка» (.) і «подвійна крапка» (..) просто є посиланнями на поточний робочий каталог і батьківський каталог відповідно, і будь-який каталог UNIX містить по-перше двох позиціях ці два імена. Іншими словами, під час створення каталогу в нього автоматично додаються ці два імена.

5.2 Права доступу к каталогам.

Так само як і із звичайними файлами, з каталогами пов'язані права доступу, що визначають можливість доступу до них різних користувачів.

Права доступу до каталогів організовані точно так, як і права доступу до звичайних файлів, розбиті на три групи бітів *rwx*, що визначають права власника файлу, користувачів з його групи і всіх останніх користувачів системи.

Права доступу до каталогу мають наступний сенс:

- право читання дає процесам можливість читати дані з каталогу;

- право запису дозволяє процесу створювати нові записи в каталозі або видаляти старі (за допомогою системних операцій *creat*(), *mknod*(), *link*() і *unlink*()), внаслідок чого змінюється вміст каталогу;
- право виконання дозволяє процесу проводити пошук в каталогу по імені файлу (оскільки "виконувати" каталог безглуздо).

5.3 Використання каталогів при програмуванні

Для роботи з каталогами існує особливе сімейство системних викликів. Головним чином ці виклики працюють із структурою *dirent*, яка містить наступні елементи:

```
struct dirent {
  ino_t  d_ino;       // Номер індексного дескриптора
  off_t  d_off;       // відстань до наступного елементу
  unsigned short d_reclen;    // длина запису
  unsigned char d_type;    // тип файлу; не підтримується
                           //більшістю реалізацій
  char    d_name[256]; /* Ім'я файлу
};
```

Специфікація XSI не визначає розмір *d_name*, але гарантує, що число байтів, передуючих нульовому символу, буде менше, ніж число, що зберігається в змінній _PC_NAME_MAX, визначеною в заголовному файлі <*unistd.h*>. Звернете увагу, що нульове значення змінної *d_ino* позначає порожній запис в каталозі.

5.4 Створення та знищення каталогів.

Для створення каталогу використовується системний виклик *mkdir*().

```
#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
```

Остаточні права доступу до каталогу, що створюється викликом *mkdir*(), визначається відповідно до маски *umask*, аналогічно створенню

файлу викликом *open*(). Посилання на каталоги «.» і «..» створюються автоматично.

Для видалення каталогу використовується системний виклик *rmdir*().

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

Виклик *rmdir*() багато в чому схожий на *unlink*(). Єдине обмеження — перед видаленням каталог має бути порожнім. Якщо каталог не порожній, то спочатку доведеться видалити всі вкладені підкаталоги і файли серією викликів *unlink*() і *rmdir*().

5.5 Питання для самоперевірки

1. Для чого потрібні каталоги?
2. Що таке каталог? Які дані зберігаються в запису каталогу?
3. Що таке порожній каталог?
4. Як перейти в батьківський каталог?
5. Як трактуються права доступу до каталогу?
6. Як створити каталог?
7. Які обмеження на створення каталогу є у системі?
8. Чому для видалення каталогу застосовується власний системний виклик?

6 ОББІГ ДЕРЕВА КАТАЛОГІВ

6.1 Відкриття та закриття каталогів. Читання каталогів

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*(), яка створює в програмі дескриптор каталогу, що використовується як посилання на відкритий каталог при виконанні необхідних операцій:

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

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 функціонування інших функцій, що виконують операції над директорією, і позиціонує потік на першому записі директорії.

При успішному завершенні функція повертає покажчик на відкритий потік каталогу, який надалі передаватиметься як параметр всім іншим функціям, що працюють з цим каталогом. При невдалому завершенні повертається значення NULL.

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

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

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

Тип даних *struct dirent* є деякою структурою даних, що описує один запис в директорії. Поля цього запису варіюються від однієї файлової системи до іншої, але одне з полів, яке власне і нас цікавитиме, завжди

присутнє в ній. Це поле *char d_name*[ ] невизначеної довжини, що не перевищує значення NAME_MAX+1, яке містить символьне ім'я файлу, що завершується символом кінця рядка. Дані, що повертаються функцією *readdir*(), переписуються 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 каталогу повертається значення NULL.

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

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

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 май файлів каталогу, не закриваючи його. Єдиним параметром цієї функції є дескриптор відкритого каталогу.

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

При читанні вмісту каталогу необхідно дотримуватися такої послідовності дій.

1. Викличте функцію *opendir*(), передавши їй шляхове ім'я необхідного каталогу.

2. Послідовно викликайте функцію *readdir*(), передаючи їй дескриптор, отриманий від функції *opendir*(). Всякий раз функція *readdir*() повертатиме покажчик на структуру типа *struct dirent*, що містить інформацію про наступний елемент каталогу. Після досягнення кінця каталогу буде набуто значення NULL.

3. Викличте функцію *closedir*(), передавши їй наявний дескриптор, щоб завершити сеанс роботи з каталогом.

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

В цій програмі нам знадобляться дві змінних:

- дескриптор каталогу *dp* типу DIR;
- змінна *ep* типу *struct dirent* для считування 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;
}
```

6.2 Зміна робочого каталогу

Для зміни поточного каталогу передбачено два системних виклики: *chdir*() і *fchdir*().

Аргумент системного виклику *chdir*() може бути і абсолютним, і відносним маршрутом до каталогу.

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

Системний виклик *chdir*() завершиться невдачею і поверне значення *-1*, якщо дорога *path* не є коректним ім'ям каталогу або якщо зухвалий процес не має доступу на виконання (проходження) для всіх каталогів в дорозі.

Системний виклик може успішно використовуватися, якщо потрібно дістати доступ до декількох файлів в заданому каталозі. Зміна каталогу і завдання імен файлів відносно нового каталогу буде ефективнішою, ніж використання абсолютних імен файлів. Це пов'язано з тим, що системі доводиться по черзі перевіряти всі каталоги в дорозі, поки не буде знайдено шукане ім'я файлу, тому зменшення числа складових в дорозі файлу заощадить час.

Системний виклик *fchdir*() приймає як аргумент дескриптор, відкритий для каталогу.

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

6.3 Виявлення імені поточного робочого каталогу

Для виявлення імені поточного робочого каталогу використовується функція *getcwd*(). Це дуже простій системний виклик, який служить для здобуття імені поточного каталогу.

```
#include  <unistd.h>
char  *getcwd(
  char *buf,   // буфер для повернення ымены
  size_t bufsize   //розмір буфера
);
// Повертає показчик на початок повного імені,
// або NULL    в випадку помилки
```

Єдина складність — визначити, якої величини має бути буфер. Але вона легко вирішується шляхом передачі NULL в якості першого параметру і 0 - в якості другого.

6.4 Оббіг дерева каталогів

Інколи необхідно виконати операцію над ієрархією каталогів, почавши від стартового каталогу, і обійти всі лежачі нижче файли і підкаталоги.

При реалізації оббігу каталогів корисно створення закладок на каталоги. Порівняєте наступні дві методики:

1. Отримати повне ім'я поточного каталогу. 2. Зробити що-небудь, що змінить поточний каталог. 3. Викликати *chdir*(), використовуючи ім'я каталогу, отримане на першому кроці.	1. Відкрити поточний каталог викликом open. 2. Зробити що-небудь, що змінить поточний каталог. 3. Викликати *fchdir*(), використовуючи дескриптор, що був отриманий на першому кроці.

Методика, яка приведена в лівому списку, декілька поступається тій, що приведена справа, тому що:

- є ряд складнощів із здобуттям імені поточного каталогу.
- це досить трудомісткий процес.

В деяких випадках, коли вам необхідно повернутися лише на один рівень вище, можна використовувати такий підхід:

```
chdir("..")
```

Приклад. Написати програму, яка видаляє із заданого каталога і всіх його підкаталогів файли з ім'ям «*core*». Ім'я каталога задавати у вигляді аргументу командного рядка.

Для того, щоб вирішити це завдання, необхідно уміти читати вміст каталога. Для цього використовуватимемо функції стандартної бібліотеки. Спочатку потрібно відкрити каталог за допомогою функції *opendir*(), потім, послідовно викликаючи функцію *readdir*(), прочитувати записи каталога, а в кінці закрити каталог за допомогою функції *closedir*().

Видалення файлів з каталога виконуватиме функція *delNotPointFiles*(), параметром якої є ім'я файлу. Ця функція викликатиметься рекурсивно, аби забезпечити видалення файлів з ім'ям *core* в підкаталогах.

Для перевірки того, що файл є каталогом написана функція *isDir*(). Для визначення типа файлу дана функція використовує системний виклик *stat*(). Така функція потрібна для організації входу в підкаталоги.

Нижче наводиться текст програми.

```
#include <dirent.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>

int delNotPointFiles(const char*);
int isDir(const char *);

int main(int argc, char * argv[])
{
   char *path =".";
   if (argc>=2)  path = argv[1];

   return delNotPointFiles(path);
}

int delNotPointFiles(const char* path)
{
    DIR* dir;
    struct dirent* entry;
    printf("path=%s\n",path);
    dir=opendir(path);
    chdir(path);
    while ((entry = readdir(dir)) != NULL)
    {
       if(isDir(entry->d_name)&&
           strcmp(entry->d_name,".")&&
           strcmp(entry->d_name,".."))
          delNotPointFiles(entry->d_name);
       else
            if (entry->d_name[0]!='.'  )
          unlink(entry->d_name);
       free(entry);
    }
    chdir("..");
    return 0;
}

int isDir(const char *path)
{
      struct stat st;
      stat(path,&st);
```

```
        return S_ISDIR(st.st_mode);
}
```

6.5 Питання для самоперевірки

1. Яка функція використовується для відкриття каталогу?
2. За допомогою якої структури можна отримати запис каталогу?
3. Як повернутися до початку каталогу?
4. Як змінити поточний каталог?
5. Як отримати ім’я поточного каталогу?
6. Що таке закладка?
7. Якы складнощы виникають при виконанні операцій над ієрархією каталогів?

7 ОСНОВИ УПРАВЛІННЯ ПРОЦЕСАМИ

Процес – це фундаментальне поняття ОС UNIX. Під процесом розуміється програма під час її виконання. Процес є єдиним активним єством в системі UNIX. Процес - унікальним чином програма, що ідентифікується, яка потребує діставання доступу до ресурсів комп'ютера.

Кожен процес запускає одну програму і спочатку отримує один потік управління. Більшість версій UNIX дозволяють процесу після того, як він запущений, створювати додаткові потоки. UNIX – багатозадачна система, так що декілька незалежних процесів можуть працювати одночасно.

7.1 Типи процесів:

У ОС UNIX виділяється три типи процесів:

- системні
- демони
- прикладні процеси.

Системні процеси є частиною ядра і завжди розташовані в оперативній пам'яті. Системні процеси не мають відповідних ним програм у вигляді виконуваних файлів і запускаються особливим чином при ініціалізації ядра системи. Виконувані інструкції і дані цих процесів знаходяться в ядрі системи, таким чином, вони можуть викликати функції і звертатися до даних, недоступних для останніх процесів.

До системних процесів можна віднести і процес початкової ініціалізації, *init*. Хоча *init* не є частиною ядра, його робота життєво важлива для функціонування всієї системи в цілому.

Демони - це не інтерактивні процеси, які запускаються звичайним способом, - шляхом завантаження в пам'ять відповідних ним програм, і виконуються у фоновому режимі. Зазвичай демони запускаються при ініціалізації системи, але після ініціалізації ядра і забезпечують роботу різних підсистем UNIX: системи термінального доступу, системи друку, мережевих служб і так далі Демони не пов'язані ні з одним користувачем. Велику частину часу демони чекають, поки той або інший процес запитає певну послугу.

Типовим демоном є *cron*. Цей демон планує активність системи на хвилини, години, дні і місяці вперед. Він пробуджується кожну хвилину,

перевіряючи, чи не потрібно чого зробити. Якщо у нього є робота, він її виконує і вирушає спати далі.

До прикладних процесів відносяться всі останні процеси, що виконуються в системі. Як правило, це процеси, породжені в рамках призначеного для користувача сеансу роботи. Найважливішим призначеним для користувача процесом є початковий командний інтерпретатор, який забезпечує виконання команд користувача в системі UNIX.

Призначені для користувача процеси можуть виконуватися як в інтерактивному (пріоритетному), так і у фоновому режимах. Інтерактивні процеси монопольно володіють терміналом, і доки такий процес не завершить своє виконання, користувач не має доступу до командного рядка.

7.2 Структури даних процесу

Виконання процесу може відбуватися в двох режимах – ядра і завдання. У режимі завдання виконується основна частина процесу, виконуючи інструкції прикладної програми, допустимі на непривілейованому рівні захисту процесора. При цьому йому недоступні системні структури ядра. Коли процесу потрібне здобуття яких-небудь послуг ядра, він робить системний виклик, який виконує інструкції ядра, що знаходяться на привілейованому рівні. Виконання процесу при цьому переходить в режим ядра.

Відповідно кожен процес в системи UNIX складається з двох частин і представлений двома структурами: призначеною (*user*) для користувача і системною (*proc*).

При завантаженні системи в оперативну пам'ять завантажується системна таблиця процесів, яка залишається там до закінчення роботи. Записами цієї таблиці є структури *proc*. Таблиця процесів є одночасно масивом і двозв’язковим списком у вигляді дерева.

Фізичний опис є статичним масивом покажчиків, довжина якого є константою, Покажчики описані системній змінній *curproc*. Curproc вказує на структуру proc активного процесу і міняється планувальником, коли ресурси процесора передаються іншому процесу. Структура *proc* містить покажчик на дані структури *user*.

Дані *user* розміщуються у визначеному місці віртуальній пам'яті ядра і адресуються системній змінній *u*.

У таблиці процесів зберігається інформація, потрібна всім процесам, навіть тим, яких на даний момент в пам'яті немає.

Структура *proc* складається з наступних компонентів:

- Різне: поточний стан, очікувані події, час до виділення інтервалу будильника, PID і PPID процесу, ідентифікатори користувача і групи.
- Параметри планування: Пріоритети, процесорний час, спожитий за останній період, що враховується, кількість часу, проведену в очікуванні.
- Образ пам'яті: покажчики на сегменти програми, даних і стека або на відповідні таблиці сторінок.
- Сигнали: маски, що вказують, які сигнали ігноруються, які перехоплюються, які тимчасово заблоковані або знаходяться в процесі доставки.

У структурі користувача міститься інформація, яка не потрібна, коли процесу фізично немає в пам'яті і він не виконується. Структура користувача вивантажується на диск, звільняючи місце в пам'яті, аби не витрачати пам'ять на непотрібну в даний момент інформацію.

Структура користувача ділиться на наступні категорії:

- Машинні регістри: заповнюються при перериванні з перемиканням в режим ядра
- Стан системного виклику: інформація про поточний виклик, включаючи параметри і результати
- Таблиця дескрипторів файлів: по дескриптору файлу знаходиться структура даних (*inode*), відповідний даному файлу
- Облікова інформація: покажчик на таблицю використання процесорного часу, максимальний розмір стека, кількість сторінок пам'яті і так далі
- Стек ядра: фіксований стек для використання процесом в режимі ядра.

7.3 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.

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

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 місця 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*().

7.4 Запуск нових програм. Сімейство викликів *exec*

Неможливо зрозуміти системні виклики *exec*() і *fork*() без чіткого розуміння відмінностей між процесом і програмою. Нагадаємо суть відмінностей: процес — це середовище виконання, що включає сегмент коду,

сегменти призначених для користувача і системних даних, а також набір додаткових ресурсів, отриманих але час виконання. Програма — це файл, який містить виконуваний код, сегменти з даними для ініціалізації і з даними користувача.

Системний виклик *exec* повторно ініціалізує процес, підміняючи його вказаною програмою — програма змінюється, а процес залишається. Системний виклик *fork*(), навпаки, запускає новий процес, який є точною копією того, що існує, простим копіюванням сегментів з кодом і даними. Знов створений процес продовжує роботу з того ж місця, що і початковий, таким чином, обидва процеси продовжують виконувати один і той же код.

Окремо один від одного *exec*() і *fork*() використовуються украй рідко. Системний виклик *exec*() — єдиний спосіб запуску програм в UNIX. Мало того, що командна оболонка використовує *exec*() для запуску наших програм, але і сама вона запускається саме такий чином. А системний виклик *fork*() — єдиний спосіб запустити новий процес.

Насправді не існує одного системного виклику *exec*. Під цим ім'ям мається на увазі ціле сімейство з шести системних викликів, імена яких в загальному вигляді можна записати як *execAB*. *A* — це один з символів, «*l*» або «*v*», вони визначають, як вхідні аргументи передаються виклику — у вигляді списку (від англ. *list*) або у вигляді масиву (від англ. *vector*). *B* (може бути відсутнім) — це або «*p*», яка вказує, що пошук файлу програми повинен виконуватися за допомогою змінної PATH, або «*e*» — такому виклику передасться специфічне середовище оточення (як це не дивно, але немає системного виклику *exec*, який поєднував би в собі характерні особливості «*e*» і «*p*»). Таким чином, ми отримуємо шість різних системних викликів: *execl*(), *execv*(), *execlp*(), *execvp*(), *execle*() і *execve*().

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*>.

Функція *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
```

Аргумент *path* повинен містити повне ім'я виконуваного файлу з програмою. Сегмент коду процесу буде «затертий» кодом нової програми, сегмент даних також буде затертий сегментом даних нової програми, а стек буде переініціалізовано. Виконання нової програми почнеться із самого початку (буде викликана функція main()).

В разі успіху повернення з *execl*() не передбачене, тому що точка повернення буде загублена. В разі помилки *execl*() поверне -1, не має сенсу перевіряти це значення, оскільки ніякого іншого отримати не можна. Найбільш часті причини, що наводять до неможливості запустити нову програму, це або відсутність файлу з програмою, або відсутність права на її запуск.

Всі останні аргументи виклику збираються в масив покажчиків на рядки, а останнім завжди повинен стояти порожній покажчик, який визначає кінець списку вхідних аргументів. Перший аргумент, відповідно до угод — це ім'я файлу програми (лише само ім'я файлу, без дороги до нього). Нова програма дістає доступ до цього списку через вже знайомі нам аргументи функції *main*() - *argc* і *argv*. Середовище оточення також передається новій програмі і доступна їй через покажчик environ або за допомогою функції *getenv*().

Оскільки процес продовжує існувати і сегмент з системними даними практично не змінюється, майже всі атрибути процесу також залишаються незмінними, включаючи ідентифікатор процесу, ідентифікатор процесу-предка, ідентифікатор групи процесу, ідентифікатор сесії, термінал, що управляє, реальні ідентифікатори користувача і групи, поточний і кореневий каталоги, пріоритет, статистичні характеристики часу виконання в просторі користувача і в просторі ядра і, як правило, дескриптори відкритих файлів. Набагато простіше перерахувати основні атрибути, які змінюються.

- Якщо процес призначав свої обробники сигналів, то всі вони скидаються у вихідний стан, оскільки функції-обробники після запуску нової програми стануть недоступні.
- Якщо в новій програмі встановлені біти *set-user-ID* або *set-group-ID*, то діючі ідентифікатори користувача і групи встановлюються заново відповідно до ідентифікаторів власника і групи файлу програми. Немає жодного способу повернути колишні діючі ідентифікатори, якщо вони відрізняються від реальних.
- Реєстрація всіх функцій, яка була виконана за допомогою *atexit*(), відміняється, оскільки код цих функцій буде затертий.
- Сегменти спільної пам'яті від'єднуються, оскільки точки з’єднання будуть загублені.
- Іменовані семафори POSIX закриваються. Семафори System V залишаються без змін.

Функція *execv*() запускає програму, вхідні аргументи передаються у вигляді масиву. Це абсолютно необхідно, коли число аргументів у момент написання програми (наприклад, командного інтерпретатора) заздалегідь невідоме.

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

Функція виконує файл, заданий рядком *path* 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.

Функція *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
```

Ідентична *execv*() за винятком того, що процесу передаються змінні середовища в аргументі *envv*.

Функція *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
);
```

```
// повертає -1 у випадку помилки
```

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

Функція *execvp*() запускає програму, аргументи передаються у вигляді масиву, пошук файлу ведеться з використанням змінної оточення PATH.

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

Ідентична *execv*() з тією різницею, що намагається знайти команду з таким ім'ям в каталогах, вказаних в змінній оточення PATH, якщо *file* не містить жодного символу слеша.

Функція *execlp*() запускає програму, аргументи передаються у вигляді списку, пошук файлу ведеться з використанням змінної оточення 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
```

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

Якщо аргумент *file* у викликах *execlp*() або *execvp*() не містить символ «/», то виконується послідовна підстановка імен каталогів із змінної PATH і проводиться пошук файлу з правом на виконання. Якщо такий файл знайдений і він є виконуваною програмою (перевіряється сигнатура файлу), то вона запускається на виконання. Якщо немає, то робиться припущення про те, що файл є сценарієм і викликається командний інтерпретатор, якому передається дорога до файлу в першому аргументі.

7.5 Завершення виконання процесу

Процес закінчує роботу в чотирьох випадках:

1. При зверненні до системного виклику *exit*(). Повернення значення з функції main еквівалентне виклику *exit*() з тим же самим значенням. Вихід з функції main без повернення значення рівносильний поверненню значення 0.

2. При зверненні до *_exit*() або *_Exit*() — двом різновидам системного виклику exit, які будуть описані нижче.

3. При отриманні сигналу на завершення.

4. В разі краху системи, причиною якого може стати все що завгодно, паю від перебоїв в мережі електроживлення і закінчуючи помилкою в або в операційній системі.

При завершенні процесу послідовно виконуються наступні дії:

- Відключаються всі сигнали.
- У процесі, що викликав, закриваються всі дескриптори відкритих файлів.
- Якщо батьківський процес знаходиться в стані виклику *wait*(), то системний виклик *wait*() завершується, видаючи батьківському процесу як результат ідентифікатор процесу, що завершився, і молодші 8 біт його коди завершення.
- Якщо батьківський процес не знаходиться в стані виклику *wait*(), то процес, що завершується, переходить в стан зомбі.
- У всіх існуючих нащадків завершених процесів, а також в зомбі-процесів ідентифікатор батьківського процесу встановлюється рівним 1. Таким чином, вони стають нащадками процесу ініціалізації (*init*).
- Якщо ідентифікатор процесу, термінальна лінія і ідентифікатор групи процесів в процесу, що завершується, збігаються, то всім процесам з тим же ідентифікатором групи процесів посилається сигнал SIGHUP. Тим самим, завершуються і всі породжені в пріоритетному режимі процеси.
- Батьківському процесу посилається сигнал SIGCHLD (завершення породженого процесу). Цей сигнал будить батьківський процес, якщо той чекає завершення породжених процесів.
-

Для завершення процесу використовуються наступні системні виклики:

- *_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
```

7.6 Питання для самоперевірки

1. Що таке процес? Які атрибути процесу є найважнішими?
2. Які атрибути дочірнього процесу відрізняються від атрибутів батьківського процесу?
3. В чому різниця між реальним та ефективним ідентифікаторами користувача?
4. Який процес вважається прикладним?
5. Яким чином може бути породжений новий процес? Яка структура нового процесу?
6. Як процес може довідатися, є він дочірнім чи батьківським?
7. Яким чином процес може змінити програму, що виконується?
8. У чому різниця між різними формами системних викликів типа *exec*()?
9. Яким чином процес завершує свою роботу?
10. Які процеси прийнято називати процесами-зомбі?

8 СИНХРОНІЗАЦІЯ ПРОЦЕСІВ

8.1 Системний виклик *wait.* Очікування завершення конкретного нащадка: виклик *waitpid*

Системні виклики *wait*() та *waitpid*() чекають, доки дочірній процес не змінить свій стан (призупинення, відновлення або завершення) і повертають його програмі, що викликає. Почнемо обговорення з системного виклику *waitpid*().

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

Аргумент *pid* може набувати наступних значень:

>0	Чекати зміну стану дочірнього процесу з вказаним ідентифікатором.
-1	Чекати зміну стану будь-якого дочірнього процесу.
0	Чекати зміну стану будь-якого дочірнього процесу, що належить до тієї ж групи процесів, що і що викликає.
< -1	Чекати зміну стану будь-якого дочірнього процесу, що належить до групи процесів з ідентифікатором *-pid*.

На виході з *waitpid*() процес, отримує ідентифікатор процесу-нащадка, чий ідентифікатор збігся з аргументом *pid*. Нуль повертається лише у тому випадку, коли був встановлений прапор WNOHANG).

Допускається чекати зміни стану лише прямих нащадків, породжених системним викликом *fork*(). Очікування процесів «онуків» не припускається, навіть якщо їх батьки (прямі нащадки процесу, що викликає) до моменту виклику вже завершили свою роботу.

Як правило, процеси-предки зацікавлені в здобутті інформації про стан своїх нащадків — інакше процеси-нащадки після закінчення перетворюються на «зомбі» і перебувають у такому вигляді, поки не завершить роботу батьківський процес. Враховуючи, що багато процесів

можуть виконуватися в системі досить тривалий час (до декількох місяців), така «неувага» до дочірніх процесів може привести до переповнення системних таблиць. Якщо очікуванняння нащадків неможливе, по тих або інших причинах, процес може використовувати сигнали для запобігання «зомбуванню».

Системний виклик *waitpid*() може повернути стан що змінив його дочірнього процесу лише один раз (такий дочірній процес називається очікуваний). Або іншими лише один раз словами: очікуваний нащадок перестає бути очікуваним, якщо звіт про зміну його стану вже отриманий. Це означає наступне: якщо в одній точці програми було отримано багатство нащадка і раптом виявилося, що це не той нащадок, якого чекали, то немає жодного способу повернути

Аргумент *options* може містити один або більш за прапори, об'єднаних операцією АБО:

WCONTINUED	Повідомляти про відновлення роботи нащадком
WNOHANG	Не чекати зміни стану нащадка. Якщо воно ще не змінилося — повертати значення 0.
WUNTRACED	Повідомляти про призупинення роботи нащадка.

Якщо до моменту виклику *waitpid*() є очікуваний дочірній процес, відповідний заданому аргументу *pid*, управління в програму повертається негайно. Якщо нащадок знайдений, але ще не змінив свій стан, системний виклик *waitpid*() блокується до появи відповідного нащадка. Якщо нащадок не був знайдений, програмі повертається значення -1 і код помилки ECHILD. Це може статися тому, що аргумент *pid* заданий неправильно або процес-нащадок перестав бути очікуваним, тобто його стан вже було отриманий.

Якщо як аргумент *statusp* переданий непорожній покажчик, то за заданою адресою записується код стану нащадка. Він є комбінацією аргументу системного виклику *_exit*() або *exit()* (якщо йдеться про завершенні нащадка) і числа, що описує причину завершення або призупинення.

Макроси, що вживаються для аналізу комбінації:

WIFEXITED(status)	*true*, якщо нащадок завершив роботу звичайним способом (зверненням до _exit або exit).
WEXITSTATUS(status)	Якщо WIFEXITED повернув *true*, молодші 8 бітом є аргумент виклику _exit() або exit().
WIFSIGNALED(status)	*true*, якщо нащадок завершився аварійно (по сигналу).
WTERMSIG(status)	Повертає номер сигналу, що викликав аварійне завершення нащадка, за умови, що WIFSIGNALED повернув *true*.

WlFSTOPPED(status)	true, якщо нащадок припинений. Можливо лише в разі установки прапора WUNTRACED).
WSTOPSIG(status)	Номер сигналу, який викликав призупинення нащадка, за умови, що WIFSTOPPED повернув true.
WIFCONTINUED(status)	true, якщо виконання нащадка було відновлене. Можливо лише в разі установки прапора WC0NTINUED.
WCOREDUMP(status)	true, якщо був створений файл з дампом пам'яті (званий в UNIX «*Core dump*» — дамп ядра). Цей файл може використовуватися при пошуку причин, що викликали «звалювання» процесу (макрос нестандартний, але він присутній в більшості систем).

Декілька прикладів, що використовують системний виклик *waitpid*():

1. Чекати завершення нащадка *pid* і отримати код завершення:

```
waitpid(pid &status, 0);
```

2. Чекати завершення будь-якого з нащадків без здобуття коду завершення:

```
pid = waitpid(-1, NULL, 0);
```

3. Отримати від будь-якого з нащадків по груповому ідентифікатору *pgid* повідомлення про завершення або про призупинення і отримати код стану. Не чекати якщо нащадок ще не змінив стан.

```
pid = waitpid(-pgid &status, WNOHANG | WUNTRACED);
```

Другий представник групи системних викликів *wait*() є спрощеним варіантом *waitpid*() і еквівалентний виклику останнього з аргументом *pid*, рівним -1, і з аргументом *options* рівним нулю:

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

Системний виклик *wait*() досить рідко використовується у великих застосуваннях, оскільки чекає завершення будь-якого нащадка. Річ у тому, що коли деяка функція, що є частиною великого застосування, створює дочірній процес і намагається його почекати, вона може випадково «діждатися» завершення абсолютно іншого нащадка, вносячи безлад і

сум'яття до ходу виконання всього застосування в цілому. Для таких випадків *waitpid*() личить набагато краще, оскільки він дозволяє чекати конкретний процес або, по крайньому заходу, члена групи процесів. Ніколи не використовуйте *wait*() при написанні бібліотечних функцій, які створюють дочірні процеси.

8.2 Зомбі-процеси и передчасне завершення програми

При завершенні процесу повинна віддалятися його структура із списку процесів. Інколи процес вже завершився, але його ім'я ще не видалене із списку процесів. В цьому випадку процес стає «зомбі» — порожнім записом в таблиці процесів, що зберігає код завершення для батьківського процесу. Таке може статися, якщо процес-нащадок (дочірній процес) завершився раніше, ніж цього чекав процес-батько.

Система повідомляє батьківський процес про завершення дочірнього за допомогою сигналу SIGCHLD. Передбачається, що після здобуття SIGCHLD він рахує код повернення за допомогою системного виклику *wait*(), після чого запис зомбі буде видалений із списку процесів.

Якщо батьківський процес ігнорує SIGCHLD (а він ігнорується за умовчанням), то зомбі залишаються до його завершення.

Ігнорування SIGCHLD в принципі не є правильним, але може бути прийнятне для короткотривалих програм (деякі програми можуть робити це навмисно, наприклад, для виключення повторення PID). Але для довготривалих і часто створюючих дочірні процеси програм це неприйнятно, тому що накопичення зомбі наводить до «витоку ресурсів» (тобто до їх поступового блокування).

Зомбі не займають пам'яті (як процеси-сироти), але блокують записи в таблиці процесів, розмір якої обмежений для кожного користувача і системи в цілому.

Процес стає зомбі тоді, коли він вже завершився, а в його батьківському процесі не була викликана функція *wait*(). Функції *wait*() і *waitpid*() призначені для здобуття батьківським процесом коди завершення його нащадка. У випадку якщо нащадок вже завершився, всі системні ресурси, займані процесом, будуть звільнені, а функція негайно поверне значення *pid* нащадка.

Розібравшись з тим, що є зомбі, варто ознайомитися з деякими способами усунення створюваної зомбі проблеми.

Один з найпростіших способів «убити» зомбі – це «убити» їх батька. Якщо в системі «вмирає» який-небудь процес, то спеціальний демон *init* наслідує всіх нащадків померлого процесу і видаляє їх, якщо вони вже

завершили своє виконання. Але у нас може не хапати прав на видалення батька. Можлива ситуація, коли батько виконує якісь необхідні нам дії, і, видаливши його, можна втратити дані.

Вочевидь, що краще запобігти зомбі, ніж з ними боротися. Одним з рішень є використання вищезазначеної функції *wait*().

8.3 Питання для самоперевірки

1. Що станеться, якщо процес-нащадок завершиться раніше, ніж процес-предок здійснить системний виклик *wait*()?
2. В чому перевага системного виклику *waitpid*() перед системним викликом *wait*()?
3. Як система повідомляє батьківський процес про завершення дочірнього процесу?
4. Які макроси використовуються для аналізу статусу нащадка?
5. При яких умовах процес набуває стану „зомбі”?
6. Чому необхідно запобігати „зомбуванню” процесів?

9 АТРИБУТИ ПРОЦЕСУ

9.1 Основні атрибути

Процес в UNIX має ряд атрибутів, що дозволяють операційній системі управляти його роботою. Основні атрибути:

1. Ідентифікатор процесу (PID)
2. Ідентифікатор батьківського процесу (PPID)
3. Поточний пріоритет PRI
4. Поправка пріоритету (NICE) – відносний пріоритет
5. Термінальна лінія (TTY)
6. Реальний (UID) і ефективний (EUID) ідентифікатори користувача
7. Реальний (GID) і ефективний (EGID) ідентифікатори групи

1) Кожен процес має унікальний ідентифікатор PID, що дозволяє ядру системи розрізняти процеси. Коли створюється новий процес, ядро привласнює йому наступний вільний ідентифікатор. Привласнення ідентифікатора зазвичай відбувається по той, що зростає, тобто ідентифікатор нового процесу більший, ніж ідентифікатор процесу, створеного перед ним. Якщо ідентифікатор досягає максимального значення (зазвичай - 65737), наступний процес отримає мінімальний вільний PID і цикл повторюється. Коли процес завершує роботу, ядро звільняє ідентифікатор, що використався ним.

Для отримання ідентифікатору процесу використовується системний виклик *getpid*().

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

2) PPID - ідентифікатор процесу - породжувача даного процесу. Всі процеси в системі, окрім системних процесів і процесу *init*, породжені одним з існуючих або існуючих раніше процесів.

Для отримання ідентифікатора батьківського процесу використовується системний виклик *getppid*().

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

```
pid_t getppid(void);
// повертає ідентифікатор батьківського процесу
```

3) Розподіл процесорних ресурсів визначається пріоритетом виконання PRI.

4) Відносний пріоритет процесу, що враховується планувальником при визначенні черговості запуску. 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.

5) Термінал або псевдотермінал, пов'язаний з процесом. З цим терміналом за умовчанням пов'язані стандартні потоки: вхідний, вихідний і потік повідомлень про помилки. Процеси-демони не пов'язані з терміналом.

6) 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 привласнюється значення ідентифікатора власника виконуваного файлу (наприклад, користувача *root*).

7) 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.

Далі наводиться група системних викликів, за допомогою яких можна взнати реальні і діючі ідентифікатори користувача і групи:

Системний виклик *getuid*() повертає реальний ідентифікатор користувача.

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

Системний виклик *geteuid*() — повертає ефективний ідентифікатор користувача.

```
#include <unistd.h>
uid_t geteuid(void);
```

```
// повертає ідентифікатор користувача(коди помилок
//не передбачені
```

Системний виклик *getgid*() — повертає реальний ідентифікатор групи.

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

Системний виклик *getegid*() — повертає ефективний ідентифікатор групи.

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

Ідентифікатор користувача або групи — це просто деяке ціле число.

Нижче наводиться приклад програми, яка виводить реальні і діючі ідентифікатори користувача і групи:

```
int main(void)
{
   uid_t uid;
   gid_t gid;
   struct passwd *pwd;
   struct group *grp;
   uid = getuid();    pwd = getpwuid(uid;)
   printf("Реальний користувач = %ld (%s)\n",
          (long)uid, pwd->pw_name);
   uid = geteuid(); pwd = getpwuid(uid);
   printf("Ефективний користувач = %ld (%s)\n",
          (long)uid, pwd->pw_name);
   gid = getgid(); grp = getgrgid(gid);
   printf("Реальна група = %ld (%s)\n",
          (long)gid, grp->gr_name);
   gid = getegid();grp = getgrgid(gid);
   printf("Ефективна група = %ld (%s)\n",
   (long)gid, grp->gr_name);
   exit(0);
}
```

Правила зміни реального і ефективного ідентифікаторів користувача і групи досить складні і розрізняються для звичайних процесів і процесів, що працюють з правами суперкористувача. Відмітимо, що окрім поточних реальних і діючих ідентифікаторів, для кожного процесу ядро зберігає

оригінальні ідентифікатори, що діють, які були встановлені останнім викликом *exec*. Вони називаються збережені ідентифікатори.

Нижче наводяться правила зміни ідентифікатора користувача (ці ж правила застосовні і для ідентифікатора групи):

1. Звичайний процес (що не володіє вдачами суперкористувача) не може змінити реальний або збережений ідентифікатори інакше як через виклик *exec* (який може змінити збережений ідентифікатор).
2. Звичайний процес може змінити ідентифікатор, що діє, на реальний або на збережений.
3. Процес, що володіє правами суперкористувача, може змінити реальний і діючий ідентифікатори на будь-якій іншій.
4. Коли призначений для суперкористувача процес змінює реальний ідентифікатор, збережений ідентифікатор набуває того ж значення.

З призначеними для суперкористувача процесами все зрозуміло — допускається все, що завгодно. Два правила, що зачіпають звичайні процеси означають наступне: якщо виклик *exec* змінив реальний і ефективний ідентифікатори, то процес має можливість перемикатися між ними.

А тепер — самі системні виклики.

Системний виклик *setegid*() встановлює ефективний ідентифікатор групи.

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

Системний виклик *seteuid*() встановлює ефективний ідентифікатор користувача

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

Можна стверджувати, що в разі процесу з правами звичайного користувача, аргумент має дорівнювати реальному або збереженому ідентифікатору. Для процесу з правами суперкористувача, аргумент може відповідати ідентифікатору будь-якого користувача.

У розпорядженні суперкористувача є два додаткові системні виклики: Системний виклик 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
```

Системний виклик *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
```

Для суперкористувача, як аргумент допускається передавати ідентифікатор будь-якого користувача, який стане відразу і реальним і збереженим. Для звичайного користувача ці виклики поводяться аналогічно двом попереднім, що може деколи вводити в оману, тому слід уникати вживання цих викликів.

9.2 Логічна організація процесів

Процеси організовані в логічну структуру, що складається з самих процесів, груп процесів і сеансів. Це спрощує управління процесами.

Кожен процес належить певній групі, яка має унікальний ідентифікатор. Група може мати в своєму складі лідера групи – процес, чий ID рівний ID групи. Процес успадковує групу від батьківського, але може покинути її і організувати власну.

Групи процесів об'єднуються в сеанси. Сеанси описують процеси, породжені користувачем за час роботи в системі. Кожен сеанс може мати один асоційований термінал, званий керівником, а групи і процеси, створені в даному сеансі, успадковують цей термінал. Отже, процеси, що належать до різних сеансів, дістають доступ до різних терміналів. Наявність терміналу, що управляє, дозволяє ядру контролювати стандартний введення/виведення, а також посилати сигнали всім процесам, що асоціюються з терміналом групи.

9.2.1 Групи процесів и ідентифікатори групи процесів. Зміна групи процесу

Група процесів включає один або більш за процеси і існує, поки в групі присутній хоч би один процес. Кожен процес обов'язково включений в яку-небудь групу. При народженні нового процесу він потрапляє в ту ж групу процесів, в якій знаходиться його батько. Процеси можуть мігрувати з групи в групу по своєму бажанню або за бажанням іншого процесу (залежно від версії UNIX). Багато системних викликів можуть бути застосовані не до одного конкретного процесу, а до всіх процесів в деякій групі. Тому те, як саме слід об'єднувати процеси в групи, залежить від того, як передбачається їх використовувати.

Кожна група процесів в системі отримує власний унікальний номер. Взнати цей номер можна за допомогою системного виклику *getpgid*(). Використовуючи його, процес може взнати номер групи для себе самого або для процесу зі свого сеансу.

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

Для переходу процесу в іншу групу процесів, можливо, з одночасним її створенням, застосовується системний виклик *setpgid*(). Перевести в іншу групу процес може або самого себе (і те не у всяку і не завжди), або своє процес-дитя, яке не виконувало системний виклик *exec*, тобто не запускав на виконання іншу програму. При певних значеннях параметрів системного виклику створюється нова група процесів з ідентифікатором, співпадаючим з ідентифікатором процесу, що перекладається, що складається спочатку лише з одного цього процесу. Нова група може бути створена лише в такий спосіб, тому ідентифікатори груп в системі унікальні. Перехід в іншу групу без створення нової групи можливий лише в межах одного сеансу.

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

Параметр *pid* є ідентифікатором процесу, який потрібно перевести в іншу групу, а параметр *pgid* – ідентифікатором групи процесів, в яку належить перевести цей процес.

Не всі комбінації цих параметрів дозволені. Перевести в іншу групу процес може або самого себе (і те не у всяку, і не завжди), або своє процес-дитя, яке не виконувало системний виклик *exec*(), тобто не запускав на виконання іншу програму.

Якщо параметр *pid* дорівнює 0, то вважається, що процес переводить в іншу групу самого себе.

Якщо параметр *pgid* дорівнює 0, то вважається, що процес переводиться в групу з ідентифікатором, співпадаючим з ідентифікатором процесу, визначуваного першим параметром.

Якщо значення, визначувані параметрами *pid* і *pgid*, рівні, то створюється нова група з ідентифікатором, співпадаючим з ідентифікатором процесу, що перекладається, що складається спочатку лише з цього процесу. Перехід в іншу групу без створення нової групи можливий лише в межах одного сеансу.

У нову групу не може перейти процес, що є лідером групи, тобто процес, ідентифікатор якого збігається з ідентифікатором його групи.

9.2.2 Сеанси и ідентифікатор сеансу

Кожен сеанс в системі також має власний номер. Для того, щоб взнати його, можна скористатися системним викликом *getsid*(). У різних версіях UNIX на нього накладаються різні обмеження.

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

Системний виклик повертає ідентифікатор сеансу для процесу з ідентифікатором *pid*. Якщо параметр *pid* дорівнює 0, то повертається ідентифікатор сеансу для даного процесу

Використання системного виклику *setsid* приводить до створення нової групи, що складається лише з процесу, який його виконав (він стає лідером нової групи), і нового сеансу, ідентифікатор якого збігається з ідентифікатором процесу, що зробив виклик. Такий процес називається лідером сеансу. Цей системний виклик може застосовувати лише процес, що не є лідером групи.

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

Цей системний виклик може застосовувати лише процес, що не є лідером групи, тобто процес, ідентифікатор якого не збігається з ідентифікатором його групи.

Системний виклик повертає значення 0 при нормальному завершенні і значення -1 при виникненні помилки.

Якщо сеанс має термінал, що управляє, то цей термінал обов'язково приписується до деякої групи процесів, що входить в сеанс. Така група процесів називається поточною групою процесів для даного сеансу. Всі процеси, що входять до поточної групи процесів, можуть здійснювати операції введення-виводу, використовуючи термінал, що управляє. Всі останні групи процесів сеансу називаються фоновими групами, а процеси, що входять в них, – фоновими процесами. При спробі введення-виведення фонового процесу через термінал, що управляє, цей процес отримає сигнали, які стандартно наводять до припинення роботи процесу. Передавати термінал, що управляє, від однієї групи процесів до іншої може лише лідер сеансу. Відмітимо, що для сеансів, що не мають терміналу, що управляє, всі процеси є фоновими.

При завершенні роботи процесу – лідера сеансу всі процеси з поточної групи сеансу отримують сигнал SIGHUP, який при стандартній обробці приведе до їх завершення. Таким чином, після завершення лідера сеансу в нормальній ситуації роботу продовжать лише фонові процеси.

9.3 Пріоритети процесів: виклик *nice*

Кожен процес має два атрибути пріоритету: поточний пріоритет, на підставі якого відбувається планування, і відносний пріоритет, званий також поправкою пріоритету, - *nice number*, який задається при породженні процесу і впливає на поточний пріоритет.

Діапазон значень поточного пріоритету різний, залежно від версії ОС UNIX і використовуваного планувальника.

У будь-якому випадку, процеси, що виконуються в призначеному для користувача режимі, мають нижчий пріоритет, ніж ті, що працюють в режимі ядра..

У всіх UNIX-системах користувачі можуть при запуску процесу задавати значення поправки пріоритету за допомогою команди *nice*.

```
nice [ -n [-|+]інкремент] команда
```

Діапазон значень інкремента в більшості систем - від -20 до 20. Якщо інкремент не заданий, використовується стандартне значення 10. Позитивний інкремент означає зниження поточного пріоритету. Звичайні користувачі можуть задавати лише позитивний інкремент і, тим самим, лише знижувати пріоритет.

Чим вище параметр *nice*, тим більше «ввічливий» і «тактовний» процес по відношенню до інших і тим нижче його пріоритет. Менше значення *nice* означає вищий його пріоритет. Параметр *nice* представляє собою позитивне значення, зазвичай — число 20 (досить дивно, але в документації до UNIX це число називається як NZERO) і є зсувом від деякого числа, яке залежить від системи. Процес стартує з параметром *nice* рівним 20 і може стати як дуже «доброзичливим», задавши параметр *nice* рівним 39, так і абсолютно «безпардонним», задавши параметр *nice* рівним 0.

Для того, щоб змінити свій пріоритет, процес звертається до системного виклику *nice*:

```
#include  <unistd.h>
int nice(
   int incr // приращение
 );
// Возвращает новое значение nice - NZERO
//или -1  в случае ошибки
//(код ошибки - в переменной errno)
```

Системний виклик *nice* додає приріст *incr* до поточного значення параметра *nice*. Результат повинен вийти в діапазоні від 0 до 39 включно. Якщо він вийшов за межі вказаного діапазону, використовується найближче допустиме значення. Лише суперкористувач може зменшити значення параметра *nice*, підвищивши тим самим пріоритет обслуговування.

Фактично, системний виклик *nice* повертає нове значення *nice,* зменшене на 20, таким чином, значенняЄ що повертається, лежить в діапазоні від -20 до 19, за умови, що NZER0==20. Проте, це значення рідко використовується в програмах. Особливо якщо врахувати, що нове значення *nice*, рівне 19, невідмітне від ознаки помилки (19 - 20 = -1). Ця помилка залишається невиправленою, якщо вона взагалі була відмічена, так багато років, що може служити показником того, як мало стурбовані UNIX-програмісти обслуговуванням помилкових ситуацій, зв’язаних з другорядними системними викликами.

9.4 Питання для самоперевірки

1. Що таке процес? Які атрибути процесу є найважливішими?
2. Чим відрізняється реальний ідентифікатор користувача від ефективного?
3. Які обмеження є у звичайного користувача на зміну ідентифікаторів користувача і групи?
4. Для чого використовуються сеанси і групи процесів?
5. Як створити групу процесів?
6. Який процес не може покинути свою групу процесів групи?
7. Який процес є лідером сеансу?
8. Що відбувається при завершенні лідеру сеансу?
9. Що таке відносний пріоритет? Як його змінити?

10 СИГНАЛИ И ЇХ ОБРОБКА

Сигнал дає можливість процесу реагувати на подію, джерелом якої може бути операційна система або інше завдання. Сигнали викликають переривання завдання і виконання заздалегідь передбачених дій. Сигнали можуть вироблятися синхронно (як результат роботи самого процесу), або асинхронно (направлені процесу іншим процесом). Синхронні сигнали найчастіше приходять від системи переривань процесора і свідчать про дії процесу, що блокуються апаратурою, наприклад, ділення на нуль, помилка адресації, порушення захисту пам'яті тощо.

Прикладом асинхронного сигналу є сигнал з терміналу. Для цього користувач може натискувати комбінацію клавіш Ctrl+C, внаслідок чого ОС UNIX виробляє сигнал SIGINT і направляє його активному процесу. Сигнал може поступити у будь-який момент виконання процесу, тобто він є асинхронним, вимагаючи від процесу негайного завершення роботи. В даному випадку реакцією на сигнал є безумовне завершення процесу. Для відправлення сигналу також служить команда *kill*:

```
kill sig_no pid
```

Сигнали генеруються ядром і забезпечують виклик певної процедури при настанні деякої події. Основні причини відправки сигналу:

- Особливі ситуації. (ділення на нуль)
- Термінальні переривання. (<Del>, <Ctrl+C>, <Ctrl+Z>)
- Інші процеси. Як засіб взаємодії між процесами за допомогою виклику *kill*()
- Управління завданнями.
- Квоти. Перевищення квоти ресурсів.
- Повідомлення. Наприклад, про готовність пристрою.
- Аларми. Пов'язано з роботою таймерів.

Програмний код процесу, якому поступив сигнал, може або проігнорувати його, або прореагувати на нього стандартною дією (наприклад, завершитися), або перехопити сигнал і самостійно виконати специфічні дії, визначені прикладним програмістом. У останньому випадку в програмному коді необхідно передбачити спеціальні системні виклики, за допомогою яких операційна система інформується, яку процедуру треба виконати у відповідь на вступ того або іншого сигналу.

Реакція на сигнал (або дія з сигналу) відноситься до всього процесу, навіть якщо цей процес виконується в декількох потоках, хоча, кожен потік може мати свою маску сигналів.

Сигнали забезпечують логічний зв'язок між процесами, а також між процесами і користувачами (терміналами).

Сигнали з'явилися вже в ранніх версіях UNIX, але їх реалізація була недостатньо надійною. Сигнал міг бути втрачений, виникали труднощі з відключенням (блокуванням) сигналів на час виконання критичних ділянок коди. В даний час стандарт POSIX.1 визначає інтерфейс надійних сигналів.

Кожен сигнал має унікальне символьне ім'я і відповідний йому номер.

10.1 Імена і типи сигналів. Нормальне і аварійне завершення

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, прийняте 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.

Виявлені помилки

SIGBUS — спроба доступу до невизначеної частини пам'яті (А)
SIGFPE — помилка арифметичної операції (Л)
SIGILL — некоректна команда (А)
SIGPIPE — запис в канал, з якого ніхто не читає (Т)
SIGSEGV — недопустиме звернення до сегменту пам'яті (А)
SIGSYS — некоректне звернення до системного виклику (А)
SIGXCPU — вичерпаний ліміт процесорного часу (А)
SIGXFSZ — перевищено обмеження на розмір файлу (А)

Сигнали, що генеруються користувачем або застосуванням

SIGABRT — звернення до системного виклику *abort*() (А)

SIGHUP — виявлений обрив зв'язку з терміналом або завершення термінального процесу (Т)

SIGINT — переривання роботи (з клавіатури) (Т)

SIGKILL — “ліквідовувати”, може мати лише штучне походження (Т)

SIGQUIT — завершити (з клавіатури) (А)

SIGTERM — завершити, може мати лише штучне походження (Т)

SIGUSR1 — сигнал користувача № 1, має лише штучне походження (Т)

SIGUSR2 — сигнал користувача №2, має лише штучне походження (Т)

Управління завданнями

SIGCHLD — дочірній процес завершив або припинив роботу (I)
SIGCONT — продовжити виконання (з клавіатури) (C)
SIGSTOP — припинити роботу, може мати лише 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)

Події таймера

SIGALRM — витік час таймера (T)
SIGVTALRM — витік час віртуального таймера (T)
SIGPROF — витік час профілюючого таймера (T)

Інші події

SIGPOLL — сталася очікувана подія (T)
SIGTRAP — пастка трасувальника — точка зупинку (A)
SIGURG — доступні позачергові дані в сокеті (I)

Символи в дужках мають наступне значення:
I — сигнал ігнорується (від англ. «*Ignore*»)
T — наводить до завершення процесу (від англ. «*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 англ. «*Abort*»)
S — зупинка (від англ. «*Stop*»)
З — продовження після зупинки (від англ. «*Continue*»)

Сигнали виявлення помилок, що мають природну природу, 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 апаратному рівні. 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.

10.2 Обробка сигналів

10.2.1Набори сигналів. Управління маскою сигналів

маска сигналів є набором бітів, для управління якими призначені наступні функції:

- *sigemptyset*() ініціалізує порожній набір сигналів

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

- *sigfillset*() ініціалізує повний набір сигналів

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

- *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
```

Робота з маскою сигналів типа *sigset_t* починається з виклику функції *sigemptyset*() або *sigfillset*(), а потім за допомогою *sigaddset*() або *sigdelset*() додаються потрібні або віддаляються непотрібні сигнали. Для перевірки наявності сигналу в наборі використовується функція *sigismember*().

```
#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
```

10.2.2Завдання обробника сигналів: виклик *sigaction*

Реакція застосування на сигнали задається за допомогою системного виклику *sigaction*(). Він викликається для кожного сигналу, дію якого необхідно змінити:

Реакція застосування на сигнали задається за допомогою системного виклику sigaction. Він викликається для кожного сигналу, дію якого необхідно змінити:

```
#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
```

У аргументі *act* передається покажчик на структуру, яка визначає реакцію на сигнал для всього процесу. Якщо аргумент *oact* не є порожнім покажчиком, за заданою адресою записується опис колишньої реакції процесу на сигнал. Якщо потрібно лише отримати опис дії сигналу, передайте в аргументі act значення NULL, в цьому випадку жодних змін вноситися не буде.

Для завдання цих аргументів використовується спеціальна структура *struct sigaction*.

```
struct sigaction
{
   void (*sa_handler)();
   void (sa_sigaction)(int, siginfo_t *, void *);
   sigset_t sa_mask;
    int sa_flags;
}
```

У структурі *sigaction* поле *sa_handler* визначає власне реакцію на сигнал і може мати одне з наступних значень:

SIG_DFL	Реакція за умовчанням, яка залежить від вигляду сигналу, але застосування завжди або ігнорує сигнал, або завершується, або припиняється, або відновлює роботу.
SIG_IGN	Застосування ігнорує сигнал, в результаті доставка сигналу не робить впливу на застосування. Є один побічний ефект: коли для сигналу SIGCHLD визначена реакція SIG_IGN, це рівносильне установці прапора SA_NOCLDWAIT. Але реакція SIG_DFL не має такого ефекту, хоча вона полягає в тому, аби ігнорувати сигнал.
Функція	Покажчик на функцію-обробник. В цьому випадку говорять: «сигнал буде перехоплений».

Функції-обробники сигналів виглядають приблизно так

```
static void fcn(int signum){
(void)write(STDOUT_FILENO, "Отриманий сигнал \n", 11);
_exit(EXIT_FAILURE);
}
```

У момент доставки сигналу викликається функція-обробник, якою як аргумент передається номер сигналу (наприклад SIGINT або SIGUSR1). Ви можете передбачити окремі функції для кожного з сигналів, можете всі сигнали обробляти однією функцією або вибрати щось середнє між цими двома крайнощами.

Якщо системою підтримуються сигнали реального часу, можна встановити прапор SA_SIGINFO (див. нижчий) і передати покажчик обробник в полі *sa_sigaction*. В цьому випадку обробник отримує значно більше відомостей про сигнал. Деякі реалізації використовують для зберігання покажчика *sa_handler* і *sa_sigaction* одну і тугіше область пам'яті, тому слід записувати адресу обробника лише в один з них.

Нижче наводиться список прапорів для поля *sa_flags*. Звернете увагу: перші два застосовуються лише для сигналу SIGCHLD.

SA_NOCLDSTOP	Не посилати сигнал при зупинці і відновленні дочірнього процесу.
SA_NOCLDWAIT	Не перетворювати дочірній процес на «зомбі». Явна установка реакції SIG_IGN на сигнал SIGCHLD дає той же ефект.
SA_NODEFER	Не додавати сигнал до маски при виклику обробника, якщо він явно не вказаний в полі sa_mask. Цей прапор залишений виключно для збереження сумісності із застарілою функцією signal.

SA_ONSTACK	Доставляти сигнал на альтернативному стеку сигналів, якщо такий був оголошений зверненням до функції sigaltstack).
SA_RESETHANO	Встановити реакцію на сигнал в значення SIG_DFL і на вході у функцію-обробник скидати прапор SA_SIGINFO. Ігнорується для сигналів SIGILL і SIGTRAP. Додатково виконуються дії, властиві сигналу SA_NODEFER і так само існує для сумісності з функцією signal.
SA_RESTART	Не переривати виконання системних викликів.
SA_SIGINFO	Покажчик на функцію-обробник слід брати з поля sa_sigaction, а не з sa_handler.

Нижче наводиться приклад програми, яка демонструє перехоплення і обробку сигналів. Вона виводить число кожні три секунди, а при здобутті сигналу переривання (SIGINT), виводить повідомлення і завершується:

```
static void fcn(int signum)
{
  (void)write(STDOUT_FILENO, "Отриманий сигнал\n", 15);
  _exit(2);
}
int main(void)
{
  int i;
  struct sigaction act;
  memset(&act, 0, sizeof(act)); act.sa_handler = fcn;
  sigaction(SIGINT &act, NULL);
  for (i = 1; ; i++)
  {
    sleep(3);
    printf("%d\n", i);
  }
  exit(0);
}
```

Зверненням до системного виклику sigaction проводиться установка функції-обробника сигналу SI6INT. Після запуску програми, коли на екрані з'явилося число 2, натискуватимемо клавіші Ctrl+C. Це привело до того, що виконання програми буде перервано і управління перейде функції fcn, яка виведе текст повідомлення і завершить роботу програми, звернувшись до системного виклику _exit.

В результаті, на екран було виведено наступне:

```
1 2 Отриманий сигнал
```

Якби функція-обробник не була встановлена, натиснення комбінації Ctrl+C привело б до негайного завершення роботи процесу, тому що в цьому полягає реакція за умовчанням будь-якого процесу на сигнал SIGINT.

За допомогою системного виклику sigaction можна змусити процес ігнорувати сигнал SIGINT:

```
int main(void)
{
  int i;
  struct sigaction act;
  memset(&act, 0, sizeof(act));
  act.sa_handler = SIQ_IGN;
  sigaction(SIGINT &act, NULL);
  for (i = 1; ; 1++)
  {
    sleep(3);
    printf("%d\n", i);
  }
  exit(0);
}
```

Цього разу програма продовжуватиме виводити числа, незважаючи на натиснення комбінації Ctrl+C.

10.3 Обробники сигналів

Після того, як зверненням до *sigaction*() сигналу буде призначений функція-обробник, вона викликатиметься у момент доставки сигналу. Якщо прапор SA_NODEFER не встановлений, перехоплений сигнал додається до маски сигналів на час виконання обробника. Крім того, до маски додаються всі сигнали з поля sa_mask структури sigaction. Після закінчення обробника оригінальна маска сигналів відновлюється в первинний стан, навіть якщо вона була явно змінена усередині обробника викликами pthread_sigmask або sigprocmask.

Дії обробника сигналу багато в чому залежать від типа сигналу і причин, що його породили.

- Сигнал може бути генерований ядром при виявленні помилки. Наприклад, SIGFPE (помилка виконання арифметичної операції) або SIGPIPE (запис в канал, який не відкритий на читання з іншого боку). У подібних випадках найбільш прийнятна реакція — завершити роботу потоку або процесу з

повідомленням про помилку. Повернення з обробника в такому стані може привести до помилок в обчисленнях. А в разі помилок, виявлених апаратними засобами, процес все одно може бути завершений системою після повернення з обробника.

- Сигнал може бути викликаний діями користувача, наприклад, натисненням комбінації клавіш Ctrl+C (SIGINT). У цій ситуації можна завершити застосування після збірки сміття або можна перервати тривалі обчислення і чекати нових команд користувача. Це лише один з прикладів реакції застосування на сигнал — у будь-якому випадку, дії, що виконуються по сигналу, тісно пов'язані з логікою роботи застосування.
- Сигнал може бути породжений самим застосуванням. Наприклад, аби повідомити про готовність файлу до обробки, можна послати сигнал SIGUSR1.
- Сигнал може бути генерований таймером.

У будь-якому випадку необхідно подумати про те:

1. Що потрібно зробити усередині обробника і як змінити стан застосування у відповідь на отриманий сигнал.
2. Куди відправитися далі. З обробника можна просто повернутися в програму, що викликає, завершити роботу застосування, виконати довгий перехід в іншу частину програми або генерувати новий сигнал.

Щоб писати досить надійні програми, слід дотримуватися наступних рекомендацій.

- Повідомлення про помилки повинні виводитися системним викликом write (або іншою безпечною функцією), а завершення роботи застосування — системним викликом _exit.
- Глобальні змінні, які змінюються усередині обробника, повинні мати типа volatile sig_atomic_t, а повернення в застосування повинне проводитися за умови, що для сигналу встановлений прапор SA_RESTART.

Приклад 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.

```
#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 міститься адреса оброблювача *handler*.

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

10.4 Питання для самоперевірки

1. Для чого використовуються сигнали в ОС UNIX?
2. Які види сигналів існують в ОС UNIX?
3. Які види реакції на сигнал за умовчанням передбачені в ОС UNIX?
4. Як задається реакція застосування на сигнал?
5. Як виглядає обробник сигналу?
6. Які стандартні обробники сигналів передбачені системою?
7. Яких рекомендацій слід дотримуватися, щоб писати досить надійні програми?

11 СИГНАЛИ І СИСТЕМНІ ВИКЛИКИ

11.1 Процедури sigsetjmp и siglongjmp

Обробка сигналів незрідка поєднується з нелокальними переходами. У таких випадках можуть виявитися корисними функції sigsetjmp і siglongjmp.

Обробник сигналів може завершуватися викликом siglongjmp(). Ввтором випадку відновлюється маска, заздалегідь збережена викликом sigsetjmp. Тому обробники сигналів можуть викликати функцію siglongjmp лише в тому випадку, якщо в тому ж потоці вже виконаний виклик функції sigsetjmp.

11.2 Блокування сигналів

Головна відмінність сигналів від інших засобів взаємодії між процесами полягає в тому, що їх обробка програмою зазвичай відбувається відразу ж після вступу сигналу (або не відбувається взагалі), незалежно від того, що програма робить в даний момент. Сигнал перериває нормальний порядок виконання інструкцій в програмі і передає управління спеціальної функції – обробникові сигналу. Якщо обробка сигналу не наводить до завершення процесу, то після виходу з функції-обробника виконання процесу поновлюється з тієї крапки, в якій воно було перерване. В програм також є можливість припинити обробку сигналів, що поступають, тимчасово, на період виконання якої-небудь важливої операції. У традиційній термінології призупинення здобуття певних сигналів називається блокуванням. Якщо для сигналу, що поступив, було встановлено блокування, сигнал буде переданий програмі, як тільки вона розблоковує даного типа сигналів. Цим блокування відрізняється від ігнорування сигналу, при якому сигнали відповідного типа ніколи не передаються програмі. Слід пам'ятати, що не всі сигнали можуть бути проігноровані.

Блокування сигналів часто повинне виконуватися нерозривно з іншою операцією. Для цього існує ряд стандартних механізмів і функцій.

Щоб припинити виконання програми до приходу певного сигналу (або будь-якого із заданої множини сигналів), існує системний виклик sigsuspend.

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

Його використання передбачає, що у весь інший час сигнал (або їх множина), що цікавить нас, заблокований, інакше можна його упустити до виклику sigsuspend.

Навіть якщо функцію установки маски sigprocmask викликати безпосередньо перед, все одно утворюється тимчасове вікно, впродовж якого очікуваний сигнал може виявитися доставлений. Тому sigsuspend приймає як аргумент маску сигналів, яку атомарно встановлює на час чекання.

Наведемо прототипи ще декількох системних викликів, пов'язаних з сигналами.

sigpause змінює маску і чекає доставки сигналу

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

sigpending повертає набір сигналів, що чекають на обробку

```
#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
```

11.3 Штучна генерація сигналів

Кожен сигнал може бути породжений як природними причинами, так і в результаті звернення до системних викликів *kill*(), *killpg*(), *abort*(), *raise*() і *sigqueue*():

Системний виклик 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
```

Послати сигнал (не маючи повноважень суперкористувача) можна лише процесу, в якого ефективний ідентифікатор користувача збігається з ефективним ідентифікатором користувача для процесу, що посилає сигнал.

Аргумент pid описує, кому посилається сигнал, а аргумент *sig* – який сигнал посилається. Цей системний виклик уміє робити багато різних речей, залежно від значення аргументів:

- Якщо pid > 0 і *sig* > 0, то сигнал номером sig (якщо дозволяють привілеї) посилається процесу з ідентифікатором *pid*.
- Якщо pid = 0, а *sig* > 0, то сигнал з номером *sig* посилається всім процесам в групі, до якої належить посилаючий процес.
- Якщо pid = -1, *sig* > 0 і посилаючий процес не є процесом суперкористувача, то сигнал посилається всім процесам в системі, для яких ідентифікатор користувача збігається з ефективним ідентифікатором користувача процесу, що посилає сигнал.
- Якщо pid = -1, *sig* > 0 і посилаючий процес є процесом суперкористувача, то сигнал посилається всім процесам в системі, за винятком системних процесів (зазвичай всім, окрім процесів з *pid* = 0 і *pid* = 1).
- Якщо pid < 0, але не –1, *sig* > 0, то сигнал посилається всім процесам з групи, ідентифікатор якої дорівнює абсолютному значенню аргументу pid (якщо дозволяють привілеї).
- Якщо значення *sig* = 0, то проводиться перевірка на помилку, а сигнал не посилається, оскільки всі сигнали мають номери > 0. Це

можна використовувати для перевірки правильності аргументу *pid* (чи є в системі процес або група процесів з відповідним ідентифікатором).

Системний виклик 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
```

Системний виклик *killpg*(), строго кажучи, взагалі не потрібний. Він посилає сигнал процесам, які належать до групи, з ідентифікатором, рівним вмісту аргументу pgrp. Таким чином, він абсолютно ідентичний зверненню:

```
kill(-pgrp,   signum);
```

Системний виклик abort посилає сигнал sigabrt SIGABRT

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

Системний виклик abort нагадує *kill*() з аргументом SIGABRT, з тією лише різницею, що системний виклик *abort*() завершить роботу процесу у будь-якому випадку (не поважно, перехоплюється цей сигнал чи ні); *abort*() ніколи не повертає управління в програму, що викликає. Наприклад, передбачимо, що сигналу SIGABRT за допомогою sigaction призначена реакція SIG_IGN. В цьому випадку системний виклик abort завершить роботу застосування, а:

```
kill(getpid(),   SIGABRT)
```

не зробить на процес жодного впливу.

Системний виклик *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
```

Системний виклик raise, насправді є звичайною стандартною функцією мови С. Он посилає сигнал потоку, що викликав. Рівноцінна операція посилки сигналу через звернення до системного виклику kill виглядає так:

```
kill(getpid(),   signum);
```

Системний виклик sigqueue розглядати не будемо.

11.4 Системний виклик pause

Ми вже зустрічалися з масою системних викликів, які можуть бути заблоковані і очікувати деякої події. Наприклад, при читанні рядків з термінального пристрою, системний виклик *read*() зазвичай блокується в очікуванні закінчення введення рядка. Системний виклик pause не робить нічого особливого і не чекає чогось конкретного — він просто блокує виконання процесу.

```
#include <unistd.h>
int  pause(void);
```

Оскільки у момент доставки сигналу практично будь-який системний виклик буде перерваний, можна сміливо стверджувати, що системний виклик pause чекає прибуття сигналу. Якщо функція-обробник сигналу повертає управління в програму, що викликає, то системний виклик pause повертає ознаку помилки з кодом EINTR, але оскільки це єдиний спосіб виходу з системного виклику pause, немає необхідності перевіряти значення, що їм повертається.

11.5 Питання для самоперевірки

1. Що таке блокування сигналу?
2. Як послати сигнал групі процесів?
3. В чому різниця між системними викликам *abort*() і *kill*() з аргументом SIGABRT?
4. В чому різниця між системними викликам *abort*() і *raise*()?
5. Для чого використовується системний виклик *pause*()?

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.

Белломо М. Unix: наглядный курс освоения операционной системы. - Киев: Диалектика, 2001. - 336 с.: ил. **Приклад.** Написать программу,

которая удаляет из заданного каталога и всех его подкаталогов файлы с именем «core». Имя каталога задавать в виде аргумента командной строки.

Для того, чтобы решить эту задачу, необходимо уметь читать содержимое каталога. Для этого будем использовать функции стандартной библиотеки. Сначала нужно открыть каталог с помощью функции opendir, затем, последовательно вызывая функцию readdir, считывать записи каталога, а в конце закрыть каталог с помощью функции closedir.

Удаление файлов из каталога будет выполнять функция delNotPointFiles, параметром которой является имя файла. Эта функция будет вызываться рекурсивно, чтобы обеспечить удаление файлов с именем core в подкаталогах.

Для проверки того, что файл является каталогом написана функция isDir. Для определения типа файла данная функция использует системный вызов stat. Такая функция нужна для организации входа в подкаталоги.

10. Ниже приводится текст программы