ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА КІБЕРНЕТИКИ
КИЇВСЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ
ІМЕНІ ТАРАСА ШЕВЧЕНКА

В.В. Зубенко
О.М. Супрун

ОПЕРАЦІЙНІ СИСТЕМИ

Методичні рекомендації
до виконання лабораторних проєктів

Київ 2025

УДК 004.451 (072)

Рецензенти: доктор технічних наук, проф. О.П. Нечипорук,
кандидат фізико-математичних наук, доц. О.І. Ченцов

Рекомендовано до друку вченою радою факультету комп'ютерних наук та кібернетики (протокол №16 від «24» червня 2025 року)

Ухвалено науково-методичною комісією факультету комп'ютерних наук та кібернетики (протокол №11 від «23» червня 2025 року)

к.ф.-м.н., доц. Зубенко Віталій Володимирович
к.ф.-м.н., доц. Супрун Ольга Миколаївна
Операційні системи: Методичні рекомендації до виконання лабораторних проєктів. – Електронне видання / В.В. Зубенко, О.М. Супрун – Київ: 2025. – 41 с.

Викладено матеріали для вивчення дисципліни «Операційні системи» студентами першого курсу магістратури факультету комп'ютерних наук та кібернетики спеціальності «Прикладна математика». Розглянуто основні принципи програмування в операційній системі *Unix*/*Linux*, методи створення та взаємодії процесів в операційних системах *Windows* та *Unix*/*Linux*. Міститься опис шести лабораторних проєктів (лабораторні проєкти 1-4 є обов'язковими, лабораторні проєкти 5-6 є додатковими).

Для здобувачів освіти факультету комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка, які навчаються за освітньо-науковою програмою «Прикладна математика» спеціальності F1 «Прикладна математика».

ЗМІСТ

Короткі теоретичні відомості 4

Лабораторний проєкт 1. Система команд та файлова структура ОС *Unix/Linux* 5

Лабораторний проєкт 2. Управління ОС *Linux* за допомогою інтерпретатора *BASH* 14

Лабораторний проєкт 3. Основні принципи програмування в ОС *Unix/Linux* 19

Лабораторний проєкт 4. Процеси та потоки в ОС *Unix/Linux* 22

Лабораторний проєкт 5. Процеси та потоки в ОС *Windows* 29

Лабораторний проєкт 6. Засоби міжпроцесної взаємодії ОС 32

Література 41

КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Однією з основних підсистем операційної системи (ОС), яка безпосередньо впливає на ефективність та функціонування комп'ютера, є підсистема керування процесами і потоками. Ця система створює процеси та потоки, організовує їх взаємодію, відповідає за виконання та завершення, а також забезпечує розподіл процесорного часу між кількома одночасно активними процесами й потоками в системі.

Сучасні ОС є багатозадачними. Багатозадачність – спосіб організації обчислювального процесу, у якому одночасно виконується кілька програм або процесів.

При цьому функції планування потоків повністю зосереджені в операційній системі. Кожному потоку для виконання в порядку черги надається обмежений неперервний період процесорного часу – квант. Якщо потік (процес) вичерпав свій квант, то він переводиться в стан готовності, очікуючи надання нового кванту процесорного часу. Операційна система, тим часом, обирає новий потік або процес із черги готових до виконання.

Кванти часу, що виділяються, можуть бути однаковими або різними для всіх потоків або процесів. Розмір кванта зазвичай вибирається невеликий (не більше 6–16 мілісекунд), щоб користувач не відчував присутності в системі одночасно кількох десятків процесів. Зміна активного потоку відбувається якщо потік завершився і залишив систему, сталася помилка, потік перейшов у стан очікування, вичерпано квант процесорного часу, або відведений даному потоку.

На рис.1 показано спрощену схему роботи системи управління процесами за допомогою двох черг.

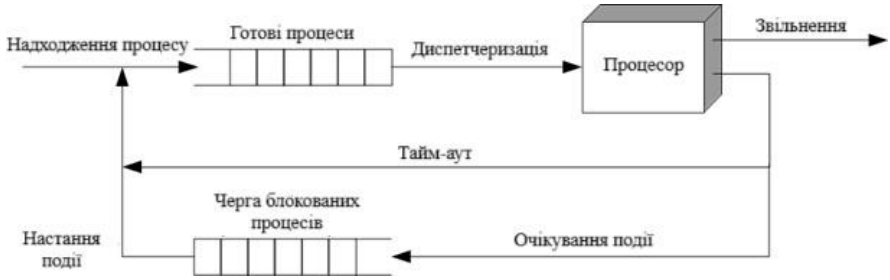

Рисунок 1 - Реалізація системи управління процесами за допомогою двох черг

Лабораторний проєкт 1

СИСТЕМА КОМАНД ТА ФАЙЛОВА СТРУКТУРА ОС *UNIX/LINUX*

Мета роботи: вивчити команди ОС для роботи з файлами, каталогами, дисками, системною датою та часом; текстовий редактор *Kate* та файловий менеджер *Midnight Commander*.

Теоретична частина

Операційна система *Linux* створена на основі ОС *UNIX* і багато в чому має схожу структуру та систему команд. Користувач може працювати в текстовому режимі за допомогою командного рядка або за допомогою графічного інтерфейсу *X Window* та одного з менеджерів робочого столу (наприклад *KDE* або *GNOME*). Причому одночасно в системі можуть працювати до 7 користувачів (6 – у текстовому режимі консолі та 1 – у графічному режимі), перемикання між користувачами здійснюється після натискання клавіш:

Ctrl + Alt + F1 або Ctrl + Alt + F7

В табл. 1 наведено основні команди системи.

Таблиця 1 – Основні команди системи

Команда	Аргументи/ключі	Приклад	Опис
`dir`	Каталог	`dir` `dir /home`	Виводить в консоль вміст каталогу
`ls`	`-all` та інші (див. man)	`ls -all`	Виводить в консоль вміст каталогу
`ps`	`-a` `-x` та інші (див. man)	`ps -a`	Виводить в консоль список процесів
`mkdir`	Ім’я каталога	`mkdir stud11`	Створює каталог

Продовження таблиці 1

Команда	Аргументи/ключі	Приклад	Опис
`rmdir`	Ім’я каталога	`rmdir stud11`	Видаляє каталог
`rm`	Файл	`rm myfile1`	Видаляє файл
`mv`	Файл нове_ім’я	`mv myfile1 myf1`	Перейменовує файл
`cat`	Файл	`cat 1.txt`	Виведення файлу в консоль
`cd`	Ім’я каталога	`cd home`	Перехід по каталогам
`grep`	(див. man)	`grep "^a" "words.txt"`	Пошук рядка у файлі
`kill`	`pid` процесу	`kill 12045`	Завершує процес
`top`			Виводить в консоль список процесів
`htop`			Виводить в консоль повний перелік запущених процесів
`su`			Перехід в режим root
`chmod`	Права_доступу файл	`chmod 777 1.txt`	Зміна прав доступу до файлів
`mount`	Будова каталогу	`mount /dev/cdrom /MyCD`	Монтування пристроїв
`dd`	`if=`файл `of=`файл `bs=n count=n`	`dd if=/dev/hda1 of=/F.bin bs=512 count=1`	Побайтне копіювання

Кінець таблиці 1

Команда	Аргументи/ключі	Приклад	Опис
`ln`	Файл1 файл2 `-l`	`ln файл1 файл2` `ln –l файл1 файл2`	Створення жорсткого або символьного посилання на файл
`uname`	`-a`	`uname -a`	Інформація про систему
`find`	`find` файл	`find /home a1.txt`	Пошук файлів
`man`		`man fgetc`	Довідка по системі
`info`		`info fgetc`	Довідка по системі

Linux та *Windows* використовують різні файлові системи для зберігання та організації доступу до інформації на дисках. В *Linux* використовуються файлові системи *Ext2/Ext3, RaiserFS* та інші. Усі файлові системи підтримують каталогування. Каталогова файлова система спочатку записує зміни, які вона буде проводити, в окрему частину файлової системи (каталог) і лише потім вносить необхідні зміни до решти файлової системи. Після успішного виконання всіх транзакцій записи видаляються з каталогу. Це забезпечує найкраще збереження цілісності системи та зменшує ймовірність втрати даних. Слід зазначити, що *Linux* підтримує доступ до розділів *Windows*.

Файлова система *Linux* має лише один кореневий каталог, який позначається косою рискою (/). В файловій структурі *Linux* немає дисків *A, B, C, D,* а є лише каталоги. В *Linux* розрізняються великі і малі літери в командах, іменах файлів і каталогів. У *Windows* у кожного файлу існує лише одне ім'я, у *Linux* їх може бути декілька. Це "жорсткі" посилання, які вказують безпосередньо на індексний дескриптор файлу. Жорстке посилання – це один із принципів організації файлової системи *Linux.*

Для виконання операцій запису та читання даних у існуючому файлі його слід відкрити за допомогою виклику `open()`. Нижче наведено опис цього виклику:

```
int open (const char *pathname, int flags, [mode_t mode]);
int fopen (const char *pathname, int flags, [mode_t mode]);
```

Другий аргумент системного виклику `open` – `flags` – має цілий тип і визначає метод доступу. Параметр `flags` приймає одне із значень, заданих сталими у заголовному файлі `fcnt1.h`. У файлі визначені три сталі:

O_RDONLY – відкрити файл тільки для читання;

O_WRONLY – відкрити файл тільки для запису;

O_RDWR – відкрити файл для читання та запису;

або "`r`", "`w`", "`rw`" для `fopen()`.

Третій параметр `mode` встановлює права доступу до файлу та є необов'язковим, він використовується лише разом із прапором *O_CREAT*. Приклад створення нового файлу:

```
# include <sys / types.h>
# include <sys / stat.h>
# include <fcnt1.h>
int Fd1;
FILE *F1;
F1=fopen ("Myfile2.txt", "w", 644);
Fd1=open ("Myfile1.txt", O_CREAT, 644);
```

Системні виклики `stat` и `fstat` дозволяють процесу визначити значення властивостей у наявному файлі:

```
#include <sys/types.h>
#include <sys/stat.h>
int stat (const char *pathname, struct stat *buf);
int fstat (int filedes, struct stat *buf);
```

де `pathname` – повне ім'я файлу, `buf` – структура типу `stat`. Ця структура після успішного виклику міститиме пов'язану з файлом інформацію.

Поля структури `stat` включають такі елементи:

```
struct stat {
dev_t         st_dev;   /* логічний пристрій, де знаходиться
файл */
ino_t        st_ino;     /* номер індексного дескриптора */
mode_t    st_mode;      /* права доступу до файлу */
nlink_t    st_nlink;   /* кількість жорстких посилань на файл */
uid_t       st_uid;     /* ID користувача-власника */
```

```
gid_t       st_gid;    /* ID групи-власника */
dev_t       st_rdev;    /* тип пристрою */
off_t       st_size;     /* загальний розмір у байтах */
unsigned long  st_blksize; /* розмір блоку введення –
виведення */
unsigned long st_blocks;  /* число блоків, які займає файл */
time_t   st_atime;  /* час останнього доступу  */
time_t   st_mtime;  /* час останньої модифікації  */
time_t   st_ctime;    /* час останньої зміни */
            };
```

Права доступу в *Linux*. Права доступу до файлів представлені у вигляді послідовності біт, де кожен біт означає дозвіл на запис (`w`), читання (`r`) або виконання (`x`). Права доступу записуються для власника файлу (`owner`); групи, до якої належить власник файлу (`group`); та всіх інших (`other`). Наприклад, при виведенні команди `dir` запис типу

```
-rwx r-x r-w  1.exe
```

означає, що власник файлу `1.exe` має права на читання, запис та виконання, група має права тільки на читання та виконання, решта мають права тільки на читання та запис. У вісімковому вигляді вийде значення 0754. Насправді маніпулює файлами не сам користувач, а запущений процес. Для перегляду прав доступу можна використовувати функцію `stat`.

Приклад: `stat("1.exe", &st1);`

Для запису прав доступу є функція `chmod`:

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

Приклад: `chmod("1.exe", 0777);`

Структура каталогів операційної системи *Linux* наведена в табл. 2. Використовують такі скорочення для імен каталогів:

— одиночна крапка (`.`) позначає поточний робочий каталог;

— дві крапки (`..`) позначають батьківський каталог поточного робочого;

— тильда (~) позначає домашній каталог користувача (зазвичай це каталог, який є поточним робочим під час запуску *Bash*).

Таблиця 2 – Структура каталогів ОС *Linux*

/	Кореневий каталог
/bin	Містить виконувані файли найнеобхідніших для роботи системи програм. Каталог /bin не містить підкаталогів
/boot	Містить ядро системи (файл vmlinuz-...) і необхідні для його завантаження файли
/dev	В цьому каталозі розміщені файли пристроїв (драйвери)
/etc	Розміщені файлів, які містять інформацію про налаштування системи (наприклад, налаштування програм). Тобто це каталог конфігураційних файлів
/home	Містить домашні каталоги користувачів системи
/lib	Тут знаходяться бібліотеки (функції, необхідні багатьом програмам)
/media	Підкаталоги, які використовуються як точки монтування для змінних пристроїв, знаходяться в цьому каталозі
/mnt	Файлові системи, що тимчасово підключаються, використовують підкаталоги або весь каталог для монтування систем
/proc	Містить файли з інформацією про процеси, що виконуються в системі
/root	Домашній каталог адміністратора системи
/sbin	Містить виконувані програми, як і каталог /bin. Використовувати програми, які розміщені в цьому каталозі, може лише адміністратор системи (root). Як і каталог /bin, містить виконувані програми
/tmp	Каталог для тимчасових файлів, що зберігають проміжні дані, необхідних для роботи тих чи інших програм і програм, що видаляються після завершення роботи
/usr	Каталог для більшості програм, які не мають значення для завантаження системи. Структура цього каталогу фактично дублює структуру кореневого каталогу

Кінець таблиці 2

/	Кореневий каталог
/var	Містить дані, які були отримані в процесі роботи одних програм і повинні бути передані іншим, і файли журналів з інформацією про роботу системи

Порядок виконання

1. Вивчити теоретичну частину лабораторного проєкту.
2. У консольному режимі, використовуючи команди з табл.1, створити в домашній папці підкаталог /номер_груп/ПІБ_здобувача, де надалі зберігатимуться всі файли здобувача. Перейти до кореневого каталогу та вивести його вміст, використовуючи команди `dir` та `ls –all`, проаналізувати відмінності.
3. Перевірити дію команд `ps`, `ps –x`, `top`, `htop`. Використовуючи команду `man`, знайти у довідковій системі довідку за функціями `fprintf`, `fputc` та командою `ls`.
4. В текстову редакторі `joe` (виклик: `joe 1.c`) написати програму 1.c, що що виводить на екран фразу "*HELLO LINUX*". Компілювати отриману програму компілятором `gcc`:

```
gcc 1.c –o 1.exe
```

Запустити отриманий файл `1.exe` на вконання:

```
./1.exe
```

Варіанти індивідуальних завдань

У всіх завданнях необхідно для читання або запису файлу використовувати функції посимвольного введення – виведення `fgetc()`, `fputc( )` або `getc()`, `putc()`. Передбачити контроль помилок відкриття, закриття, читання та запису файлу чи каталогу. Виведення повідомлень про помилки повинно здійснюватися у стандартний потік виведення повідомлень про помилки (`stderr`) у вигляді: і'мя_модуля: текст_повідомлення. Ім'я модуля обирається з аргументів командного рядка.

1. Програма введення символів з клавіатури та запису їх у файл (ім'я файлу вводиться як аргумент під час запуску програми). Передбачити вихід після введення певного символу (наприклад, ctrl-F).

2. Програма перегляду текстового файлу та виведення його вмісту на екран (ім'я файлу передається як аргумент при запуску програми, другий аргумент) *N* встановлює виведення за групами рядків (по *N* рядків) або суцільним текстом (*N* = 0)).

3. Програма копіювання одного файлу в інший, імена файлів передаються як аргументи командного рядка під час запуску програми. Передбачити копіювання прав доступу до файлу.

4. Програма підрахунку числа символів, що відображаються в рядках текстового файлу. Результати підрахунку записуються в другий текстовий файл (імена файлів передаються як аргументи командного рядка під час запуску програми). Приклад роботи програми:
вихідний текстовий файл із трьох рядків:

FGHJK

VBNMertyuimda

NRTcvbnmjgfdSTUIOP

файл, отриманий в результаті роботи програми:

1. 5
2. 13
3. 18

загалом: 3 рядки 36 символів

5. Програма підрахунку числа слів у текстовому файлі. Результати підрахунку записуються в другий текстовий файл (імена файлів передаються як аргументи командного рядка при запуску програми). Приклад виведення програми для текстового файлу:

REEEt TY

YH UU WFG T NN GHJ UU ghj

Ss gyu GG RGH gH

файл, отриманий в результаті роботи програми:

1. 2 слова
2. 8 слів
3. 5 слів

загалом: 3 рядки 15 слів

6. Програма, яка підраховує кількість символів з однаковими кодами *ASCII* у текстовому файлі. Результати підрахунку записуються в інший текстовий файл (імена файлів передаються як аргументи командного рядка під час запуску програми). Приклад виведення програми для текстового файлу:

EWwER REEEt

rg E ERT Ety SI I IO NN

файл, отриманий в результаті роботи програми:

1. *E* код *ASCII* 69 =8
2. *W* код *ASCII* 87 =1
3. *w* код *ASCII* 119 =1
4. *E* код *ASCII* 69 =8

…

загалом: 26 символів

Лабораторний проєкт 2

УПРАВЛІННЯ ОС *LINUX* ЗА ДОПОМОГОЮ ІНТЕРПРЕТАТОРА *BASH*

Мета роботи: дослідити основні об'єкти, команди, типи даних та оператори управління інтерпретатора *BASH*; створити скрипт-файл.

Теоретична частина

Bash – це *sh*- сумісний інтерпретатор командної мови, що виконує команди, прочитані зі стандартного вхідного потоку або файлу. Скрипт-файл – це звичайний текстовий файл, що містить послідовність команд bash, для якого встановлені права на виконання.

Приклад скрипта, що виводить вміст поточного каталогу в консоль та файл:

```
#!/bin/bash dir
dir > 1.txt
```

Командний інтерпретатор використовує такі змінні:

`$0`, `$1`, `$2`, `$3`… значення аргументів командного рядка під час запуску скрипту, де `$0` – ім'я самого файлу скрипту, `$1` – перший аргумент, `$2` – другий аргумент і так далі; `$@` всі аргументи командного рядка, кожен у лапках; `$?` код повернення останньої команди.

Приклад простого скрипта, що виводить в консоль і у файл вміст каталогу, де ім'я каталогу передається скрипту в якості аргументів при запуску:

запуск скрипту: `>./mydir /home/stud`

скрипт:

```
#!/bin/bash
dir $1
dir  $1 > 1.txt
```

Можна створити власну змінну та присвоїти їй значення:

```
A=121
A=”121”
let A=121
let “A=A+1”
```

Виведення значення в консоль: `echo $A`

Перевірка умови: `test[expr]`

де `expr`: а) для рядків: $S_1 = S_2$ S_1 містить S_2

S_1 != S_2 S_1 не містить S_2

-n S_1 якщо довжина S_1 >0

-z S_1 якщо довжина S_1 =0

б) цілі i_1 та i_2

i_1 – `ge` i_2

i_1 – `gt` i_2

i_1 – `ie` i_2

i_1 – `et` i_2

i_1 – `nt` i_2

в) файли

`-d name_file` чи є файл каталогом

`-f name_file` чи є файл звичайним файлом

`-r name_file` чи доступний файл для читання

`-s name_file` чи має файл ненульову довжину

`-w name_file` чи доступний файл для запису

`-x name_file` чи є файл виконуваним

г) логічні операції

`!exp` логічне заперечення (не)

`exp1 –a exp2` множення умов (і)

`exp1 –o exp2` додавання умов (або)

Перевірка умови: `if [expr ]`

```
then com 1      якщо умова expr=true, то команда
   …              com 1… com n
   com n
(elif expr2
com1
   …
   com n
)
else
com 1
   …
   com n
fi
```

Перевірка кількох умов: `case string1 in`

```
str 1)
```

```
        com 1
        …
        com n
        ; ;
      str 2)
        com 1
        …
        com n
        ; ;
      str 3)
        com 1
        …
        com n
        ; ;
      *)              // default
        com 1
        …
        com n
        ; ;
        esac
```

Функція користувача: `fname2 (arg1,arg2...argN)`

```
{
commands
}
```

Організація циклів:

1. `for var1 in list`
   `do`
   `com1`
   …
   `com n`
   `done`
2. `while exp`
   `com1`
   …
   `com n`
   `end`
3. `until exp` // аналог do-while
   `do`
   `com1`
   …

```
com n
done
```

Порядок виконання

1. Вивчити теоретичну частину лабораторного проєкту.
2. Написати скрипт, що виводить у консоль і файл всі аргументи командного рядка.
3. Написати скрипт, що виводить у файл (ім'я файлу задається користувачем як перший аргумент командного рядка) імена всіх файлів із заданим розширенням (третій аргумент командного рядка) із заданого каталогу (ім'я каталогу задається користувачем як другий аргумент командного рядка).
4. Написати скрипт, що компілює та запускає програму (ім'я вихідного файлу та *exe*-файлу результату задається користувачем як аргументи командного рядка). У разі помилок під час компіляції вивести в консоль повідомлення про помилки та не запускати програму на виконання.

Варіанти індивідуальних завдань

1. Створити скрипт, який виконує пошук файлів певного розміру в заданому каталозі. Користувач задає розмірний діапазон (min і max) як перший і другий аргументи командного рядка, а ім'я каталогу - як третій аргумент.
2. Використовуючи циклу `for`, написати скрипт, який для всіх файлів у заданому каталозі та всіх його підкаталогах виводить у консоль розміри та права доступу. Користувач задає ім'я каталогу в якості першого аргументу командного рядка.
3. Розробити скрипт, який здійснює пошук вказаного користувачем рядка у всіх файлах зазначеного каталогу та його підкаталогів. Користувач використовує рядок для пошуку та ім'я каталогу як перший та другий аргументи командного рядка відповідно. У консоль виводяться повний шлях та імена файлів, що містять заданий рядок, а також їхній розмір. Якщо доступ до будь-якого каталогу відсутній, потрібно вивести відповідне повідомлення і продовжити виконання скрипта.
4. Створити скрипт пошуку однакових за вмістом файлів у двох каталогах, наприклад *Dir1* та *Dir2*. Перший і другий аргументи командного рядка задаються користувачем як імена *Dir1* та *Dir2*

відповідно. В результаті роботи програми файли, наявні в *Dir1*, порівнюються з файлами в *Dir2* за їхнім вмістом. На екран виводяться кількість переглянутих файлів та результати порівняння.

5. Розробити скрипт, який у вказаному каталозі та всіх його підкаталогах знаходить файли, власником яких є заданий користувач. Ім’я користувача та каталог задаються користаачем в якості першого і другого аргументів командного рядка. Скрипт виводить результати у файл, який є третім аргументом командного рядка, у вигляді: повний шлях, ім'я файлу та його розмір. В консоль виводиться загальна кількість переглянутих файлів.

6. Написати скрипт, що знаходить у заданому каталозі та його підкаталогах всі файли заданого розміру. Користувач задає ім'я каталогу як перший аргумент командного рядка, а також розмірний діапазон файлів (min і max), як другий і третій аргументи командного рядка. Файл, що буде результатои роботи скрипта, повинен мати вигляд: повний шлях, ім'я файлу, його розмір. Цей файл є четвертим аргументом командного рядка. В консоль виводиться загальна кількість переглянутих файлів.

7. Створити скрипт, який обчислює загальний розмір усіх файлів у вказаному каталозі та його підкаталогах. В якості першого аргументу командного рядка користовач задає ім’я каталогу. Результати зберігаються у файл, ім’я якого задається другим аргументом командного рядка. Результуючий файл повинен містити інформацію: каталог (повний шлях), сумарний розмір файлів, кількість переглянутих файлів.

Лабораторний проєкт 3

ОСНОВНІ ПРИНЦИПИ ПРОГРАМУВАННЯ В ОС *LINUX*

Мета роботи: вивчити файлову систему ОС *Linux* та основних функцій для роботи з каталогами та файлами.

Теоретична частина

Каталоги в ОС *Linux* – це особливі файли. Для відкриття або закриття каталогів існують виклики:

```
#include <dirent.h>
DIR *opendir (const char *dirname);
int closedir( DIR *dirptr);
```

Для читання записів каталогу існує виклик:

```
struct dirent *readdir(DIR *dirptr);
```

Структура наступна:

```
struct dirent {
          long      d_ino;
          off_t     d_off;
          unsigned short d_reclen;
          char      d_name [1];
          };
```

Поле `d_ino` – це число, яке є унікальним для кожного файлу у файловій системі. Значенням поля `d_off` є зміщення даного елемента у реальному каталозі. Поле `d_name` - це початок масиву символів, який задає ім'я елемента каталогу. Це ім'я обмежено нульовим байтом і може містити не більше *MAXNAMLEN* символів. Тим самим описувана структура має змінну довжину, що зберігається в полі `d_reclen`.

Приклад виклику:

```
DIR *dp;
struct dirent *d;
d=readdir(dp);
```

При першому виклику функції `readdir` в структуру `dirent` будет прочитаний перший запис каталога. Після прочитання всього каталогу в результаті наступних викликів `readdir` буде повернуто значення `NULL`. Для повернення вказівника на початок каталогу на перший запис

існує виклик:

```
void rewindir(DIR *dirptr);
```

Для отримання імені поточного робочого каталогу використовується функція:

```
char *getcwd(char *name, size_t size);
```

Порядок виконання

1. Вивчити теоретичну частину лабораторного проєкту.
2. Написати програму виведення на екран вмісту, заданого користувачем каталогу. Вивести з використанням програми вміст поточного та кореневого каталогів. Передбачити контроль помилок відкриття, закриття, читання каталогу. Виведення повідомлень про помилки має здійснюватися у стандартний потік виведення повідомлень про помилки (`stderr`) у такому вигляді: ім'я_модуля текст_повідомлення.

Варіанти індивідуальних завдань

Передбачити виконання контролю помилок для всіх операцій з файлами та каталогами.

1. Відсортувати у заданому каталозі (аргумент 1 командного рядка) та у всіх його підкаталогах файли за такими критеріями (аргумент 2 командного рядка, задається у вигляді цілого числа): 1 – за розміром файлу, 2 – за іменем файлу. Записати відсортовані файли до нового каталогу (аргумент 3 командного рядка).
2. Знайти в заданому каталозі (аргумент 1 командного рядка) та всіх його підкаталогах заданий файл (аргумент 2 командного рядка). Вивести в консоль повний шлях до файлу, ім'я файлу, його розмір, дату створення, право доступу, номер індексного дескриптора. Вивести також загальну кількість переглянутих каталогів та файлів.
3. Для заданого каталогу (аргумент 1 командного рядка) та всіх його підкаталогів вивести у заданий файл (аргумент 2 командного рядка) та в консоль імена файлів, їх розмір та дату створення, що задовольняють заданим умовам: 1 – размір файлу знаходиться в межах від *N1* до *N2* (*N1, N2* задаються в аргументах командного рядка), 2 – дата створення знаходиться в межах від *M1* до *M2* (*M1, M2* задаються в аргументах командного рядка).
4. Знайти файли, що збігаються за вмістом, у двох заданих каталогах (аргументи 1 і 2 командного рядка) і всіх їх підкаталогах.

Вивести у консоль та у файл (аргумент 3 командного рядка) їх ім'я, розмір, дату створення, права доступу, номер індексного дескриптора.

5. Підрахувати сумарний розмір файлів у заданому каталозі (аргумент 1 командного рядка) та для кожного його підкаталогу окремо. Вивести у консоль та у файл (аргумент 2 командного рядка) назву підкаталогу, кількість файлів у ньому, сумарний розмір файлів, ім'я файлу з найбільшим розміром.

6. Створити програму, яка виконує пошук файлів заданого розміру у вказаному каталозі та всіх його підкаталогах. Шлях до каталогу задається першим аргументом командного рядка, мінімальний і максимальний розміри файлів — другим і третім аргументами відповідно. Результати пошуку записуються у файл, ім’я якого передається як четвертий аргумент командного рядка. Формат запису: повний шлях, ім’я файлу та його розмір. На екран виводиться загальна кількість переглянутих файлів

Лабораторний проєкт 4

ПРОЦЕСИ ТА ПОТОКИ В ОС *LINUX*

Мета роботи: дослідити методи створення процесів у ОС *Linux,* основні функції створення та управління процесами, обмін даними між процесами.

Теоретична частина

В ОС *Linux* для створення процесів використовується системний виклик

```
fork():
#include <sys/types.h>
#include <unistd.h>
pid_t fork (void);
```

В результаті успішного виклику `fork()` ядро створює новий процес, що є майже точною копією процесу, який викликає. Іншими словами, новий процес виконує копію тієї ж програми, що й процес, що створив його, при цьому всі його об'єкти даних мають ті ж самі значення, що і в процесі, який викликає.

Створений процес називається дочірнім процесом, а процес, що здійснив виклик `fork()`, називається батьківським. Після виклику батьківський процес і його новостворений нащадок виконуються одночасно, при цьому обидва процеси продовжують виконання з оператора, який слідує відразу за викликом `fork()`. Процеси виконуються в різних адресних просторах, тому прямий доступ до змінних одного процесу з іншого процесу неможливий.

Наступна програма наочно показує роботу виклику `fork()` та використання процесу:

```
#include <stdio.h>
#include <unistd.h>
int main ()
{
   pid_t pid;        /* ідентифікатор процесу */
   printf (“Наразі лише один процес\n”);
   pid = fork ();      /* створення нового процесу */
```

```
      printf (“Вже два процеси\n”);
      if (pid = = 0)
         {
         printf (“Це Дочірній процес його pid=%d\n”,
getpid());
         printf (“А pid його Батьківського процесу=%d\n”,
getppid());
         }
      else if (pid > 0)
         printf (“Це Батьківський процес pid=%d\n”, getpid());
      else
         printf    (“Помилка    виклику    fork,    нащадок    не
створено\n”);
   }
```

Для коректного завершення дочірнього процесу у батьківському процесі необхідно використовувати функцію `wait()` або `waitpid()`:

```
pid_t   wait(int *status);
pid_t   waitpid(pid_t pid, int *status, int options);
```

Функція `wait` призупиняє виконання батьківського процесу доти, допоки дочірнійпроцес не припинитьвиконання, або до появисигналу, якийабо завершує поточний процес, або вимагає виклику функції-обробника. Якщо дочірній процес на момент виклику функції вже завершився (так званий «зомбі»), то функція негайно повертається. Системні ресурсы, що пов’язані з дочірнім процесом, звільняються.

Функція `waitpid` призупиняє виконання батьківського процесу до тих пір, допоки дочірній процес, що вказаний в параметрі `pid`, не завершить виконання або допоки не з’явиться сигнал, який або завершує батьківський процес, або вимагає викликати функцію-обробник. Якщо вказаний дочірній процес на момент виклику функції вже завершився (так званий «зомбі»), то функція негайно повертається. Системні ресурсы, щопов’язані з дочірнім процесом, звільняються.

Параметр `pid` може набувати декількох значень:

`pid < −1` означає, що потрібно чекати будь-який дочірній процес, чий ідентифікатор групи процесів дорівнює абсолютному значенню `pid`;

`pid = −1` означає, що слід очікувати будь-який дочірній процес; `wait` поводиться так само;

`pid = 0` означає, що слід очікувати будь-який дочірній процес, чий

ідентифікатор групи процесів дорівнює такому в поточного процесу;

`pid > 0` означає, що слід очікувати дочірній процес, чий ідентифікатор дорівнює `pid`.

Значення `options` створюється шляхом бітової операції АБО над такими константами:

WNOHANG означає повернути управління негайно, якщо жоден дочірній процес не завершив виконання;

WUNTRACED означає повертати управління також для зупинених дочірніх процесів, про статус яких ще не було повідомлено.

Кожен дочірній процес при завершенні роботи посилає своєму батьківському процесу спеціальний сигнал *SIGCHLD*, на який у всіх процесів за замовчуванням встановлено реакцію «ігнорувати сигнал». Наявність такого сигналу спільно з системним викликом `waitpid()` дозволяє організувати асинхронний збір інформації про статус породжених процесів, що завершилися батьківським процесом.

Для перезавантаження програми, що виконується, можна використовувати функції сімейства `exec`. Основна відмінність між функціями у сімействі ***exec*** полягає у способі передачі параметрів.

```
int execl(char *pathname, char *arg0, arg1, ..., argn, NULL);
int execle(char *pathname, char *arg0, arg1, ..., argn, NULL, char **envp);
int execlp(char *pathname, char *arg0, arg1, ..., argn, NULL);
int execlpe(char *pathname, char *arg0, arg1, ..., argn, NULL, char **envp);
int execv(char *pathname, char *argv[]);
int execve(char *pathname, char *argv[],char **envp);
int execvp(char *pathname, char *argv[]);
int execvpe(char *pathname, char *argv[],char **envp);
```

Існує розширене трактування поняття процесу, за якого процес розглядається як сукупність виділених йому ресурсів і набору потоків виконання. Потоки (threads) спільно використовують програмний код, глобальні змінні та системні ресурси процесу, однак кожен з них має власний лічильник команд, індивідуальний набір регістрів і окремий стек. Всі глобальні змінні доступні для будь-якого з потоків, що належать цьому процесу. Кожен потік у системі ідентифікується унікальним номером — ідентифікатором потоку. Оскільки

традиційний процес у моделі потоків розглядається як процес із єдиним потоком виконання, для такого процесу також можна отримати ідентифікатор відповідного потоку, скориставшись функцією `pthread_self()`. Потік, що створюється під час ініціалізації нового процесу, зазвичай називають початковим або головним потоком виконання цього процесу. Для створення потоків використовується функція `pthread_create`:

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

Функція створює новий потік, в якому виконується функція користувача `start_routine`, і передає їй як аргумент параметр `arg`. За потреби передачі більше одного параметра, всі такі параметри збираються до однієї структури, адреса якої і передається. Якщо виклак успішний, функція `pthread_create` повертає значення 0 і поміщає ідентифікатор нового потоку виконання за адресою, на яку вказує параметр `thread`. У разі помилки повертається додатнє число, яке є кодом помилки, опис якого наведено у файлі `<errno.h>`. Значення системної змінної `errno` при цьому не визначається. Параметр `attr` використовується для визначення різних атрибутів потоку, який створюється. Функція потоку повинна мати заголовок виду

```
void * start_routine (void *)
```

Завершення функції потоку відбувається у таких випадках:

- функція потоку вкликала функцію `pthread_exit()`;
- функція потоку досягла точки виходу;
- потік був достроково завершений іншим потоком.

Функція `pthread_join()` використовується для переведення потоку в стан очікування:

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

Функція `pthread_join()` блокує роботу виконуваного потоку, який викликав її, допоки не закінчить роботу потік з ідентифікатором `thread`.Адреса, яку повернув потік, що завершив виконання (або в результаті виходу з пов'язаної з ним функції, або при виклику функції `pthread_exit()`), після розблокування записується у вказівник за

адресою status_addr. Якщо користувач не зацікавлений у значенні, яке повернув виконуваний потік, в якості параметра можна використовувати значення NULL.

Для компіляції програми з потоками необхідно підключити бібліотеку pthread.lib таким чином:

```
gcc  1.c -o 1.exe -lpthread
```

Час в Linux відраховується в секундах, що минули з початку цієї епохи (00:00:00 UTC, 1 Січень 1970 року). Для отримання системного часу можна використовувати такі функції:

```
#include <sys/time.h>
time_t  time (time_t *tt);
int  gettimeofday(struct timeval *tv, struct timezone
*tz);

struct timeval {
            long tv_sec;        /* секунди */
            long tv_usec;     /* мікросекунди */
            };
```

Порядок виконання

1. Вивчити теоретичну частину лабораторного проєкту.

2. Написати програму, що створює два дочірні процеси, з використанням двох викликів fork(). Батьківський та два дочірні процеси повинні виводити на екран свій pid та pid батьківського процесу (для дочірніх процесів) та поточний час у форматі години: хвилини: секунди: мілісекунди. Використовуючи виклик system(), виконати команду ps -x у батьківському процесі. Знайти свої процеси у списку запущених процесів.

3. В основній програмі створити два потоки. Після цього процес-батько створює файл, записує в нього рядки виду: *N* pid, поточний час у форматі години: хвилини: секунди: мілісекунди (де *N* – номер рядка, що виводиться) і виводить в лівій половині екрану рядки, що формуються. Кількість рядків, що записуються у файл, $k = 100$. Обидва потоки читають рядки з файлу і виводять їх у правій частині екрана у вигляді потік id потоку поточний час (мілісекунди (мсек)) рядок з файлу.

Варіанти індивідуальних завдань

1. Розробити програму за умовою п. 3 порядку виконання роботи, але замість потоків створити 3 процеси, які здійснюють ті ж функції, що і в п. 3 (зчитати рядки з файлу та вивести їх у правій частині екрану).

2. Написати програму знаходження масиву *K* послідовних значень функції *y[i]=sin(2*PI*i/N)* (*i*=0,1,2…*K*-1) з використанням ряду Тейлора. Користувач задає значення *K, N* та кількість n членів ряду Тейлора. Для розрахунку кожного члена ряду Тейлора запускається окремий потік. Кожен потік виводить на екран свій id та розраховане значення ряду. Головний процес підсумовує всі члени ряду Тейлора та отримане значення *y[i]* записує у файл.

3. Написати програму синхронізації двох каталогів, наприклад *Dir1* та *Dir2*. Користувач задає імена *Dir1* та *Dir2*. В результаті роботи програми файли, наявні в *Dir1*, але відсутні в *Dir2*, повинні скопіюватися в *Dir2* разом із правами доступу. Процедури копіювання повинні запускатися з використанням функції `fork()` в окремому процесі для кожного файлу, що копіюється. Кожен процес виводить на екран свій pid, ім'я файлу, що копіюється, і число скопійованих байт. Число запущених процесів не повинно перевищувати *N* (вводиться користувачем).

4. Написати програму пошуку однакових за вмістом файлів у двох каталогах, наприклад *Dir1* та *Dir2*. Користувач задає імена *Dir1* та *Dir2*. В результаті роботи програми файли, що є в *Dir1*, порівнюються з файлами з *Dir2* за їх вмістом. Процедури порівняння повинні запускатися з використанням функції `fork()` в окремому процесі для кожної пари порівнюваних файлів. Кожен процес виводить на екран свій pid, ім'я файлу, число переглянутих байт та результати порівняння. Число запущених процесів не повинно перевищувати *N* (вводиться користувачем).

5. Написати програму пошуку заданої користувачем комбінації з m байт ($m < 255$) у всіх файлах поточного каталогу. Користувач задає ім'я каталогу. Головний процес відкриває каталог і запускає для кожного файлу каталогу окремий процес пошуку заданої комбінації з *m* байт. Кожен процес виводить на екран свій pid, ім'я файлу, число переглянутих байт та результати пошуку. Число запущених процесів не повинно перевищувати *N* (вводиться користувачем).

6. Розробити програму, яка сприймає команди, що вводяться з

клавіатури, та здійснює їхнє коректне виконання. Для цього кожна команда, що вводиться, повинна виконуватися в процесі, який окремо запускається з використанням виклику `exec()`. Передбачити контроль помилок.

7. Створити дерево процесів/потоків за індивідуальним завданням. Кожен процес/потік постійно, через час *t* виводить на екран наступну інформацію:

номер процесу/потоку pid ppid поточний час (мілісекунди (мсек)).

Час *t* =((номер процесу/потоку по дереву)*200 (мілісекунди (мсек)).

Лабораторний проєкт 5

ПРОЦЕСИ ТА ПОТОКИ В ОС *WINDOWS*

Мета роботи: вивчити методи створення процесів і потоків в ОС *Windows*, основні функції управління процесами та потоками, обмін даними між процесами та потоками.

Теоретична частина

Для створення процесу в *Windows* використовуються функції:

```
BOOL CreateProcess(
LPCTSTR lpApplicationName,  // ім'я виконуваного модуля
LPTSTR lpCommandLine,       // командний рядок
LPSECURITY_ATTRIBUTES lpProcessAttributes,      // вказівник на структуру
LPSECURITY_ATTRIBUTES lpThreadAttributes,    // вказівник на структуру
BOOL bInheritHandles,          // маркер успадкування поточного процесу
DWORD dwCreationFlags,      // маркери способів створення процесу
LPVOID lpEnvironment,       // вказівник на блок середовища
LPCTSTR lpCurrentDirectory,        // поточний диск або каталог
LPSTARTUPINFO lpStartupInfo, // вказівник на структуру STARTUPINFO
LPPROCESS_INFORMATION lpProcessInformation // вказівник на структуру
);
```

Основним параметром є ім'я модуля, що виконується, відповідне імені програми, яка запускається в новоствореному процесі. Приклад:

```
void main()
{
 STARTUPINFO si;
 ZeroMemory(&si,sizeof(STARTUPINFO));
 PROCESS_INFORMATION pi;
 CreateProcess("c:\\1.exe",null,null,null,false,null,null,
nulL,&si,&pi)
```

Для створення потоку (*thread*) у *Windows* використовується функція

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES   lpThreadAttributes,   // атрибути захисту
  DWORD dwStackSize,      // размір стека
  LPTHREAD_START_ROUTINE lpStartAddress, // адреса функції потоку
  LPVOID lpParameter, // параметр, який буде передано функції потоку
  DWORD dwCreationFlags, // 0 – виконання починається негайно
  LPDWORD lpThreadId  // ідентифікатор id нового потоку
);
```

При кожному виклику цієї функції система створює об'єкт ядра (потік) і виділяє пам'ять під стек потоку адресного простору процесу. Новий потік виконується в контексті того ж процесу, що і батьківський потік. Він отримує доступ до всіх описників об'єктів ядра, всієї пам'яті та стеків всіх потоків у процесі. За рахунок цього потоки в рамках одного процесу можуть легко взаємодіяти один з одним.

Для передачі повідомлень використовується спеціальна `win32`-функція:

```
LRESULT SendMessage( HWND hWnd, // handle вікна
   UINT Msg, // повідомлення, що надсилається
   WPARAM wParam, // перший параметр повідомлення
   LPARAM lParam // другий параметр повідомлення
);
```

З'ясувавши *Window handle* потрібного вікна, користувач може надсилати йому *Windows Messages*.

Порядок виконання

1. Вивчити теоретичну частину лабораторного проєкту.
2. Написати програму, яка запускає у новому процесі програму notepad.
3. Написати програму, яка запускає новий потік у поточному процесі. У потоці виконується функція, що виводить кожні 50 мс на екран: номер_повідомлення id_потоку поточний_час (сек:мсек).
4. Написати програму *A*, яка запускає в новому процесі програму *B*.

Обидві програми повинні надсилати одна одній повідомлення – *Windows message*.

Варіанти індивідуальних завдань

1. Створити дерево процесів/потоків за індивідуальним завданням. Кожен процес/потік постійно, через час ***t***, виводить на екран наступну інформацію:

номер процесу/потоку pid поточний час (милісекунди(мсек)).

Час *t* =((номер процесу/потоку по дереву)*200 (милісекунди (мсек))**.**

2. Написати програму, що виводить на екран список запущених процесів та потоків та їх атрибути (pid та ін.) у вигляді таблиці.

3. Написати програму знаходження масиву K послідовних значень функції *y[i]=sin(2*PI*i/N)* (*i*=0,1,2…*K*-1) з використанням ряду Тейлору. Користувач визначає значення *K, N* і кількість *n* членів ряду Тейлора. Для розрахунку кожного члена ряду Тейлора запускається окремий потік. Кожен потік виводить на екран свій pid та розраховане значення ряду. Головний процес підсумовує всі члени ряду Тейлора і отримане значення *y[i]* записує в масив та файл користувача.

4. Написати програму синхронізації двох каталогів, наприклад *Dir1* та *Dir2*. Користувач задає імена *Dir1* и *Dir2*. В результаті роботи програми файли, що є в *Dir1,* але відсутні в *Dir2*, повинні скопіюватися в *Dir2* разом. Процедури копіювання повинні запускатися в окремому потоці для кожного файлу, що копіюється. Кількість запущених потоків не повинна перевищувати *N* (вводиться користувачем). Кожен потік виводить на екран свій id, ім'я файлу, що копіюється, і число скопійованих байт.

5. Написати програму пошуку однакових за вмістом файлів у двох каталогах, наприклад *Dir1* та *Dir2*. Користувач задає імена *Dir1* та *Dir2*. В результаті роботи програми файли, наявні в *Dir1*, порівнюються з файлами в *Dir2* за їх вмістом. Процедури порівняння повинні запускатися в окремому потоці кожної пари порівнюваних файлів. Кожен потік виводить на екран свій id, ім'я файлу, число переглянутих байт та результати порівняння. Число запущених потоків не повинно перевищувати *N* (вводиться користувачем).

6. Написати програму пошуку заданої користувачем комбінації з *m* байт (*m* < 255) у всіх файлах поточного каталогу. Користувач задає ім'я каталогу. Головний процес відкриває каталог і запускає для кожного файлу каталогу окремий потік пошуку заданої комбінації з *m* байт. Кожен потік виводить на екран свій id, ім'я файлу, число переглянутих байт та результати пошуку. Число запущених потоків не повинно перевищувати *N* (вводиться користувачем).

Лабораторний проєкт 6

ЗАСОБИ МІЖПРОЦЕСНОЇ ВЗАЄМОДІЇ ОС

Мета роботи: вивчити методи та засоби взаємодії процесів в ОС *Linux.*

Теоретична частина

Кожен процес у *Linux* виконуються у власному, ізольованому адресному просторі. Для забезпечення взаємодії між процесами необхідно застосовувати відповідні спеціальні методи: спільні файли; загальна або розділювальна пам'ять; черги повідомлень (queue); сигнали (signal); канали (pipe); семафори.

1. Спільні файли. Якщо процеси використовують спільні файли, то обмін інформ ацією між ними відбувається за допомогою відкриття одного і того ж файлу. Для пришвидшення роботи потрібно використовувати файли, які відображаються в пам'яті за допомогою виклику `mmap()`:

```
#include <unistd.h>
#include <sys/mman.h>
void * mmap(void *start, size_t length, int prot , int
flags, int fd, off_t offset);
```

ФункцІя `mmap` відображає length байтів починаючи зі зміщення offset файлу, яке визначається описувачем `fd` в пам’ять, починаючи з адреси `start`. Параметр `offset` не є обов'язковим. Зазвичай він дорівнює 0. Реальне розміщення відображених даних повертається власне функцією `mmap`, в жодному разі не рівне 0. Аргумент `prot` описує бажаний режим захисту пам’яти (він не повинен конфліктувати з режимом відкриття файлу):

PROT_EXEC – дані, що знаходяться в пам'яті, можуть викристовуватись;

PROT_READ – дані, що знаходяться в пам'яті, можна зчитувати;

PROT_WRITE – в сегмент можна записувати інформацію;

PROT_NONE – доступ до цього сегменту пам'яті заборонено.

Параметр `flags` для об’єкту, що відображується, задає тип, опції відображення, вказує, чи належать дані тільки цьому процесу або їх

можуть читати й інші процеси. Цей параметр містить комбінації таких біт:

MAP_FIXED – не рекомендується використовувати цю опцію;

MAP_SHARED – розділити використання відображення з іншими процесами, що відображають той самий об'єкт. Запис даних у цю область пам'яті еквівалентний запису у файл. Файл може не оновлюватися до виклику функцій `msync(2)` чи `munmap(2)`;

MAP_PRIVATE – створити нерозділюване відображення, з механізмом copy-on-write. Усі записи до цієї області пам'яті не впливають на вміст оригінального файлу. Не визначено, є чи ні зміни у файлі після виклику `mmap` видимими у відповідному діапазоні пам'яті.

2. Загальна або розділювальна пам'ять. Використання розділюваної пам'яті передбачає створення спеціальної області пам'яті, до якої можуть одночасно звертатися кілька процесів. Для роботи з такою пам'яттю застосовуються відповідні системні виклики:

```
#include <sys/mman.h>
   int shm_open (const char *name, int oflag, mode_t mode);
   int shm_unlink (const char *name);
```

Виклик `shm_open` створює і відкриває новий (або вже існуючий) об'єкт пам'яті, який розділюється. При відкритті за допомогою функції `shm_open()` повертається файловий дескриптор. Ім'я name трактується стандартним чином для аналізованих засобів міжпроцесної взаємодії. За допомогою аргументу oflag можуть вказуватися прапори *O_RDONLY*, *O_RDWR, O_CREAT, O_EXCL* і/або *O_TRUNC*. Режим доступу до створюванго об'єкту формується згідно значення mode і маски створення файлів процесу. За допомогою функції `shm_unlink` виконується обернета операція, яка видаляє попередньо створений за допомогою `shm_open` об'єкт.

Після доєднання сегмента пам'яті до віртуального адресного простору процесу, він може отримувати доступ до відповідних ділянок пам'яті за допомогою звичайних машинних команд читання та запису, без необхідності виклику додаткових системних функцій.

```
int main (void) {
  int fd_shm;  /* дескриптор об'єкта в пам'яті, яка розділяється */
  if ((fd_shm = shm_open (“myshered.shm”, O_RDWR | O_CREAT,
0777)) < 0) { perror ("error create shm");     return (1); }
```

Для компіляції програми необхідно підключити бібліотеку `rt.lib` наступним чином: `gcc  1.c  –o  1.exe  -lrt`

3. Черги повідомлень (queue). Черги повідомлень є складнішим методом зв'язку взаємодіючих процесів в порівнянні з програмними каналами.

За допомогою черг можна з одного або кількох завдань незалежним чином надсилати повідомлення деякому завданню-отримувачу. При цьому тільки процес-отримувач може читати та видаляти повідомлення з черги, а процеси-клієнти мають право лише поміщати у чергу свої повідомлення. Черга працює тільки в одному напрямку, якщо необхідний двосторонній зв'язок, слід створити дві черги. Черги повідомлень надають можливість використовувати кілька дисциплін обробки повідомлень:

— *FIFO* – повідомлення, яке записано першим, буде прочитане першим;

— *LIFO* – повідомлення, яке записано останнім, буде прочитане першим;

— пріоритетна – повідомлення читаються з урахуванням їх пріоритетів;

— довільний доступ – можна читати будь-яке повідомлення, а програмний канал забезпечує лише дисципліну *FIFO*.

Для відкриття черги служить функція `mq_open()`, яка за аналогією з файлами створює опис відкритої черги і дескриптор типу `mqd_t`, що посилається на нього, повертається як нормальний результат.

```
#include <mqueue.h>
mqd_t mq_open (  const char *name, int oflag, ...);
```

Під час читання повідомлення з черги не видаляється, тобто може бути прочитане неодноразово. У самих чергах фактично зберігаються не повідомлення як такі, а лише вказівники на них у пам’яті та їхні розміри. Ця інформація розміщується системою в сегменті пам’яті, спільному для всіх задач, які взаємодіють через відповідну чергу. Оскільки черга є системним механізмом і потребує використання ресурсів операційної системи, то кожен процес, що використовує чергу, повинен попередньо отримати доступ до цього спільного сегмента пам’яті за допомогою системних викликів.

4. Сигнали (*signal*). Із точки зору користувача, отримання процесом сигналу схоже на виникнення переривання: процес призупиняє своє виконання, і керування передається функції-обробнику сигналу. Після завершення обробки сигналу процес може продовжити нормальне

виконання. Типи сигналів зазвичай задаються за допомогою спеціальних символьних констант. Системний виклик `kill()` призначений для надсилання сигналу одному або кільком визначеним процесам у межах прав доступу користувача.

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

Надіслати сигнал без прав суперкористувача дозволяється лише тому процесу, ефективний ідентифікатор користувача якого збігається з ефективним ідентифікатором користувача процесу, що надсилає сигнал. Аргумент *pid* визначає процес-отримувач, а аргумент *sig* — сигнал, який потрібно надіслати. Поведінка виклику залежить від значень переданих аргументів:

pid > 0 – сигнал надсилається процесу з ідентифікатором *pid*;

pid = 0 – сигнал надсилається всім процесам у групі, до якої належить процес, що надсилає сигнал;

pid = –1 – процес, що надсилає сигнал, не є процесом суперкористувача, тому сигнал надсилається всім процесам у системі, для яких ідентифікатор користувача збігається з ефективним ідентифікатором користувача процесу, що надсилає сигнал;

pid = 1 – процес, що надсилає сигнал, є процесом суперкористувача, тому сигнал надсилається всім процесам в системі, за винятком системних процесів (зазвичай всім, крім процесів з pid = 0 та pid = 1);

pid < 0, але не –1 – сигнал надсилається всім процесам із групи, ідентифікатор якої дорівнює абсолютному значенню аргументу pid (якщо дозволяють привілеї);

sig = 0 – сигнал фактично не надсилається, лише виконується перевірка на наявність помилок. Це можна використовувати для перевірки коректності значення pid, тобто з'ясування, чи існує у системі процес або група процесів із вказаним ідентифікатором.

Системні виклики для встановлення власного обробника сигналів:

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

Системний виклик `signal` призначений для зміни реакції процесу на певний сигнал. Параметр sig задає номер сигналу, для якого слід змінити оброблення. Параметр handler визначає новий спосіб оброблення цього сигналу: покажчик на користувацьку функцію-обробник, спеціальне значення *SIG_DFL* (для відновлення стандартної

реакції на сигнал *sig*) або *SIG_IGN* (для ігнорування сигналу). Виклик *signal* повертає покажчик на попередній обробник сигналу, який можна зберегти та використати для відновлення попередньої обробки за потреби.

Приклад користувальницького оброблення сигналу *SIGUSR1*.

```
void *my_handler(int nsig) {
   код функції-обробника сигналу }
int main() {
   (void) signal(SIGUSR1, my_handler); }
```

5. Канали (*pipe*). Програмний канал – це файл особливого типу (*FIFO*: «першим увійшов – першим вийшов»). Процеси можуть записувати та зчитувати дані з каналу як із звичайного файлу. Якщо канал заповнено, процес запису в канал зупиняється, поки не з'явиться вільне місце, щоб знову заповнити його даними. З іншого боку, якщо канал порожній, то процес, що читає, зупиняється до тих пір, поки процес, що пише, не запише дані в цей канал. На відміну від звичайного файлу, тут немає можливості позиціонування по файлу з використанням покажчика.

В ОС *Linux* розрізняють два види програмних каналів:

— іменований програмний канал. Може слугувати для спілкування та синхронізації довільних процесів, які знають ім'я даного програмного каналу та мають відповідні права доступу. Для створення використовується виклик:

```
int mkfifo(const char *filename, mode_t mode);
```

— неіменований програмний канал. Даним каналом можуть користуватися тільки процес, що його створив, і його нащадки. Для створення використовується виклик: `int pipe(int fd[2]);`

6. Семафори. Семафор це змінна певного типу, доступна для паралельних процесів, над якою виконуються лише дві операції:

— *A*(*S, n)* – збільшити значення семафору *S* на величину *n*;

— *D*(*S, n)* – якщо значення семафору $S < n$, процес блокується. Далі $S = S - n$;

— *Z*(*S*) – процес блокується доти, допоки значення семафору *S* не стане рівним 0.

Семафор виконує роль допоміжного механізму для керування критичним ресурсом, оскільки операції *A* та *D* є неподільними і

взаємно виключають одна одну. Механізм семафора працює за принципом попередньої перевірки стану критичного ресурсу, після чого надається дозвіл на доступ або відмовляється на певний час. Основною перевагою семафорних операцій є відсутність «активного очікування», що значно підвищує ефективність роботи мультипрограмної обчислювальної системи.

Системні виклики, що використовуються для работи з семафорами:

а) Створення та отримання доступу до набору семафорів:

```
int semget(key_t key, int nsems, int semflg);
```

Параметр *key* є ключем, що слугує ім'ям масиву семафорів. Значення цього параметра може бути отримане за допомогою функції `ftok()` або мати спеціальне значення *IPC_PRIVATE*. Використання *IPC_PRIVATE* завжди призводить до створення нового масиву семафорів із унікальним ключем, який не збігається з ключами жодного з існуючих масивів і не може бути отриманий функцією `ftok()` з будь-якими параметрами. Параметр *nsems* визначає кількість семафорів у створюваному або існуючому масиві. Якщо масив із заданим ключем уже існує, але його розмір не співпадає зі значенням *nsems*, виникає помилка.

Параметр *semflg* — це прапори, які мають значення лише під час створення нового масиву семафорів. Вони визначають права доступу різних користувачів до масиву, а також умови створення нового масиву і поведінку системного виклику при спробі створення. Значення *semflg* складається з комбінації (операцією побітового АБО — "|") визначених констант і вісімкових прав доступу.

IPC_CREAT – якщо для вказаного ключа масив не існує, то його потрібно створити;

IPC_EXCL – використовується разом із *IPC_CREAT*. Якщо обидва прапори задані й масив семафорів із вказаним ключем уже існує, доступ до нього не надається, і виклик завершується помилкою. У цьому випадку змінна *errno*, визначена у файлі `<errno.h>`, набуде значення *EEXIST*;

0400 – користувач, який створив масив, має право на його читання;

0200 – користувач, який створив масив має право на запис в масив;

0040 – читання дозволено групі, до якої належить користувач, який створив масив;

0020 – запис дозволено групі, до якої належить користувач, який

створив масив;

0004 – дозволено читання всім користувачам;

0002 – дозволено запис всім користувачам.

Приклад: `semflg= IPC_CREAT | 0022`

b) Зміна значень семафорів:

```
int semop(int semid, struct sembuf *sops, int nsops);
```

Параметр *semid* є дескриптором *System V IPC* для набору семафорів, тобто значенням, яке повертає системний виклик `semget()` під час створення або пошуку набору семафорів за ключем. Кожен із *nsops* елементів масиву, на який вказує параметр *sops*, описує операцію, що має бути виконана над одним із семафорів у масиві *IPC*-семафорів, і є структурою відповідного типу:

```
struct sembuf {
short  sem_num; // номер семафору в масиві IPC семафорів (починаючи з 0);
short sem_op;  // виконувана операція;
short sem_flg; // прапори для виконання операції.
}
```

Значення елементу структури *sem_op* визначається:

— для виконання операції *A(S,n)* семафор повинен мати значення, що дорівнює *n*;

— для виконання операції *D(S,n)* семафор повинен мати значення, що дорівнює *n*;

— для виконання операції *Z(S)* семафор повинен мати значення, що дорівнює 0.

Відповідно до семантики системного виклику, усі операції над семафорами виконуються лише безпосередньо перед успішним завершенням виклику. Якщо під час виконання операцій *D* або *Z* процес переходить у стан очікування, він може бути примусово виведений з цього стану в випадках: масив семафорів було видалено з системи або процес отримав сигнал, який підлягає обробці.

Виконання різноманітних керуючих операцій (включно з видаленням) над набором семафорів:

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

Спочатку всі семафори ініціюються нульовим значенням.

Порядок виконання

1. Вивчити теоретичну частину лабораторного проєкту.
2. Написати програму, що створює дочірній процес. Батьківський процес створює семафор (сем1) та загальний файл. Дочірній процес записує у файл по одному рядку всього 100 рядків виду номер рядка pid_процесу поточний час (мілісекунди (мсек)). Батьківський процес читає з файлу рядки і виводить їх на екран у наступному вигляді: pid рядок_прочитаний_з_файлу. Семафор сем1 використовується процесами для вирішення, кому з процесів отримати доступ до файлу.
3. Написати програму, що створює дочірній процес. Батьківський процес створює неіменований канал. Дочірній процес записує в канал 100 рядків виду номера рядка pid_процесу поточний час (мілісекунди (мсек)). Батьківський процес читає з каналу рядки і виводить їх на екран у наступному вигляді: pid_рядок_прочитаний_з_файлу.

Варіанти індивідуальних завдань

1. Створити два дочірні процеси. Батьківський процес створює семафор (сем1) та загальний файл, відображений у пам'яті. Обидва дочірні процеси безперервно записують у файл по 100 рядків виду номер_рядка pid_процесу поточний_час (мілісекунди (мсек)). Усього процеси мають записати 1000 рядків. Семафор сем1 використовується процесами для дозволу, кому з процесів отримати доступ до файлу. Батьківський процес читає з файлу по 75 рядків і виводить їх на екран. Дочірні процеси розпочинають операції з файлом після отримання сигналу *SIGUSR1* від батьківського процесу.
2. Створити два дочірні процеси. Батьківський процес створює семафори (сем1), (сем2) та 2 неіменовані канали (кан1 та кан2). Обидва дочірні процеси неперервно записують в канали по 100 рядків виду номер_рядка pid_процесу поточний_час (мілісекунди (мсек)). Усього процеси мають записати 1000 рядків. Семафори (сем1), (сем2) використовуються процесами для дозволу, кому з процесів отримати доступ до каналу. Батьківський процес читає з кожного каналу по 75 рядків і виводить їх на екран. Дочірні процеси розпочинають операції з каналами після отримання сигналу *SIGUSR2* від батьківського процесу.
3. Створити два дочірні процеси. Батьківський процес створює семафор (сем1) і пам'ять, що розділяється. Обидва дочірні процеси безперервно записують в пам'ять, що розділяється, по 100 рядків виду

номер_рядка *pid*_процесу поточний_час (мілісекунди (мсек)). Усього процеси мають записати 1000 рядків. Семафор сем1 використовується процесами для дозволу, кому з процесів отримати доступ до пам'яті, що розділяється. Батьківський процес читає з пам'яті, що розділяється, по 75 рядків і виводить їх на екран. Дочірні процеси розпочинають операції з файлом після отримання сигналу *SIGUSR1* від батьківського процесу.

ЛІТЕРАТУРА

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. Operating system concepts.- 10thEdition - Hoboken, NJ : Wiley, 2018. 1278 p.
2. Andrew S. Tanenbaum, Herbert Bos. Modern Operating Systems 5th Edition – Prentice Hall,2022. 1184 p.
3. Christopher Negus. Linux BIBLE 3th Edition – Wiley, 2022. 928 p.
4. Stallings W. Operating systems: internals and design principles.- 9th Edition -Upeer SaddleRiver, New Jersey.: Prentice- Hall, 2018. 1128 p.
5. Зайцев В.Г., Дробязко І.П. Операційні системи: навч. посіб. Київ: КПІ ім. Ігоря Сікорського, 2019. 240 с.
6. Операційні системи : навч. посіб. / І. М. Федотова-Півень та ін. Харків : ТОВ «ДІСА ПЛЮС», 2019. 217 с.
7. Погребняк Б. І., Булаєнко М. В. Операційні системи : навч. посіб. Харків : ХНУМГ ім. О. М. Бекетова, 2018. 104 с.
8. Шеховцов В.А. Операційні системи. Київ : BHV, 2005. 576 с.