Лабораторна робота № 7
Потоки, перенаправлення потоків

Мета роботи:
- ознайомлення з системними викликами для управління потоками в ОС Linux,
- перенаправлення потоків при роботі файлами і командами.

Теоретичні відомості

Управління потоками

Процес розбивається на виконувані одиниці - потоки (*threads*) (один і більше потоків).

Потік – набір послідовність виконуваних команд процесора, які використовують загальний адресний простір процесу, це елемент виконання всередині процесу: віртуальний процесор, стек або статус програми. У порівнянні з процесами взаємодія і синхронізація потоків вимагає менше часу, оскільки потоки одного процесу виконуються в одному адресному просторі.

Процес містить один або кілька потоків. Якщо процес містить тільки один потік, такий процес називається однопоточними. Це класичні процеси UNIX. Якщо процес містить більше одного потоку, такі процеси називаються багатопоточними.

Існує дві основні категорії реалізації потоків: *користувацькі потоки* - потоки, що реалізуються через спеціальні бібліотеки потоків і працюють в просторі користувача. *Потоки ядра* - потоки, що реалізуються через системні виклики і працюють в просторі ядра.

В ОС UNIX/Linux для потоків реалізований стандарт Р-потоків - POSIX (*Portable Operating System Interface*) - **pthreads** ("P" - от POSIX). Для написання багатопотокової програми API для роботи з Р-потоками надає біля 100 інтерфейсів. Кожна функція в API забезпечена префіксом **pthread_.**

Прототипи функцій роботи з потоками і необхідні типи даних містяться в заголовки <**pthread.h>.** Ці функції не включені в стандартну бібліотеку мови С, вони знаходяться в бібліотеці ***libthread***. Тому в командному рядку для ***gcc*** необхідно додати опцію «***-pthread»:***

gcc -Wall -Werror -pthread beard.c -o beard

Кожний потік має свій ідентифікатор потоку, ID потоку. В програмах на С/С++ для ID потоків слід використовувати тип ***pthread_t*** з ***<sys/types.h>.***

При роботі з потоками використовуються основні функції:
- створення потоку;
- блокування роботи потоку в очікування завершення іншого;
- дострокове завершення потоків;
- завершення роботи потоків.

Створення потоку. Потоки створюються функцією ***pthread_create***, яка має наступну сигнатуру

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

Ця функція визначена в заголовки <*pthread.h*>.

Перший параметр цієї функції є вказівником на змінну типу ***pthread_t***, в яку буде записано адресу ідентифікатора створюваного потоку – ID.

Другий параметр є вказівником на змінну типу ***pthread_attr_t*** , використовується для установки атрибутів потоку. Цей об'єкт управляє деталями взаємодії потоку з іншою програмою. Якщо параметр дорівнює NULL, то потік буде створений з атрибутами за замовчуванням.

Третім параметром функції *pthread_create* повинна бути адреса функції потоку – вказівник на функцію потоку. Ця функція відіграє для потоку ту ж роль, що функція *main* для головної програми. Функція потоку приймає один параметр типу покажчик на *void* і повертає значення типу вказівник на *void*.

Четвертий параметр функції *pthread_create* має тип void*. Цей параметр може використовуватися для передачі значення як аргумент у функцію потоку. Через нього можна передавати новому потоку параметри.

Після виклику *pthread_create* функція потоку буде запущена на виконання паралельно з іншими потоками програми.

Функція повертає 0 в разі успіху, код помилки – в разі невдачі.

Ідентифікатори потоків (TID -*Thread ID***)** для потоків є аналогами ідентифікаторів процесів (PID). У той час як *PID призначаються ядром Linux, TID призначаються лише бібліотекою Р-потоків*. Цей тип представлений *pthread_t*, і POSIX не вимагає, щоб він був арифметичним. TID нового потоку визначається за допомогою аргументу *thread* при успішному виклику *pthread_create()*. Потік може отримати свій TID при запуску за допомогою функції ***pthread_self():***

```
#include <pthread.h>
pthread_t pthread_self (void);
```

Використання функції: ***const pthread_t me=pthread_self();***

Блокування роботи потоку. Приєднання дозволяє одному з потоків блокуватися в очікуванні завершення іншого:

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

Після успішного виконання викликаючий потік блокується до тих пір, поки потік, вказаний як *thread*, не завершиться (якщо потік *thread* вже завершено, функція *pthread_join()* повертається негайно). Як тільки *thread* завершується, викликаючий потік активізується і, якщо *retval* не дорівнює NULL, отримує значення завершеного процесу, яке повертається, і передане *pthread_exit()* або повернене від його стартової процедури. Після цього можна сказати, що потоки *приєдналися* один до одного. *Приєднання завжди дозволяє потокам синхронізувати своє виконання по відношенню до періоду існування інших потоків*. Всі потоки в Р-потоках є рівноправними; кожний потік може приєднуватися до будь-якого іншого. Один потік може приєднуватися до багатьох (фактично, як ми скоро побачимо, найчастіше один головний потік очікує інших потоків, які сам і створив), але тільки один потік може намагатися приєднатися до певного іншого, декілька потоків не повинні намагатися приєднатися до будь-якого одного.

В разі успіху функція *pthread_join()* повертає 0, в разі помилки - код помилки.

Дострокове завершення потоків. Р-потоки викликають завершення інших потоків через їх скасування. Це забезпечує функція *pthread_cancel():*

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

Успішний виклик *pthread_cancel()* надсилає запит на скасування потоку, представленому через ідентифікатор потоку *thread*. Чи може потік бути скасований і коли, залежить від його *стану відміни і типу скасування відповідно*.

Стан відміни потоку може бути доступно або недоступно. За замовчуванням він є доступним для нових потоків. З іншого боку, тип скасування вказує, коли відбувається скасування. Потоки можуть змінювати свій стан через ***pthread_setcancelstate():***

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

Тип скасування потоку може бути *асинхронним* або *відкладеним*; за замовчуванням зазвичай встановлений останній. З асинхронним типом скасування потік може бути убитий в будь-якій точці після отримання команди на скасування. З відкладеним типом потік може бути убитий тільки в спеціальних точках скасування, які є функціями Р-потоків або бібліотеки С і являють собою безпечні моменти, в яких викликаючий потік може бути перерваний.

Функція повертає 0 в разі успіху, в разі невдачі – код помилки.

Завершення роботи потоків. Завершення роботи потоків дуже схоже на завершення роботи процесів, за винятком того, що, коли потік завершується, інші потоки в процесі продовжують виконуватися. Потоки можуть перериватися за наступних обставин:

- якщо потік повертається зі стартової процедури, він переривається; це аналог «виходу за границі» в *main();*
- якщо потік викликає функцію *pthread_exit()*, він завершується; це аналог виклику *exit();*
- якщо потік скасовується іншим потоком через функцію *pthread_cancel(),* він завершується; це аналог відправки сигналу SIGKILL через *kill().*

Найпростіший шлях потоку для завершення самого себе, - це «вихід за границі» своєї початкової процедури. Однак часто потрібно завершити потік десь в глибині стека виклику функції, достатньо далеко від стартової процедури. Для таких випадків в Р-потоках є виклик ***pthread_exit(),*** потоковий еквівалент *exit():*

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

Після виконання викликаючий потік завершується; *retval* (вилучення) забезпечується для кожного потоку, що очікує завершення.

Потік завершується при звичайному поверненні з функції потоку, коли величина, яка повертається ***return***, буде значенням, яке повертається потоком.

***Приклад 1 роботи з потоком*:**

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
int i = 0;
void* thread_func(void *arg) {
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
        for (i=0; i < 4; i++)  {
```

```
                                printf("I'm still running!\n");
                                 }
            pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
            pthread_testcancel();
            printf("YOU WILL NOT STOP ME!!!\n");
            return NULL;
    }
    int main(int argc, char * argv[]) {
           pthread_t thread;
           pthread_create(&thread, NULL, thread_func, NULL);
           pthread_cancel(thread);
           printf("Requested to cancel the thread\n");
           pthread_join(thread, NULL);
           printf("The thread is stopped.\n");
    return EXIT_SUCCESS;
    }
```

Приклад 2 багатопоточної програми

```
    #include <pthread.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <time.h>
    #include <unistd.h>
    #define NUM_THREADS 6
    void *thread_function(void *arg);
    int main() {
        int res;
        int lots_of_threads;
        pthread_t a_thread[NUM_THREADS];
        void *thread_result;
        srand((unsigned)time(NULL));
        for (lots_of_threads=0; lots_of_threads < NUM_THREADS;
    lots_of_threads++)
             {
              res=pthread_create (&(a_thread[lots_of_threads]),NULL,
                             thread_function, (void *)&lots_of_threads);
              if (res != 0) {
                      perror("Thread creation failed");
                      exit(EXIT_FAILURE);
                             }
              }
    printf("Waiting for threads to finish...\n");
    for (lots_of_threads = NUM_THREADS - 1; lots_of_threads >= 0;
    lots_of_threads--)
             {
              res=pthread_join               (a_thread[lots_of_threads],
&thread_result);
              if (res == 0) printf("Picked up a thread\n");
              else perror("pthread_join failed");
              }
    printf("All done\n");
    exit(EXIT_SUCCESS);
    }
    void *thread_function(void *arg) {
```

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

Синхронізація потоків

Усі потоки виконуються в одному адресному просторі. У зв'язку з цим постає проблема спільного використання загальних змінних, доступу до певного ресурсу, оскільки в один момент часу тільки єдиний потік повинен працювати з певним розділюваним ресурсом. Така задача має назву *забезпечення взаємовиключення*, а ділянки програмного коду, в яких потоки виконують операції з розділюваними ресурсами, називаються *критичними секціями.* З іншого боку можлива ситуація, коли потоку для продовження своєї роботи потрібно результат виконання іншого потоку, що потребує синхронізації дій. Для узгодженості взаємодії потоків розроблені *засоби синхронізації потоків*:

м'ютекси (взаємні виключення), семафори і умовні змінні.

Синхронізація і взаємовиключення забезпечуються за рахунок атомарности виконуваних операцій над м'ютексів і семафора. Атомарної називають операцію, яка не може бути перервана в ході свого виконання.

М'ютекс дозволяє потокам управляти доступом до даних. При використанні м'ютекса тільки один потік в певний момент часу може заблокувати м'ютекс і отримати доступ до ресурсу (право на його використання). По завершенні роботи з ресурсом потік повинен повернути це право, розблокувавши м'ютекс. Якщо будь-який потік звернеться до вже заблокованого м'ютексу, то він буде змушений чекати розблокування м'ютекса потоком, який їм володіє.

Прототипи функцій для виконання операцій над м'ютексів описуються в файлі *pthread.h.* Нижче наводяться прототипи найбільш часто використовуваних функцій разом з поясненням їх синтаксису і виконуваних ними дій.

#include <pthread.h>

pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

ініціалізує м'ютекс *mutex* із зазначеними атрибутами attr або з атрибутами за замовчуванням (при вказівці 0 в якості *attr*).

#include <pthread.h>

int pthread_mutex_destroy (pthread_mutex_t *mutex); знищує м'ютекс mutex.

#include <pthread.h>

int pthread_mutex_lock (pthread_mutex_t *mutex);

виконує блокування або замикання викликаючого потоку, поки м'ютекс, вказаний як mutex не стане доступним.

Якщо м'ютекс вже заблокований, то потік, який викликав, буде заблокований до розблокування мьютекса.

```
#include <pthread.h>
```

int pthread_mutex_unlock (pthread_mutex_t *mutex);

негайне розблокування або відмикання, або вивільнення м'ютекса mutex.

Приклад 3 використання мьютекса для контролю доступу до змінної. У наведеному нижче коді функція *increment_count* використовує м'ютекс, щоб гарантувати атомарность (цілісність) модифікації розділюваної змінної *count*. Функція *get_count()* використовує м'ютекс, щоб гарантувати, що змінна *count* атомарному зчитується.

```
#include <stdio.h>
#include <pthread.h>
pthread_mutex_t count_mutex;
long count;
void *increment_count(void *arg) {
        pthread_mutex_lock(&count_mutex);
        count=count+1;
        pthread_mutex_unlock(&count_mutex);
        return NULL;
}
long get_count() {
      long c;
      pthread_mutex_lock(&count_mutex);
      c = count;
      pthread_mutex_unlock(&count_mutex);
      return (c);
}
int main(){
      pthread_t thread[5];
      pthread_mutex_init(&count_mutex, NULL);
      for (int i=0; i<5;i++){
           pthread_create(&(thread[i]), NULL, increment_count, NULL);
           printf("%ld\n", get_count());
      }
      printf("\n");
      return 0;
}
```

Приклад 3а) Робота з м'ютексом для доступу до змінної:

```
#include <stdio.h>
#include <pthread.h>
#define LIMIT 16
pthread_mutex_t count_mutex;
long count = 0;
void *increment_count(void *arg) {
        pthread_mutex_lock(&count_mutex);
        count=count+1;
        printf("%ld\n", count);
        pthread_mutex_unlock(&count_mutex);
        return NULL;
}
int main (void) {
       pthread_t a_thread[LIMIT]
       pthread_mutex_init(&count_mutex, NULL);
       for(register int=1; i<LIMIT; i++){
          pthread_create(&(a_thread[i]),NULL, increment_count, NULL);
         //printf("%ld\n", get_count());
        }
       for(register int i =0; i<LIMIT; i++ {
```

```
            pthread_join(a_thread[i], NULL);
          }
        pthread_mutex_destroy(%count_mutex);
        return 0;
}
```

Приклад 4 багатопотокової програми з синхронізацією з використанням м'ютексів

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <math.h>
#define SIZE_I 2
#define SIZE_J 2
float X[SIZE_I][SIZE_J];
float S[SIZE_I][SIZE_J];
int count = 0; // глобальний лічильник
struct DATA_ {
        double x;
        int i;
        int z;
};
typedef struct DATA_ DATA;
pthread_mutex_t lock; // Блокування
// Функція для обчислень
double f(double x) { return x*x; }
// Потокова функція для обчислень
void *calc_thr (void *arg) {
         DATA* a = (DATA*) arg;
         X[a->i][a->z] = f(a->x);
         // установка блокування
         pthread_mutex_lock(&lock);
         // зміна глобальної змінної
         count ++;
         // зняття блокування
         pthread_mutex_unlock(&lock);
         delete a;
         return NULL;
}
// Потокова функція для введення
void *input_thr(void *arg) {
         DATA* a = (DATA*) arg;
         printf("S[%d][%d]:", a->i, a->z);
         scanf("%f", &S[a->i][a->z]);
         delete a;
         return NULL;
}
int main() {
//масив ідентифікаторів потоків
pthread_t thr[ SIZE_I * SIZE_J ];
// ініціалізація м'ютекса
pthread_mutex_init(&lock, NULL);
DATA *arg;
// Введення даних для обробки
```

```
for (int i=0;i<SIZE_I; i++) {
for (int z=0; z<SIZE_J; z++) {
            arg = new DATA;
            arg->i = i;
            arg->z = z;
// створення потоку для введення елементів матриці
pthread_create (&thr[ i* SIZE_J + z ], NULL, input_thr, (void *)arg);
}      // for (int z=0; z<SIZE_J; P ++z)
}   // for (int i=0;i<SIZE_I; P ++i)
// Очікування завершення усіх потоків введення даних
// ідентифікатори потоків зберігаються у масиві thr
for(int i = 0; i < SIZE_I*SIZE_J; i++) pthread_join (thr[i], NULL);
// Обчислення елементів матриці
pthread_t thread;
printf("Start calculation\n");
for (int i=0;i<SIZE_I; i++) {
       for (int z=0; z<SIZE_J; z++) {
            arg=new DATA;
            arg->i=i;
            arg->z=z;
            arg->x = S[i][z];
            // створення потоку для обчислень
            pthread_create (&thread, NULL, calc_thr, (void *)arg);
            // переведення потоку у режим від'єнання
      pthread_detach(thread);
            // for (int z=0; z<SIZE_J; z++)
}      // for (int i=0;i<SIZE_I; i++)
do {
// Основний процес "засинає" на 1с
// Перевірка стану обчислень
printf("finished %d threads.\n", count);
} while ( count < SIZE_I*SIZE_J);
// Виведення результатів
for (int i=0;i<SIZE_I; i++) {
       for (int z=0; z<SIZE_J; z++) {
            printf("X[%d][%d] = %f\t", i, z, X[i][z]);
} printf("\n");
}
// видалення м'ютекса
pthread_mutex_destroy(&lock);
return 0;
}
```

Семафор призначений для синхронізації потоків щодо дій та даних. Семафор - це захищена змінна, значення якої можна опитувати і міняти тільки за допомогою спеціальних операцій *P* і *V* і операції ініціалізації. Семафор може приймати ціле невід'ємне значення. При виконанні потоком операції *P* над семафором *S* значення семафора зменшується на 1 при S> 0 або потік блокується, «чекаючи на семафорі», при S=0. При виконанні операції *V(S)* відбувається пробудження одного з потоків, які очікують на семафорі *S*, а якщо таких немає, то значення семафора збільшується на 1. Як випливає з вищесказаного, при вході в критичну секцію потік повинен виконувати операцію *P*, а при виході з критичної секції операцію *V*.

Прототипи функцій для маніпуляції з семафора описуються у файлі ***semaphore.h.*** Нижче наводяться прототипи функцій разом з поясненням їх синтаксису і виконуваних ними дій.

int sem_init (sem_t *sem, int pshared, unsigned int value); - ініціалізація семафора *sem* значенням *value*. В якості *pshared* завжди необхідно вказувати 0.

int sem_wait (sem_t *sem); - «очікування на семафорі». Виконання потоку блокується до тих пір, поки значення семафора не стане позитивним. При цьому значення семафора зменшується на 1.

int sem_post (sem_t *sem); - збільшує значення семафора *sem*.

int sem_destroy (sem_t *sem); - знищує семафор *sem*.

int sem_trywait (sem_t *sem); - неблокуючий варіант функції *sem_wait*. При цьому замість блокування викликав потоку функція повертає управління з кодом помилки в якості результату роботи.

Приклад 5 багатопотокової програми з синхронізацією семафора

```
    #include "main.h"
    #include <iostream.h>
    #include <semaphore.h>
    #include <fstream.h>
    #include <stdio.h>
    #include <error.h>
    using namespace std;

    void* WriteToFile(void*);
    int errno;
    sem_t psem;
    ofstream qfwrite;

    int main(int argc, char **argv) {
    pthread_t tidA,tidB;
    //int n;
    char filename[]="./rezult.txt";
    qfwrite.open(&filename[0]);
    sem_init(&psem,0,0);
    sem_post(&psem))
pthread_create(&tidA,NULL,&WriteToFile,(void*)100));
         pthread_create(&tidB,NULL,&WriteToFile,(void*)100));
         pthread_join(tidA,NULL));
         pthread_join(tidB,NULL));
    sem_destroy(&psem));
    qfwrite.close();
    return 0;
    }
    void* WriteToFile(void *f){
    int max = (int)f;
    for (int i=0; i<=max; i++)
    {
           sem_wait(&psem);
           qfwrite<<pthread_self()<<"-writetofilecounter i="<<i<<endl;
           qfwrite<<flush;
           sem_post(&psem);
    }
    return NULL;
```

```
}
```

Умовна змінна дозволяє потокам очікувати виконання деякої умови (події), пов'язаної з розділюваними даними. Над умовними змінними визначені дві основні операції: *інформування* про настання події і *очікування* події. При виконанні операції «*інформування*» один з потоків, які очікують на умовну змінну, відновлює свою роботу. Умовна змінна завжди використовується спільно з м'ютексів. Перед виконанням операції «очікування» потік повинен заблокувати м'ютекс. При виконанні операції «очікування» зазначений м'ютекс автоматично розблокується. Перед відновленням очікує потоку виконується автоматичне блокування м'ютекса, що дозволяє потоку увійти в критичну секцію, після критичної секції рекомендується розблокувати м'ютекс. При подачі сигналу іншим потокам рекомендується так само функцію «сигналізації» захистити м'ютексом.

Прототипи функцій для роботи з умовними змінними містяться в файлі ***pthread.h.*** Нижче наводяться прототипи функцій разом з поясненням їх синтаксису і виконуваних ними дій.

pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t *attr); - ініціалізує умовну змінну *cond* із зазначеними атрибутами *attr* або з атрибутами за замовчуванням (при вказівці 0 в якості attr).

int pthread_cond_destroy (pthread_cond_t *cond); - знищує умовну змінну *cond*.

int pthread_cond_signal (pthread_cond_t *cond); - інформування про настання події потоків, які очікують на умовній змінної *cond*.

int pthread_cond_broadcast (pthread_cond_t *cond); - інформування про настання події потоків, які очікують на умовній змінної *cond*. При цьому відновлені будуть все очікують потоки.

int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex); - очікування події на умовну змінну *cond*.

Перенаправлення введення/виведення в Linux

Стандартні потоки введення і виведення в Linux є одним з найбільш поширених засобів для обміну інформацією процесів, а перенаправлення «>», «>>» і «|» є однією з найбільш популярних конструкцій командного інтерпретатора.

Введення і виведення розподіляється між трьома стандартними потоками:

stdin - стандартне введення (клавіатура), номер потоку - 0;

stdout - стандартне виведення (екран), номер потоку - 1;

stderr - стандартна помилка (виведення помилок на екран), номер потоку - 2.

Зі стандартного введення команда може тільки зчитувати дані, а два інших потоки можуть використовуватися тільки для запису. Дані виводяться на екран і зчитуються з клавіатури, так як стандартні потоки за замовчуванням асоційовані з терміналом користувача. Потоки можна підключати до чого завгодно: до файлів, програм і навіть пристроїв. У командному інтерпретаторі *bash* така операція називається перенаправленням:

<*file* - використовувати файл як джерело даних для стандартного потоку введення,

>*file* - направити стандартний потік виведення в файл; якщо файл не існує, він буде створений, якщо існує - перезаписаний зверху;

2>*file* - направити стандартний потік помилок в файл; якщо файл не існує, він буде створений, якщо існує - перезаписаний зверху;
>>*file* - направити стандартний потік виведення в файл; якщо файл не існує, він буде створений, якщо існує - дані будуть дописані до нього в кінець;
2>>file - направити стандартний потік помилок в файл; якщо файл не існує, він буде створений, якщо існує - дані будуть дописані до нього в кінець;
&>file або> *&file* - направити стандартний потік виведення і стандартний потік помилок в файл; інша форма запису*:> file 2>&1*.

Стандартне введення - стандартний вхідний потік передає дані від користувача до програми.

cat> myfile - введення тексту з клавіатури у файл, після введення кожного рядка натискається *Enter*, по завершенні введення тексту натискається Ctrl + D, що означає кінець файлу EOF.

cat myfile - виведення інформації на екран з файлу.

cat file1 >> file2 - дописати вміст файлу file1 у файл file2.

Команда ***cat*** зазвичай використовується для об'єднання вмісту файлів.

cat file1 file2 file3 > file4 команда об'єднання трьох файлів в один файл file4.

Стандартне виведення - стандартний вихідний потік не перенаправляється в який-небудь файл, а виводить текст на дисплей терміналу. Команда ***echo*** виводить на екран будь-який аргумент (текст або значення змінних), який передається йому в командному рядку: ***echo*** Example.

echo> file1 "текст" - перенаправлення виведення за допомогою символу ">", якщо файл із таким ім'ям вже існує, то він буде перезаписаний;

echo>>file2 "***текст додається***" - перенаправлення виведення за допомогою символу ">>", новий текст буде додано в кінець файлу;

echo "текст на принтер" | lp - передача стандартного виведення однієї команди на стандартний вхід іншої за допомогою символу "|", текст необхідно роздрукувати на принтері.

Канали. Канали використовуються для перенаправлення потоку з однієї програми в іншу.

Особливим варіантом перенаправлення виведення є організація програмного каналу (іноді називає трубопроводом або конвеєром, оператор «|»). Для цього дві або декілька команд, таких, як виведення попередньої слугує введенням для наступної та розділяються символом вертикальної риски – «|». При цьому стандартний вихідний потік команди, розташованої ліворуч від символу «|», направляється на стандартне введення програми, розташованої праворуч від символу «|», наприклад:

cat myfile | grep Linux | wc -l .

Команда ***grep*** відшукує слово ***Linux*** у файлі, команда ***wc -l*** обчислює кількість рядків у слові. Рядок означає, що виведення команди *cat*, тобто з файлу *myfile*, буде спрямовано на вхід команди *grep*, яка виділить тільки рядки, що містять слово *"Linux"*. Виведення команди *grep* буде, в свою чергу, спрямовано на вхід команди *wc -l*, яка підрахує кількість таких рядків.

Програмні канали використовуються для того, щоб скомбінувати кілька маленьких програм, кожна з яких виконує тільки певні перетворення над своїм вхідним потоком, для створення узагальненої команди, результатом якої буде якесь більш складне перетворення.

Завдання:

1. Опанувати команди по роботі з потоками.
2. Ознайомитися із засобами синхронізації потоків.
3. Ознайомитися із стандартними потоками введення/виведення.
4. Підготувати звіт для викладача про виконання лабораторної роботи і представити його.

Хід виконання роботи

1. Використати вихідні тексти, наведених вище п’яти прикладів для потоків, м’ютексів та семафорів, для створення програм.
2. Пояснити та прокоментувати результати роботи програм та їх особливості.
3. Створити файл з особистими даними (група, прізвище, ім’я, по-батькові), використовуючи стандартне введення з клавіатури. Вивести дані на екран. Додати у файл інформацію про ваше хоббі. Створити канал.

Підготувати звіт

1. Описати хід виконання поставлених завдань, надаючи знімок екрану (screenshot).
2. Висновки по роботі.

Контрольні питання

1. У чому полягає відмінність потоку і процесу?
2. Які вам відомі типи потоків?
3. Які вам відомі основні функції при роботі з потоками?
4. Які вас відомі засоби синхронізації потоків?
5. Які стандартні потоки введення/виведення?
6. Які види перенаправлення вам відомі?
7. Що таке канали?

Література

1. Уорд Б. Внутреннее устройство Linux. Санкт-Петербург : Питер, 2016. 384 с.
2. Негус К., Каэн Ф. Ubuntu и Debian Linux для продвинутых: более 1000 незаменимых команд. Санкт-Петербург : Питер, 2014. 384 с.
3. Немет Эви, Гарт Снайдер, Трент Хейн, Бэн Уэйли. Unix и Linux: руководство системного администратора, 4-е изд. : Пер. с англ. Москва : ООО “И.Д. Вильямс”, 2012. 1312 с.
4. Колисниченко Д. Linux от новичка к профессионалу. Санкт-Петербург : БХВ-Питер, 2016. 672 с.