Лабораторна робота №2. Системні виклики

Мета

Метою виконання цього комп'ютерного практикуму є знайомство з середовищем розробки в ОС сімейства Unix і системними викликами цих ОС.

В результаті його виконання будуть отримані навички написання програм для ОС сімейства Unix і відбудеться оволодіння методом взаємодії з ОС за допомогою системних викликів.

Завдання

Необхідно написати програму на мові C, яка виконує те ж завдання, що і в роботі №1. Ця программа може і повинна використовувати ті ж самі утиліти, які використовувались в роботі №1 (`cat`, `grep`, `sort`, `head`, `awk`, `uniq`, `cut`, `paste` і т.д.), але логіка роботи самого Shell має бути реалізована за допомогою операторів мови C і системних викликів (обов'язковим є використання викликів `fork`, варіантів виклику `exec`, `wait` або `waitpid`, `open`, `close`, `pipe`, dup2). Під логікою роботи Shell мається на увазі:

- запуск процесів і очікування їх результатів
- відкриття файлів, перенаправлення вводу-виводу, pipe
- умовні вирази і цикли

Також в C може бути реалізовано підрахунок агрегованих значень і форматний друк.

Приклад подібної програми наведено нижче.

Системні виклики

Системні виклики в UNIX-системах — це інтерфейс, через який ОС надає сервіси користувацкій програмі. Вони реалізують такі функції, як:

- управління вводом-виводом
- робота з файлами
- управління процесами
- управління механізмами IPC

* pdf generated on 2014-10-20 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

- управління пам'яттю
- робота з мережею
- і т.і.

Стандартна бібліотека C, а також інших мов програмування, реалізує свої функції у наведених сферах (наприклад, роботу з файлами і ввод-вивід) як обгортки навколо системних викликів. Але з системними викликами можна працювати напряму, в обхід стандартної бібліотеки. Більш того, не всі системні виклики мають аналоги в стандартній бібліотеці: функції управління процесами, роботи з сокетами і нитками управління є прикладами таких функцій.

Нехай нам необхідно вирішити наступну задачу: знайти в текстовому файлі `1.txt` всі рядки, які містять в собі дату `1/01/2000`. В UNIX середовищі це можна зробити за допомогою такого конвеєру: `cat 1.txt | grep 1/01/2000`. Нижче наведений приклад програми на мові C з використанням системних викликів, яка реалізує те ж саме.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    int fd[2], status;

    /* Створюємо анонімний канал */
    pipe(fd);

    /* Клонуємо поточний процес */
    pid_t pid1 = fork();

    /* Зверніть увагу на те, що змінна pid буде
     * існувати і в оригінальному процесі,
     * і в його клоні, але в другому випадку буде
     * дорівнювати нулю. Саме цей факт використовується
     * в наступній конструкції */

    if (!pid1) { // childpid == 0 => це дочірній процес
        /* В цьому процесі ми будемо писати в канал */
        // перенапрявляємо STDOUT (fd==1) у канал
        dup2(fd[1], 1);

        /* Необхідно закрити обидва кінця каналу */
```


* pdf generated on 2014-10-20 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

```
    close(fd[0]);
    close(fd[1]);

    /* Запускаємо cat */
    char* command[3] = {"/bin/cat", "1.txt", 0};
    execvp(command[0], command);

    /* Якщо не відбулося ніяких помилок,
     * execvp() не завершується і ми
     * ніколи не досягнемо цієї ділянки коду.
     * Якщо ж ми все-таки сюди дістались,
     * клону слід завершитися з кодом повернення,
     * який сигналізує про помилку */
    exit(EXIT_FAILURE);
} else if (pid1 == -1) {
    /* fork() повертає -1 у випадку помилки */
    fprintf(stderr, "Can't fork, exiting...\n");
    exit(EXIT_FAILURE);
}

/* Це батьківський процес */

/* Клонуємо його знову */
pid_t pid2 = fork();

if (!pid2) {
    /* В цьому процесі будемо зчитувати з каналу */
    // перенапрявляємо вивід каналу у STDIN (fd==0)
    dup2(fd[0], 0);
    close(fd[0]);
    close(fd[1]);

    /* Запускаємо grep */
    char* command[3] = {"/bin/grep", "1/01/2010", 0};
    execvp(command[0], command);

    exit(EXIT_FAILURE);
} else if (pid2 == -1) {
    fprintf(stderr, "Can't fork, exiting...\n");
    exit(EXIT_FAILURE);
}

/* Це батьківський процес */
```


* pdf generated on 2014-10-20 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

```
        close(fd[0]);
        close(fd[1]);

        /* Очікуємо завершення породжених процесів */
        waitpid(pid1, NULL, 0);
        waitpid(pid2, &status, 0);

        /* Завершуємося з кодом повернення grep */
        exit(status);

        return 0;
    }
```

Під час роботи з системними викликами обов'язково необхідно обробляти код повернення з них, оскільки інформація про помилки, які могли відбутися у виклику повертається у цьому коді.

Отримати докладну справку по системним викликам можна за допомогою `man` — ним присвячений розділ 2 (наприклад, ввівши в консолі `man 2 fork` можна отримати справку про системний виклик `fork`).

Література

- Learn C The Hard Way

Компіляція, збирання і відлагодження програм в UNIX-середовищі

- http://users.actcom.co.il/~choo/lupg/tutorials/
- http://www.gnu.org/software/make/manual/make.html
- http://www.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html
- Tutorial of gcc and gdb
- Debugging with GDB
- http://sysadvent.blogspot.com/2010/12/day-15-down-ls-rabbit-hole.html
- Learning C with gdb

Системні виклики для запуску процесів і управлінням вводом-виводом

* pdf generated on 2014-10-20 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting

* pdf generated on 2014-10-20 from markdown sources with SBCL 1.1.14 on Linux 3.11.0-12-generic with 3bmd, cl-pdf & cl-typesetting