# Лабораторная работа №2. Системные вызовы

## Цель

Целью выполнения этого компьютерного практикума является знакомство со средой разработки в ОС семейства Unix и системными вызовами этой ОС.

В результате его выолнения будет получены бызовые навыки написания программ для ОС семейства Unix и произойдет овладение методом взаимодействия с ОС с помощью системных вызовов.

## Задание

Необходимо написать программу на языке C, которая выполняет то же задание, что и в работе №1. Эта программа должна использовать те же самые утилити, которые использовались в работе №1 (`cat`, `grep`, `sort`, `head`, `awk`, `uniq`, `cut`, `paste` и т.д.), но логика рыботи самого Shell должна быть реализованна с помощью операторов языка C и системных вызовов (обязательно использование вызовов `fork`, `exec`, `wait` или `waitpid`, `open`, `close`, `pipe`, `dup2`).

Под логикой работы Shell имеется в виду:

- запуск процессов и ожидание их результатов
- открытие файлов, перенаправление ввода-вывода, pipe
- условные выражения и циклы

Также в C может быть реализован подсчет агрегированных значений и форматная печать.

Пример подобной программы приведен ниже.

## Системные вызовы

Системные вызовы в UNIX-системах — это интерфейс, через который ОС предоставляет сервисы пользовательской программе. Они реализуют такие функции, как:

- управление вводом-выводом
- работа с файлами

 

* 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

- управление процесами
- управление механизмами IPC
- управление памятью
- роабота с сетью
- и т.д.

Стандартная библиотека C, а также других языков програмирования, реализует свои функции в указаных выше сферах (например, работу с файлами и ввод-вивод) как обертки над системными вызовами. Но с системными вызовами можно работать напрямую, в обход стандартной библиотеки. Более того, не все системные вызовы имеют аналоги в стандартной библиотеке: функции управления процессами, работи с сокетами и нитями управления являются примерами таких функций.

Пускай нам необходимо решить соледующую задачу: найти в текстовом файле `1.txt` все строки, которые содержат в себе дату `1/01/2000`. В UNIX среде это можно сделать с помощью следующего конвеера: `cat 1.txt | grep 1/01/2000`. Ниже приведен пример програмы на языке C с использованием системных вызовов, который реализует то же самое:

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

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

    /* Создаем анонимный канал */
    pipe(fd);

    /* Клонируем текущий процесс */
    pid_t pid1 = fork();

    /* Обратите внимание, что переменная pid будет
     * существовать и в исходном процессе, и в его клоне,
     * но в дочернем процессе она будет равна нулю.
     * Именно этот факт используется ниже */

    if (!pid1) { // childpid == 0 => это дочерний процесс
        /* В этом процессе мы будем писать в канал */
        // перенаправляем STDOUT (fd==1) в канал
```


* 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

```
    dup2(fd[1], 1);

    /* Обязательно закрыть оба конца канала */
    close(fd[0]);
    close(fd[1]);

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

    /* Если не произошло ошибок, execvp()
     * не завершается и мы никогда не попадем
     * в этоту часток кода. Если же мы все-таки
     * здесь окажемся, клону следует завершиться
     * с кодом возврата, который сигнализирует
     * про ошибку */
    exit(EXIT_FAILURE);
} else if (pid1 == -1) {
    /* fork() возвращает -1 в случае ошибки */
    fprintf(stderr, "Can't fork, exiting...\n");
    exit(EXIT_FAILURE);
}

/* Это родительский процесс */

/* Клонируем его снова */
pid_t pid2 = fork();

if (!pid2) {
    /* В этом процессе мы будем читать из канала */
    // перенапрявляем вывод канала в STDIN (fd==0)
    dup2(fd[0], 0);

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

    /* Запускаем 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);
```

 

* 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

```
        }

        /* Это родительский процесс */

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

        /* Ожидаем завершения порожденных процессов */
        waitpid(pid1, NULL, 0);
        waitpid(pid2, &status, 0);

        /* Завершаемся с кодом возврата grep */
        exit(status);

        return 0;
    }
```

Работая с системными вызовами, необходимо обязательно обрабатывать код возврата из них, поскольку информация об ошибках, которые могли произойти в этом вызове, передается через этот код, и если она не будет обработана, но, во-первых, она будет утеряна, а, во-вторых, последующая работа программы станет не предсказуемой.

Получить подробную справку по системным вызовам можно в разделе 2 команды `man` (например, введя в консоли `man 2 fork` можно получить справку о системном вызове `fork`).

# Литература

- Learn C The Hard Way

## Компиляция, сборка и отладка программ в UNIX-среде

- 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


* 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

- http://sysadvent.blogspot.com/2010/12/day-15-down-ls-rabbit-hole.html
- Learning C with gdb

## Системные вызовы для запуска процессов и управления вводом-выводом

- Командная оболочка и системные вызовы
- Fork, Exec and Process control



* 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