Министерство науки и образования РФ  
Новосибирский государственный технический университет  
Кафедра ТПИ

Лабораторная работа №6

по дисциплине «Управление ресурсами в вычислительных системах»

Факультет: ПМИ

Группа: ПМИ-31

Выполнили: Махнёва Е. А.

Соловьёв Б. С

Преподаватели: Хайленко Е. А.  
 Филиппова Е. В.

Вариант: 9

1. **Цель работы**

Практическое освоение средств управления ресурсами ОС UNIX на основе

разработки программы, моделирующей работу интерпретатора в плане создания

процессов, реализующих команды в командной строке, их синхронизации и

взаимодействию по данным.

1. **Задание**

Составить программу, моделирующую работу Shell-интерпретатора при

обработке командной строки, указанной в варианте. При реализации программы

путем выдачи сообщений информировать обо всех этапах ее работы (создан

процесс, выполнение команды закончено и т.д.).

Вариант №9:

tr -d "[0-9]" < a.txt | sort | uniq > b.txt

1. **Описание использованных системных вызовов**
   1. **Функции работы с файлами/каналами**

**int dup2(int \_\_fd, int \_\_fd2) <unistd.h>**

Дублирует дескриптор, заданный первым параметром, чтобы он указывал на \_\_fd2, перед этим закрывая \_\_fd2, чтобы они указывали на один и тот же файл.

**int open(const char \*\_\_path, int \_\_oflag, ...) <fcntl2.h>**

Открывает файл, возвращая дескриптор, указывающий на него (а не потоковый тип, как в случае с fopen()). Значение третьего параметра варьируется в зависимости от выставленных во втором параметре флагов. Возможные значения флагов: O\_RDONLY, O\_WRONLY, O\_CREAT, тысячи их. Флаги можно комбинировать с использованием логической операции **|** (OR)

**int fcntl (int \_\_fd, int \_\_cmd, ...) <fcntl.h>**

Совершает указанную в \_\_cmd операцию над файловым дескриптором \_\_fd. Значения необязательных аргументов зависят от выбранной операции.

* 1. **Функции работы с процессами**

**pid\_t wait(void \*\_\_stat\_loc) <sys/wait.h>**

Ожидает поступления сигнала SIGCHLD от дочернего процесса о его завершении. Возвращает pid завершившегося процесса, а в аргумент по ссылке кладётся код, с которым потомок завершился. Если на момент вызова дочерних процессов нет, возвращает -1 и 0 соответственно.

**int execlp(const char \*\_\_file, const char \*\_\_arg, ...) <unistd.h>**

Запускает на выполнение файл \_\_file, выполняя поиск в порядке, заданном переменной окружения PATH, если путь до файла не является полным. При запуске передаются аргументы командной строки, указанные в параметрах функции по одному. Обязательным является задание нулевого указателя в конце списка аргументов.

**void exit(int \_\_status) <stdlib.h>**

Выполняет все обработчики, зарегистрированные на завершение процесса, выполняет очистку файловых дескрипторов и завершает программу с кодом завершения \_\_status.

1. **Текст программы**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <fcntl.h>

#define RD 0 // Read end of pipe

#define WR 1 // Write end of pipe

#define FORK\_ERROR 1 // Exit status for failed fork()

#define DATA\_ERROR 2

typedef unsigned char byte; // this is missing in standard libs for whatever reason.

typedef struct { byte flags; byte code; } wait\_t; // union for wait result

// process creation parameters

typedef struct {

char \*command; // executable file to be found in PATH

char \*argv0; // argv[0] string, required for exec-() functions

char \*arg1, \*arg2; // first arg, second arg. no more due to complexity

int in, out, err; // descriptors of standard streams

} param\_t;

// Fill param\_t with process creation data.

void create\_startup\_param(

param\_t \*param,

char \*command,

char \*arg1,

char \*arg2 = NULL,

int out = STDOUT\_FILENO,

int err = STDERR\_FILENO,

int in = STDIN\_FILENO) {

param->command = command;

param->argv0 = command;

param->arg1 = arg1;

param->arg2 = arg2;

param->in = in;

param->out = out;

param->err = err;

}

// Overlays current executing process with COMMAND.

int shell\_exec(param\_t \*param) {

dup2(param->out, STDOUT\_FILENO); // this is identical to fcntl(param->out, F\_DUPFD, 1)

dup2(param->err, STDERR\_FILENO);

dup2(param->in, STDIN\_FILENO);

int ret = execlp(param->command, param->argv0, param->arg1, param->arg2, NULL);

return ((ret == -1) ? -1 : 0); // return -1 on failure, 0 otherwise

}

// Create new thread in new process via fork() and run callback in it.

pid\_t fork\_thread(int(func(param\_t \*param)), void \*param = NULL) {

int childpid;

switch (childpid = fork()) {

case -1:

fprintf(stderr, "[shell]\t Could not fork to subcommand '%s'\n", ((param\_t\*)param)->command);

exit(FORK\_ERROR);

break;

case 0:

exit(func((param\_t\*)param)); // run callback and exit

break;

default:

fprintf(stderr, "[shell]\t Process %d created for '%s'\n", childpid, ((param\_t\*)param)->command);

return childpid;

break;

}

}

// Wait till ALL running subcommands end

int join\_threads() {

pid\_t wait\_pid;

wait\_t wait\_ret;

int exec\_result = 0;

while ((wait\_pid = wait(&wait\_ret)) > 0) {

printf("[shell]\t Process %d exited with code %d\n", wait\_pid, wait\_ret.code);

exec\_result |= wait\_ret.code; // OR all codes

}

return exec\_result;

}

int main(int argc, char \*\*argv) {

param\_t param;

int a = open("a.txt", O\_RDONLY); // if file doesn't exists, returns -1

if (a == -1) {

fprintf(stderr, "[MAIN]\t Could not open file 'a.txt'\n");

return DATA\_ERROR;

}

int pipes[2], pipestwo[2];

pipe(pipes);

fprintf(stderr, "[MAIN]\t First pipe was opened.\n");

pipe(pipestwo);

fprintf(stderr, "[MAIN]\t Second pipe was opened.\n");

int ret;

char buf[512];

create\_startup\_param(&param, "tr", "-d", "[0-9]", pipes[WR], STDERR\_FILENO, a);

fork\_thread(shell\_exec, &param);

fprintf(stderr, "[MAIN]\t tr -d [0-9] completed.\n");

if((ret = join\_threads()) != 0)

return ret;

close(pipes[WR]);

fprintf(stderr, "[MAIN]\t First pipe was closed for writing.\n");

create\_startup\_param(&param, "sort", NULL, NULL, pipestwo[WR], STDERR\_FILENO, pipes[RD]);

fork\_thread(shell\_exec, &param);

fprintf(stderr, "[MAIN]\t sort completed.\n");

ret = join\_threads();

close(pipestwo[WR]);

fprintf(stderr, "[MAIN]\t Second pipe was closed for writing.\n");

int b = open("b.txt", O\_WRONLY | O\_CREAT, 0600);

if (b == -1) {

fprintf(stderr, "[MAIN]\t Could not open file 'b.txt'\n");

return DATA\_ERROR;

}

create\_startup\_param(&param, "uniq", NULL, NULL, b, STDERR\_FILENO, pipestwo[RD]);

fork\_thread(shell\_exec, &param);

fprintf(stderr, "[MAIN]\t uniq completed, result is in 'b.txt'.\n");

return join\_threads();

}

1. **Результат работы программы**

|  |  |
| --- | --- |
| Содержимое a.txt: | Содержимое b.txt |
| table  67ghk  1717171  upr  upr  maxim  soap  grajdanin  upr  soap | ghk  grajdanin  maxim  soap  table  upr  pr |

Содержимое консоли:

[MAIN] First pipe was opened.

[MAIN] Second pipe was opened.

[shell] Process 8352 created for 'tr'

[MAIN] tr -d [0-9] completed.

[shell] Process 8352 exited with code 0

[MAIN] First pipe was closed for writing.

[shell] Process 8353 created for 'sort'

[MAIN] sort completed.

[shell] Process 8353 exited with code 0

[MAIN] Second pipe was closed for writing.

[shell] Process 8354 created for 'uniq'

[MAIN] uniq completed, result is in 'b.txt'.

[shell] Process 8354 exited with code 0