Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

**Лабораторная работа №1 по курсу**

**«Операционные системы»**

Группа: М8О-216БВ-24

Студент: Показеев Д.Д.

Преподаватель: Бахарев В.Д.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 25.09.25

Москва, 2024

**Постановка задачи**

**Вариант 20.**

Родительский процесс создает два дочерних процесса. Первой строкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия File с таким именем на запись для child1. Аналогично для второй строки и процесса child2. Родительский и дочерний процесс должны быть представлены разными программами. Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в pipe1 или в pipe2 в зависимости от правила фильтрации. Процесс child1 и child2 производят работу над строками. Процессы пишут результаты своей работы в стандартный вывод.

**Общий метод и алгоритм решения**

Использованные системные вызовы:

* pid\_t fork(void); – создает дочерний процесс. Возвращается PID дочернего процесса в родительском процессе и 0 в дочернем процессе. Используется для создания двух дочерних процессов, которые будут обрабатывать данные по-разному.
* int pipe(int \*fd); – создает анонимный канал (pipe) для межпроцессного взаимодействия. В программе создаются два канала: pipefd1[2] и pipefd2[2], где fd[0] – файловый дескриптор для чтения, а fd[1] – для записи.
* ssize\_t read(int fd, void \*buf, size\_t count); – читает данные из файлового дескриптора. Используется для чтения имен файлов из стандартного ввода и для чтения входных данных.
* ssize\_t write(int fd, const void \*buf, size\_t count); – записывает данные в файловый дескриптор. Используется для записи ошибок в STDERR\_FILENO и для записи данных в каналы.
* int open(const char \*pathname, int flags, mode\_t mode); – открывает или создает файл. Используется для открытия выходных файлов с правами доступа 0600.
* int dup2(int oldfd, int newfd); – дублирует файловый дескриптор. Используется для перенаправления стандартного вывода дочерних процессов в созданные файлы и стандартного ввода в каналы.
* int close(int fd); – закрывает файловый дескриптор. Используется для закрытия каналов после их использования.
* int execv(const char \*pathname, char \*const argv[]); – заменяет текущий образ процесса новой программой. Используется для запуска дочерних программ child1 и child2.
* pid\_t waitpid(pid\_t pid, int \*status, int options); – ожидает завершения дочернего процесса. Используется для ожидания завершения обоих дочерних процессов.

Программа реализует параллельную обработку входных данных с использованием межпроцессного взаимодействия через каналы.

1. Создаются два канала с помощью pipe() для передачи данных двум дочерним процессам.
2. Считываются имена двух выходных файлов из стандартного ввода с помощью read() и open().
3. Создаются два дочерних процесса с помощью fork():
   * Первый дочерний процесс перенаправляет свой стандартный вывод в первый файл с помощью dup2() и читает данные из первого канала.
   * Второй дочерний процесс перенаправляет свой стандартный вывод во второй файл и читает данные из второго канала.
4. В родительском процессе входящие данные направляются в разные каналы в зависимости от их длины (строки длиной до 10 символов в первый канал, остальные во второй).
5. Данные, поступающие в дочерние процессы, обрабатываются (в обоих случаях происходит реверс строки) и записываются в соответствующие выходные файлы.
6. Родительский процесс ожидает завершения дочерних процессов с помощью waitpid().

**Код программы**

**main.c**

#include <fcntl.h>

#include <stdint.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/wait.h>

#include <unistd.h>

int main() {

const char CHILD\_PROGRAM\_NAME[] = "child";

const char OUTPUT\_FOLDER\_NAME[] = "output\_files";

int pipefd1[2];

int pipefd2[2];

pid\_t child1;

pid\_t child2;

char main\_binary\_path[1024];

char output\_path[1024];

{

char progpath[1024];

ssize\_t len =

readlink("/proc/self/exe", progpath, sizeof(progpath) - 1);

if (len == -1) {

const char msg[] = "error: failed to read full program path\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

while (progpath[len] != '/') --len;

progpath[len] = '\0';

snprintf(main\_binary\_path, sizeof(main\_binary\_path) - 1, "%s/%s",

progpath, CHILD\_PROGRAM\_NAME);

while (progpath[len] != '/') --len;

progpath[len] = '\0';

snprintf(output\_path, sizeof(output\_path) - 1, "%s/%s", progpath,

OUTPUT\_FOLDER\_NAME);

}

if (pipe(pipefd1) == -1) {

const char msg[] = "error: failed create pipe\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

if (pipe(pipefd2) == -1) {

const char msg[] = "error: failed create pipe\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

char buf[4096];

ssize\_t bytes\_read;

bytes\_read = read(STDIN\_FILENO, buf, sizeof(buf));

if (bytes\_read < 0) {

const char msg[] = "error: failed to read from stdin\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

} else if (buf[0] == '\n' || bytes\_read == 0) {

const char msg[] = "error: invalid file name\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

char file\_path[1024];

buf[bytes\_read - 1] = '\0';

snprintf(file\_path, sizeof(file\_path) - 1, "%s/%s", output\_path, buf);

int32\_t file1 = open(file\_path, O\_WRONLY | O\_CREAT | O\_TRUNC, 0600);

if (file1 == -1) {

const char msg[] = "error: failed to open requested file\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

bytes\_read = read(STDIN\_FILENO, buf, sizeof(buf));

if (bytes\_read < 0) {

const char msg[] = "error: failed to read from stdin\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

} else if (buf[0] == '\n') {

const char msg[] = "error: invalid file name\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

buf[bytes\_read - 1] = '\0';

snprintf(file\_path, sizeof(file\_path) - 1, "%s/%s", output\_path, buf);

int32\_t file2 = open(file\_path, O\_WRONLY | O\_CREAT | O\_TRUNC, 0600);

if (file2 == -1) {

const char msg[] = "error: failed to open requested file\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

child1 = fork();

if (child1 == -1) {

const char msg[] = "error: failed to spawn new process\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

if (child1 == 0) {

dup2(file1, STDOUT\_FILENO);

close(file1);

close(pipefd1[1]);

dup2(pipefd1[0], STDIN\_FILENO);

close(pipefd1[0]);

close(pipefd2[0]);

close(pipefd2[1]);

char\* const args[] = {"child", NULL};

execv(main\_binary\_path, args);

const char msg[] = "error: failed to execv\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

child2 = fork();

if (child2 == -1) {

const char msg[] = "error: failed to spawn new process\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

if (child2 == 0) {

dup2(file2, STDOUT\_FILENO);

close(file2);

close(pipefd2[1]);

dup2(pipefd2[0], STDIN\_FILENO);

close(pipefd2[0]);

close(pipefd1[0]);

close(pipefd1[1]);

char\* const args[] = {"child", NULL};

execv("./child", args);

const char msg[] = "error: failed to execv\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

while (bytes\_read = read(STDIN\_FILENO, buf, sizeof(buf))) {

if (bytes\_read < 0) {

const char msg[] = "error: failed to read from stdin\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

} else if (buf[0] == '\n') {

break;

}

if (bytes\_read > 11) {

write(pipefd2[1], buf, bytes\_read);

} else {

write(pipefd1[1], buf, bytes\_read);

}

}

close(pipefd2[1]);

close(pipefd1[1]);

waitpid(child1, NULL, 0);

waitpid(child2, NULL, 0);

return 0;

}

**child.c**

#include <stdint.h>

#include <stdlib.h>

#include <unistd.h>

int main() {

char buf[4096];

ssize\_t bytes\_read;

int32\_t written;

int32\_t left;

int32\_t right;

while (bytes\_read = read(STDIN\_FILENO, buf, sizeof(buf))) {

if (bytes\_read < 0) {

const char msg[] = "error: failed to read from stdin\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

left = 0;

right = bytes\_read - 2;

while (left < right) {

char tmp = buf[left];

buf[left] = buf[right];

buf[right] = tmp;

++left;

--right;

}

written = write(STDOUT\_FILENO, buf, bytes\_read);

if (written != bytes\_read) {

const char msg[] = "error: failed to echo\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

}

return 0;

}

**Протокол работы программы**

./main

a

b

123

1234567890

12345678900

~/code/mai/OSi/lab1/build -> main ?

cat a

321

0987654321

~/code/mai/OSi/lab1/build -> main ?

cat b

00987654321

execve("./main", ["./main"], 0x7ffd647f8490 /\* 70 vars \*/) = 0

brk(NULL) = 0x560bd8b88000

access("/etc/ld.so.preload", R\_OK) = -1 ENOENT (No such file or directory)

openat(AT\_FDCWD, "/etc/ld.so.cache", O\_RDONLY|O\_CLOEXEC) = 3

fstat(3, {st\_mode=S\_IFREG|0644, st\_size=196755, ...}) = 0

mmap(NULL, 196755, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0x7f1e130f7000

close(3) = 0

openat(AT\_FDCWD, "/usr/lib/libc.so.6", O\_RDONLY|O\_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0000x\2\0\0\0\0\0"..., 832) = 832

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 896, 64) = 896

fstat(3, {st\_mode=S\_IFREG|0755, st\_size=2149728, ...}) = 0

mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7f1e130f5000

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 896, 64) = 896

mmap(NULL, 2174000, PROT\_READ, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7f1e12e00000

mmap(0x7f1e12e24000, 1515520, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x24000) = 0x7f1e12e24000

mmap(0x7f1e12f96000, 454656, PROT\_READ, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x196000) = 0x7f1e12f96000

mmap(0x7f1e13005000, 24576, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x204000) = 0x7f1e13005000

mmap(0x7f1e1300b000, 31792, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0x7f1e1300b000

close(3) = 0

mmap(NULL, 12288, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7f1e130f2000

arch\_prctl(ARCH\_SET\_FS, 0x7f1e130f2740) = 0

set\_tid\_address(0x7f1e130f2a10) = 8372

set\_robust\_list(0x7f1e130f2a20, 24) = 0

rseq(0x7f1e130f2680, 0x20, 0, 0x53053053) = 0

mprotect(0x7f1e13005000, 16384, PROT\_READ) = 0

mprotect(0x560bce931000, 4096, PROT\_READ) = 0

mprotect(0x7f1e13161000, 8192, PROT\_READ) = 0

prlimit64(0, RLIMIT\_STACK, NULL, {rlim\_cur=8192\*1024, rlim\_max=RLIM64\_INFINITY}) = 0

getrandom("\xa7\x82\xc9\x8e\xc9\x3b\xdf\x59", 8, GRND\_NONBLOCK) = 8

munmap(0x7f1e130f7000, 196755) = 0

**pipe2([3, 4], 0) = 0**

**pipe2([5, 6], 0) = 0**

**read(0, "a\n", 4096) = 2**

**openat(AT\_FDCWD, "a", O\_WRONLY|O\_CREAT|O\_TRUNC, 0600) = 7**

**read(0, "b\n", 4096)** = 2

**openat(AT\_FDCWD, "b", O\_WRONLY|O\_CREAT|O\_TRUNC, 0600) = 8**

rt\_sigprocmask(SIG\_BLOCK, ~[], [], 8) = 0

**clone(child\_stack=NULL, flags=CLONE\_CHILD\_CLEARTID|CLONE\_CHILD\_SETTID|SIGCHLD, child\_tidptr=0x7f1e130f2a10) = 8373**

rt\_sigprocmask(SIG\_SETMASK, [], NULL, 8) = 0

rt\_sigprocmask(SIG\_BLOCK, ~[], [], 8) = 0

**clone(child\_stack=NULL, flags=CLONE\_CHILD\_CLEARTID|CLONE\_CHILD\_SETTID|SIGCHLD, child\_tidptr=0x7f1e130f2a10) = 8374**

rt\_sigprocmask(SIG\_SETMASK, [], NULL, 8) = 0

**read(0, "123\n", 4096) = 4**

**write(4, "123\n", 4) = 4**

**read(0, "1234567890\n", 4096) = 11**

**write(4, "1234567890\n", 11) = 11**

**read(0, "12345678900\n", 4096) = 12**

**write(6, "12345678900\n", 12) = 12**

**read(0, "\n", 4096) = 1**

**close(6) = 0**

**close(4) = 0**

wait4(8373, NULL, 0, NULL) = ? ERESTARTSYS (To be restarted if SA\_RESTART is set)

--- SIGCHLD {si\_signo=SIGCHLD, si\_code=CLD\_EXITED, si\_pid=8374, si\_uid=1000, si\_status=0, si\_utime=0, si\_stime=0} ---

**wait4(8373, NULL, 0, NULL) = 8373**

**wait4(8374, NULL, 0, NULL) = 8374**

exit\_group(0) = ?

+++ exited with 0 +++

**Вывод**

Лабораторная работа демонстрирует применение системных вызовов для создания многопроцессных программ с межпроцессным взаимодействием. В ходе работы были использованы такие системные вызовы, как pipe для создания каналов передачи данных, fork для создания дочерних процессов, read и write для ввода-вывода, а также wait для синхронизации завершения дочерних процессов.