**Московский авиационный институт**

**(Национальный исследовательский университет)**

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Операционные системы»

**Лабораторная работа № 1**

Студент: Ваньков Д. А.

Группа: 80-207Б

Преподаватель: Соколов А. А.

Дата: 20.10.2018

Оценка:

Москва, 2018

1. **Постановка задачи**

Целью лабораторной работы является:

Приобретение практических навыков диагностики работы программного обеспечения.

1. **Описание**

Системные вызовы – один из уровней абстракции, предоставляемый пользовательским программам. Само управление ресурсами проходит незаметно для пользователя и осуществляется в автоматическом режиме. Любой однопроцессорный компьютер одномоментно может выполнить только одну команду. Когда процесс выполняет пользовательскую программу в режиме пользователя и нуждается в какой-нибудь услуге ОС, он должен выполнить команду системного прерывания, чтобы передать управление ОС. Затем ОС по параметрам вызова определяет, что именно требуется вызывающему процессу. После этого она обрабатывает системный вызов и возвращает управление той команде, которая следует за системным вызовом. В некотором смысле выполнение системного вызова похоже на выполнение особой разновидности вызова процедуры, с той лишь разницей, что системные вызовы входят в ядро, а процедурные – нет.

1. **Решение задачи**

Используемые системные вызовы:

* **ssize\_t write(int***fd***, const void \****buf***, size\_t***count***) -** пишет до *count* байт из буфера, на который указывает *buf*, в файле, на который ссылается файловый дескриптор *fd*.
* **ssize\_t read(int***fd***, void \****buf***, size\_t***count***) -** пытается прочитать *count* байт из файлового дескриптора *fd* в буфер, начинающийся по адресу *buf*. Для файлов, поддерживающих смещения, операция чтения начинается с текущего файлового смещения, и файловое смещение увеличивается на количество прочитанных байт. Если текущее файловое смещение находится за концом файла, то ничего не читается и **read**() возвращает ноль. Если значение *count* равно 0, то **read**() *может* обнаружить ошибки, описанные далее. При отсутствии ошибок, или если **read**() не выполняет проверки, то **read**() с *count* равным 0 возвращает 0 и ничего не меняет.
* **pid\_t fork(void) -** создаёт новый процесс посредством копирования вызывающего процесса. Новый процесс считается *дочерним* процессом. Вызывающий процесс считается *родительским* процессом. Дочерний и родительский процессы находятся в отдельных пространствах памяти. Сразу после **fork**() эти пространства имеют одинаковое содержимое.
* **int close(int***fd***) -** закрывает файловый дескриптор.
* **int pipe2(int***pipefd***[2], int***flags***) -** создаёт однонаправленный канал данных, который можно использовать для взаимодействия между процессами. Массив *pipefd* используется для возврата двух файловых описателей, указывающих на концы канала. *pipefd[0]* указывает на конец канала для чтения. *pipefd[1]* указывает на конец канала для записи. Данные, записанные в конец канала, буферизируются ядром до тех пор, пока не будут прочитаны из конца канала для чтения.

Каждая из функций - родитель и ребёнок - принимают идентификаторы потоков, на чтение и на запись. Каждый из них передаётся так, что "чтение с читабельного конца потока 1 и запись в писательный конец потока 2" и "чтение с читабельного конца потока 2 и запись в писательный конец потока1".

Функция pipe() создаёт собственный поток обмена данными. При этом в передаваемый ей массив записываются числовые идентификаторы (дескрипторы) двух "концов" потока: один на чтение, другой на запись. Поток (pipe) работает по принципу "положенное первым - первым будет считано".

int pid = fork(); Начиная с этого момента, процессов становится два. У каждого своя память. в процессе-родителе pid хранит идентификатор ребёнка. в ребёнке в этой же переменной лежит 0. Далее в каждом случае надо закрыть "лишние" концы потоков. поскольку сама программа теперь существует в двух экземплярах, то фактически у каждого потока появляются вторые дескрипторы. И если произошла хоть какая-то ошибка, то общая переменная ошибок != 0.

**Диагностика программы лабораторной работы №2 с помощью strace**

chappybunny@chappybunny:~/OS$ strace ./a.out

execve("./a.out", ["./a.out"], 0x7fff3915f950 /\* 53 vars \*/) = 0

brk(NULL) = 0x561b14723000

stat("/home/chappybunny/OS/lab5/haswell", 0x7ffc549163f0) = -1 ENOENT (No such file or directory)

openat(AT\_FDCWD, "/home/chappybunny/OS/lab5/x86\_64/libc.so.6", O\_RDONLY|O\_CLOEXEC) = -1 ENOENT (No such file or directory)

stat("/home/chappybunny/OS/lab5/x86\_64", 0x7ffc549163f0) = -1 ENOENT (No such file or directory)

openat(AT\_FDCWD, "/home/chappybunny/OS/lab5/libc.so.6", O\_RDONLY|O\_CLOEXEC) = -1 ENOENT (No such file or directory)

stat("/home/chappybunny/OS/lab5", {st\_mode=S\_IFDIR|0755, st\_size=4096, ...}) = 0

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

fstat(3, {st\_mode=S\_IFREG|0644, st\_size=86621, ...}) = 0

mmap(NULL, 86621, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0x7fec2d03a000

close(3) = 0

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

openat(AT\_FDCWD, "/lib/x86\_64-linux-gnu/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\0\260\34\2\0\0\0\0\0"..., 832) = 832

fstat(3, {st\_mode=S\_IFREG|0755, st\_size=2030544, ...}) = 0

mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7fec2d038000

mmap(NULL, 4131552, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7fec2ca38000

mprotect(0x7fec2cc1f000, 2097152, PROT\_NONE) = 0

mmap(0x7fec2ce1f000, 24576, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1e7000) = 0x7fec2ce1f000

mmap(0x7fec2ce25000, 15072, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0x7fec2ce25000

close(3) = 0

arch\_prctl(ARCH\_SET\_FS, 0x7fec2d039500) = 0

mprotect(0x7fec2ce1f000, 16384, PROT\_READ) = 0

mprotect(0x561b13ae5000, 4096, PROT\_READ) = 0

mprotect(0x7fec2d050000, 4096, PROT\_READ) = 0

munmap(0x7fec2d03a000, 86621) = 0

dup(2) = 3

fcntl(3, F\_GETFL) = 0x8402 (flags O\_RDWR|O\_APPEND|O\_LARGEFILE)

brk(NULL) = 0x561b14723000

brk(0x561b14744000) = 0x561b14744000

fstat(3, {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(136, 0), ...}) = 0

write(3, "FORK: Success\n", 14FORK: Success

) = 14

close(3) = 0

fstat(1, {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(136, 0), ...}) = 0

write(1, "\n", 1

) = 1

write(1, "==========\n", 11==========

) = 11

write(1, "Enter name file\n", 16Enter name file

) = 16

fstat(0, {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(136, 0), ...}) = 0

read(0, 4

"4\n", 1024) = 2

write(1, "Enter n\n", 8Enter n

) = 8

read(0, 3

"3\n", 1024) = 2

write(1, "Enter m\n", 8Enter m

) = 8

read(0, 4

"4\n", 1024) = 2

pipe([3, 4]) = 0

clone(child\_stack=NULL, flags=CLONE\_CHILD\_CLEARTID|CLONE\_CHILD\_SETTID|SIGCHLD, child\_tidptr=0x7fec2d0397d0) = 2028

close(3) = 0

Child

write(1, "Enter number\n", 13Enter number

) = 13

write(4, "\0\0\0\0", 4) = 4

write(4, "\1\0\0\0", 4) = 4

write(4, "\2\0\0\0", 4) = 4

write(4, "\3\0\0\0", 4) = 4

close(4) = 0

write(1, "End Enter\n", 10End Enter

) = 10

pipe([3, 4]) = 0

clone(child\_stack=NULL, flags=CLONE\_CHILD\_CLEARTID|CLONE\_CHILD\_SETTID|SIGCHLD, child\_tidptr=0x7fec2d0397d0) = ? ERESTARTNOINTR (To be restarted)

--- SIGCHLD {si\_signo=SIGCHLD, si\_code=CLD\_EXITED, si\_pid=2028, si\_uid=1000, si\_status=255, si\_utime=0, si\_stime=0} ---

clone(child\_stack=NULL, flags=CLONE\_CHILD\_CLEARTID|CLONE\_CHILD\_SETTID|SIGCHLD, child\_tidptr=0x7fec2d0397d0) = 2033

close(3) = 0

Child

write(1, "Enter number\n", 13Enter number

) = 13

write(4, "\0\0\0\0", 4) = 4

write(4, "\1\0\0\0", 4) = 4

write(4, "\2\0\0\0", 4) = 4

write(4, "\3\0\0\0", 4) = 4

close(4) = 0

write(1, "End Enter\n", 10End Enter

) = 10

pipe([3, 4]) = 0

--- SIGCHLD {si\_signo=SIGCHLD, si\_code=CLD\_EXITED, si\_pid=2033, si\_uid=1000, si\_status=255, si\_utime=0, si\_stime=0} ---

clone(child\_stack=NULL, flags=CLONE\_CHILD\_CLEARTID|CLONE\_CHILD\_SETTID|SIGCHLD, child\_tidptr=0x7fec2d0397d0) = 2034

close(3Child

) = 0

write(1, "Enter number\n", 13Enter number

) = 13

write(4, "\0\0\0\0", 4) = 4

write(4, "\1\0\0\0", 4) = 4

write(4, "\2\0\0\0", 4) = 4

write(4, "\3\0\0\0", 4) = 4

close(4) = 0

write(1, "End Enter\n", 10End Enter

) = 10

lseek(0, -1, SEEK\_CUR) = -1 ESPIPE (Illegal seek)

exit\_group(-1) = ?

+++ exited with 255 +++

1. **Листинг программы**

#include <unistd.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <fcntl.h>

void parentProces(int\* pipe\_fd, int m, char \*fname) {

int d;

close(pipe\_fd[0]);

printf("Enter number\n");

for (int i = 0; i < m; ++i)

write(pipe\_fd[1], &i, sizeof(i));

close(pipe\_fd[1]);

}

void childProces(int\* pipe\_fd, char \*fname) {

int d;

int fd;

printf("Child\n");

close(pipe\_fd[1]);

fd = open(fname, O\_CREAT | O\_APPEND | O\_WRONLY, S\_IWUSR | S\_IRUSR); // Открыть на дозапись, если нет создать с правами без sudo

dup2(fd, 1); // Перенаправить вывод в файл fd

while(read(pipe\_fd[0], &d, sizeof(d)) > 0) {

printf("%d ", d);

}

printf("\n");

close(fd);

}

int main(int argc, char const \*argv[]) {

int pipe\_fd[2];

pid\_t pid;

char name\_file[20];

int count\_process;

int m;

int err = 0;

err = -1;

perror("FORK");

printf("\n==========\nEnter name file\n");

scanf ("%s", name\_file);

printf("Enter n\n");

scanf ("%d", &count\_process);

printf("Enter m\n");

scanf ("%d", &m);

int i = 0;

for (i; i < count\_process; ++i) {

if(pipe(pipe\_fd) == -1) {

perror("PIPE");

err = -2;

}

pid = fork();

if(pid == -1) {

perror("FORK");

err = -1;

}

else if (pid == 0) {

childProces(pipe\_fd, name\_file);

break;

} else

parentProces(pipe\_fd, m, name\_file);

printf("End Enter\n");

}

return err;

}

1. **Вывод**

Изучение системных вызовов позволило понять, что делает операционная система в момент их исполнения. Важно правильно обрабатывать системные вызовы, это поможет избежать ошибок при некорректном исполнении программы и поможет в ее отладке.

Если что-то пошло не так, при правильной обработке ошибок мы сразу получим сообщение о ней в этом месте программы. Утилиты диагностики позволяют быстро выяснить, как программа взаимодействует с операционной системой. Это происходит путем мониторинга системных вызовов и сигналов.

Также можно использовать утилиты диагностики для отладки своего кода. Они позволяют увидеть не только, где программа "упала" но и почему это произошло. Для эффективного использования трассировки необходимо знать системные вызовы и понимать их работу. Этому мы и учились в этой лабораторной работе.