Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №1 по курсу**

**«Операционные системы»**

Студент: Базаргармаев Нима Дондокович

Группа: М80 – 301Б-18

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2020

**Содержание**

1. Постановка задачи
2. Общие сведения о программе
3. Средство диагностики
4. Основные файлы программы
5. Демонстрация работы программы
6. Вывод

**Постановка задачи**.

Приобретение практических навыков диагностики программного обеспечения.

**Общие сведения о программе**

Программа компилируется из одного файла fib.c. В данном файле используются заголовочные файлы sys/types.h, sys/wait.h, unistd.h, errno.h, stdlib.h, string.h. В программе используются следующие системные вызовы:

1. **read/write** – предназначены для осуществления потоковых операций ввода (чтения) и вывода (записи) информации над каналами связи, описываемыми файловыми дескрипторами, т.е. для pipe, файлов и для потокового ввода.
2. **pipe** – для создания однонаправленного канала, через который могут общаться два процесса. При нормальном завершении вызова в первый элемент массива(аргумент pipe) – fd[0] – будет занесен файловый дескриптор, соответствующий выходному потоку данных pip’а и позволяющий выполнять только операцию чтения, а во второй элемент массива – fd[1] – будет занесен файловый дескриптор, соответствующий входному потоку данных и позволяющий выполнять только операцию записи. Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибок.
3. **fork** – системный вызов для порождения нового процесса. Процесс, который инициировал системный вызов fork, принято называть родительским процессом (parent process). Вновь порожденный процесс принято называть процессом-ребенком (child process). Процесс-ребенок является почти полной копией родительского процесса. У порожденного процесса по сравнению с родительским изменяются значения следующих параметров: PID, PPID. При однократном системном вызове возврат из него может произойти дважды: один раз в родительском процессе, а второй раз в порожденном процессе. Если создание нового процесса произошло успешно, то в порожденном процессе системный вызов вернет значение 0, а в родительском процессе – положительное значение, равное идентификатору процесса-ребенка. Если создать новый процесс не удалось, то системный вызов вернет в инициировавший его процесс отрицательное значение.

**Средство диагностики**

Утилита **strace.**

**Основные файлы программы**.

**Файл fib.c**

#include <sys/types.h> // библиотека с типами данных size\_t

#include <sys/wait.h> // библиотека с waitpid()

#include <unistd.h> // библиотека с системными вызовами fork(), pipe(), read(), write(), close()

#include <errno.h> // бибоиотека с perror()

#include <stdlib.h>

#include <string.h>

int fd\_1[2]; // массив для pipe(), дескриптор 0 - чтение, 1 - запись

int Fibonacci(int n) {

int result, n\_1, n\_2 = 0;

int status = 0; // переменная для ожидания завершения процесса

ssize\_t wr = 0; // переменная для проверки записи в поток

if (n == 0) {

return 0;

} else if (n < 2) {

return n;

}

pid\_t pid, ppid; // id процесса-родителя и ребенка

pid = fork();

if (pid < 0) {

perror("Can't create fork\n");

exit(-1);

} else if (pid == 0) { // процесс-ребенок

n\_1 = Fibonacci(n - 1);

close(fd\_1[0]);

wr = write(fd\_1[1], &n\_1, sizeof(n\_1));

if (wr < 0) {

perror("Can't write in pipe\n");

}

close(fd\_1[1]);

exit(0);

} else { // процесс-родитель

n\_2 = Fibonacci(n - 2);

}

// ожидаем завершения процессов и сохраняем все в буффер

if (waitpid(pid, &status, 0) < 0) {

perror("Error in waitpid\n");

}

if (read(fd\_1[0], &n\_1, sizeof(n\_1)) < 0) {

perror("Can't read from pipe\n");

}

result = n\_1 + n\_2;

return result;

}

void reverse(char s[]) {

int i, j;

char c;

for (i = 0, j = strlen(s)-1; i<j; i++, j--) {

c = s[i];

s[i] = s[j];

s[j] = c;

}

}

void itoa(int n, char s[]) {

int i, sign;

i = 0;

do { // генерируем цифры в обратном порядке

s[i] = n % 10 + '0'; // берем следующую цифру

++i;

} while ((n /= 10) > 0); // удаляем

s[i] = '\0';

reverse(s);

}

int main() {

int n, num\_len = 0; // номер числа в последовательности и его длина

char read\_sym[10]; // массив, содержащий считываемое число

write(1, "Enter the number: ", sizeof("Enter the number: ")); // 1 - это стандартный поток вывода

if (pipe(fd\_1) < 0) {

perror("Can't create pipe\n");

exit(-1);

}

num\_len = read(2, &read\_sym, 10); // 2 - стандартный поток ввода

read\_sym[num\_len - 1] = '\0';

n = atoi(read\_sym); // преобразуем считанную строку в int

if (n <= 0) {

write(1, "Number must be > 0\n", sizeof("Number must be > 0\n"));

} else {

n = Fibonacci(n);

char result[10];

for (int i = 0; i < 10; ++i) {

result[i] = ' ';

}

itoa(n, result);

write(1, result, sizeof(result));

write(1, "\n", sizeof("\n"));

}

return 0;

}

**Демонстрация работы программы.**

**Запустим утилиту strace для 2 лабораторной работы:**

user@DESKTOP-4RFVSK2:~/os/2lab$ strace ./a.out

execve("./a.out", ["./a.out"], 0x7fffd6f20960 /\* 19 vars \*/) = 0

brk(NULL) = 0x7fffdf53e000

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

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=30224, ...}) = 0

mmap(NULL, 30224, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0x7f46eb27c000

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) = 0x7f46eb270000

mmap(NULL, 4131552, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7f46eac00000

mprotect(0x7f46eade7000, 2097152, PROT\_NONE) = 0

mmap(0x7f46eafe7000, 24576, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1e7000) = 0x7f46eafe7000

mmap(0x7f46eafed000, 15072, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0x7f46eafed000

close(3) = 0

arch\_prctl(ARCH\_SET\_FS, 0x7f46eb2714c0) = 0

mprotect(0x7f46eafe7000, 16384, PROT\_READ) = 0

mprotect(0x7f46eb601000, 4096, PROT\_READ) = 0

mprotect(0x7f46eb227000, 4096, PROT\_READ) = 0

munmap(0x7f46eb27c000, 30224) = 0

write(1, "Enter the number: \0", 19Enter the number: ) = 19

pipe([3, 4]) = 0

read(2, 5

"5\n", 10) = 2

clone(child\_stack=NULL, flags=CLONE\_CHILD\_CLEARTID|CLONE\_CHILD\_SETTID|SIGCHLD, child\_tidptr=0x7f46eb271790) = 104

clone(child\_stack=NULL, flags=CLONE\_CHILD\_CLEARTID|CLONE\_CHILD\_SETTID|SIGCHLD, child\_tidptr=0x7f46eb271790) = 106

wait4(106, [{WIFEXITED(s) && WEXITSTATUS(s) == 0}], 0, NULL) = 106

--- SIGCHLD {si\_signo=SIGCHLD, si\_code=CLD\_EXITED, si\_pid=106, si\_uid=1000, si\_status=0, si\_utime=0, si\_stime=0} ---

read(3, "\2\0\0\0", 4) = 4

--- SIGCHLD {si\_signo=SIGCHLD, si\_code=CLD\_EXITED, si\_pid=104, si\_uid=1000, si\_status=0, si\_utime=0, si\_stime=0} ---

wait4(104, [{WIFEXITED(s) && WEXITSTATUS(s) == 0}], 0, NULL) = 104

read(3, "\2\0\0\0", 4) = 4

write(1, "5\0 ", 105 ) = 10

write(1, "\n\0", 2

) = 2

exit\_group(0) = ?

+++ exited with 0 +++

Как видно из работы утилиты, наиболее часто выполняемыми системными вызовами являются вызовы read/write. Так же имеется вызов pipe.

**Вывод.**

Я научился наблюдать за системными вызовами Unix, используя утилиту strace. Данная утилита имеет много ключей, тем самым она является гибким инструментом для нахождения ошибок, связанных с системными вызовами. Утилита идеально подходит для отслеживания дочерних процессов, присоединению к процессу на лету и для других задач, возникающих при отладке программы. Помимо этого, с помощью данной утилиты можно получить много информации о программе еще до изучения кода самой программы.