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»

Выполнил: М. А. Понизяйкин
Группа: М8О-207БВ-24
Преподаватель: Е. С. Миронов

Москва, 2025

Условие

Цель работы:
Приобретение практических навыков в:

- Управление процессами в ОС
- Обеспечение обмена данных между процессами посредством каналов

Задание:
Составить и отладить программу на языке Си, осуществляющую работу с процессами и взаимодействие между ними в одной из двух операционных систем. В результате работы программа (основной процесс) должен создать для решение задачи один или несколько дочерних процессов. Взаимодействие между процессами осуществляется через системные сигналы/события и/или каналы (pipe). Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

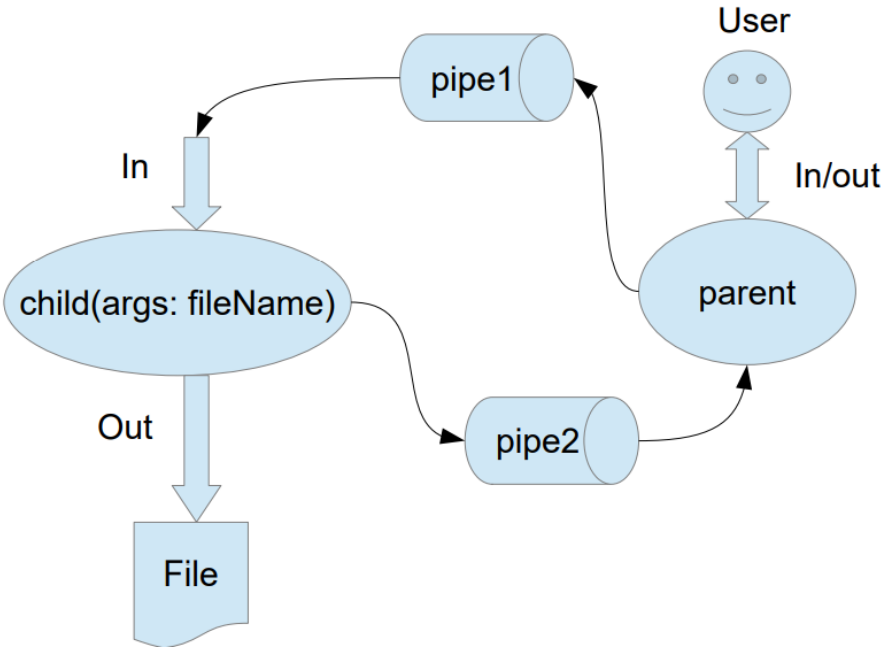

Рис. 1: Схема работы процессов.

Вариант: 4
Родительский процесс создает дочерний процесс. Первой строчкой пользователь в консоль родительского процесса пишет имя файла, которое будет передано при создании дочернего процесса. Родительский и дочерний процесс должны быть представлены разными программами. Родительский процесс передает команды пользователя через pipe1, который связан с стандартным входным потоком дочернего процесса. Дочерний процесс принеобходимости передает данные в родительский процесс через pipe2. Результаты своей работы дочерний процесс пишет в созданный им файл. Допускается просто открыть файл и писать туда, не перенаправляя стандартный поток вывода. Пользователь вводит команды вида: «число число число<endline>». Далее эти числа передаются от родительского процесса в дочерний. Дочерний процесс производит деление первого числа, на последующие, а результат выводит в файл. Если происходит деление на 0, то тогда дочерний и родительский процесс завершают свою работу. Проверка деления на 0 должна

осуществляться на стороне дочернего процесса. Числа имеют тип float. Количество чисел может быть произвольным.

Метод решения

Алгоритм решения задачи:

1. Пользователь в консоль родительского процесса вводит имя файла, в который дочерний процесс запишет результат вычислений, а затем — последовательность чисел типа `float`, разделённых пробелами.

2. Создаётся объект логики родительского процесса (реализован в `parent.cpp`), инициализирующий работу основной программы.

3. Родительский процесс создаёт два канала (pipe).

   - `input_pipe` — для передачи последовательности чисел от родителя к дочернему процессу;
   - `err_pipe` — для передачи диагностических сообщений (ошибок) от дочернего процесса к родителю. Затем вызывается `fork()` для создания дочернего процесса.

4. В контексте дочернего процесса:

   - закрывается записывающий конец `input_pipe` и читающий конец `err_pipe`;
   - читающий конец `input_pipe` перенаправляется на стандартный поток ввода (`stdin`) с помощью `dup2()`;
   - записывающий конец `err_pipe` перенаправляется на стандартный поток ошибок (`stderr`);
   - выполняется системный вызов `execl()`, запускающий отдельный исполняемый файл `child`, которому передаётся имя выходного файла как аргумент командной строки. Если запуск дочернего исполняемого файла не удался, процесс завершается с ошибкой.

5. Родительский процесс закрывает читающий конец `input_pipe` и записывающий конец `err_pipe`, после чего отправляет в `input_pipe` строку с числами, введёнными пользователем.

6. Дочерний процесс (`child`) получает имя выходного файла через аргумент командной строки, считывает числа из `stdin` (который перенаправлен на `input_pipe`), выполняет последовательное деление первого числа на все последующие. Перед каждым делением проверяется, не равен ли делитель нулю. В случае деления на ноль процесс немедленно завершается с выводом сообщения об ошибке в `stderr`.

7. Если деление выполнено успешно, дочерний процесс открывает (создаёт) указанный файл и записывает в него результат вычислений с точностью до шести знаков после запятой.

8. Родительский процесс ожидает завершения дочернего с помощью `wait()`. Если дочерний процесс завершился с ненулевым кодом (например, из-за деления на ноль), родитель считывает сообщение об ошибке из `err_pipe` и также завершает свою работу. В случае успешного завершения программа завершается корректно.

Архитектура программы:

```
lab1-var4/
├── bin/
├── child.cpp
├── build/
├── include/
│   ├── child.h
│   ├── exceptions.h
│   ├── os.h
│   └── parent.h
├── src/
│   ├── child.cpp
│   ├── os.cpp
│   └── parent.cpp
└── main.cpp
```

Ссылки:

- https://pubs.opengroup.org/onlinepubs/009696799/functions/write.html
- https://pubs.opengroup.org/onlinepubs/009696799/functions/execl.html
- https://pubs.opengroup.org/onlinepubs/009696799/functions/getppid.html
- https://pubs.opengroup.org/onlinepubs/009696799/functions/getpid.html
- https://pubs.opengroup.org/onlinepubs/009696799/functions/waitpid.html
- https://pubs.opengroup.org/onlinepubs/009696799/functions/sleep.html
- https://pubs.opengroup.org/onlinepubs/009696799/functions/exit.html
- https://pubs.opengroup.org/onlinepubs/009696799/functions/kill.html
- https://pubs.opengroup.org/onlinepubs/009696799/functions/dup2.html
- https://pubs.opengroup.org/onlinepubs/009696799/functions/pipe.html
- https://pubs.opengroup.org/onlinepubs/009696799/functions/fork.html
- https://pubs.opengroup.org/onlinepubs/009696799/functions/close.html

Описание программы

`main.cpp` — точка входа в родительский процесс. Вызывает функцию `RunParentProcess()`, реализующую основную логику взаимодействия с пользователем и дочерним процессом.

`bin/child.cpp` — точка входа в дочерний процесс. Принимает имя выходного файла как аргумент командной строки и вызывает функцию `RunChildProcess()` для выполнения вычислений.

`exceptions.h` — объявление пользовательских исключений.

`os.h` — объявление функций-обёрток над системными вызовами операционной системы (для поддержки кроссплатформенности).

`src/os.cpp` — реализация функций.

Основные функции:

- `int CreatePipe(pipe_t fd[2]);` — создаёт анонимный канал. Используется системный вызов `pipe()`.
- `pid_t CloneProcess();` — создаёт копию текущего процесса. Используется системный вызов `fork()`.
- `int LinkStdinWithPipe(pipe_t pipe);` — перенаправляет стандартный ввод на указанный канал. Используется системный вызов `dup2()`.
- `int LinkStderrWithPipe(pipe_t pipe);` — перенаправляет стандартный поток ошибок на указанный канал. Используется системный вызов `dup2()`.
- `int ClosePipe(pipe_t pipe);` — закрывает файловый дескриптор канала. Используется системный вызов `close()`.
- `int Exec(const char* path);` — заменяет текущий процесс на новый исполняемый файл. Используется системный вызов `execl()`.
- `int WaitForChild();` — ожидает завершения любого дочернего процесса. Используется системный вызов `wait()`.
- `int WritePipe(pipe_t pipe, void* buffer, size_t bytes);` — записывает данные в канал. Используется системный вызов `write()`.
- `int ReadPipe(pipe_t pipe, void* buffer, size_t bytes);` — читает данные из канала. Используется системный вызов `read()`.
- `void AddSignalHandler(signal_t sig, SignalHandler_t handler);` — регистрирует обработчик сигнала. Используется системный вызов `signal()`.
- `int ReadFromStdin(char* buffer, size_t size);` — читает данные напрямую из стандартного ввода. Используется системный вызов `read()`.
- `void PrintLastError();` — выводит описание последней ошибки ОС. Используется функция `perror()`.

`child.h` — объявление функции логики дочернего процесса.
`src/child.cpp` — реализация вычислительной логики.
Основные функции:

- `void RunChildProcess(const char* output_file);` — считывает числа из `stdin` (перенаправленного на канал), выполняет последовательное деление первого числа на все последующие, проверяет деление на ноль. При успехе записывает результат в файл `output_file` с точностью до шести знаков после запятой. При ошибке завершает процесс с кодом 1.

parent.h — объявление функции логики родительского процесса.
src/parent.cpp — реализация управления процессами и взаимодействия с пользователем.
Основные функции:

- `void RunParentProcess();` — запрашивает у пользователя имя выходного файла и последовательность чисел, создаёт два канала (`input_pipe` и `err_pipe`), порождает дочерний процесс через `fork()`, перенаправляет его `stdin` и `stderr` на соответствующие концы каналов, запускает исполняемый файл `child` с передачей имени файла как аргумента, отправляет числа в `input_pipe`, ожидает завершения дочернего процесса и обрабатывает возможные ошибки.
- `void OnChildKilled(signal_t signum);` — обработчик сигнала `SIGCHLD`. При завершении дочернего процесса читает диагностическое сообщение из `err_pipe` и завершает родительский процесс.
- `void PrintErrorFromChild(pipe_t pipe);` — читает и выводит ошибки, полученные от дочернего процесса через канал.

Результаты

Программа получает на вход название файла, создает дочерний процесс, этот процесс в директории открывает (создает) этот файл. После этого дочерний процесс обрабатывает все введеные пользователем числа, последовательно делит первое число на остальные, при этом избегая деления на 0, и результат записывает в указанный файл.
Результатом является файл в директории. Если не удалось создать канал для передачи данных между процессами или дочерние процессы завершились, то программа безопасно прекращает свою работу.

Выводы

В ходе выполнения лабораторной работы были приобретены практические навыки в управлении процессами в ОС, обеспечение обмена данных между процессами посредством каналов.
Была составлена и отлажена программа на языке C++, осуществляющая работу с процессами и взаимодействие между ними в операционной системе с ядром Linux.
В результате работы программа (основной процесс) создает один дочерний процесс.
Взаимодействие между процессами осуществляется через системные сигналы и каналы.
Ошибки в результате рыботы дочернего процесса обрабатываются и передаются через отдельный канал `stderr`.
Обрабатаны системные ошибки, которые могут возникнуть в результате работы.
Есть возможно поддержки кроссплатформенности за счет изменений системных вызовов в файле `inc/src/os.cpp`.

Исходная программа

```cpp
#include "child.h"
#include <cstdio>

int main(int argc, char* argv[]) {
    if (argc != 2) {
        std::fprintf(stderr, "Usage: %s <output_file>\n", argv[0]);
        return 1;
    }
    RunChildProcess(argv[1]);
    return 0;
}
```

Листинг 1: bin/child.cpp

```cpp
#pragma once

void RunChildProcess(const char* output_file);
```

Листинг 2: child.h

```cpp
#pragma once

#include <stdexcept>
#include <string>

class OsException : public std::runtime_error {
public:
    explicit OsException(const std::string& message) : std::runtime_error(message) {}
};
```

Листинг 3: exceptions.h

```cpp
#pragma once

#include <string>
#include <cstddef>

using pipe_t = int;
using pid_t = int;
using signal_t = int;
using SignalHandler_t = void(*)(int);

extern const int ChildDeathSig;
extern const int BrokenPipeSig;

int CreatePipe(pipe_t fd[2]);
pid_t CloneProcess();
int Exec(const char* path);
void PrintLastError();
int WaitForChild();
int ClosePipe(pipe_t pipe);
int WritePipe(pipe_t pipe, void* buffer, std::size_t bytes);
int ReadPipe(pipe_t pipe, void* buffer, std::size_t bytes);
int LinkStdinWithPipe(pipe_t pipe);
int LinkStderrWithPipe(pipe_t pipe);
```

```
24 void AddSignalHandler(signal_t sig, SignalHandler_t handler);
25 int ReadFromStdin(char* buffer, std::size_t size);
```

Листинг 4: os.h

```
1  #pragma once
2  #include "os.h"
3
4  void RunParentProcess();
5  void PrintErrorFromChild(pipe_t pipe);
6  void OnChildKilled(signal_t signum);
```

Листинг 5: parent.h

```
1  #include "child.h"
2  #include <cstdio>
3  #include <cstdlib>
4
5  void RunChildProcess(const char* output_file) {
6      float num = 0.0;
7      int scanned = std::scanf("%f", &num);
8      if (scanned != 1) {
9          std::fprintf(stderr, "Error: failed to read first number\n");
10         std::exit(1);
11     }
12
13     float divider = 1.0;
14     while ((scanned = std::scanf("%f", &divider)) == 1) {
15         if (divider == 0.0) {
16             std::fprintf(stderr, "Error: division by zero occurred\n");
17             std::exit(1);
18         }
19         num /= divider;
20     }
21
22     if (scanned == 0 && !std::feof(stdin)) {
23         std::fprintf(stderr, "Error: invalid input format\n");
24         std::exit(1);
25     }
26
27     if (std::ferror(stdin)) {
28         std::fprintf(stderr, "Error: error on stdin\n");
29         std::exit(1);
30     }
31
32     FILE* out = std::fopen(output_file, "w");
33     if (!out) {
34         std::perror("fopen");
35         std::exit(1);
36     }
37
38     std::fprintf(out, "%.6f\n", num);
39     std::fclose(out);
40 }
```

Листинг 6: child.cpp

```cpp
#include "os.h"
#include "exceptions.h"
#include <unistd.h>
#include <sys/wait.h>
#include <cstring>
#include <csignal>
#include <cstdio>

const int ChildDeathSig = SIGCHLD;
const int BrokenPipeSig = SIGPIPE;

int CreatePipe(pipe_t fd[2]) {
    return pipe(fd);
}

pid_t CloneProcess() {
    return static_cast<pid_t>(fork());
}

int Exec(const char* path) {
    return execl(path, path, nullptr);
}

void PrintLastError() {
    perror("OS Error");
}

int WaitForChild() {
    return wait(nullptr);
}

int ClosePipe(pipe_t pipe) {
    return close(pipe);
}

int WritePipe(pipe_t pipe, void* buffer, std::size_t bytes) {
    return static_cast<int>(write(pipe, buffer, bytes));
}

int ReadPipe(pipe_t pipe, void* buffer, std::size_t bytes) {
    return static_cast<int>(read(pipe, buffer, bytes));
}

int LinkStdinWithPipe(pipe_t pipe) {
    return dup2(pipe, STDIN_FILENO);
}

int LinkStderrWithPipe(pipe_t pipe) {
    return dup2(pipe, STDERR_FILENO);
}

void AddSignalHandler(signal_t sig, SignalHandler_t handler) {
    signal(sig, handler);
}

int ReadFromStdin(char* buffer, std::size_t size) {
    return static_cast<int>(read(STDIN_FILENO, buffer, size));
```

```
}
```

Листинг 7: os.cpp

```
#include "parent.h"
#include "os.h"
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <string>
#include <unistd.h>

const std::size_t kBuffer = 100;
static pipe_t err_pipe_in;

void PrintErrorFromChild(pipe_t pipe) {
    char buffer[kBuffer];
    int bytes = ReadPipe(pipe, buffer, kBuffer);
    if (bytes > 0) {
        std::fprintf(stderr, "Error from child: ");
    }
    while (bytes > 0) {
        std::fwrite(buffer, sizeof(char), bytes, stderr);
        bytes = ReadPipe(pipe, buffer, kBuffer);
    }
}

void OnChildKilled(signal_t signum) {
    PrintErrorFromChild(err_pipe_in);
    std::exit(-1);
}

void RunParentProcess() {
    AddSignalHandler(ChildDeathSig, OnChildKilled);

    std::printf("Enter result file name: ");
    char output_file[kBuffer];
    if (!std::fgets(output_file, kBuffer, stdin)) {
        PrintLastError();
        std::exit(1);
    }
    output_file[strcspn(output_file, "\n")] = '\0';

    std::printf("Enter numbers (separator - space): ");
    char numbers_line[kBuffer];
    if (!std::fgets(numbers_line, kBuffer, stdin)) {
        PrintLastError();
        std::exit(1);
    }

    pipe_t input_pipe[2];
    pipe_t err_pipe[2];
    if (CreatePipe(input_pipe) == -1 || CreatePipe(err_pipe) == -1) {
        PrintLastError();
        std::exit(1);
    }

    err_pipe_in = err_pipe[0];
```

```
    pid_t child_id = CloneProcess();

    if (child_id == 0) { // child
        if (LinkStdinWithPipe(input_pipe[0]) == -1 ||
            LinkStderrWithPipe(err_pipe[1]) == -1) {
            PrintLastError();
            std::exit(1);
        }
        ClosePipe(input_pipe[1]);
        ClosePipe(err_pipe[0]);

        const char* child_argv[] = {"./child", output_file, nullptr};
        execl("./child", "./child", output_file, nullptr);

        PrintLastError();
        std::exit(1);
    } else if (child_id == -1) {
        PrintLastError();
        std::exit(1);
    } else { // parent
        ClosePipe(input_pipe[0]);
        ClosePipe(err_pipe[1]);

        WritePipe(input_pipe[1], numbers_line, std::strlen(numbers_line));
        ClosePipe(input_pipe[1]);

        PrintErrorFromChild(err_pipe[0]);
        ClosePipe(err_pipe[0]);

        WaitForChild();
    }
}
```

Листинг 8: parent.cpp

```
#include "parent.h"

int main() {
    RunParentProcess();
    return 0;
}
```

Листинг 9: main.cpp

Strace

```
execve("./main", ["./main"], 0x7ffc71d22640 /* 26 vars */) = 0
brk(NULL)                               = 0x64b419a95000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x796307c6
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=20163, ...}) = 0
mmap(NULL, 20163, PROT_READ, MAP_PRIVATE, 3, 0) = 0x796307c64000
close(3)                                = 0
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\220\243\2\0\0\0\0\0"..., 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"..., 784, 6
fstat(3, {st_mode=S_IFREG|0755, st_size=2125328, ...}) = 0
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"..., 784, 6
mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x796307a00000
mmap(0x796307a28000, 1605632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRIT
mmap(0x796307bb0000, 323584, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0
mmap(0x796307bff000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE
mmap(0x796307c05000, 52624, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x796307c
arch_prctl(ARCH_SET_FS, 0x796307c61740) = 0
set_tid_address(0x796307c61a10)         = 16642
set_robust_list(0x796307c61a20, 24)     = 0
rseq(0x796307c62060, 0x20, 0, 0x53053053) = 0
mprotect(0x796307bff000, 16384, PROT_READ) = 0
mprotect(0x64b3ed8dc000, 4096, PROT_READ) = 0
mprotect(0x796307ca1000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x796307c64000, 20163)           = 0
rt_sigaction(SIGCHLD, {sa_handler=0x64b3ed8da41f, sa_mask=[CHLD], sa_flags=SA_RESTORE
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
getrandom("\x82\xe7\x50\xa0\xa4\x4c\xac\xa1", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x64b419a95000
brk(0x64b419ab6000)                     = 0x64b419ab6000
fstat(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
write(1, "Enter result file name: ", 24) = 24
read(0, "ans.txt\n", 1024)              = 8
write(1, "Enter numbers (separator - space"..., 35) = 35
read(0, "1 2 3455\n", 1024)             = 9
pipe2([3, 4], 0)                        = 0
pipe2([5, 6], 0)                        = 0
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_
close(3)                                = 0
close(6)                                = 0
write(4, "1 2 3455\n", 9)               = 9
close(4)                                = 0
read(5, "", 100)                        = 0
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=16655, si_uid=1000, si_stat
read(5, "", 100)                        = 0
exit_group(-1)                          = ?
+++ exited with 255 +++
```