Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики Кафедра вычислительной математики и программирования

Лабораторная работа по предмету "операционные системы" №4

Студент: Мокеева С.А.
Преподаватель: Соколов А.А.
Группа: М8О-206Б-20
Дата: 12.04.2022
Оценка:
Подпись:

Москва 2022г.

Вариант 2.

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 решения задачи один или несколько дочерних процессов. Взаимодействие между процессами осуществляется через системные сигналы/события и/или через отображаемые файлы (memory-mapped files).

Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы

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 при необходимости 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 считает их сумму и выводит её в файл. Числа имеют тип float. Количество чисел может быть произвольным.

Реализация

Файл main.cpp

```
#define  _GNU_SOURCE  // for getline

#include <assert.h>
#include <fcntl.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <sys/file.h>
#include <unistd.h>

bool is_child_ready = false;

void sig_handler(int signum) { //обработчик сигнала
    is_child_ready = true;
```

```
}

char* read_filename() { //считываем имя файла
    char* line = NULL;
    size_t bufsize = 0;
    int nread = getline(&line, &bufsize, stdin);
    if (nread == -1) {
        return NULL;
    }
    size_t len = strlen(line);
    line[len - 1] = '\0';
    return line;
}


bool translate_stdin_to_map(int fd_map, float* map, size_t n_numbers_max) {
    fprintf(stderr, "DEBUG: translate_stdin_to_map: started\n");

    bool is_ok = true;
    bool is_done = false;

    while (! is_done) { // для каждой команды:
        flock(fd_map, LOCK_EX); //"захватываем" файл
        fprintf(stderr, "DEBUG: locked\n");

        is_child_ready = false; //выставляем, что ребёнок не готов принять
текущую команду

        fprintf(stderr, "DEBUG: --- reading a new command ---\n");
        for (size_t i_number = 1; ; i_number++) { //считываем команду
            fprintf(stderr, "DEBUG: i_number: %zu\n", i_number);

            if (i_number > n_numbers_max) { //слишком много чисел для
выделенной памяти
                fprintf(stderr, "too many numbers\n");
                is_ok = false;
                is_done = true;
                break; //выходим из цикла
            }

            // Считываем очередное число из входного файла и преобразуем
его из текста во float.
            float number;
            int result_scanf = scanf("%f", &number);
            if (result_scanf == EOF) { // встретили конец файла
                fprintf(stderr, "DEBUG: EOF\n");
                is_ok = i_number == 1;
                is_done = true; //если начали читать новую команду, а там
вначале конец файла, то всё хорошо
                break;
            }
            if (result_scanf == 0) { //если там не число
                fprintf(stderr, "expected a number\n");
                is_ok = false;
                is_done = true;
                break;
            }
            assert(result_scanf == 1); //проверяем, что одно число
считалось и сохраняем это число

            fprintf(stderr, "DEBUG: number: %f\n", number);
```

```
            map[i_number] = number; //записываем это число

            char sep = getchar();
            fprintf(stderr, "DEBUG: sep: %02x\n", sep);
            if (sep != '\n') { //если это не символ новой строки,
продолжаем
                continue;
            }

            // Записываем числа команды:
            assert(i_number == (float) i_number); // представимо точно
(проверяем, что можем записать
                                                  //максимальное количество
чисел в точном представлении float)
            map[0] = i_number;
            msync(map, (i_number + 1) * sizeof(float), MS_SYNC);
//синхронизация памяти
            fprintf(stderr, "DEBUG: msynced\n");
            break; // переходим к новом команде
        }

        if (is_done) {
            map[0] = -1; //служебное число
            msync(map, sizeof(float), MS_SYNC);
        }

        flock(fd_map, LOCK_UN); //разблокируем файл
        fprintf(stderr, "DEBUG: unlocked\n");

        while (! is_child_ready) { //ждём, пока ребёнок не завершится
            sleep(1);
        }
        fprintf(stderr, "DEBUG: child is ready\n");
    }

    return is_ok;
}

bool wait_for_child_exit() {
    int wstatus;
    if (wait(&wstatus) == -1) {
        perror("wait");
        return false;
    }
    // Ребёнок завершился.
    if (! (WIFEXITED(wstatus) && WEXITSTATUS(wstatus) == 0)) {
        fprintf(stderr, "error in child\n");
        return false;
    }

    return true;
}

int main(int argc, char *argv[]) {
    if (argc != 2) { // arg count
        fprintf(stderr, "Usage: %s MAPFILE\n", argv[0]);
        return 1;
    }

    char* filename_map = argv[1];
```

```
    //open(имя файла, права чтение&запись | создаём файл | если файл был
создан, обрезаем, права доступа)
    // 0600 =  rwx rwx rwx

    int fd_map = open(filename_map, O_RDWR | O_CREAT | O_TRUNC, 0600);
//открываем файл
    if (fd_map == -1) { //если не получилось, то выходим
        perror(filename_map); // глобальная переменная errno(что именно
произошло), perror смотрит,
                              //что написано в errno и отображает
соответствующее сообщение
        return 1;
    }
    flock(fd_map, LOCK_EX); // блокируем файл, чтобы ребёнок пока не читал

    char* filename_result = read_filename();
    if (filename_result == NULL) { //если есть ошибка
        fprintf(stderr, "expected a filename\n"); //выводим ошибку на
стандартный поток ошибок
        return 1;
    }
    fprintf(stderr, "DEBUG: filename_result: %s\n", filename_result);

    signal(SIGUSR1, sig_handler); //сигнал о том, что ребёнок готов,
вызывается функция sig_handler

    //создаём ребёнка
    int pid_child = fork();
    if (pid_child == -1) { если не вышло, вывели сообщение об ошибке
        perror("fork");
        return 1;
    }
    // pid_child=0 -- у дочернего процесса
    // pid_child=номер_дочернего -- у родительского процесса

    if (pid_child == 0) { // Мы в дочернем процессе.
        //закрываем файл, чтобы в ребёнке его снова открыть
        if (close(fd_map) == -1) {
            perror(filename_map);
            return 1;
        }
        //запускаем ребёнка, передаём имя вспомогательного файла и куда
записывать результаты
        char* argv_child[] = {"./child", filename_map, filename_result,
NULL};
        if (execv("child", argv_child) == -1) { //эта функция не должна
завершиться, если
                                                //завершилась, выводим
ошибку
            perror("execv");
        }
        return 1; // Если вдруг вернулись из функции execv.
    }

    // Мы в родительском процессе.
    assert(pid_child > 0);

    size_t n_numbers_max = 100; //максимум чисел, которые будем хранить
    size_t length_map = (n_numbers_max + 1) * sizeof(float); //расширяем
файл, чтобы эти числа туда влезли
    //(сколько байт)
```

```
    //забиваем файл нулями
    //fallocate(fd, int mod, off_t offset, off_t, len);
    if (fallocate(fd_map, 0, 0, length_map) == -1) {
        perror("fallocate");
        return 1;
    }

    //файл отображаем на массив, с которым дальше будем работать
    float* map = mmap(NULL, length_map, PROT_WRITE, MAP_SHARED, fd_map, 0);
    //mmap(то, на какой адрес надо отобразить, сколько байт, будем
обращаться на запись,
    //те изменения, которые мы сделаем, должны быть и в самом файле, что
это за файл, смещение(с начала));
    if (map == MAP_FAILED) { //если не удалось, выходим
        perror("mmap");
        return 1;
    }
    fprintf(stderr, "DEBUG: map: %p\n", map);

    bool is_ok = true;

    if (! translate_stdin_to_map(fd_map, map, n_numbers_max)) { //запускаем
translate_stdin_to_map
        is_ok = false;
    }

    if (munmap(map, length_map) == -1) { //закрываем отображение файла на
память
        perror("munmap");
        is_ok = false;
    }

    if (close(fd_map) == -1) { //закрываем сам файл
        perror(filename_map);
        is_ok = false;
    }

    if (! wait_for_child_exit()) {//ждём, когда ребёнок выйдет
        is_ok = false;
    }

    return is_ok ? 0 : 1;
}
```

Файл child.cpp

```
#define _POSIX_SOURCE

#include <assert.h>
#include <fcntl.h>
#include <float.h>
#include <signal.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <sys/file.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

bool ready(pid_t pid_parent) {
```

```
    if (kill(pid_parent, SIGUSR1) == -1) { // говорим, что готовы ждать
запрос
        perror("kill");
        return false;
    }
    return true;
}

int main(int argc, char* argv[]) { //принимает на вход вспомогательный файл
и результирующий
    fprintf(stderr, "DEBUG: child: started\n");

    if (argc != 3) {
        return 1;
    }
    char* filename_map = argv[1]; //вспомогательный файл
    char* filename_result = argv[2]; //результирующий файл

    int fd_map = open(filename_map, O_RDWR, 0600); //открываем файл
    if (fd_map == -1) {
        perror(filename_map);
        return 1;
    }

    size_t n_numbers_max = 100;
    size_t length_map = (n_numbers_max + 1) * sizeof(float);
    //вспомогательный файл отображаем на область памяти
    float* map = mmap(NULL, length_map, PROT_READ | PROT_WRITE, MAP_SHARED,
fd_map, 0);
    if (map == MAP_FAILED) {
        perror("mmap");
        return 1;
    }

    FILE* file_result = fopen(filename_result, "w"); //открываем файл с
результатами
    //через fopen, тк при записи результатов мы можем их кэшировать, чтобы
сразу на
    //жёсткий диск не писать
    if (file_result == NULL) {
        perror(filename_result);
        return 1;
    }

    pid_t pid_parent = getppid(); //узнаём номер родителя, чтобы ему
отправлять сигнал
    fprintf(stderr, "DEBUG: child: pid_parent=%d\n", pid_parent);

    bool is_ok = true;
    bool is_done = false;
    while (! is_done) {
        fprintf(stderr, "DEBUG: child: trying to lock\n");
        flock(fd_map, LOCK_EX); //захватываем файл
        fprintf(stderr, "DEBUG: child: locked\n");

        int k_numbers = (int) map[0];//смотрим, что в нулевом служебном
элементе
        if (k_numbers == -1) { //команды закончилимь
            is_done = true;
        } else if (k_numbers > 0) { //суммируем
            float sum = 0;
```

```
            for (int i_number = 1; i_number <= k_numbers; i_number++) {
                sum += map[i_number];
            }

            if (fprintf(file_result, "%f\n", sum) < 0) {
                printf("fprintf error\n");
                is_ok = false;
                is_done = true;
            }

            map[0] = 0; // чтобы ребёнок не выполнял ту же команду снова
            msync(map, sizeof(float), MS_SYNC);
        }

        flock(fd_map, LOCK_UN); //разблокируем файл
        fprintf(stderr, "DEBUG: child: unlocked\n");

        ready(pid_parent); //сообщаем, что готовы
    }

    if (munmap(map, length_map) == -1) { //закрываем отображение файла на
память
        perror("munmap");
        is_ok = false;
    }

    if (close(fd_map) == -1) { //закрываем сам файл
        perror(filename_map);
        is_ok = false;
    }

    if (fclose(file_result) != 0) { //закрываем результирующий файл
        printf("fclose error\n");
        is_ok = false;
    }

    return is_ok ? 0 : 1;
}
```

Файл run.sh

```
#!/bin/bash

set -e  # exit on error
set -x  # trace (print commands)

(
  gcc main.c -o main
  gcc child.c -o child

  rm file.map
  printf 'result.txt\n1 2 3\n4 5 6\n' | ./main file.map
  ls -l file.map

  od --address-radix=d -f file.map
  cat result.txt
) |& tee run.log
```

Пример работы

```
sophie@sophie-VirtualBox:~/os/os4$ ./run.sh
+ tee run.log
+ gcc main.c -o main
+ gcc child.c -o child
+ rm file.map
+ ./main file.map
+ printf 'result.txt\n1 2 3\n4 5 6\n'
DEBUG: filename_result: result.txt
DEBUG: map: 0x7ff9c21ac000
DEBUG: translate_stdin_to_map: started
DEBUG: locked
DEBUG: --- reading a new command ---
DEBUG: i_number: 1
DEBUG: number: 1.000000
DEBUG: sep: 20
DEBUG: i_number: 2
DEBUG: number: 2.000000
DEBUG: sep: 20
DEBUG: i_number: 3
DEBUG: number: 3.000000
DEBUG: sep: 0a
DEBUG: child: started
DEBUG: child: pid_parent=32747
DEBUG: child: trying to lock
DEBUG: msynced
DEBUG: unlocked
DEBUG: child: locked
DEBUG: child: unlocked
DEBUG: child is ready
DEBUG: locked
DEBUG: --- reading a new command ---
DEBUG: i_number: 1
DEBUG: number: 4.000000
DEBUG: sep: 20
DEBUG: i_number: 2
DEBUG: number: 5.000000
DEBUG: sep: 20
DEBUG: i_number: 3
DEBUG: number: 6.000000
DEBUG: sep: 0a
DEBUG: child: trying to lock
DEBUG: msynced
DEBUG: unlocked
DEBUG: child: locked
DEBUG: child: unlocked
DEBUG: child is ready
DEBUG: locked
DEBUG: --- reading a new command ---
DEBUG: i_number: 1
DEBUG: EOF
DEBUG: child: trying to lock
DEBUG: unlocked
DEBUG: child: locked
DEBUG: child: unlocked
DEBUG: child is ready
+ ls -l file.map
-rw------- 1 sophie sophie 404 апр 11 21:46 file.map
+ od --address-radix=d -f file.map
0000000              -1               4               5               6
0000016               0               0               0               0
*
0000400               0
0000404
+ cat result.txt
6.000000
15.000000
```

Вывод

В ходе лабораторной работы я научилась работать с shared_memory и использовала её для передачи информации между процессами. Также я использовала такие системные вызовы как open, fork, mmap, fallocate.