Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №4 по курсу**

**«Операционные системы»**

Студент: Ткаченко Егор Юрьевич

Группа: М8О-207Б-21

Вариант: 18

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2022

**Содержание**

1. Репозиторий
2. Постановка задачи
3. Общие сведения о программе
4. Общий метод и алгоритм решения
5. Исходный код
6. Демонстрация работы программы
7. Выводы

**Репозиторий**

[https://github.com/Tnirpps/OS\_lab](https://vk.com/away.php?to=https%3A%2F%2Fgithub.com%2FTnirpps%2FOS_lab&cc_key=)

**Постановка задачи**

**Цель работы**

Изучить управление процессами, обеспечение синхронизации между процессами.

**Задание**

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

child1 или в child2 в зависимости от правила фильтрации. Процесс child1 и child2 производят работу над строками. Процессы пишут результаты своей работы в стандартный вывод.

Правило фильтрации: нечетные строки отправляются в child1, четные в child2.

Дочерние процессы удаляют все гласные из строк.

**Общие сведения о программе**

Программа компилируется из файлов main.c, child.c. Также используется заголовочные файлы: fcntl.h, pthread.h, stdio.h, string.h, sys/mman.h, sys/stat.h, sys/types.h, assert.h . В программе используются следующие системные вызовы:

1. fork() – создает новый процесс.
2. execv() – передает процесс на исполнение другой программе.
3. shm\_open() – создаёт/открывает объект общей памяти.
4. shm\_unlink() – обратная к shm\_open().
5. ftruncate() – устанавливает файлу заданную длину в байтах.
6. close() – закрывает файловый декриптор.
7. pthread\_mutex\_lock() – блокирует мьютекс.
8. pthread\_mutex\_unlock() – разблокирует мьютекс.
9. pthread\_cond\_wait() – открывает мьютекс и ждёт изменения состояния переменной условия.
10. pthread\_cond\_signal() – посылает сигнал на разблокировку.

**Общий метод и алгоритм решения**

Родительский процесс создаёт два дочерних, затем передаёт им данные. Дочерние процессы выполняют фильтрацию полученных от родительского процесса данных в соответствии с требованием варианта. Работа заканчивается, когда родительский процесс закрывает каналы с дочерними процессами и их дальнейшее чтение становится невозможным. Взаимодействие между процессами осуществляется через отображаемые файлы (memory-mapped files).

**Исходный код**

================================= main.c ===================================

#include <fcntl.h>

#include <pthread.h>

#include <stdio.h>

#include <string.h>

#include <sys/mman.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

#include <assert.h>

#define check\_ok(VALUE, OK\_VAL, MSG) if (VALUE != OK\_VAL) { printf("%s", MSG); return 1; }

#define check\_wrong(VALUE, WRONG\_VAL, MSG) if (VALUE == WRONG\_VAL) { printf("%s", MSG); return 1; }

/\* Ubuntu has 255 symbol filename limit \*/

const unsigned int FILENAME\_LIMIT = 255;

const unsigned int SHARED\_MEMORY\_SIZE = 1;

const int child\_count = 2;

const char\* SHARED\_FILE\_NAME[] = {"1\_shared\_file", "2\_shared\_file"};

const char\* SHARED\_MUTEX\_NAME[] = {"1\_shared\_mutex","2\_shared\_mutex"};

const char\* SHARED\_COND\_NAME[] = {"1\_shared\_cond", "2\_shared\_cond"};

int write\_to\_process(char\* sharedFile, pthread\_mutex\_t\* mutex, pthread\_cond\_t\* condition, const char c) {

check\_ok(pthread\_mutex\_lock(mutex), 0, "Error locking mutex in parent!\n")

while (sharedFile[0] != 0) {

check\_ok(pthread\_cond\_wait(condition, mutex), 0, "Error waiting cond in parent!\n")

}

sharedFile[0] = c;

check\_ok(pthread\_cond\_signal(condition), 0, "Error sending signal in parent!\n")

check\_ok(pthread\_mutex\_unlock(mutex), 0, "Error unlocking mutex in parent!\n")

return 0;

}

int main() {

assert(child\_count <3);

char file[FILENAME\_LIMIT + 1];

int output[child\_count];

for (int i = 0; i < child\_count; ++i) {

memset(file, 0, FILENAME\_LIMIT + 1);

if (fgets(file, (int) FILENAME\_LIMIT, stdin) == NULL) {

printf("Error reading file name!\n");

return 1;

}

file[strcspn(file, "\n")] = '\0';

output[i] = open(file, O\_CREAT | O\_RDWR, 0777);

check\_wrong(output[i], -1, "Error opening output file!\n")

}

int fd[child\_count];

int fdMutex[child\_count];

int fdCond[child\_count];

pthread\_mutex\_t\* mutex[child\_count];

pthread\_cond\_t\* condition[child\_count];

pthread\_mutexattr\_t mutex\_attribute;

check\_ok(pthread\_mutexattr\_init(&mutex\_attribute), 0, "Error initializing mutex attribute!\n")

check\_ok(pthread\_mutexattr\_setpshared(&mutex\_attribute, PTHREAD\_PROCESS\_SHARED), 0, "Error sharing mutex attribute!\n")

pthread\_condattr\_t condition\_attribute;

check\_ok(pthread\_condattr\_init(&condition\_attribute), 0, "Error initializing cond attribute!\n")

check\_ok(pthread\_condattr\_setpshared(&condition\_attribute, PTHREAD\_PROCESS\_SHARED), 0, "Error sharing cond attribute!\n")

for (int i = 0; i < child\_count; ++i) {

/\* Shared file \*/

fd[i] = shm\_open(SHARED\_FILE\_NAME[i], O\_RDWR | O\_CREAT, S\_IRWXU);

check\_wrong(fd[i], -1, "Error creating shared file!\n")

check\_ok(ftruncate(fd[i], SHARED\_MEMORY\_SIZE), 0, "Error truncating shared file!\n")

/\* Shared mutex \*/

fdMutex[i] = shm\_open(SHARED\_MUTEX\_NAME[i], O\_RDWR | O\_CREAT, S\_IRWXU);

check\_ok(ftruncate(fdMutex[i], sizeof(pthread\_mutex\_t)), 0, "Error creating shared mutex file!\n")

mutex[i] = (pthread\_mutex\_t\*) mmap(NULL, sizeof(pthread\_mutex\_t), PROT\_READ | PROT\_WRITE, MAP\_SHARED, fdMutex[i], 0);

check\_wrong(mutex[i], MAP\_FAILED, "Error mapping shared mutex!\n")

check\_ok(pthread\_mutex\_init(mutex[i], &mutex\_attribute), 0, "Error initializing mutex!\n")

/\* Shared cond \*/

fdCond[i] = shm\_open(SHARED\_COND\_NAME[i], O\_RDWR | O\_CREAT, S\_IRWXU);

check\_ok(ftruncate(fdCond[i], sizeof(pthread\_cond\_t)), 0, "Error creating shared cond file!\n")

condition[i] = (pthread\_cond\_t\*) mmap(NULL, sizeof(pthread\_cond\_t), PROT\_READ | PROT\_WRITE, MAP\_SHARED, fdCond[i], 0);

check\_ok(pthread\_cond\_init(condition[i], &condition\_attribute), 0, "Error initializing cond!\n")

}

check\_ok(pthread\_mutexattr\_destroy(&mutex\_attribute), 0, "Error destoying mutex attribute!\n")

check\_ok(pthread\_condattr\_destroy(&condition\_attribute), 0, "Error destoying cond attribute!\n")

/\* Creating child process \*/

int id\_1 = fork();

check\_wrong(id\_1, -1, "Error creating the first process!\n")

if (id\_1 == 0) {

check\_wrong(dup2(output[0], STDOUT\_FILENO), -1, "Dup2 error\n")

char \*Child1\_argv[] = {"child\_1", (char\*) SHARED\_FILE\_NAME[0], (char\*) SHARED\_MUTEX\_NAME[0], (char\*) SHARED\_COND\_NAME[0], NULL};

check\_wrong(execv("child", Child1\_argv), -1, "Error executing child process!\n")

} else {

int id\_2 = fork();

check\_wrong(id\_2, -1, "Error creating the second process!\n")

if (id\_2 == 0) {

check\_wrong(dup2(output[1], STDOUT\_FILENO), -1, "Dup2 error\n")

char \*Child2\_argv[] = {"child\_2", (char\*) SHARED\_FILE\_NAME[1], (char\*) SHARED\_MUTEX\_NAME[1], (char\*) SHARED\_COND\_NAME[1], NULL};

check\_wrong(execv("child", Child2\_argv), -1, "Error executing child process!\n")

} else {

char\* sharedFile[child\_count];

for (int i = 0; i < child\_count; ++i) {

sharedFile[i] = mmap(NULL, SHARED\_MEMORY\_SIZE, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd[i], 0);

check\_wrong(sharedFile[i], MAP\_FAILED, "Error creating shared file!")

sharedFile[i][0] = 0;

}

char c;

int proc\_index = 0;

while (scanf("%c", &c) > 0) {

if (proc\_index != 0 && proc\_index != 1) {

perror("Code error, it's a bug\n");

return -1;

}

check\_ok(write\_to\_process(sharedFile[proc\_index], mutex[proc\_index], condition[proc\_index], c), 0, "")

if (c == '\n') proc\_index ^= 1;

}

for (int i = 0; i < child\_count; ++i) {

check\_ok(pthread\_mutex\_lock(mutex[i]), 0, "Error locking mutex in parent!\n")

while (sharedFile[i][0] != 0) {

check\_ok(pthread\_cond\_wait(condition[i], mutex[i]), 0, "Error waiting cond in parent!\n")

}

sharedFile[i][0] = -1;

check\_ok(pthread\_cond\_signal(condition[i]), 0, "Error sending signal in parent!\n")

check\_ok(pthread\_mutex\_unlock(mutex[i]), 0, "Error unlocking mutex in parent!\n")

check\_wrong(munmap(sharedFile[i], SHARED\_MEMORY\_SIZE), -1, "Error unmapping sharedFile!")

}

}

}

for (int i = 0; i < child\_count; ++i) {

check\_ok(munmap(mutex[i], sizeof(pthread\_mutex\_t)), 0, "Error unmapping mutex!\n")

check\_ok(munmap(condition[i], sizeof(pthread\_cond\_t)), 0, "Error unmapping cond!\n")

check\_wrong(shm\_unlink(SHARED\_FILE\_NAME[i]), -1, "Error unlinking shared file!\n")

check\_wrong(shm\_unlink(SHARED\_MUTEX\_NAME[i]), -1, "Error unlinking shared mutex file!\n")

check\_wrong(shm\_unlink(SHARED\_COND\_NAME[i]), -1, "Error unlinking shared cond file!\n")

check\_ok(close(output[i]), 0, "Error closing input file!\n")

}

return 0;

}

================================= child.c ===================================

#include <fcntl.h>

#include <pthread.h>

#include <stdio.h>

#include <sys/mman.h>

#include <sys/stat.h>

#include <string.h>

#define check\_ok(VALUE, OK\_VAL, MSG) if (VALUE != OK\_VAL) { printf("%s", MSG); return 1; }

#define check\_wrong(VALUE, WRONG\_VAL, MSG) if (VALUE == WRONG\_VAL) { printf("%s", MSG); return 1; }

int is\_vowel(const char c) {

return (int) (

c == 'a' ||

c == 'e' ||

c == 'u' ||

c == 'o' ||

c == 'i');

}

int main(int argc, char\*\* argv) {

if (argc < 4) {

perror("Not enough arguments in child process\n");

return 1;

}

/\* Shared file \*/

int fd = shm\_open(argv[1], O\_RDWR, S\_IRWXU);

check\_wrong(fd, -1, "Error opening shared file in child process!\n")

struct stat statbuf;

check\_wrong(fstat(fd, &statbuf), -1, "Error getting shared file size in child!\n")

char\* sharedFile = mmap(NULL, statbuf.st\_size, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);

check\_wrong(sharedFile, MAP\_FAILED, "Error mapping shared file in child process!\n")

/\* Shared mutex \*/

int fdMutex = shm\_open(argv[2], O\_RDWR, S\_IRWXU);

check\_wrong(fdMutex, -1, "Error opening shared mutex file in child process!\n")

pthread\_mutex\_t\* mutex = mmap(NULL, sizeof(pthread\_mutex\_t), PROT\_READ | PROT\_WRITE, MAP\_SHARED, fdMutex, 0);

check\_wrong(mutex, MAP\_FAILED, "Error mapping shared mutex file in child process!\n")

/\* Shared cond \*/

int fdCond = shm\_open(argv[3], O\_RDWR, S\_IRWXU);

check\_wrong(fdCond, -1, "Error opening shared cond file in child process!\n")

pthread\_cond\_t\* condition = mmap(NULL, sizeof(pthread\_cond\_t), PROT\_READ | PROT\_WRITE, MAP\_SHARED, fdCond, 0);

check\_wrong(condition, MAP\_FAILED, "Error mapping shared cond file in child process!\n")

while (1) {

check\_ok(pthread\_mutex\_lock(mutex), 0, "Error locking mutex in child!\n")

while (sharedFile[0] == 0) {

check\_ok(pthread\_cond\_wait(condition, mutex), 0, "Error waiting cond in child!\n")

}

if (sharedFile[0] == -1) {

check\_ok(pthread\_mutex\_unlock(mutex), 0, "Error unlocking mutex in child!\n")

break;

}

if (!is\_vowel(sharedFile[0])) {

printf("%c", sharedFile[0]);

}

sharedFile[0] = 0;

check\_ok(pthread\_cond\_signal(condition), 0, "Error sending signal in child!\n")

check\_ok(pthread\_mutex\_unlock(mutex), 0, "Error unlocking mutex in child!\n")

}

check\_ok(pthread\_mutex\_destroy(mutex), 0, "Error destroying mutex!\n")

check\_ok(pthread\_cond\_destroy(condition), 0, "Error destroying cond!\n")

check\_wrong(munmap(mutex, sizeof(pthread\_mutex\_t)), -1, "Error unmapping shared mutex file in child!")

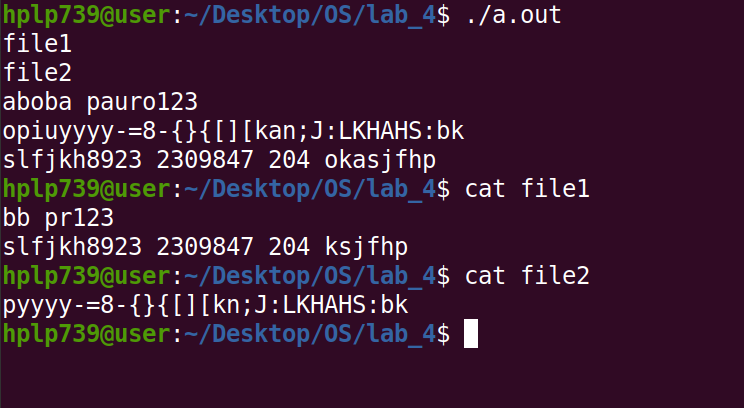
check\_wrong(munmap(condition, sizeof(pthread\_cond\_t)), -1, "Error unmapping shared cond file in child!")

check\_wrong(munmap(sharedFile, statbuf.st\_size), -1, "Error unmapping shared file in child!")

return 0;

}

**Демонстрация работы программы**



**Выводы**

В ходе выполнения работы я изучил основы работы с файлами, отображаемыми в память, составил программу, в которой синхронизировал работу двух процессов с помощью общих файлов, узнал, что в ОС Ubuntu общие файлы располагаются в /dev/shm. В современных реалиях пользователю приходится открывать сразу много приложений. Поместить в память все данные может быть невозможным, поэтому при разработке ОС важно предусмотреть выгрузку фоновых процессов на диск и вовремя подгрузить их.