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

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' ||
```

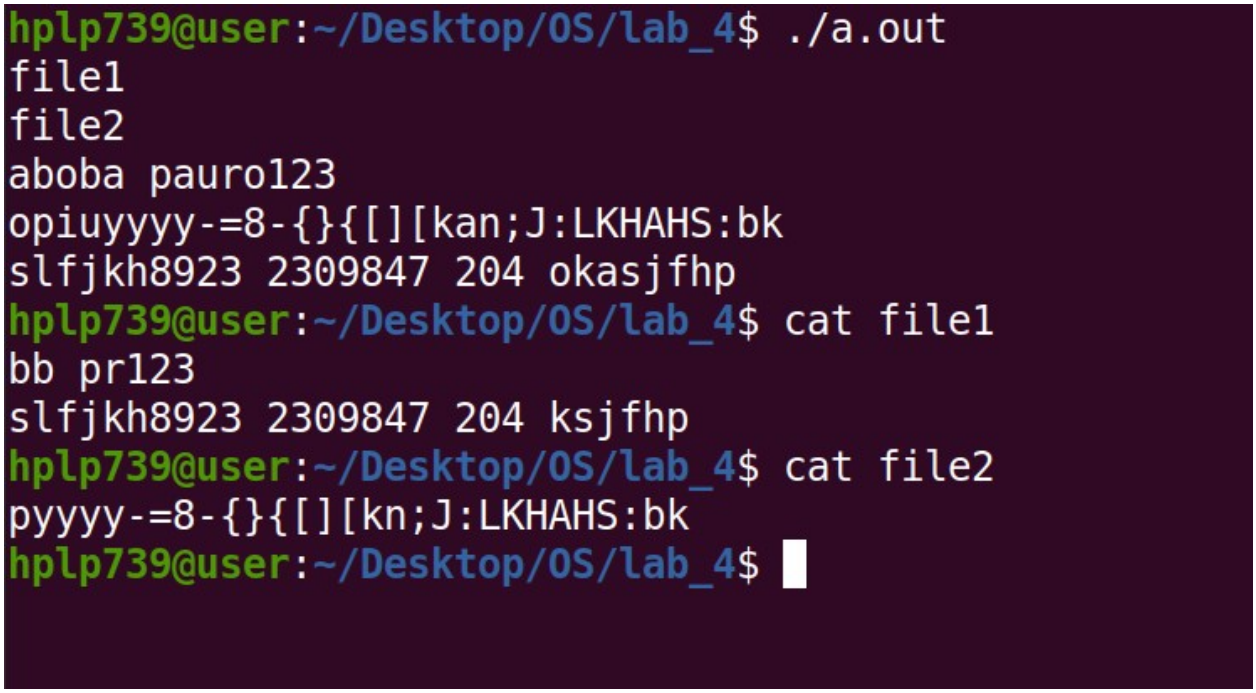
```
        c == 'i');
}

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.