Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»
Дисциплина «Операционные системы»

Лабораторная работа №4

Тема: Отображение файла в память

Студент: Инютин М. А.
Группа: М8О-207Б-19
Преподаватель: Миронов Е. С.
Дата:
Оценка:

Москва, 2020

1. Постановка задачи

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 через отображаемые файлы (memory-mapped files). Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

Вариант 6. В файле записаны команды вида: «число число число <endline>». Дочерний процесс считает их сумму и выводит результат в стандартный поток вывода. Числа имеют тип int.

2. Описание программы

Создадим общие файлы для синхронизации процессов: один для записи и чтения суммы, второй для мьютекса и третий для условной переменной. Нужно не забыть перед инициализацей общего мьютекса изменить длину файла. В дочерний процесс передадим названия общих файлов. Для передачи суммы основному процессу будем писать результат в общий файл и ждать, пока сумму не прочитают и не сотрут. В основном процессе после всех вычислений нужно уничтожить все общие файлы.

3. Набор тестов

Основной процесс читает имя файла, в котором записаны команды. Команды представляют собой несколько чисел в строке.

input.txt
```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
-1 -2 -3 -4 -5 1 2 3 4 5
1
2147483647
-2147483648
1 2 3 4 5 6 7 8 9 10
```

4. Результат выполнения тестов

Основной процесс для каждой строки выводит результат выполнения команд.

```
210
0
1
2147483647
-2147483648
55
```

5. Листинг программы

Программа представлена в двух файлах main.c и child.c. Первый для основного процесса, а второй для дочернего.

child.c

```
#include <fcntl.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>

#define check_ok(VALUE, OKVAL, MSG) if (VALUE != OKVAL) {
printf("%s", MSG); return 1; }
#define check_wrong(VALUE, WRONGVAL, MSG) if (VALUE == WRONGVAL) {
printf("%s", MSG); return 1; }

int main(int argc, char** argv) {
     /* 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");

     int num = 0, sum = 0, minus = 0;
     char c;
```

```
    char* sumString = malloc(sizeof(char) * statbuf.st_size);
    check_wrong(sumString, NULL, "Error allocating memory in
child!\n");
    while (scanf("%c", &c) > 0) {
        if (c == ' ' || c == '\t') {
            sum = minus ? sum - num : sum + num;
            num = 0;
            minus = 0;
        } else if (c == '-') {
            minus = 1;
        } else if (c == '\n') {
            sum = minus ? sum - num : sum + num;
            num = 0;
            minus = 0;
            int tmpSum = sum, i = 0;
            while (tmpSum != 0) {
                sumString[i++] = '0' + abs(tmpSum % 10);
                tmpSum = tmpSum / 10;
            }
            if (sum == 0) {
                sumString[i++] = '0';
            }
            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 (sum < 0) {
                sharedFile[0] = '-';
                for (int j = 0; j < i; ++j) {
                    sharedFile[1 + j] = sumString[i - j - 1];
                }
            } else {
                for (int j = 0; j < i; ++j) {
                    sharedFile[j] = sumString[i - j - 1];
                }
            }
            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");
            sum = 0;
        } else if ('0' <= c && c <= '9') {
            num = num * 10 + c - '0';
        }
    }
    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");
        }
        sharedFile[0] = 'a';
        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_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!");
        free(sumString);
        return 0;
}
```

main.c

```
#include <fcntl.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

#define check_ok(VALUE, OKVAL, MSG) if (VALUE != OKVAL) {
printf("%s", MSG); return 1; }
#define check_wrong(VALUE, WRONGVAL, MSG) if (VALUE == WRONGVAL) {
printf("%s", MSG); return 1; }

/* Ubuntu has 255 symbol filename limit */
const unsigned long long FILENAME_LIMIT = 255;
const unsigned long long SHARED_MEMORY_SIZE = 16;
const char* CHILD_EXECUTABLE_NAME = "child.out";
const char* SHARED_FILE_NAME = "shared_file";
const char* SHARED_MUTEX_NAME = "shared_mutex";
const char* SHARED_COND_NAME = "shared_cond";

int main() {
     char* s = malloc(sizeof(char) * (FILENAME_LIMIT + 1));
     check_wrong(s, NULL, "Error allocating memory!\n");
     for (int i = 0; i < FILENAME_LIMIT + 1; i++) {
          s[i] = 0;
     }
     if (!(scanf("%s", s) > 0)) {
          printf("Error reading file name!\n");
          return 1;
     }
     FILE* input = fopen(s, "r");
     check_wrong(input, NULL, "Error opening input file!\n");
     /* Shared file */
     int fd = shm_open(SHARED_FILE_NAME, O_RDWR | O_CREAT,
S_IRWXU);
     check_wrong(fd, -1, "Error creating shared file!\n");
     check_ok(ftruncate(fd, SHARED_MEMORY_SIZE), 0, "Error
truncating shared file!\n");
     /* Shared mutex */
     int fdMutex = shm_open(SHARED_MUTEX_NAME, O_RDWR | O_CREAT,
S_IRWXU);
     check_ok(ftruncate(fdMutex, sizeof(pthread_mutex_t)), 0,
"Error creating shared mutex file!\n");

     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_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!\
n");
    check_ok(pthread_mutex_init(mutex, &mutex_attribute), 0,
"Error initializing mutex!\n");
    check_ok(pthread_mutexattr_destroy(&mutex_attribute), 0,
"Error destoying mutex attribute!\n");
    /* Shared cond */
    int fdCond = shm_open(SHARED_COND_NAME, O_RDWR | O_CREAT,
S_IRWXU);
    check_ok(ftruncate(fdCond, sizeof(pthread_cond_t)), 0, "Error
creating shared cond file!\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");

    pthread_cond_t* condition = (pthread_cond_t*)mmap(NULL,
sizeof(pthread_cond_t), PROT_READ | PROT_WRITE, MAP_SHARED,
fdCond, 0);
    check_wrong(mutex, MAP_FAILED, "Error mapping shared cond!\
n");
    check_ok(pthread_cond_init(condition, &condition_attribute),
0, "Error initializing cond!\n");
    check_ok(pthread_condattr_destroy(&condition_attribute), 0,
"Error destoying cond attribute!\n");
    /* Creating child process */
    int id = fork();
    check_wrong(id, -1, "Error creating process!\n");
    if (id == 0) {
        check_wrong(dup2(fileno(input), fileno(stdin)), -1,
"Error changing stdin in child process!");
        char** argv = malloc(sizeof(char*) * 5);
        check_wrong(argv, NULL, "Erorr allocating memory!");
        argv[0] = malloc(sizeof(char) * 10);
        memcpy(argv[0], CHILD_EXECUTABLE_NAME, 10);
        argv[1] = malloc(sizeof(char) * 12);
        memcpy(argv[1], SHARED_FILE_NAME, 12);
        argv[2] = malloc(sizeof(char) * 13);
        memcpy(argv[2], SHARED_MUTEX_NAME, 13);
        argv[3] = malloc(sizeof(char) * 12);
        memcpy(argv[3], SHARED_COND_NAME, 12);
        argv[4] = NULL;
        check_wrong(execv(CHILD_EXECUTABLE_NAME, argv), -1,
"Error executing child process!\n");
```

```
    } else {
        char* sharedFile = mmap(NULL, SHARED_MEMORY_SIZE,
PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
        check_wrong(sharedFile, MAP_FAILED, "Error creating
shared file!");
        while (1) {
            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");
            }
            if (sharedFile[0] == 'a') {
                check_ok(pthread_mutex_unlock(mutex), 0, "Error
unlocking mutex in parent!\n");
                break;
            }
            printf("%s\n", sharedFile);
            for (int j = 0; j < SHARED_MEMORY_SIZE; ++j) {
                sharedFile[j] = 0;
            }
            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");
        }
        check_wrong(munmap(sharedFile, SHARED_MEMORY_SIZE), -1,
"Error unmapping fd1!");
    }
    check_ok(pthread_mutex_destroy(mutex), 0, "Error destroying
mutex!\n");
    check_ok(munmap(mutex, sizeof(pthread_mutex_t)), 0, "Error
unmapping mutex!\n");
    check_ok(pthread_cond_destroy(condition), 0, "Error destroying
cond!\n");
    check_ok(munmap(condition, sizeof(pthread_cond_t)), 0, "Error
unmapping cond!\n");

    check_wrong(shm_unlink(SHARED_FILE_NAME), -1, "Error unlinking
shared file!\n");
    check_wrong(shm_unlink(SHARED_MUTEX_NAME), -1, "Error
unlinking shared mutex file!\n");
    check_wrong(shm_unlink(SHARED_COND_NAME), -1, "Error unlinking
shared cond file!\n");

    check_ok(fclose(input), 0, "Error closing input file!\n");
    free(s);
    return 0;
}
```

6. 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.

Список литературы

1. shm_open(3) — Linux manual page — man7.org
   URL: https://man7.org/linux/man-pages/man3/shm_open.3.html
   (дата обращения 24.11.2020)
2. mmap(2) — Linux manual page — man7.org
   URL: https://man7.org/linux/man-pages/man2/mmap.2.html
   (дата обращения 24.11.2020)
3. Mutual Exlusion Lock Attributes — Multithreaded Programming Guide
   URL: https://docs.oracle.com/cd/E19683-01/806-6867/sync-83/index.html
   (дата обращения 25.11.2020)
4. Condition Variable Attributes — Multithreaded Programming Guide
   URL: https://docs.oracle.com/cd/E19683-01/806-6867/sync-91921/index.html
   (дата обращения 25.11.2020)