Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу
«Операционные системы»

ОТОБРАЖАЕМЫЕ ФАЙЛЫ

Студент: Харьков Павел Александрович
Группа: М8О–206Б–19
Вариант: 20
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2021.

Постановка задачи

Цель работы

Приобретение практических навыков в:

- Освоение принципов работы с файловыми системами
- Обеспечение обмена данных между процессами посредством технологии «File mapping»

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 через отображаемые файлы (memory-mapped files).

Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

Вариант 20: 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 pipe1 или в pipe2 в зависимости от правила фильтрации. Процесс child1 и child2 инвертируют строки. Процессы пишут результаты своей работы в стандартный вывод.

Общие сведения о программе.

Программа компилируется из файлов laba_4.c, get_line.c . Также используется заголовочные файлы: unistd.h, stdlib.h, stdio.h, fcntl.h, string.h, sys/mman.h, signal.h, semaphore.h. В программе используются следующие системные вызовы:

- fork – создает копию процесса.
- sem_open – создание семафора.
- sem_close – закрытие семафора.
- sem_unlink – отвязка именного семафора от имени.
- read – читает в память из файлового дескриптора определенное количество байт.

- write – пишет в файловый дескриптор из памяти определенное количество байт.
- open – открывает определенный файл и возвращает его файловый дескриптор.
- close – закрывает файловый дескриптор.
- mmap – отображение файла в память.
- mummap – удаление отображений.

Общий метод и алгоритм решения.

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы fork, sem_open, open, close, read, write, mmap, mummap.
2. Написать программу, которая будет работать с 3-мя процессами: один родительский и два дочерних, процессы связываются отображенными файлами и работают с именными семафорами.
3. Отладить программу и протестировать на тестах.

Основные файлы программы.

laba_4.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <sys/mman.h>
#include <signal.h>

#include <semaphore.h>

#include "get_line/get_line.h"

#define SEM_FAILED ((sem_t *) 0)

const int BUFF = 128;
sem_t* child_sem[2];
char* chsem_names[2] = { "/child0", "/child1" };
sem_t* par_sem[2];
char* parsem_names[2] = { "/par0", "/par1" };
char* mmap_files[2];
int id[2];
```

```
void OpenSems() {
    for (int i = 0; i < 2; ++i) {
        child_sem[i] = sem_open(chsem_names[i], O_CREAT, 0666, 0);
        if (child_sem[i] == SEM_FAILED) {
            perror("Error open: ");
            exit(-1);
        }
    }
    for (int i = 0; i < 2; ++i) {
        par_sem[i] = sem_open(parsem_names[i], O_CREAT, 0666, 1);
        if (par_sem[i] == SEM_FAILED) {
            perror("Error open: ");
            exit(-1);
        }
    }
}

void CloseSems() {
    for (int i = 0; i < 2; ++i) {
        if (sem_close(child_sem[i])) {
            perror("Error close: ");
        }
    }
    for (int i = 0; i < 2; ++i) {
        if (sem_close(par_sem[i])) {
            perror("Error close: ");
        }
    }
}

void UnlinkSems() {
    for (int i = 0; i < 2; ++i) {
        if (sem_unlink(chsem_names[i]) < 0) {
            perror("Error unlink: ");
        }
        if (sem_unlink(parsem_names[i]) < 0) {
            perror("Error unlink: ");
        }
    }
}

void Munmap(){
    munmap(mmap_files[0], BUFF);
    munmap(mmap_files[1], BUFF);
}

void ReverseLine(char* line, int size)
{
    for(int i = 0; i < size / 2; ++i)
    {
```

```
        char tmp = line[size - i - 1];
        line[size - i - 1] = line[i];
        line[i] = tmp;
    }
}

void sig_handlerchd(int sig){
    CloseSems();
    exit(0);
}

int Child(int id, char* file_path){
    if(signal(SIGTERM, sig_handlerchd) == SIG_ERR){
        perror("Error signal: ");
        exit(-1);
    }
    int fd;
    if ((fd = open(file_path, O_WRONLY|O_CREAT|O_TRUNC, 0666)) < 0) {
        perror(file_path);
        exit(-1);
    }

    int size = 0;
    while(1) {
        sem_wait(child_sem[id]);

        memcpy(&size, mmap_files[id], sizeof(size));
        if (size == 0) {
            sem_post(par_sem[id]);
            break;
        }
        char line[size+2];
        memcpy(line, mmap_files[id] + sizeof(size), size*sizeof(char));
        line[size] = '\n';
        line[size+1] = '\0';

        ReverseLine(line, size);

        write(fd, line, size+1);

        sem_post(par_sem[id]);
    }

    close(fd);

    return 0;
}

void WriteToMmap(int id, int size, char* line)
{
```

```
    sem_wait(par_sem[id]);
    memcpy(mmap_files[id], (char*)(&size), sizeof(size));
    memcpy(mmap_files[id] + sizeof(size), line, size*sizeof(char));
    sem_post(child_sem[id]);
}

void Parent()
{
    char* line = NULL;
    int size;

    while ((size = get_line(&line, STDIN_FILENO)) != 0) {
        if(size > 10){
            WriteToMmap(1, size, line);
        }else{
            WriteToMmap(0, size, line);
        }

    }
    free(line);
}

void Close(){
    Munmap();
    UnlinkSems();
    CloseSems();
}

void sig_handler(int sig){
    kill(id[0], SIGTERM);
    kill(id[1], SIGTERM);
    Close();
    exit(0);
}

int main()
{
    char* file_path = NULL;
    if(get_line(&file_path, STDIN_FILENO) == 0){
        free(file_path);
        exit(0);
    }
    char* file_path2 = NULL;
    if(get_line(&file_path2, STDIN_FILENO) == 0){
        free(file_path);
        free(file_path2);
        exit(0);
    }
```

```
   mmap_files[0] = mmap(NULL, BUFF, PROT_READ | PROT_WRITE, MAP_SHARED | M
AP_ANONYMOUS, -1, 0);
   mmap_files[1] = mmap(NULL, BUFF, PROT_READ | PROT_WRITE, MAP_SHARED | M
AP_ANONYMOUS, -1, 0);

   if (mmap_files[0] == MAP_FAILED) {
      perror("Error mmap: ");
      exit(-1);
   }
   if (mmap_files[1] == MAP_FAILED) {
      perror("Error mmap: ");
      munmap(mmap_files[0], BUFF);
      exit(-1);
   }
   OpenSems();

   if(signal(SIGINT, sig_handler) == SIG_ERR){
      perror("Error signal: ");
      exit(-1);
   }

   id[0] = fork();
   if (id[0] == -1)
   {
      perror("Error fork: ");
      Close();
      exit(-1);
   }
   else if(id[0] == 0)
   {
      Child(0, file_path);
   }
   else
   {
      id[1] = fork();
      if (id[1] == -1)
      {
         perror("Error fork: ");
         Close();
         exit(-1);
      }
      else if(id[1] == 0)
      {
         Child(1, file_path2);
      }
      else
      {
         Parent();
         UnlinkSems();
         kill(id[0], SIGTERM);
         kill(id[1], SIGTERM);
```

```
        }
    }
    free(file_path);
    free(file_path2);

    Munmap();
    CloseSems();
    return 0;
}
```

get_line.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

#include "get_line.h"

int get_char(int fd)
{
    char c;
    if (read(fd, &c, 1) == 1)
        return (unsigned char)c;
    return EOF;
}

int get_line(char** line_, int fd)
{
    free(*line_);
    int c;
    int size = 0;
    int cap = 4;
    char* line = (char*)malloc(cap * sizeof(char));
    while((c = get_char(fd)) != EOF && c != '\n' && c!= '\r')
    {
        if(size == cap)
        {
            cap *= 2;
            line = (char*)realloc(line, cap * sizeof(char));
            if(line == NULL)
                exit(-1);
        }
        line[size] = c;
        ++size;
    }
    if(size == cap)
    {
        cap += 1;
        line = (char*)realloc(line, cap * sizeof(char));
        if(line == NULL)
            exit(-1);
```

```
    }
    line[size] = '\0';
    *line_ = line;
    return size;
}
```

Пример работы программы.

```
pablo$ make
gcc -lpthread -pthread   -c -o src/get_line/get_line.o src/get_line/get_line.c
gcc -lpthread -pthread   -c -o src/laba_4.o src/laba_4.c
gcc -lpthread -pthread -o laba_4 src/laba_4.o src/get_line/get_line.o
pablo$ cat tests/test1
filef
files
hello world
hi
something long
short
pablo$ ./laba_4 < tests/test1
pablo$ cat filef
ih
trohs
pablo$ cat files
dlrow olleh
gnol gnihtemos
```

Выводы.

Выполнив данную лабораторную работу, я научился использовать семафоры. Благодаря им можно не бояться, что произойдет «гонка данных», которая может привести к неверным результатам. Также я узнал, что нужно обязательно отвязывать именной семафор от имени, так как сам он не удаляется и если создать неудаленный семафор, то значение при инициализации может быт другим. И еще я научился работать с отображенными в память файлами, они могут пригодиться, если через однонаправленные каналы будет неудобно отправлять текст.