Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет: “Информационные технологии и прикладная математика”
Кафедра: 806 “Вычислительная математика и программирование”

Лабораторная работа №4
по курсу “Операционные системы”

Студент	Полей-Добронравова А.В.
Группа	М8О-307Б-18
Вариант	7
Преподаватель	Миронов Е.С.
Дата	
Оценка	

Москва, 2020

1. 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.

7. На вход программе подается название 2-ух файлов. Необходимо отсортировать оба файла (каждый в отдельном процессе) произвольной сортировкой (на усмотрение студента). Содержимое обоих файлов вывести в стандартный поток вывода родительским процессом.

2. Решение задачи

Используемые системные вызовы:

- **ssize_t write(int fd, const void* buf, size_t count)** - пишет в файл, на который ссылается файловый дескриптор fd, count байт из буфера buf.
- **ssize_t read(int fd, void* buf, size_t count)** - считывает из файла, на который ссылается файловый дескриптор fd, count байт в буфер, начинающийся по адресу buf. Для файлов, поддерживающих смещения, операция чтения начнется с текущего смещения, потом сдвинется на count байт. Если текущее смещение за концом файла,

считывание не произойдет, и **read()** вернет **0**. Если count = 0, **read()** вернет **0**.

- **pid_t fork(void)** - Создаёт новый процесс копированием вызывающего процесса. Новый процесс называется *дочерним*, вызывающий процесс - *родительский*. Находятся в разных пространствах памяти, но имеют одинаковое содержимое сразу после вызова.
- **int close(int fd)** - закрывает файловый дескриптор.
- **void * mmap(void ****start****, size_t** ***length*****, int** ***prot*** **, int** ***flags*****, int** ***fd*****, off_t** ***offset*****) -** отражает *length* байтов, начиная со смещения *offset* файла (или другого объекта), определенного файловым описателем *fd*, в память, начиная с адреса *start*. Последний параметр (адрес) необязателен, и обычно бывает равен 0. Настоящее местоположение отраженных данных возвращается самой функцией mmap, и никогда не бывает равным 0. Аргумент *prot* описывает желаемый режим защиты памяти (он не должен конфликтовать с режимом открытия файла). Оно является либо **PROT_NONE** либо побитовым ИЛИ одного или нескольких флагов PROT_*.

  PROT_EXEC (данные в страницах могут исполняться);

  PROT_READ(данные можно читать);

  PROT_WRITE(в эту область можно записывать информацию);

  PROT_NONE(доступ к этой области памяти запрещен).

  Параметр *flags* задает тип отражаемого объекта, опции отражения и указывает, принадлежат ли отраженные данные только этому процессу или их могут читать другие. Он состоит из комбинации следующих битов:

  MAP_FIXED

  Не использовать другой адрес, если адрес задан в параметрах функции. Если заданный адрес не может быть использован, то функция**mmap** вернет сообщение об ошибке. Если используется

MAP_FIXED, то *start* должен быть пропорционален размеру страницы. Использование этой опции не рекомендуется.

MAP_SHARED

Разделить использование этого отражения с другими процессами, отражающими тот же объект. Запись информации в эту область памяти будет эквивалентна записи в файл. Файл может не обновляться до вызова функций **msync**(2) или **munmap**(2)**.**

MAP_PRIVATE

Создать неразделяемое отражение с механизмом copy-on-write. Запись в эту область памяти не влияет на файл. Не определено, являются или нет изменения в файле после вызова **mmap** видимыми в отраженном диапазоне.

fd должно быть корректным описателем файла, если только не установлено MAP_ANONYMOUS, так как в этом случае аргумент игнорируется.

offset должен быть пропорционален размеру страницы.

При удачном выполнении **mmap** возвращает указатель на область с отраженными данными. При ошибке возвращается значение MAP_FAILED (-1), а переменная *errno* приобретает соответствующее значение. При удачном выполнении **munmap** возвращаемое значение равно нулю. При ошибке возвращается -1, а переменная *errno* приобретает соответствующее значение. (Вероятнее всего, это будет EINVAL).

- **int munmap(void **start*, size_t *length*) -** освобождение за собой ресурсов.

Демонстрация работы программы:

```
MacBook-Pro-Amelia:Documents amelia$ gcc -o l4 lab4.c
MacBook-Pro-Amelia:Documents amelia$ ./l4 lab2os.txt lab2os1.txt buf1.txt buf2.txt
in parent process
 child process 1
```

child process 2
heloo
please i want be very cute
so fo gkgr
sooooooooooooooooooooooooooooooooooo
no if lfg
a a aaaaaaaa aaa

heloo
please i want be very cute
so fo gkgr
sooooooooooooooooooooooooooooooooooo
no if lfg
a a aaaaaaaa aaa

ended child process 1: 0

begining of file1
want te
sose i ry cuooooooooooooooooooooooooooooooooooo
nokgr
sheloofg
a be veaaaaaa aaa if l fo g aaa

pleaf
aa

end of file1

ended child process 2: 0

begining of file2
want te
sose i ry cuooooooooooooooooooooooooooooooooooo
nokgr
sheloofg
a be veaaaaaa aaa if l fo g aaa

pleaf
aa

end of file2

3. Листинг программы

```
#include <stdio.h>

#include <string.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <sys/mman.h>

#include <sys/stat.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <fcntl.h>

#include <string.h>

#include <signal.h>


struct FileMapping {

    int fl;

    size_t fsize;

    char* dataPtr;
```

```
};


int Sort(const char* argv, const char* name, char* dataPtr) // сортировка вставкой
{
    int fp, n = 100;
    if ((fp=open(argv, O_RDONLY)) < 0) {
        printf("Cannot open file.\n");
        printf("%d \n", fp);
        exit(-1);
        return -1;
    }
    char mass[n][6];
    char newElement[6], s[6], ss[6];
    int i = -1;
    while(read(fp,&newElement,(sizeof(char)* 5)) > 0) {
        if (i >= n) {
            printf("above limit \n");
            break;
        }
        i++;
        for (int f = 0; f < 6; f++) {
```

```
        mass[i][f] = newElement[f];
    }
}
for (int k = 0; k < i; k++)
{
    for (int d = 0; d < 6; d++) {
        printf("%c", mass[k][d]);
    }
}
printf(" \n");
for (int j = 0; j <= i; j++)
{
    for (int k = (i - 1); k > j; k--) // для всех элементов после i-ого
    {
        for (int f = 0; f < 6; f++) {
            s[f] = mass[k][f];
            ss[f] = mass[k - 1][f];
        }
        /* printf(" \n");
         printf("%s \n", s);
         printf("%s \n", ss);
         printf(" \n");
```

```
        */
        if (strcmp(ss,s) < 0) // если текущий элемент меньше предыдущего
        {
            strcpy(newElement,ss); // меняем их местами
            for (int f = 0; f < 6; f++) {
                mass[k - 1][f] = mass[k][f];
            }
            for (int f = 0; f < 6; f++) {
                mass[k][f] = newElement[f];
            }
        }
    }
}
printf(" \n");
close(fp);
int fl = open(name, O_WRONLY, 0); //Открытие для записи
if(fl < 0) {
    printf("FileMappingOpen - open failed, fname = %s \n", name);
    return(-1);
}
printf(" \n");
int k = 0;
```

```
    for(int j = 0; j <= i; j++) {
        for (int y = 0; y < 6; y++) {
            dataPtr[k] = mass[j][y];
            k++;
        }
    }
    close(fl);
    return 0;
}

int main(int argc, const char * argv[]) {
    pid_t pid1, pid2;
    int status1, status2;
    struct FileMapping g1, g2;
      g1.fl = open(argv[3], O_CREAT | O_APPEND | O_RDWR, S_IWUSR | S_IRUSR);
    //  O_RDWR открытие для записи и чтения
      g2.fl = open(argv[4], O_CREAT | O_APPEND | O_RDWR, S_IWUSR | S_IRUSR);
    //  O_RDWR открытие для записи и чтения
    g1.fsize = 200;
    g2.fsize = 200;
```

```
    ftruncate(g1.fl, g1.fsize); // усечение файла до размера 100
    ftruncate(g2.fl, g2.fsize);
     g1.dataPtr = (char*)mmap(NULL, g1.fsize, PROT_READ | PROT_WRITE, MAP_SHARED, g1.fl, 0); //Создаем отображение файла в память
    // PROT_READ страницы могут быть прочитаны
    // PROT_WRITE стр могут быть описаны
     //  MAP_SHARED стр могут сипользоваться совместно с др процессами, которые также проектируют этот объект в память
    if (g1.dataPtr == MAP_FAILED)
       // при ошибке возвращается значение MAP_FAILED
    {
       perror("Map");
       printf("FileMappingCreate - open failed 2, fname = %s \n", argv[3]);
       close(g1.fl);
       exit(-1);
    }
     g2.dataPtr = (char*)mmap(NULL, g2.fsize, PROT_READ | PROT_WRITE, MAP_SHARED, g2.fl, 0); //Создаем отображение файла в память
    // PROT_READ страницы могут быть прочитаны
    // PROT_WRITE стр могут быть описаны
     //  MAP_SHARED стр могут сипользоваться совместно с др процессами, которые также проектируют этот объект в память
    if (g2.dataPtr == MAP_FAILED)
```

```
    // при ошибке возвращается значение MAP_FAILED
{
    perror("Map");
    printf("FileMappingCreate - open failed 2, fname = %s \n", argv[4]);
    close(g2.fl);
    exit(-1);
}

pid1 = fork();
if (pid1 == 0) { // child process 1
    printf(" child process 1 \n");
    exit(Sort(argv[1], argv[3],g1.dataPtr));
} else if (pid1 < 0) {
    perror("fork");
} else {
    printf("in parent process \n");
    pid2 = fork();
    if (pid2 == 0) { // child process 2
        printf(" child process 2 \n");
        exit(Sort(argv[2], argv[4],g2.dataPtr));
    }
    else if (pid2 < 0) {
```

```
    perror("fork");
}
if (waitpid(pid1, &status1, 0) == -1) {
    perror("waitpid");
}
else {
    printf("\n ended child process 1: %d \n", status1);
    printf("\n begining of file1 \n");
    for(int k = 0; k <= g1.fsize; k++){
        printf("%c", g1.dataPtr[k]);
    }
    printf("\n end of file1 \n");
}
if (waitpid(pid2, &status2, 0) == -1) {
    perror("waitpid");
}
else {
    printf("\n ended child process 2: %d \n", status2);
    printf("\n begining of file2 \n");
    for(int k = 0; k <= g2.fsize; k++){
        printf("%c", g2.dataPtr[k]);
    }
```

```
            printf("\n end of file2 \n");
        }
    }
    munmap(g1.dataPtr,g1.fsize);
    munmap(g2.dataPtr,g2.fsize);
    close(g1.fl);
    close(g2.fl);
    return 0;
}
```

6. Вывод

Существует несколько способов передачи данных между процессами, в этой лабораторной использован способ mmap.

Когда мы обращаемся к памяти, в которую отображен файл, данные загружаются с диска в кэш(если их там ещё нет), затем делается отображение кэша в адресное пространство нашей программы. Если эти данные удаляются — отображение отменяется. Таким образом, мы избавляемся от операции копирования из кэша в буфер. Кроме того, нам не нужно париться по поводу оптимизации работы с диском — всю грязную работу берёт на себя ядро ОС.