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:	Голубев В.С.
Группа:	М8О-206Б-18
Вариант:	26
Преподаватель:	Миронов Е.С.
Оценка:	
Дата:	

Москва, 2019

1. Постановка задачи

Дочерний процесс при создании принимает имя файла. При работе дочерний процесс получает числа от родительского процесса и пишет их в файл. Родительский процесс создает n дочерних процессов и передает им поочередно числа из последовательности от 1...m.

Операционная система: Unix.

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.

1. Решение задачи

Создать файл, затем с помощью утилиты ftruncate() задать ему размер. Далее с помощью mmap сделать отображение этого файла в память, сделав на нее указатель в виде массива типа int. Считывая данные из этой памяти, а также запись в нее осуществляется посредством обращения к данному массиву типа int.

Используемые системные вызовы:

- **pid_t fork(void) -** создаёт новый процесс посредством копирования вызывающего процесса. Новый процесс считается *дочерним* процессом. Вызывающий процесс считается *родительским* процессом. Дочерний и родительский процессы находятся в отдельных пространствах памяти. Сразу после **fork**() эти пространства имеют одинаковое содержимое.

- **int pipe(int** *pipefd***[2])-** создаёт однонаправленный канал данных, который можно использовать для взаимодействия между процессами. Массив *pipefd* используется для возврата двух файловых описателей, указывающих на концы канала. *pipefd[0]* указывает на конец канала для чтения. *pipefd[1]* указывает на конец канала для записи. Данные, записанные в конец канала, буферизируются ядром до тех пор, пока не будут прочитаны из конца канала для чтения.
- **void * mmap (void *****start***, size_t** *length***, int** *prot***, int** *flags***, int** *fd*, **off_t** *offset***);**

- **int munmap (void ****start***, size_t** *length***);**

В функции **MMAP** проецируемая *длиной* байт из файла (или другого объекта) *Fd* от смещения *смещения* в области памяти, предпочтительно от адреса *начала* . Последний адрес - это всего лишь подсказка и обычно не указывается путем ввода 0. Фактическое пространство, в которое проецируется объект, возвращается **mmap** . Параметр *prot*o писывает требуемую защиту памяти. Он состоит из следующих бит:

PROT_EXEC
Страницы могут быть выполнены.
PROT_READ
Страницы могут быть прочитаны.
PROT_WRITE
Страницы могут быть описаны.
Параметр *flags* указывает тип объекта для параметров проекта и проекции, а также то, являются ли изменения в копии проецируемого объекта для процесса частными или совместно с другими ссылками. Он состоит из следующих бит:
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
Страницы могут использоваться совместно с другими процессами, которые также проектируют этот объект в память.
MAP_PRIVATE
Создайте приватную проекцию объекта копирования на запись.
Вышеуказанные три флага описаны в POSIX.4. В Linux также есть MAP_DENYWRITE, MAP_EXECUTABLE и MAP_ANON (YMOUS).
Munmap Системный вызов удаляет проекции в указанной области хранения. Будущие обращения к этому адресному пространству будут генерировать неверную ошибку ссылки на память - Недопустимый доступ к памяти.

ДЕМОНСТРАЦИЯ РАБОТЫ ПРОГРАММЫ:

1.
```
vsgolubev@vsgolubev-VirtualBox:~/OS/lab4$ ./a.out
Enter name of out file
1.txt
Enter name of memory file
2.txt
Enter n
5
Enter m
5
vsgolubev@vsgolubev-VirtualBox:~/OS/lab4$ cat 1.txt
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
vsgolubev@vsgolubev-VirtualBox:~/OS/lab4$
```

2. Руководство по использованию программы

Компиляция и запуск программного кода в *Ubuntu* :

gcc lab4.c

3. Листинг программы

```
#include
<unistd.h>
            #include <sys/mman.h>
            #include <stdio.h>
            #include <sys/stat.h>
            #include <sys/types.h>
            #include <sys/wait.h>
            #include <fcntl.h>
            #include <stdlib.h>
            struct FileMapping {
                    int file;
                    size_t fsize;
                    unsigned char* dataPtr;
            };
            void parentProcess(int* pipe_fd, int m, char *fname, char *fproc, int* dataPtr) {
                    int d;
                    int file = open(fproc, O_WRONLY, 0);
                    if(file < 0) {
                            printf("FileMappingOpen - open failed, fname = %s \n", fproc);
                            exit(-1);
                    }
                    for (int i = 0; i < m; ++i) {
                             dataPtr[i] = i + 1;
                    }
                    close(file);
            }
            void childProcess(int* pipe_fd, char *fname, char *fproc, int* dataPtr, int
            fsize) {
                    int d;
                    int fd;
                    fd = open(fname, O_CREAT | O_APPEND | O_WRONLY, S_IWUSR | S_IRUSR);
                    int file = open(fproc, O_RDONLY, 0);
                    if(file < 0) {
                            printf("FileMappingOpen - open failed, fname = %s \n", fproc);
                            exit(-1);
                    }
                    dup2(fd, 1);
                    int k = 0;
                    for(k; k < fsize; ++k){
                            if (dataPtr[k] != 0)
                                    printf("%i ", dataPtr[k]);
```

```
        }
        printf("\n");
        close(fd);
        close(file);
}
int main(int argc, char const *argv[]) {
        int pipe_fd[2];
        pid_t pid;
        char name_file[20];
        char proc_file[20];
        int count_process;
        int m;
        int err = 0;
        printf("Enter name of out file\n");
        scanf ("%s", name_file);
        printf("Enter name of memory file\n");
        scanf("%s", proc_file);
        printf("Enter n\n");
        scanf ("%d", &count_process);
        printf("Enter m\n");
        scanf ("%d", &m);
        int i = 0;
        int file = open(proc_file, O_CREAT | O_APPEND | O_RDWR, S_IWUSR |
S_IRUSR);
        size_t fsize = 100;
        ftruncate(file, fsize);
        int* dataPtr = (int*)mmap(NULL, fsize, PROT_READ | PROT_WRITE, MAP_SHARED,
file, 0);
        if (dataPtr == MAP_FAILED)
        {
                perror("Map");
                printf("FileMappingCreate - open failed 2, fname = %s \n",
proc_file);
                close(file);
                exit(-1);
        }
        for (i; i < count_process; ++i) {
                if (pipe(pipe_fd) == -1) {
                        perror("PIPE");
                        err = -2;
                }
                pid = fork();
                if (pid == -1) {
                        perror("FORK");
                        err = -1;
                }
                else if (pid == 0) {
                        childProcess(pipe_fd, name_file, proc_file, dataPtr, m);
```

```
                        break;
                } else
                        parentProcess(pipe_fd, m, name_file, proc_file, dataPtr);
        }
        return err;
}
```

4. Вывод

Разделяемая память является самым быстрым средством обмена данными между процессами.

В других средствах межпроцессового взаимодействия (IPC) обмен информацией между процессами проходит через ядро, что приводит к переключению контекста между процессом и ядром, т.е. к потерям производительности.

Техника разделяемой памяти позволяет осуществлять обмен информацией через общий для процессов сегмент памяти без использования системных вызовов ядра. Сегмент разделяемой памяти подключается в свободную часть виртуального адресного пространства процесса. Таким образом, два разных процесса могут иметь разные адреса одной и той же ячейки подключенной разделяемой памяти.