**Московский авиационный институт**

**(Национальный исследовательский университет)**

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Операционные системы»

**Лабораторная работа № 4**

Студент: Ваньков Д. А.

Группа: 80-207Б

Преподаватель: Соколов А. А.

Дата: 18.11.2018

Оценка:

Москва, 2018

1. **Постановка задачи**

Вариант 26.

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*о 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 используйте какой-либо другой адрес так, как указано. Если указанный адрес не может быть использован **mmap не** удастся. Если указано MAP\_FIXED, *начало* должно быть кратно размеру страницы. Мы не рекомендуем использовать этот параметр.

**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.

**Тесты программы:**

1. **Руководство по использованию программы**

Компиляция и запуск программного кода в *Ubuntu* :

*gcc -pthread four.c -lm*

1. **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 <pthread.h>

#include <stdlib.h>

struct FileMapping {

int filik;

size\_t fsize;

unsigned char\* dataPtr;

};

void parentProces(int\* pipe\_fd, int m, char \*fname, char \*fproc, int\* dataPtr) {

int d;

printf("Enter number\n");

int filik = open(fproc, O\_WRONLY, 0); //Открытие для записи

if(filik < 0) {

printf("FileMappingOpen - open failed, fname = %s \n", fproc);

exit(-1);

}

for (int i = 0; i < m; ++i) {

dataPtr[i] = i;

}

close(filik);

}

void childProces(int\* pipe\_fd, char \*fname, char \*fproc, int\* dataPtr, int fsize) {

int d;

int fd;

printf("Child\n");

fd = open(fname, O\_CREAT | O\_APPEND | O\_WRONLY, S\_IWUSR | S\_IRUSR); // Открыть на дозапись, если нет создать с правами без sudo

int filik = open(fproc, O\_RDONLY, 0); //Открытие для чтения

if(filik < 0) {

printf("FileMappingOpen - open failed, fname = %s \n", fproc);

exit(-1);

}

dup2(fd, 1); // Перенаправить вывод в файл fd

int k = 0;

for(k; k < fsize; ++k){

if (dataPtr[k] != 0)

printf("[%i] ", dataPtr[k]);

}

printf("\n");

close(fd);

close(filik);

}

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 filik = open(proc\_file, O\_CREAT | O\_APPEND | O\_RDWR, S\_IWUSR | S\_IRUSR);

size\_t fsize = 100;

ftruncate(filik, fsize);

int\* dataPtr = (int\*)mmap(NULL, fsize, PROT\_READ | PROT\_WRITE, MAP\_SHARED, filik, 0); //Создаем отображение файла в память

if (dataPtr == MAP\_FAILED) {

perror("Map");

printf("FileMappingCreate - open failed 2, fname = %s \n", proc\_file);

close(filik);

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) {

childProces(pipe\_fd, name\_file, proc\_file, dataPtr, m);

break;

} else

parentProces(pipe\_fd, m, name\_file, proc\_file, dataPtr);

printf("End Enter\n");

}

return err;

}

1. **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.