Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работ №4 по курсу**

**«Операционные системы»**

**Работа между процессами посредством memory-mapped file**

Студент: Петрухин Дмитрий Олегович

Группа: М8О –201Б-18

Вариант: 13

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2019

**Содержание**

1. Постановка задачи
2. Общие сведения о программе
3. Общий метод и алгоритм решения
4. Основные файлы программы
5. Тестирование
6. Демонстрация работы программы
7. Вывод

**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.

**Задание согласно 13 варианту:**

Родительский процесс считывает матрицы из стандартного ввода, дочерний процесс вычисляет среднеарифметическое матриц и передает их родительского процессу. Родительский процесс печатает результат в стандартный вывод.

**Общие сведения о программе**

В main.c используются заголовочные файлы pthread.h, stdio.h, stdlib.h, sys/types.h, sys/mman.h, semaphore.h. В программе используются следующие системные вызовы:

1. **open()** - Процедура открытия файла. При открытии файла операционная система проверяет, соответствуют ли права, которые запросил процесс для операций над файлом, правам доступа, установленным для этого файла. В случае соответствия она помещает необходимую информацию в системную таблицу файлов и, если этот файл не был ранее открыт другим процессом, в таблицу индексных дескрипторов открытых файлов. Далее операционная система находит пустой элемент в таблице открытых файлов процесса, устанавливает необходимую связь между всеми тремя таблицами и возвращает на пользовательский уровень дескриптор этого файла.
2. **sem\_t \*sem\_open(const char \*name, int offlag, mode\_t mode, unsigned int value) -** Системный вызов создает новый семафор. Первый аргумент - имя семафора, второй аргумент - флаг, отвечающий за работу вызова(O\_CREAT, то создается если не существует), третий аргумент отвечает за настройки прав доступа(аналогично open), четвертый аргумент задает начальное значение семафора. При успешном выполнении возвращает адрес данного семафора, при ошибке возвращает SEM\_FAILED, а в переменную errno записывается номер ошибки
3. **sem\_unlink(const char \*name)** - Удаляет именованный семафор на который ссылается аргумент name. Имя семафора удаляется немедленно. Семафор уничтожается после того, как все остальные процессы, в которых он открыт, закроют его. При успешном завершении возвращается 0, при ошибке -1, а в errno содержится код ошибки.
4. **ftruncate(const char\* path, off\_t length)** - Устанавливают длину обычного файла с именем *path* или файловым дескриптором *fd* в *length* байт. Если файл до этой операции был длиннее, то отсеченные данные теряются. Если файл был короче, то он увеличивается, а добавленная часть заполняется нулевыми байтами. Указатель на файл не меняется. При использовании ftruncate файл должен быть открыт для записи. При успешном завершении работы функции возвращает 0. При ошибке возвращается -1, а переменной errno присваивается номер ошибки.
5. **sem\_close(sem\_t \*sem)** - Закрывает именованный семафор, на который указывает *sem*, позволяя освободить все ресурсы, которые система выделила под семафор вызывающему процессу. При успешном завершении возвращает 0, при ошибке -1. Операция закрытия выполняется автоматически при завершении процесса для всех семафоров, которые были им открыты. Автоматическое закрытие осуществляется как при добровольном завершении работы (вызове exit или \_exit), так и при принудительном (с помощью сигнала). Закрытие семафора не удаляет его из системы. Именованные семафоры Posix обладают по меньшей мере живучестью ядра. Значение семафора сохраняется, даже если ни один процесс не держит его открытым.
6. **close(int fd)** - Функция закрывает файловый дескриптор, который после этого не ссылается ни на один файл и может быть использован повторно. При успешном завершении возвращает 0, в случае ошибки -1.
7. **void \* mmap(void \****start***, size\_t***length***, int***prot* **, int***flags***, int***fd***, off\_t***offset***)** - Системный вызов *mmap* служит для отображения предварительно открытого файла в адресное пространство вычислительной системы. Параметр start указывает начало области адресного пространства, в которую будет отображен файл, length отвечает за длину отображения файла в байтах, prot указывает возможные операции с файлом(операции не могу быть шире, чем они указаны при открытии). flags определяет способ отображения файла в адресное пространство.(MAX\_SHARED - полученное отображение используется другими процессами, вызвавшими mmap для этого файла, MAX\_PRIVATE - отображение файла в монопольном распоряжении процесса), fd файловый дескриптор для файла, offset - смещение отображения в память от начала файла в байтах. При успешном завершении возвращает начальный адрес в области памяти, в которую отображен файл, иначе MAP\_FAILED.
8. **int munmap (void \*start, size\_t length)** - Системный вызов *munmap* служит для прекращения отображения memory mapped файла в адресное пространство вычислительной системы. Параметр start является адресом начала области памяти, выделенной для отображения файла(то что вернул mmap), length - определяет ее длину, и его значение должно совпадать со значением соответствующего параметра в системном вызове *mmap()*. При нормально завершении возвращает 0, иначе -1;

**Общий метод и алгоритм решения**.

Для реализации поставленной задачи необходимо:

1. Создаю файл для чтения и записи, а также 2 семафора.
2. Отображаю файл в память.
3. Порождаю дочерний процесс.
4. В родительском процессе ввожу матрицы и затем записываю их в файл, дожидаюсь выполнения работы дочернего процесса.
5. Дочерний процесс считывает данные файла с памяти и вычисляет среднее арифметическое матриц.
6. Родительский процесс выводит результат.

**Листинг программы.**

**Файл main.c**

#include <unistd.h> // for fileno, read/write

#include <stdlib.h> //for atoi/exit

#include <string.h>

#include <stdio.h>

#include <semaphore.h>

#include <sys/mman.h>

#include <pthread.h>

#include <sys/types.h>

void write\_to\_memory(int a, int\* memory, int position){

int\* n = (int\*)(memory + 4\*position);

\*n = a;

}

int get\_from\_memory(int\* memory, int position){

int\* res = (int\*)(memory + 4\*position);

return \*res;

}

int main(){

char num1[10];

int num, summ;

int i, m, n, length;

int \*ptr, \*ptr2;

int res1, res2;

pid\_t p;

int fd = open("mapped.dat", O\_RDWR | O\_CREAT, 0666);

if(fd == -1){

printf("File open failed!\n");

exit(1);

}

sem\_t\* semaphore = sem\_open("lovesem", O\_CREAT, 777, 0);

sem\_t\* semaphore1 = sem\_open("lovesem1", O\_CREAT, 777, 0);

if(semaphore == SEM\_FAILED || semaphore1 == SEM\_FAILED){

printf("Semaphores doesn't create\n");

exit(1);

}

sem\_unlink("losesem");

sem\_unlink("lovesem1");

printf("Enter m \n");

read(STDIN\_FILENO,num1,10);

m = atoi(num1);

if (!atoi(num1)){

printf("m should not be zero \n");

printf("enter number m\n");

read(STDIN\_FILENO,num1,10);

m = atoi(num1);

}

printf("Enter n not equal to zero\n");

read(STDIN\_FILENO,num1,10);

n = atoi(num1);

length = m\*n\*sizeof(int) + 4\*sizeof(int);

ftruncate(fd,length);

ptr = (int\*)mmap(NULL, length, PROT\_WRITE |

PROT\_READ, MAP\_SHARED, fd, 0);

if (ptr == MAP\_FAILED) {

printf("Memmory mapping failed");

exit(1);

}

p = fork();

if(p == -1){ // everything is bad, the fork is not working

printf("Can't fork child");

exit(1);

} else if(p > 0){

printf("Enter first matrix \n");

for(i=2;i<(m\*n+2);i++) {

scanf("%d", &num);

write\_to\_memory(num,ptr,i);

}

printf("Enter second matrix \n");

for(i=(m\*n+2);i<(2\*m\*n+2);i++) {

scanf("%d", &num);

write\_to\_memory(num,ptr,i);

}

sem\_post(semaphore);

sem\_wait(semaphore1);//тут ждем

printf("Means value first matrix\n");

res1 = get\_from\_memory(ptr,0);

printf("%d\n", res1);

printf("Means value second matrix\n");

res2 = get\_from\_memory(ptr,1);

printf("%d\n", res2);

sem\_close(semaphore);

sem\_close(semaphore1);

munmap(ptr,length);

close(fd);

} else if (p == 0){

int res1, res2;

sem\_wait(semaphore);

for(i = 2;i < (m\*n+2);i++){

num = get\_from\_memory(ptr,i);

res1+=num;

}

res1 = res1 / (m\*n);

for(i = (m\*n+2);i < (2\*m\*n+2);i++){

num = get\_from\_memory(ptr,i);

res2+=num;

}

res2 = res2 / (m\*n);

write\_to\_memory(res1,ptr,0);

write\_to\_memory(res2,ptr,1);

sem\_post(semaphore1);

sem\_close(semaphore);

sem\_close(semaphore1);

close(fd);

}

return 0;

}

**Тестирование.**

**test1:**

abcd

2

2

5 5 5 5

10 10 10 10

**test2:**

3

4

99 15 21

30 1 9

0 0 1

19 19 19

1 1 1

1 1 1

1 1 1

100 111 111

**Демонстрация работы программы.**

➜ os\_lab\_04 git:(master) ✗ ./os4

Enter m

abcd

m should not be zero

enter number m

2

Enter n not equal to zero

2

Enter first matrix

5 5 5 5

Enter second matrix

10 10 10 10

Means value first matrix

5

Means value second matrix

10

➜ os\_lab\_04 git:(master) ✗ ./os4

Enter m

3

Enter n not equal to zero

4

Enter first matrix

99 15 21

30 1 9

0 0 1

19 19 19

Enter second matrix

1 1 1

1 1 1

1 1 1

100 111 111

Means value first matrix

19

Means value second matrix

27

**Вывод.**

Кажется, что отображение файла на дисковое пространство не имеет каких-либо преимуществ, но оно явно может ускорить перенос файла в память. 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.

Помимо этого, если компьютер не обладает достаточным количеством оперативной памяти, то c помощью mmap() реализуется виртуальная память, которая виртуально расширяет объем оперативной памяти, и, тем самым обеспечивая более эффективный процесс решения разного рода задач как для пользователей, так и для программистов.