****Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №4 по курсу**

**«Операционные системы»**

Группа: М80-207Б-18

Студент: Ильминский Никита Сергеевич

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Москва, 2020.

***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 основной процесс программы должен создать для решения задачи один или несколько дочерних процессов. Взаимодействие между процессами должно осуществляться через отображаемые файлы (memory-mapped files).

Также необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

*Вариант 23:*

Родительский процесс считывает две координаты, передает их через канал дочернему процессу. Дочерний процесс определяет, к какой четверти относится точка, а далее передает результат родительскому процессу.

***Алгоритм и описание программы***

Данная задача уже рассматривалась в лабораторной работе 2, однако тогда в качестве инструмента межпроцессного взаимодействия использовался pipe. Теперь же для этой же цели мы будем использовать системный вызов mmap, позволяющий выполнить отображение файла на память.

Для использования mmap необходим файл. Будем создавать файлы в программе (фнукиця int getFile()) при помощи функции mkstemp (генерация уникального временного имени файла, поданного параметром templateString, создание и открытие файла; возвращаемое значение – дескриптор); также будет лучше, если созданный файл будет временным, поэтому при помощи функции unlink мы удалим имя файла, а сам файл будет удален при завершении всех процессов, которые его используют; сразу будем обрабатывать возможные ошибки выполнения (неудачное создание файла, неудачный вызов unlink, неудачная попытка записи в созданный файл).

Создав файл, мы можем использовать его как “передатчик” вводимых координат от процесса к процессу, для этого нужно отобразить его на память. Применим mmap:

void \*mmap(void \*addr, size\_t len, int prot, int flags, int fildes, off\_t off) –

здесь:

- addr – адрес ячейки памяти, которая будет являться началом отображаемого в память файла; лучше всего использовать значение 0 – система сама определит нужный адрес;

- len – длина отображаемой на память информации, кратная размеру одной страницы виртуальной памяти (если введенная длина не кратна размеру страницы, то происходит округление, при этом, если значение len == 0, то mmap должен завершиться с ошибкой); лучше всего предварительно узнать размер страницы памяти через функцию getpagesize(), что применяется в нашей программе;

- prot – флаг “защиты”, позволяющий устанавливать желаемые условия доступа к отображенному файлу (PROT\_READ, PROT\_WRITE, PROT\_EXEC – доступ только к чтению, записи и выполнению соответственно); установленное значение должно быть таким же, как и значение, определённое при выполнении системного вызова open() для получения дескриптора;

- flags – различные флаги, используемые mmap; например, MAP\_SHARED – разрешение различным процессам вносить изменения в файл, (в противном случае - MAP\_PRIVATE, процессы получают копию отображенной области памяти, поэтому изменения отображаться не будут);

- fildes – дескриптор соответствующего файла;

- off – смещение позиции отображения в файле; данное значение обязательно должно быть кратным размеру страницы виртуальной памяти.

Также обработаем возможную ошибку MAP\_FAILED.

Так как изменение файла доступно для многих процессов, то мы получаем общий ресурс, поэтому процессы входят в состояние гонки. Мы должны предусмотреть это, введя и используя какой-либо примитив синхронизации между процессами. В данной ситуации, когда всего лишь два процесса должны поочередно использовать один ресурс, логичнее всего использовать семафор (можно и мюьтекс, но его рациональнее использовать с потоками). Введем два семафора sem1 и sem2 с заданным счетчиком 0, значение которых будет изменяться всего лишь на 1; сделаем это при помощи функции sem\_open:

sem\_t \*sem\_open(const char \*name, int oflag, mode\_t mode, unsigned int value) – здесь:

* name – имя семафора;
* oflag – флаги выполнения, флаг O\_CREAT обеспечивает создание семафора при его несуществовании;
* mode – соответствующее значение доступа для создания семафора (сочетание символьных констант для описания уровня доступа, см. системный вызов open()); 777 – полные права (все пользователи и группы могут читать, записывать и выполнять файл);
* value – начальное значение счетчика семафора.

И также обработаем возможные ошибки.

Теперь мы можем так же, как ранее, использовать fork(), породить дочерний процесс и для разных возвращаемых значений pid описать алгоритм. В родительском процессе, как прежде, будет приниматься координата, но теперь ее значение будет записываться не в pipe, а в массив mappedData; после записи значение счетчика семафора 1 при помощи sem\_post будет увеличено на единицу, что позволит родительскому процессу, до этого ожидающему освобождения ресурса, при помощи sem\_wait получить доступ к критической секции, считать координату из mappedData и увеличить значение второго семафора, открыв путь для дальнейшего выполнения родительского процесса – считывания результата и вывода на экран. Все это будет продолжаться, пока пользователь будет вводить два целочисленных значения координат данной точки.

Наконец, в конце при помощи munmap происходит высвобождение занимаемой памяти, закрытие семафоров и их уничтожение, а также закрытие созданного исходного файла.

***Листинг***

*main.c:*

#include <sys/types.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <sys/mman.h>

#include <pthread.h>

#include <string.h>

#include <fcntl.h>

#include <semaphore.h>

int getFile() {

char\* templateString = strdup("/tempXXXXXX");

int fd = mkstemp(templateString);

if (fd == -1) {

perror("File couldn't be created!");

exit(1);

}

if(unlink(templateString) == -1) {

perror("Couldn't unlink filename!");

exit(1);

}

free(templateString);

if(write(fd, "test", 4) == -1) {

perror("Couldn't test write in temp file!");

exit(1);

}

return fd;

}

int main()

{

int point[2] = {0};

char quarter;

int filedes = getFile();

size\_t pagesize = getpagesize();

int\* mappedData = (int\*)mmap(0, pagesize, PROT\_WRITE | PROT\_READ, MAP\_SHARED, filedes, 0);

if (mappedData == MAP\_FAILED) {

perror("Mmap failed!");

exit(1);

}

unsigned int count = 0;

sem\_t\* sem1 = sem\_open("sem1", O\_CREAT, 777, count);

sem\_t\* sem2 = sem\_open("sem2", O\_CREAT, 777, count);

if(sem1 == SEM\_FAILED) {

perror("Couldn't create semaphore1!");

exit(1);

}

if(sem2 == SEM\_FAILED) {

perror("Couldn't create semaphore2!");

exit(1);

}

pid\_t pid = fork();

if(pid > 0) {

printf("Enter the coordinates: ");

while(scanf("%d %d", &point[0], &point[1])) {

mappedData[0] = point[0];

mappedData[1] = point[1];

printf("parent process waiting...\n");

sem\_post(sem1);

//int c;

//sem\_getvalue(sem1, &c);

//printf("1 sem1 val %d\n", c);

//sem\_getvalue(sem2, &c);

//printf("2 sem2 val %d\n", c);

sem\_wait(sem2);

//printf("4\n");

quarter = mappedData[2];

printf("The answer is: %c\n", quarter);

printf("\nEnter the coordinates: ");

}

} else if(pid == 0) {

while(1) {

sem\_wait(sem1);

point[0] = mappedData[0];

point[1] = mappedData[1];

if(point[0] == 0 && point[1] == 0) {

printf("The point is the origin! Assigning quarter to 0...\n");

quarter = '0';

}

if(point[0] == 0 && point[1] != 0) {

printf("The point lays on the Oy axis! Assigning quarter to 0...\n");

quarter = '0';

}

if(point[0] != 0 && point[1] == 0) {

printf("The point lays on the Ox axis! Assigning quarter to 0...\n");

quarter = '0';

}

if(point[0] > 0 && point[1] > 0) {

quarter = '1';

}

if(point[0] < 0 && point[1] > 0) {

quarter = '2';

}

if(point[0] < 0 && point[1] < 0) {

quarter = '3';

}

if(point[0] > 0 && point[1] < 0) {

quarter = '4';

}

printf("child process writing...\n");

mappedData[2] = quarter;

int c;

sem\_getvalue(sem2, &c);

printf("3 sem2 val %d\n", c);

sem\_post(sem2);

}

} else if(pid == -1) {

perror("Unable to fork!");

sem\_close(sem1); sem\_close(sem2);

sem\_unlink("sem1"); sem\_unlink("sem2");

close(filedes);

exit(1);

}

if(munmap(mappedData, pagesize) == -1) {

perror("Unmapping failed!");

sem\_close(sem1); sem\_close(sem2);

sem\_unlink("sem1"); sem\_unlink("sem2");

close(filedes);

exit(1);

}

sem\_close(sem1); sem\_close(sem2);

sem\_unlink("sem1"); sem\_unlink("sem2");

close(filedes);

return 0;

}

***Демонстрация работы программы***

root@DESKTOP-5B8SU4Q:~/labs/3sem/os/4# gcc -g main.c -lpthread

root@DESKTOP-5B8SU4Q:~/labs/3sem/os/4# ./a.out

Enter the coordinates: 0 0

parent process waiting...

1 sem1 val 1

The point is the origin! Assigning quarter to 0...

2 sem2 val 0

child process writing...

3 sem2 val 0

4

The answer is: 0

Enter the coordinates: 0 3

parent process waiting...

1 sem1 val 1

The point lays on the Oy axis! Assigning quarter to 0...

2 sem2 val 0

child process writing...

3 sem2 val 0

4

The answer is: 0

Enter the coordinates: 123 0

parent process waiting...

1 sem1 val 1

The point lays on the Ox axis! Assigning quarter to 0...

2 sem2 val 0

child process writing...

3 sem2 val 0

4

The answer is: 0

Enter the coordinates: -342 -500

parent process waiting...

1 sem1 val 1

child process writing...

2 sem2 val 0

3 sem2 val 0

4

The answer is: 3

Enter the coordinates: 60000 1

parent process waiting...

1 sem1 val 1

child process writing...

2 sem2 val 0

3 sem2 val 0

4

The answer is: 1

Enter the coordinates: -352235 9

parent process waiting...

1 sem1 val 1

child process writing...

2 sem2 val 0

3 sem2 val 0

4

The answer is: 2

Enter the coordinates: quit

root@DESKTOP-5B8SU4Q:~/labs/3sem/os/4#

***Вывод***

В результате выполнения данной лабораторной работы я научился работать с отображением файлов на память при помощи системного вызова mmap. В процессе выполнения задачи возникла необходимость в синхронизации двух процессов, поэтому я также приобрел опыт по работе с примитивами синхронизации, в частности с семафорами.  
По принципу действия mmap напоминает malloc – в результате определенная область памяти, которую также необходимо освободить.  
Семафоры оказались удобными при реализации синхронизации родительского и дочернего процесса.

***Список литературы***

https://pubs.opengroup.org/onlinepubs/009695399/functions/mmap.html

http://beej.us/guide/bgipc/html/multi/mmap.html

http://man7.org/linux/man-pages/man2/open.2.html

http://man7.org/linux/man-pages/man3/sem\_open.3.html