Оглавление

лабораторная работа разделяемая память.....1
Теория 1
Чему нужно научиться 7
Задание.....8

ЛАБОРАТОРНАЯ РАБОТА РАЗДЕЛЯЕМАЯ ПАМЯТЬ

Теория

Обмен данными между процессами с использованием механизмов межпроцессорного взаимодействия каналов, FIFO и очередей сообщений может привести к падению производительности системы. Это, связано с тем, что данные, передаваемые с помощью этих механизмов, копируются из буфера передающего процесса в буфер ядра и затем в буфер принимающего процесса. Механизм разделяемой памяти позволяет избавиться от накладных расходов передачи данных через ядро, предоставляя процессам возможность получить доступ к одной области памяти для обмена данными.

Область разделяемой памяти – это некоторая часть физической памяти, которая используется совместно несколькими процессами. Процесс может присоединить эту область в качестве диапазона виртуальной памяти в свое адресное пространство. Диапазон может быть различным для каждого процесса (см Рис. 1. Разделяемая область памяти). После присоединения процесс получает возможность доступа к этой области как к любому другому участку

памяти, больше не нужно использовать системные вызовы для чтения/записи. Поэтому, механизм разделяемой памяти предоставляет процессу максимально быстрый способ доступа к данным. Если процесс записывает данные в ячейки разделяемой памяти, их измененное содержимое сразу же становится видимым остальным процессам, разделяющим между собой эту область.

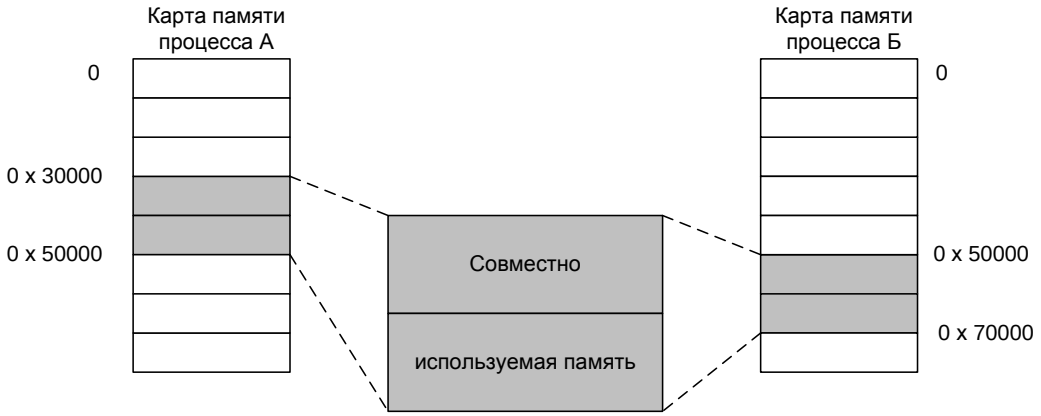

Рис. 1. Разделяемая область памяти

Таким образом, механизм разделяемой памяти является эффективным средством, позволяющим совместно использовать большие объемы данных без применения копирования и системных вызовов. Основным ограничением механизма является отсутствие средств синхронизации. Если два процесса пытаются изменить одну и ту же разделяемую область памяти ядро системы не может обеспечить последовательность этих операций, а это может привести к смешению записанных данных. Процессы, разделяющие между собой область памяти, должны самостоятельно поддерживать собственный протокол синхронизации. Когда один из процессов записывает данные в разделяемую память, остальные процессы должны ожидать завершения операции. Обычно для этой цели используются семафоры, назначение и число которых определяется

конкретным использованием разделяемой памяти. Правда использование семафоров требует вызова одной или нескольких системных функций, а это уменьшает производительность работы с разделяемой памятью.

Разделяемая память POSIX

Для отображения файла в память необходимо сначала его открыть при помощи ***open*** и затем вызвать ***mmap***. Рассмотрим системный вызов ***mmap***:

paddr=***mmap***(addr, len, prot, flags, fd, offset);

Результатом работы функции становится создание области с образом размером [offset, offset+len] файла fd в адресном пространстве [paddr, paddr+len] процесса. Параметр flag указывает на тип отображения и может принимать значения:

- MAP_SHARED
- MAP_PRIVATE.

Переменная prot устанавливается как любая комбинация из следующих возможных значений:

- PROT_READ;
- PROT_WRITE;
- PROT_EXECUTE.

Некоторые системы, не поддерживающие привилегии выполнения. Согласованное значение paddr выбирается системой. Оно не может быть равным нулю, а образ не вправе накладываться на уже существующие отображения. Вызов ***mmap*** игнорирует параметр addr, если не установлен флаг MAP_FIXED. В этом случае значение paddr должно совпадать с addr. Если значение addr окажется

неподходящим (например, не находится в диапазоне корректных адресов процесса), вызов завершится ошибкой.

Системный вызов ***mmap*** работает с целыми страницами памяти. Это означает, что параметр offset должен быть выровнен по величине страницы. При указании флага MAP_FIXED этому требованию должен соответствовать и параметр addr. Если значение len не совпадает с целым числом страниц, система сама произведет округление до следующего целого числа.

Отображение будет поддерживаться системой до тех пор, пока не будет выполнен вызов:

munmap(addr, len);

Может быть выполнено замещение адресного диапазона другим файлом при помощи вызова ***mmap*** с флагом MAP_RENAME. Для изменения признака защиты страницы используется системный вызов:

mprotect(addr, len, prot);

Разделяемая память System V

Для каждого сегмента разделяемой памяти ядро хранит следующую структуру, определенную в заголовочном файле <sys/shm.h>:

```
struct shmid_ds {
        struct ipc_perm shm_perm; // структура разрешений
        size_t shm_segsz; // размер разделяемой памяти
        pid_t shm_lpid;
        // идентификатор процесса, выполнившего последнюю операцию
```

pid_t shm_cpid; // идентификатор процесса создателя

shmatt_t shm_nattch; // текущее количество подключений

shmatt_t shm_cnattch; // количество подключений in-core

time_t shm_atime; // Время последнего подключения

time_t shm_dtime; // Время последнего отключения

time_t shm_ctime; // Время последнего изменения данной структуры

};

Для создания или для доступа к уже существующей разделяемой памяти используется системный вызов ***shmget*:**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflag);
```

Функция возвращает дескриптор разделяемой памяти в случае успеха, и –1 в случае неудачи. Аргумент size определяет размер создаваемой области памяти в байтах. Значения аргумента shmflag задают права доступа к объекту и специальные флаги IPC_CREAT и IPC_EXCL. Вызов ***shmget*** лишь создает или обеспечивает доступ к разделяемой памяти, но не позволяет работать с ней.

Для работы с разделяемой памятью (чтение/запись) необходимо сначала присоединить (attach) область вызовом ***shmat***:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

int ***shmat***(int shmid,char ∗shmaddr, int shmflag);

Вызов ***shmat*** возвращает адрес начала области в адресном пространстве процесса размером size, заданным предшествующим вызовом ***shmget***. В этом адресном пространстве взаимодействующие процессы могут размещать требуемые структуры данных для обмена информацией. Правила получения этого адреса следующие:

- Если аргумент shmaddr нулевой, то система самостоятельно выбирает адрес.
- Если аргумент shmaddr отличен от нуля, значение возвращаемого адреса зависит от наличия флажка SHM_RND в аргументе shmflag:
    - Если флажок SHM_RND не установлен, система присоединяет разделяемую память к указанному shmaddr адресу.
    - Если флажок SHM_RND установлен, система присоединяет разделяемую память к адресу, полученному округлением в меньшую сторону shmaddr до некоторой определенной величины SHMLBA.

По умолчанию разделяемая память присоединяется с правами на чтение и запись. Эти права можно изменить, указав флажок SHM_RDONLY в аргументе shmflag.

Таким образом, несколько процессов могут отображать область разделяемой памяти в различные участки собственного виртуального адресного пространства, как это показано на рисунке (Рис. 2 Отображение разделяемой памяти).

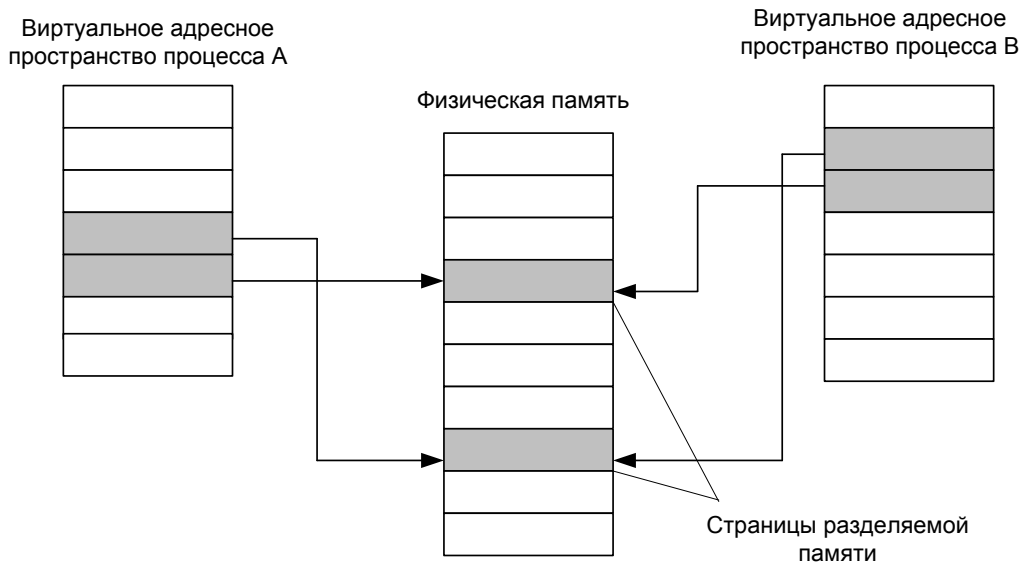

Рис. 2 Отображение разделяемой памяти

Окончив работу с разделяемой памятью, процесс отключает (detach) область вызовом ***shmdt***:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt(char *shmaddr);
```

Для полного удаления области процессу необходимо использовать системный вызов ***shmctl*** с IPC_RMID.

Чему нужно научиться

Изучить использование разделяемой памяти System V и Posix.

Задание

Уровень 1 (A)

Посмотрите на

Скелет кода для задания 1 (A), разберитесь и объясните, что там происходит.

Уровень 2 (A)

Посмотрите на Скелет кода для задания 2 (A), разберитесь и объясните, что там происходит.

Уровень 3 (A)

Разработайте наше любимое клиент серверное приложение с использованием разделяемой памяти:

- Клиент будет считывать полное имя файла из стандартного потока ввода и записывать его в разделяемую память;
- Сервер будет считывать это имя из разделяемой памяти и пытаться открыть файл с этим именем;
    - Если файл успешно откроется, то сервер будет передавать его содержимое в разделяемую память. В противном случае он запишет туда сообщение об ошибке.
- Клиент будет считывать данные из разделяемой памяти и записывать их в стандартный поток вывода.
    - Клиент либо выведет содержимое файла, либо сообщение об ошибке.
    -
    - Скелет кода для задания 1 (A)

shmem.h

```
#define MAXBUFF         80
#define PERM            0666
/*Структура данных в разделяемой памяти*/
typedef struct mem_msg{
        int segment;
        char buff[MAXBUFF];
} Message;
/*Ожидание начала выполнения клиента*/
static struct sembuf proc_wait[1] = {
        1, -1, 0};
/*Уведомление сервера о том, что клиент начал работу*/
static struct sembuf proc_start[1] = {
        1, 1, 2};
/*Блокировка разделяемой памяти*/
static struct sembuf mem_lock[2] = {
        0, 0, 0,
        0, 1, 0};
/*Освобождение ресурса*/
static struct sembuf mem_unlock[1] = {
        0, -1, 0};
```

Server.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
```

```
#include “shmem.h”
main()
{
Message      ∗msgptr;
key_t key;
int      shmid, semid;
/∗Получим ключ. Один и тот же ключ можно использовать как для семафора, так и для разделяемой памяти∗/
        if(( key = ftok(“server”, ‘A’))<0){
                printf(“Невозможно получить ключ\n”): exit(1);}
/∗Создадим область разделяемой памяти∗/
        if((shmid = shmget(key, sizeof(Message),
                                        PERM|IPC_CREAT))<0){
                printf(“Невозможно создать область\n”);
exit(1);}
/∗Присоединим ее∗/
        if((msgptr = (Message ∗)shmat(shmid, 0, 0))<0){
                printf(“Ошибка присоединения\n”); exit(1);}
/∗Создадим группу из двух семафоров:
Первый семафор – для синхронизации работы с разделяемой памятью
Второй семафор – для синхронизации выполнения процессов∗/
        if((semid = semget(key, 2, PERM|IPC_CREAT))<0){
                printf(“Невозможно создать семафор\n”);
exit(1);}
/∗Ждем, пока клиент начнет работу и заблокирует разделяемую память∗/
        if(semop(semid, &proc_wait[0], 1)<0){
```

```
                printf("Невозможно выполнить операцию\n");
exit(1);}
/*Ждем, пока клиент закончит запись в разделяемую память и совободит ее. После этого заблокируем ее*/
        if(semop(semid, &proc_lock[0], 2)<0){
                printf("Невозможно выполнить операцию\n");
exit(1);}
/*Выведем сообщение на терминал*/
        printf("%s", msgptr->buff);
/*Освободим разделяемую память*/
        if(shmdt(msgptr)<0){
                printf("Ошибка отключения\n"); exit(1);}
/*Всю остальную работу по удалению объектов сделает клиент*/
        exit(0);
}
```

Client.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include "shmem.h"
main()
{
Message      *msgptr;
key_t key;
int      shmid, semid;
```

```
/*Получим ключ. Один и тот же ключ можно использовать как для семафора, так и для разделяемой памяти*/
        if(( key = ftok("server", 'A'))<0){
                printf("Невозможно получить ключ\n"): exit(1);}
/*Получим доступ к разделяемой памяти*/
        if((shmid = shmget(key, sizeof(Message), 0))<0){
                printf("Ошибка доступа\n"); exit(1);}
/*Присоединим ее*/
        if((msgptr = (Message *)shmat(shmid, 0, 0))<0){
                printf("Ошибка присоединения\n"); exit(1);}
/*Получим доступ к семафору*/
        if((semid = semget(key, 2, PERM))<0){
                printf("Ошибка доступа\n"); exit(1);}
/*Заблокируем разделяемую память*/
        if(semop(semid, &proc_lock[0], 2)<0){
                printf("Невозможно выполнить операцию\n");
exit(1);}
/*Уведомим сервер о начале работы*/
        if(semop(semid, &proc_start[0], 1)<0){
                printf("Невозможно выполнить операцию\n");
exit(1);}
/*Запишем в разделяемую память сообщение*/
        sprintf(msgptr->buff, "Good luck!\n");
/*Освободим разделяемую память*/
        if(semop(semid, &proc_unlock[0], 1)<0){
                printf("Невозможно выполнить операцию\n");
exit(1);}
```

```
/*Ждем, пока сервер в свою очередь не освободит разделяемую память*/
        if(semop(semid, &proc_lock[0], 2)<0){
                printf("Невозможно выполнить операцию\n");
exit(1);}
/*Отключимся от области*/
        if(shmdt(msgptr)<0){
                printf("Ошибка отключения\n"); exit(1);}
/*Удалим созданные объекты IPC*/
        if(shmctl(shmid, IPC_RMID, 0)<0){
                printf("Невозможно удалить область\n");
                exit(1);}
        if(semctl(semid, 0, IPC_RMID)<0){
                printf("Невозможно удалить семафор\n");
                exit(1);}
        exit(0);
}
```

Скелет кода для задания 2 (А)

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <semaphore.h>

int main(int argc,char **argv)
{
        int fd,i,nloop;
```

```
        char *ptr;
        sem_t *mutex;
        nloop=10;

        if((fd=open("home/tvk/IPC/filedata",O_RDONLY))<0)
        {
                printf("Can not open file name\n");
                exit(1);
        }
        printf("File is opened!\n");
        ptr=mmap(NULL,sizeof(char),PROT_READ|PROT_WRITE,
                        MAP_SHARED,fd,0);
        close(fd);
        mutex=sem_open(px_ipc_name("mysem"),O_CREAT|O_EXCL,0
);
        sem_unlink(px_ipc_name("mysem"));
        setbuf(stdout,NULL);
        if(fork()==0) {
                for(i=0;i<nloop;i++)
                {
                        sem_wait(mutex);
                        printf("Child: %c\n",(*ptr)++);
                        sem_post(mutex);
                }
                exit(0);
                        }
        for(i=0;i<nloop;i++)
        {
                sem_wait(mutex);
```

```
            printf("Parent: %c\n",(*ptr)++);
            sem_post(mutex);
        }

        exit(0);
}
```