# Лабораторная работа 8. РАЗДЕЛЯЕМАЯ ПАМЯТЬ (*Shared memory*), СИНХРОНИЗАЦИЯ

**Цель**. Освоить механизм взаимодействия между процессами на основе использования общей памяти.

# Краткие теоретические сведения.

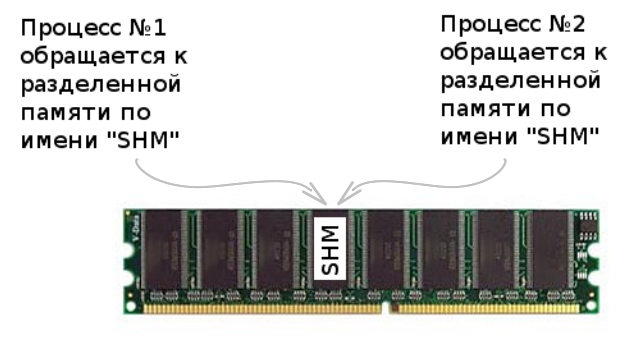
**Перечень рассматриваемых вопросов**

* Выделение совместно используемой (общей) памяти.
* Активация и отключение совместного доступа к сегментам памяти.
* Контроль над общей памятью.
* Реализация семафоров.
* Управление семафорами.

Интенсивный обмен данными между процессами с использованием каналов, FIFO, сокетов, очереди сообщений, может вызвать падение производительности системы. Это связано с тем, что данные, передаваемые с помощью этих объектов, копируются из буфера передающего процесса в буфер ядра и затем в буфер принимающего процесса.

# Механизм разделяемой памяти

Механизм разделяемой памяти позволяет избавиться от накладных расходов передачи данных через ядро, предоставляя двум или более процессам возможность непосредственного получения доступа к одной области памяти для обмена данными.



При этом процессы должны предварительно "договориться" о правилах использования разделяемой памяти. Сначала один из процессов выделяет некоторый объем памяти, который называется **сегментом**. После выделения совместно используемого сегмента другие процессы могут задействовать его. Эта операция называется **подключением сегмента**.

Например, пока один из процессов производит запись данных в разделяемую память, другие процессы должны воздержаться от работы с ней.

Такое *кооперативное* использование разделяемой памяти называется, ***синхронизации* выполнения процессов**. **Синхронизация** разделяет среди двух или более процессов (потоков) **общий ресурс**, который может быть *одновременно доступен только одному потоку*.

**Системные вызовы** для работы с разделяемой памятью объявлены в заголовочном файле **sys/shm.h** .

**Получение совместного доступа**

Процесс может получить доступ к созданному сегменту двумя способами:

1. Использовать идентификатор сегмента, полученный от другого процесса.
2. Вызвать shmget(), применяя для этого заранее известный ключ.

Получить информацию о совместно используемых сегментах памяти и их идентификаторах можно с помощью программы **ipcs**.

Выделение общей памяти осуществляется при помощи системного вызова shmget()

int **shmget** (key\_t KEY, size\_t SIZE, int FLAGS);

Этот системный вызов возвращает идентификатор сегмента или –1, если произошла ошибка.

**Активизация совместного доступа**

Для работы с общим сегментом памяти нужно, чтобы каждый из взаимодействующих процессов обратился к системному вызову shmat().

void \* **shmat** (int ID, void \* ADDRESS, int FLAGS);

Данный системный вызов возвращает адрес совместно используемого сегмента памяти. В случае ошибки системный вызов shmat() возвращает указатель (void \*) -1.

**Отключение совместного доступа**

Системный вызов shmdt() вызывается в каждом процессе после того, как тот закончил работу с общей памятью.

int **shmdt** (void \* ADDRESS);

При успешном завершении shmdt() возвращает 0, в случае ошибки возвращается –1.

**Управление использованием памяти**

Системный вызов shmctl() позволяет осуществлять различные операции над общим сегментом памяти.

int **shmctl** (int ID, int COMMAND, struct shmid\_ds \* DESC);

При успешном завершении shmctl() возвращает 0. В случае ошибки возвращается –1.

Для понимания работы механизма разделяемой памяти проделайте следующие упражнения.

## Упражнение 1

Простейший пример межпроцессного взаимодействия на основе совместно используемых сегментов памяти. Сначала создадим процесс, который создает общий сегмент и открывает доступ к нему для всех желающих.

**Программа shm1-owner.c**

#include <stdio.h>

#include <string.h>

#include <sys/shm.h>

#define SHMEM\_SIZE 4096

#define SH\_MESSAGE "Hello World!\n"

int **main** (void)

{

int shm\_id;

char \* shm\_buf;

int shm\_size;

struct shmid\_ds ds;

shm\_id = shmget (IPC\_PRIVATE, SHMEM\_SIZE, IPC\_CREAT | IPC\_EXCL | 0600);

if (shm\_id == -1) {

fprintf (stderr, "shmget() error\n");

return 1;

}

shm\_buf = (char \*) shmat (shm\_id, NULL, 0);

if (shm\_buf == (char \*) -1) {

fprintf (stderr, "shmat() error\n");

return 1;

}

shmctl (shm\_id, IPC\_STAT, &ds);

shm\_size = ds.shm\_segsz;

if (shm\_size < strlen (SH\_MESSAGE)) {

fprintf (stderr, "error: segsize=%d\n", shm\_size);

return 1;

}

strcpy (shm\_buf, SH\_MESSAGE);

printf ("ID: %d\n", shm\_id);

printf ("Press <Enter> to exit...");

fgetc (stdin);

shmdt (shm\_buf);

shmctl (shm\_id, IPC\_RMID, NULL);

return 0;

}

Объясните принцип работы данной программы.

Программа, которая читает данные из совместно используемого сегмента.

**Программа shm1-user.c**

#include <sys/shm.h>

#include <stdio.h>

int main (int argc, char \*\* argv)

{

int shm\_id;

char \* shm\_buf;

if (argc < 2) {

fprintf (stderr, "Too few arguments\n");

return 1;

}

shm\_id = atoi (argv[1]);

shm\_buf = (char \*) shmat (shm\_id, 0, 0);

if (shm\_buf == (char \*) -1) {

fprintf (stderr, "shmat() error\n");

return 1;

}

printf ("Message: %s\n", shm\_buf);

shmdt (shm\_buf);

return 0;

}

Поскольку процессы "не договорились" заранее о выделении общего ключа для сегмента памяти, то единственным адекватным способом взаимодействия будет непосредственная передача программе-клиенту идентификатора сегмента через аргумент.

Для проверки работы этой программы желательно иметь два терминальных окна. В первом окне запускаем процесс-сервер:

$ gcc -o shm1-owner shm1-owner.c

$ ./shm1-owner

Теперь, не нажимая клавиши <Enter>, переходим в другое терминальное окно и запускаем процесс-клиент:

$ gcc -o shm1-user shm1-user.c

$ ./shm1-user 31391750

Но не торопитесь переходить в исходное окно и нажимать клавишу <Enter>. **Эксперимент еще не закончен**! Наберите в клиентском окне команду **ipcs -m**:

$ ipcs -m

Эта команда выводит на экран список совместно используемых сегментов памяти с их ключами и идентификаторами. В приведенном примере последняя строка содержит сведения о нашем сегменте. Теперь нажмите клавишу <Enter> в исходном терминале и вызовите ipcs -m еще раз:

$ ipcs -m

Убедитесь, что сегмента больше нет. Но обратите внимание на то, что если программа завершится, не вызвав shmctl() с командой IPC\_RMID, то сегмент продолжит существовать в памяти. Однако пользователь, которому принадлежит сегмент (см. колонку owner), может самостоятельно удалить его при помощи команды **ipcrm**. (**Проведите эксперимент**).

## Упражнение 2

Объясните принцип работы данной программы. Результаты выполнения подтвердите *скриншотами*.

# Семафоры

Семафоры позволяют процессам задействовать общие ресурсы организованно, не допуская конфликтов. Типичный пример — организация межпроцессного взаимодействия с использованием общей памяти. **Синхронизации** выполнения процессов, легко решается с помощью семафоров.

В предыдущих примерах нам приходилось изобретать собственные не слишком надежные средства контроля доступа к общему сегменту памяти. В большинстве случаев приложения требуют **более надежных подходов** к решению данной задачи.

Предположим, что два процесса делят какой-нибудь общий ресурс. Чтобы не произошло конфликтов, процессы договариваются об использовании дополнительного разделяемого ресурса — семафора. Каждый из процессов перед доступом к общему ресурсу пытается установить для себя семафор (занять место). Если семафор уже установлен кем-то другим, то процесс блокируется до тех пор, пока другой процесс не освободит ресурс. Это самая простая схема взаимодействия.

Семафоры в Linux представляют собой очень гибкий механизм, который может контролировать доступ к совместным ресурсам именно так, как нужно вам.

Примерный сценарий работы с разделяемой памятью с использованием семафоров:

1. Сервер получает доступ к разделяемой памяти, используя семафор.
2. Сервер производит запись данных в разделяемую память.
3. После завершения записи сервер освобождает разделяемую память с помощью семафора.
4. Клиент получает доступ к разделяемой памяти, запирая ресурс с помощью семафора.
5. Клиент производит чтение данных из разделяемой памяти и освобождает ее, используя семафор.

Для каждой области разделяемой памяти, ядро поддерживает структуру данных shmid\_ds.

Применение семафоров в Linux осуществляется на основе следующих механизмов, объявленных в заголовочном файле sys/sem.h:

**struct sembuf** {

unsigned short int sem\_num;

short int sem\_op;

short int sem\_flg;

};

**union semnum** { // объединение

int val;

struct semid\_ds \* buf;

unsigned short \* array;

};

int **semget** (key\_t KEY, int SEMS, int FLAGS);

int **semop** (int ID, struct **sembuf** \* SB, size\_t SIZE);

int **semctl** (int ID, int SNUM, int COMMAND, ...);

Дополнительные возможности структуры sembuf можно изучить на man-странице системного вызова semop().

**Управление семафорами**

Администрирование семафоров осуществляется при помощи системного вызова semctl()

int **semctl** (int ID, int SNUM, int COMMAND, ...);

Этот системный вызов обычно задает две команды: инициализацию и удаление семафоров.

## Упражнение 3

Пример использования программами сегментов разделяемой памяти с применением семафора. Напишем две программы **shm2-owner.c** и **shm2-user.c**.

**Программа shm2-owner.c**

#include <stdio.h>

#include <string.h>

#include <sys/shm.h>

#include <sys/sem.h>

#define SHMEM\_SIZE 4096

#define SH\_MESSAGE "Hello World!\n"

#define SEM\_KEY 2007

#define SHM\_KEY 2007

union semnum {

int val;

struct semid\_ds \* buf;

unsigned short \* array;

} sem\_arg;

int **main** (void)

{

int shm\_id, sem\_id;

char \* shm\_buf;

int shm\_size;

struct shmid\_ds ds;

struct sembuf sb[1];

unsigned short sem\_vals[1];

shm\_id = shmget (SHM\_KEY, SHMEM\_SIZE, IPC\_CREAT | IPC\_EXCL | 0600);

if (shm\_id == -1) {

fprintf (stderr, "shmget() error\n");

return 1;

}

sem\_id = semget (SEM\_KEY, 1, 0600 | IPC\_CREAT | IPC\_EXCL);

if (sem\_id == -1) {

fprintf (stderr, "semget() error\n");

return 1;

}

printf ("Semaphore: %d\n", sem\_id);

sem\_vals[0] = 1;

sem\_arg.array = sem\_vals;

if (semctl (sem\_id, 0, SETALL, sem\_arg) == -1) {

fprintf (stderr, "semctl() error\n");

return 1;

}

shm\_buf = (char \*) shmat (shm\_id, NULL, 0);

if (shm\_buf == (char \*) -1) {

fprintf (stderr, "shmat() error\n");

return 1;

}

shmctl (shm\_id, IPC\_STAT, &ds);

shm\_size = ds.shm\_segsz;

if (shm\_size < strlen (SH\_MESSAGE)) {

fprintf (stderr, "error: segsize=%d\n", shm\_size);

return 1;

}

strcpy (shm\_buf, SH\_MESSAGE);

printf ("ID: %d\n", shm\_id);

sb[0].sem\_num = 0;

sb[0].sem\_flg = SEM\_UNDO;

sb[0].sem\_op = -1;

semop (sem\_id, sb, 1);

sb[0].sem\_op = -1;

semop (sem\_id, sb, 1);

semctl (sem\_id, 1, IPC\_RMID, sem\_arg);

shmdt (shm\_buf);

shmctl (shm\_id, IPC\_RMID, NULL);

return 0;

}

**Программа shm2-user.c**

#include <sys/shm.h>

#include <stdio.h>

#include <sys/sem.h>

#define SEM\_KEY 2007

#define SHM\_KEY 2007

int **main** (int argc, char \*\* argv)

{

int shm\_id, sem\_id;

char \* shm\_buf;

struct sembuf sb[1];

shm\_id = shmget (SHM\_KEY, 1, 0600);

if (shm\_id == -1) {

fprintf (stderr, "shmget() error\n");

return 1;

}

sem\_id = semget (SEM\_KEY, 1, 0600);

if (sem\_id == -1) {

fprintf (stderr, "semget() error\n");

return 1;

}

shm\_buf = (char \*) shmat (shm\_id, 0, 0);

if (shm\_buf == (char \*) -1) {

fprintf (stderr, "shmat() error\n");

return 1;

}

printf ("Message: %s\n", shm\_buf);

sb[0].sem\_num = 0;

sb[0].sem\_flg = SEM\_UNDO;

sb[0].sem\_op = 1;

semop (sem\_id, sb, 1);

shmdt (shm\_buf);

return 0;

}

Объясните принцип работы данной программы.

# Задания

## Задание №1

Используя листинги упражнения №1 создайте еще несколько программ **shm1-ownerХХ.c** (количество программ «ownerХХ» должно соответствовать количеству членов команды). Программы должны выводить в сообщениях имена членов команды. Перестройте коды программ так, чтобы процессы обращались к одному сегменту памяти.

Подсказка, подробно изучите системный вызов **shmget()** и его атрибуты.

Поэкспериментируйте с передачей сообщений между процессами.

## Задание №2

Использования условия задания №1, создайте несколько программ **shm1-ownerХХ.c**, с использованием одного семафора.

1. Подключить к *общей памяти* двух клиентов (один из них выполняется как поток).

2. Создать два *объекта общие памяти*. Первый процесс пишет в первый *объект* и читает из *второго объекта памяти*, второй процесс, наоборот, читает из первого объекта и пишет во второй (программа-чат).

3. Сделать два сервера сообщений и одного клиента. Серверы используют одну и ту же общую память. Клиент читает и определяет, от какого сервера пришло сообщение.

*Выполните упражнения, объясните принцип работы программы, ответьте на контрольные вопросы.*

# Контрольные вопросы

1. Что такое **разделяемая память**?
2. Что такое **сегмент** памяти, и его атрибуты?
3. Что такое **синхронизация процессов**?
4. Какие системные вызовы используются для работы с разделяемой памятью? Какие атрибуты они имеют, какие значения возвращают? Какие библиотеки необходимы для их подключения,
5. Что такое **семафоры**, для чего они используются?
6. Какие системные вызовы используются для работы с семафорами? Какие атрибуты они имеют, какие значения возвращают?
7. Опишите примерный сценарий работы с разделяемой памятью с использованием семафоров.
8. Что Вы знаете о синхронизации и семафорах в языках более высокого уровня **C#** или **Java**?

## Дополнительная информация

# Литературные и интернет-источники

Робачевский А. М. Операционная система UNIX®. - СПб.: 2002. - 528 ил.

Иванов Н. Н. И20 Программирование в Linux. Самоучитель. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2012. — 400 с.: ил.