# Семафоры

Для синхронизации процессов, в смысле синхронизации доступа нескольких процессов к разделяемым ресурсам, используются семафоры.
Являясь одной из форм IPC, семафоры не предназначены для обмена большими объемами данных, как в случае FIFO или очередей сообщений.
Вместо этого, они выполняют функцию, полностью соответствующую своему названию — разрешать или запрещать процессу использование того или иного разделяемого ресурса.

Например, допустим, имеется разделяемый ресурс в виде файла. Необходимо в течение времени, когда некий процесс производит операцию над ресурсом (например, записывает в файл) блокировать доступ к ресурсу для других процессов.
Для этого свяжем с данным ресурсом некую целочисленную величину — счетчик, доступный для всех процессов.
Примем, что значение счетчика 1 означает доступность ресурса, значение 0 — его недоступность.
Тогда перед началом работы с ресурсом процесс должен проверить значение счетчика.
Если оно равно 0 — ресурс занят и операция недопустима — процессу остается ждать.
Если значение счетчика равно 1 — можно работать с ресурсом.
Для этого, прежде всего, необходимо заблокировать ресурс, т. е. изменить значение счетчика на 0.

После выполнения процессом операции необходимо освободить ресурс, а для этого значение счетчика необходимо изменить на 1.
В приведенном примере счетчик играет роль семафора.

Для нормальной работы необходимо обеспечить выполнение следующих условий :

1. Значение семафора должно быть доступно различным процессам.

Поэтому семафор находится не в адресном пространстве процесса, а в адресном пространстве ядра.

2. Операция проверки и изменения значения семафора должна быть реализована в виде одной *атомарной* по отношению к другим процессам (т.е. непрерываемой другими процессами) операции.

В противном случае возможна ситуация, когда после проверки значения семафора выполнение процесса будет прервано другим процессом, который в свою очередь проверит семафор и изменит его значение.
Единственным способом гарантировать атомарность критических участков операций является выполнение этих операций в режиме ядра.
Таким образом, семафоры являются системным ресурсом, действия над которым производятся через интерфейс системных вызовов.

Семафоры, входящие в состав средств IPC Linux, обладают следующими характеристиками:

- Семафор представляет собой не один счетчик, а группу, состоящую из нескольких счетчиков, объединенных общими признаками (дескриптором объекта, правами доступа и т.д.)
- Каждое из этих чисел может принимать любое неотрицательное значение в пределах, определенных системой (а не только значения 0 и 1).

Для каждой группы семафоров ядро поддерживает структуру данных *semid_ds , в*ключающую следующие поля:

| | |
|---|---|
| *struct ipc_perm sem_perm* | Описание прав доступа |
| *struct sem *sem_base* | Указатель на первый элемент массива семафоров |
| *ushort sem_nsems* | Число семафоров в группе |
| *time_t sem_otime* | Время последней операции |
| *time_t sem_ctime* | Время последнего изменения |

Значение конкретного семафора из набора хранится во внутренней структуре *sem ,* включающей следующие поля:

| | | |
|---|---|---|
| *ushort* | *semval* | Значение семафора |
| *pid_t* | *sempid* | Идентификатор процесса, выполнившего последнюю операцию над семафором |
| *ushort* | *semncnt* | Число процессов, ожидающих увеличения значения семафора |
| *ushort* | *semzcnt* | Число процессов, ожидающих обнуления семафора |

Помимо собственно значения семафора, в структуре *sem* хранится идентификатор процесса, вызвавшего последнюю операцию над семафором, число процессов, ожидающих увеличения значения семафора, и число процессов, ожидающих, когда значение семафора станет равным нулю.
Эта информация позволяет ядру производить операции над семафорами.

## Создание и получение доступа

Для получения доступа к семафору (и для его создания, если он еще не существует) используется системный вызов *semget()*.

```
#include <sys/itypes.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key_t key, int nsems, int semflags);
```

В случае успешного завершения операции функция *semget()* возвращает дескриптор объекта,
в случае неудачи - 1.

Аргумент *nsems* задает число семафоров в группе. Часто это единица.
В случае, когда мы не создаем, а лишь получаем доступ к существующему семафору, этот аргумент игнорируется.
Третий аргумент — *semflags* определяет права доступа к семафору, а также флаги:
- для его создания или получения доступа к существующему (IPC_CREAT),
- для создания нового семафора без возможности получения доступа к существующему (IPC_EXCL),
- для создания ресурса специально для данного процесса (IPC_PRIVATE , в этом случае ключ не требуется).

Пример программы, создающей три набора из 3-х семафоров но с *разными флагами*:

```
/* Программа gener_sem.cpp */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include<sys/ipc.h>
#include <sys/sem.h>
#include<sys/types.h>

main(void)
{
int     sem1, sem2, sem3;
key_t  ipc_key;
ipc_key = ftok(".", `S`);

if ((sem1=semget(ipc_key, 3, IPC_CREAT | 0666)) == -1){
        perror("semget: IPC_CREAT | 0666");

        }
printf("sem1 identifier is %d\n", sem1);

if((sem2 = semget(ipc_key, 3, IPC_CREATE | IPC_EXCL | 0666)) == -1) {
        perror("semget: IPC_CREATE | IPC_EXCL | 0666");

        }
printf("sem2 identifier is %d\n", sem2);

if((sem3 = semget(IPC_PRIVATE, 3, 0600)) == -1) {
        perror("semget: IPC_PRIVATE");

        }
printf("sem3 identifier is %d\n", sem3);
exit(0);
}
```

При запуске данной программы многократно будет видно, что

- набор *sem1* будет создан единожды, а затем каждая новая попытка будет всего лишь открывать доступ к уже существующему ресурсу;
- попытки создания набора *sem2* на том же ключе всегда будут приводить к ошибке из-за наличия флагов *IPC_CREATE* | *IPC_EXCL,* не допускающих открытия ресурса вместо его создания;
- набор *sem3* будет создаваться при каждом новом запуске программы. Причем каждый раз с новым уникальным идентификатором.

Просмотреть список созданных семафором и *прав доступа* к ним можно также и командой *ipcs*.

## Операции над семафорами

После получения *дескри*птора объекта процесс может производить операции над семафором,
(подобно тому, как после получения файлового дескриптора процесс может читать и записывать данные в файл).
Для этого используется системный вызов *semop() :*

```
#include <sys/itypes.h>
#include <sys/ipc.h>
#include<sys/sem.h>

int semop (int semid, struct sembuf *semop, size_t nops);
```

Первый параметр — это идентификатор (дескриптор) объекта, возвращенный предыдущим вызовом *semget().*

В качестве второго аргумента функции *semop()* передается указатель на структуру данных, определяющую операции, которые требуется произвести над семафором с дескриптором *semid*.

Операций может быть несколько, и их число указывается в последнем аргументе *nops*. Важно, что ядро обеспечивает
*атомарность выполнения критических участков операций*
(например, *проверка значения семафора — изменение значения семафора*) по отношению к другим процессам.
Поля структуры *sembuf*, определяющей операции имеют вид:

```
struct sembuf {
        short sem_num;        /* номер семафора в группе */
        short sem_op;         /* операция */
        short sem_flg;        /* флаги операции */
        };
```

Поле *sem_num* - номер семафора в группе, обычно нулевое значение.

Поле *sem_op* - определяет три возможные операции над семафором *:*

1. если величина *semop* положительна, то текущее значение семафора увеличивается на эту величину;
2. если значение *semop* равно нулю, процесс ожидает, пока семафор не обнулится (т.е. происходит проверка значения семафора);
3. если величина *semop* отрицательна, процесс ожидает, пока значение семафора не станет большим или равным абсолютной величине *semop*, а затем абсолютная величина *semop* вычитается из значения семафора);

При работе с семафорами взаимодействующие процессы должны договориться об их использовании и кооперативно проводить операции над семафорами. Операционная система не накладывает ограничений на использование семафоров.
В частности, процессы вольны решать,
какое значение семафора является *разрешающим*,
на какую величину изменяется значение
семафора и т. п.

Таким образом, при работе с семафорами процессы используют различные комбинации из трех операций, определенных системой, по своему трактуя значения семафоров.

Поле *sem_flg* - флаг, обычно установленный в SEM_UNDO , если процесс завершается без освобождения семафора (операция остается блокированной), то это позволяет семафору сброситься автоматически (отменить операцию).

**Примеры использования бинарного семафора**
Бинарный семафор - это такой семафор значения которого могут принимать только 0 и 1.

В первом примере значение 0 является разрешающим,
а 1 запирает некоторый разделяемый ресурс
(файл, разделяемая память и т.п.),
ассоциированный с семафором.

Определим операции, запирающие ресурс и освобождающие его:

```
static struct sembuf sop_lock[2] = {
        0, 0, 0,    /* ожидать обнуления семафора */
        0, 1, 0     /* затем увеличить значение семафора на 1 */
};
static struct sembuf sop_unlock[1] = {
        0, -1, 0    /* обнулить значение семафора */
};
```

Итак для *запирания ресурса* процесс производит вызов

*semop (semid, &sop_lock[0], 2);*

обеспечивающий атомарное выполнение двух операций:

- Ожидание доступности ресурса. В случае, если ресурс уже занят (значение семафора равно 1), выполнение процесса будет приостановлено до освобождения ресурса (значение семафора равно 0).
- Запирание ресурса. Значение семафора устанавливается равным 1.

Для *освобождения ресурса* процесс должен произвести вызов:

*semop (semid, &sop_unlock[0], 1);*

который уменьшит текущее значение семафора (равное 1) на 1
и оно станет равным 0,
что соответствует освобождению ресурса.
Если какой-либо из процессов ожидает ресурса
(т. е. Произвел ранее вызов операции *sop_lock*),
он будет "разбужен" системой, и сможет, в свою очередь,
запереть ресурс и работать с ним.

Во втором примере изменим трактовку значений семафора:
значению 1 семафора соответствует доступность
некоторого ассоциированного с семафором ресурса,
а нулевому значению — недоступность.
В этом случае содержание операций несколько изменится.

```
static struct sembuf sop_lock[2] = {
        0, -1, 0,       /* ожидать разрешающего сигнала (1),
                                затем обнулить семафор */
};
static struct sembuf sop_unlock[1] = {
        0, 1, 0         /* увеличить значение семафора на 1 */
};
```

Процесс запирает ресурс вызовом:

*semop (semid, &sop_lock[0], 1);*

а освобождает:

*semop (semid, &sop_unlock[0], 1);*

Во втором случае операции получаются проще (по крайней мере их код компактнее), однако этот подход имеет потенциальную опасность:
при создании семафора, его значения устанавливаются равными 0,
и во втором случае он сразу же запирает ресурс.

Для преодоления данной ситуации процесс, первым создавший семафор, должен вызвать операцию *sop_unlock*
однако в этом случае процесс инициализации семафора
перестанет быть атомарным и может быть прерван другим процессом, который, в свою очередь, изменит значение семафора.
В итоге, значение семафора станет равным 2, что повредит нормальной работе с разделяемым ресурсом.

Можно предложить следующее решение данной проблемы:

```
/* Создаем семафор, если он уже существует semget() возвращает
                        ошибку, поскольку указан флаг  IPC_EXCL */

if ( (semid = semget( key, nsems, perms | IPC_CREAT | IPC_EXCL )) < 0)

        if  (errno = EEXIST)
        {
        /* Действительно, ошибка вызвана существованием объекта */
                if ((semid = semget(key, nsems, perms ) ) < 0)
                        return(-1); /* Возможно не хватает системных ресурсов */
        }
        else
                return(-1); /* Возможно не хватает системных ресурсов */
}
/* Если семафор создан нами, проинициализируем его */
else
        semop (semid, &sop_unlock[0], 1);
```

# Разделяемая память

Обмен через разделяемую память (*Shared Memory*) наиболее быстрый (но не самый простой по организации) способ обмена данными между независимыми процессами.

Интенсивный обмен данными между процессами с использованием рассмотренных ранее других механизмов межпроцессного взаимодействия (каналы, FIFO, очереди сообщений) может вызвать падение производительности системы.
Это, в первую очередь, 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 позволяет избавиться от накладных расходов передачи данных через ядро, предоставляя двум и более процессам возможность непосредственного получения доступа к одной области памяти для обмена данными.

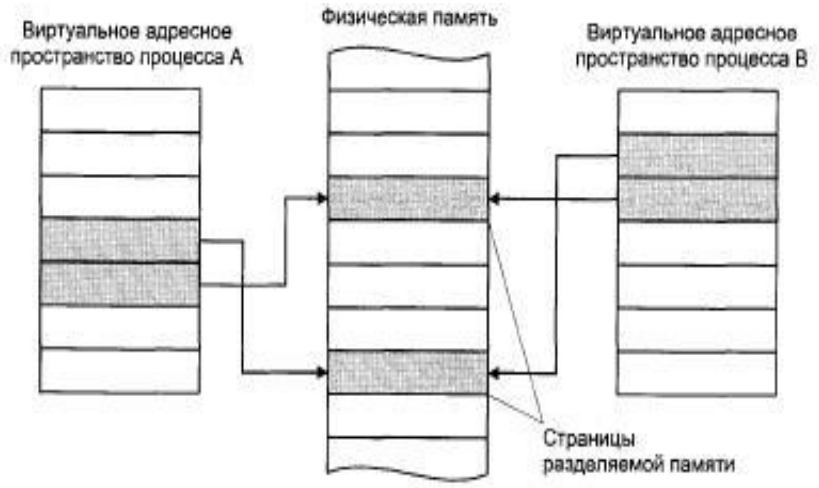
Безусловно, процессы должны предварительно "договориться" о правилах использования разделяемой памяти.
Например, пока один из процессов производит запись данных в разделяемую память, другие процессы должны воздержаться от работы с ней. Задача кооперативного использования разделяемой памяти, заключающаяся в синхронизации выполнения процессов, легко решается с помощью семафоров.

Примерный сценарий работы с разделяемой памятью может выглядеть таким образом:

1. Сервер создает сегмент разделяемой (общей) памяти, задавая его размер и права доступа к нему. Затем присоединяет этот сегмент к своей, выделенной операционной системой, памяти, отображая его при этом на область данных.
2. Сервер получает доступ к разделяемой памяти, используя семафор, производит запись данных в разделяемую память.
После завершения записи сервер освобождает разделяемую память с помощью семафора.
4. Другой процесс, (клиент) получающий права доступа к сегменту, может также отобразить его на свою область данных. Клиент при получении доступа к разделяемой памяти, запирает ресурс с помощью семафора.
5. Клиент производит чтение данных из разделяемой памяти и освобождает ее, используя семафор.

Когда все процессы заканчивают с коммуникациями через сегмент общей памяти, процесс владелец обычно отвечает за удаление этого сегмента. Процесс владелец (создавший сегмент общей памяти) может делегировать свои полномочия другому процессу.

Таким образом, несколько процессов могут отображать область разделяемой памяти в различные участки собственного виртуального адресного пространства:




## Создание сегмента

Для каждой создаваемой области разделяемой памяти, ядро поддерживает структуру данных *shmid_ds (определена в <sys/shm.h>),*
основными полями которой являются:

| | | |
|---|---|---|
| *struct ipc_perm* | *shm_perm* | *- Права доступа, владельца области* |
| *int* | *shm_segsz* | *- Размер выделяемой памяти* |
| *ushort* | *shm_nattch* | *- Число процессов, использующих разделяемую память* |
| *time_t* | *shm_atime* | *- Время последнего присоединения к разделяемой памяти* |
| *time_t* | *shm_dtime* | *- Время последнего отключения от разделяемой памяти* |
| *time_t* | *shm_ctime* | *- Время последнего изменения* |

Системный вызов *shmget()* используется для создания разделяемого сегмента памяти и генерации ассоциированной с ним системной структуры данных *shmid_ds* или для получения доступа к уже существующему сегменту.
Системный вызов *shmget()* возвращает уникальный идентификатор (дескриптор) разделяемого сегмента, а в случае неудачи -1.

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

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

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*

При условии, что запрос не превышает параметров системы, вызов *shmget()* создаёт *новый* разделяемый сегмент памяти, если:

- значением аргумента *key* является символьная константа *IPC_PRIVATE*;
- параметр *key* не ассоциирован ни с каким идентификатором существующей разделяемой памяти, а в аргументе *shmflag* установлен флаг *IPC_CREAT* (в противном случае, будет возвращён идентификатор существующей разделяемой памяти, ассоциированной с *key*);
- значение параметра *key* не ассоциировано с идентификатором существующей разделяемой памяти и флаги *IPC_CREAT*, *IPC_EXCL* установлены. Установление флагов *IPC_CREAT* и *IPC_EXCL* гарантирует создание уникального разделяемого сегмента памяти, а не получение доступа к существующему сегменту.

В программе *gener_shma* иллюстрируется процедура создания двух сегментов (разного размера) разделяемой памяти.

```
/* Программа gener_shma.cpp */
/* Отведение сегмента разделяемой памяти */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

main(void)
{
key_t key=15;
int    shmid_1, shmid_2;

if ((shmid_1=shmget(key, 1000, 0644|IPC_CREAT)) == -1){
   perror("shmget shmid_1");
   exit(1);
   }

printf("First memory identifire is %d \n", shmid_1);
   if((shmid_2 = shmget(IPC_PRIVATE, 20, 0644)) == -1) {
       perror("shmget shmid_2");
       exit(2);
       }
printf("Second shared memory identifire is %d \n", shmid_2);
exit(0);
}
```

В случае неуспешного выполнения, *shmget()* возвращает -1 и устанавливает переменную *errno* равную коду ошибки.
Примеры кодов ошибок:

EONT - идентификатор разделяемой памяти не существует для данного *key* и *IPC_CREAT*не установлен;

EACC - идентификатор разделяемой памяти существует для данного *key*, но данная операция запрещена текущими правами доступа;

ENOMEM - недостаточно памяти для выполнения операции;

EEXIST - идентификатор разделяемой памяти существует для данного *key*, но установлены флаги I*PC_CREAT* и *IPC_EXCL*;

EINVAL - неверный размер сегмента;

ENOSPC - достигнут предел числа разделяемых сегментов системы.

Ограничения для сегмента разделяемой памяти :

| *Параметр* | *Значение* |
|---|---|
| Максимальный размер сегмента | 1,048,576 байт |
| Минимальный размер сегмента | 1 байт |
| Максимальное число сегментов в системе | 100 |
| Максимальное число сегментов, связанных с процессом | 6 |

Заметим, что вызов *shmget()*
лишь создает новую область разделяемой памяти или обеспечивает доступ к уже существующей разделяемой памяти, но
не позволяет работать с ней.

## Присоединение и отсоединение

Для работы с разделяемой памятью (чтение и запись)
необходимо сначала присоединить (a*ttach*) область вызовом *shmat():*

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

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

Вызов *shmat()* возвращает *адрес начала области* в пространстве процесса размером *size ,* заданном в предшествующем вызовом *shmget() .*

В этом адресном пространстве взаимодействующие процессы могут размещать требуемые структуры данных для обмена информацией.

Правила получения  этого адреса следующие:

1. Если аргумент *shmaddr* нулевой, то система самостоятельно выбирает адрес.
2. 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* (используется системой как размер страницы памяти).

По умолчанию разделяемая память присоединяется с правами на *чтение и запись*.

Эти права можно изменить,
указав флажок *SHM_RDONLY* в аргументе *shmflag .*

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);
```

Программа *attach_shm* иллюстрирует процедуры присоединения сегмента общей памяти, отображения ее на память процесса и операции с ней.

```
/* Программа attach_shm.cpp */

/* Создание и присоединение сегмента,
   операции с разделяемой памятью  */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 30

extern etext, edata, end;
```

```
main(void) {
	pid_t pid;
	int	shmid;
	char	c, *shm, *s;

	if ((shmid=shmget(IPC_PRIVATE, SHM_SIZE, IPC_CREAT|0666))< 0) {
		perror("shmget fail");
		exit(1);
	}
	if ((shm = (char *) shmat( shmid, 0, 0)) == (char *) -1) {
		perror("shmat : parent ");
		exit(2);
	}
	printf("Addresses in parent\n\n");
	printf("shared mem: %X etext: %X edata: %X end: %X\n\n",
		shm, &etext, &edata, &end);

	s = shm;		/* Указатель s ссылается на разделяемую память */
	for (c = 'A'; c <= 'Z'; ++c)	/* Выложить в область разделяемой
						памяти */
		*s++ = c;
	*s = NULL;			/* Маркер конца строки */
	printf("In parent before fork, memory is : %s \n", shm);
	pid = fork();

    switch (pid) {
	case -1:
		perror("fork ");
		exit(3);
	default:
		sleep(5);			/* Ожидание завершения дочернего
							процесса */
		printf("\nIn parent after fork, memory is  : %s\n", shm);
		printf("Parent removing shared memory\n");
		shmdt(shm);
		shmctl(shmid, IPC_RMID, (struct shmid_ds *) 0 );
		exit(0);
	case 0:
		printf("In child after fork, memory is : %s \n",shm);
		for (; *shm; ++shm)	/* Модификация разделяемой памяти */
			*shm += 32;
		shmdt (shm );
		exit(0);
	}
 }
```

В приведенной программе создаётся новый сегмент разделяемой памяти длиной 30 байт. Этот сегмент привязывается к области данных процесса, используя первый доступный адрес. Реальные адреса привязки, хранимые в *etext*, *edata*, *end*, выводятся на консоль.
Указатель типа *char* * содержит адрес начала сегмента разделяемой памяти, в который записываются данные
(последовательность прописных букв алфавита).
Системный вызов *fork* создаёт процесс-потомок, который повторно выводит на экран содержимое разделяемого сегмента, а затем модифицирует содержимое разделяемой памяти (преобразованием прописных букв в строчные). По окончании преобразования, процесс-потомок отъединяет разделяемый сегмент и завершает работу.
Процесс-родитель, после 5-ти секундного ожидания, выводит на консоль  содержимое разделяемого сегмента, отъединяет сегмент разделяемой памяти и удаляет его системным вызовом *shmctl()*.

## Управление сегментом разделяемой памяти

Системный вызов *shmctl()* дает возможность производить ряд операций по управлению существующим сегментом разделяемой памяти и системной структурой данных *shmid_ds* этого сегмента.

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

int shmctl (int shmid, int cmd, struct shmid_ds *buf);
```

Вызов *shmctl()* имеет три аргумента.
Первый, *shmid* ,
это идентификатор разделяемого сегмента памяти, сгенерированный раннее вызванным shmget().
Второй аргумент, *cmd*,
определяет операцию, которую должен выполнить *shmctl()*.
Третий аргумент, *buf*,
это указатель на структуру типа *shmid_ds*.

Вызов *shmctl()* исполняет следующие операции *cmd*:

- *IPC_RMID* - удалить системную структуру данных с указанным идентификатором разделяемого сегмента *shmid* .
  На месте аргумента *buf* указывают 0, приведённый к нужному типу *(shmid_ds *)*.
- *IPC_STAT* - возвращает текущее значение структуры *shmid_ds* для разделяемого сегмента с идентификатором *shmid*. Процесс должен иметь право на чтение данного разделяемого сегмента;

- *IPC_SET* - модифицировать некоторые элементы структуры *shm_perm* в структуре *shmid_ds*. Такие элементы, как *shm_perm.uid, shm_perm.gid и shm_perm.mode*. Процесс должен иметь эффективный идентификатор суперпользователя или ID, совпадающий с *shm_perm.cuid* или *shm_perm.uid (создателя или владельца сегмента)* .
- *SHM_LOCK* - блокирование в памяти (запрет свопинга) сегмента разделяемой памяти, соответствующий *semid* аргументу;
- *SHM_UNLOCK* - разблокирование сегмента (разрешение свопинга) разделяемой памяти, на который указывает *semid*. Операции *SHM_LOCK и SHM_UNLOCK* может выполнить процесс, имеющий эффективный идентификатор сегмента или *ID* суперпользователя.

**Синхронизация операций с разделяемой памятью**

При работе с разделяемой памятью необходимо синхронизировать выполнение взаимодействующих процессов:

когда один из процессов записывает данные в разделяемую память, остальные процессы ожидают завершения операции.

Обычно синхронизация обеспечивается с помощью семафоров, назначение и число которых определяется конкретным использованием разделяемой памяти.

Можно привести примерную схему обмена данными между двумя процессами (клиентом и сервером) с использованием разделяемой памяти. Для синхронизации процессов использована группа из двух семафоров.

Первый семафор служит для блокирования доступа к разделяемой памяти, его разрешающий сигнал — 0,
а 1 является запрещающим сигналом.

Второй семафор служит для сигнализации серверу о том,
что клиент начал работу.

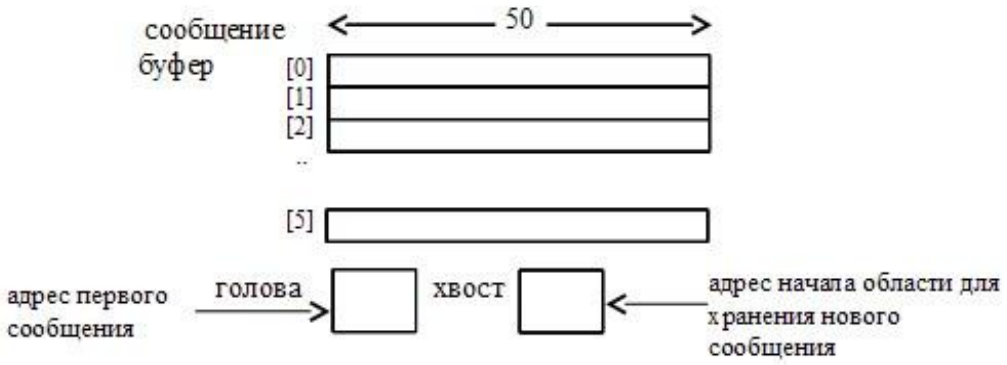
Необходимость применения второго семафора обусловлена следующими обстоятельствами:
начальное состояние семафора, синхронизирующего работу с памятью, является открытым (0), и вызов сервером операции *mem_lock* заблокирует обращение к памяти для клиента.

Таким образом, сервер должен вызвать операцию только после того, как разделяемую память заблокирует клиент.

Назначение второго семафора заключается в уведомлении сервера, что клиент начал работу, заблокировал разделяемую память и начал записывать данные в эту область.
Теперь, при вызове сервером операции *mem_lock* его выполнение будет приостановлено до освобождения памяти клиентом.

**Пример организации обмена сообщениями**

Рассмотрим пример взаимодействия процессов с организацией обмена через сегмент разделяемой памяти. Один процесс будет вырабатывать случайные сообщения, которые будут храниться в разделяемом сегменте памяти и считываться вторым процессом.
Для обеспечения взаимодействия между двумя процессами, которые могут выполняться с *различными скоростями*, используется массив с шестью буферами для сообщений. Данный массив буферов обслуживается как очередь (FIFO). Конфигурация сегмента разделяемой памяти при этом имеет вид:



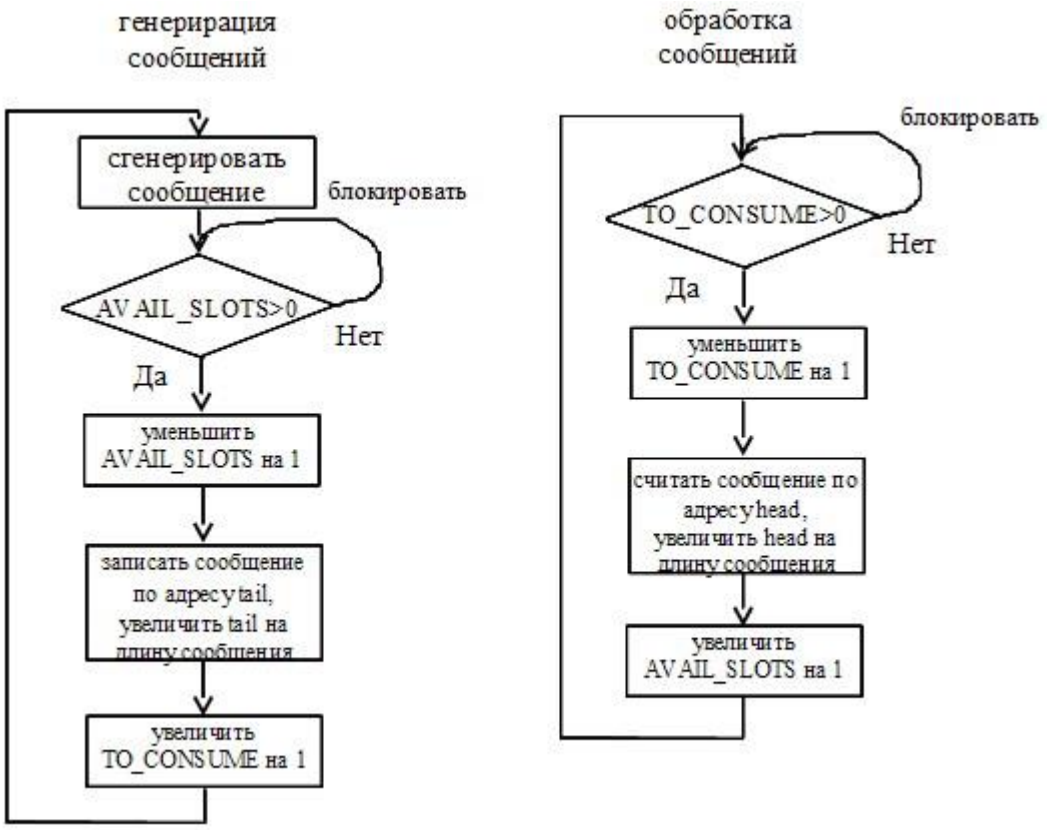

Для координации доступа к разделяемому сегменту данных используются две переменные-семафоры.

Первый семафор, используемый как семафор-счётчик, будет хранить число доступных для записи слотов. Пока его значение отлично от нуля, процесс может продолжать запись сообщений в сегмент разделяемой памяти.
При инициализации счётчик доступных слотов устанавливается равным 6.

Второй семафор, также являющийся счётчиком, будет указывать число слотов, доступных для чтения.
Оба процесса — записывающий (*producer*) и считывающий (*consumer*) — выполняются параллельно и используют один разделяемый сегмент памяти.

В примере вначале запускается процесс-родитель (*parent*), который отвечает за создание и инициализацию разделяемого сегмента памяти и двух переменных-семафоров.
Затем процесс-родитель запускает вызовом *fork()* двух потомков : первым процесс-генератор (*producer*), вторым - обработчик сообщений (*consumer*).

Исполнение процессов *producer* и *consumer* схематично показано на рисунке :




В файлах *parent*, *producer* и *consumer* содержится исходный код примера. Для снижения объёма кода используется локальный файл заголовка, *local.h* . Он содержит все директивы *include* и объявления переменных, требуемых каждой из программ (*parent*, *producer* и *consumer*), составляющих данный пример

*/* Файл  local.h */*

```
 /*  common header file: parent, producer and consumer */
 #include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <wait.h>
#include <signal.h>
```

```
#define ROWS 5
#define COLS 3
#define SLOT_LEN 50
#define N_SLOTS 6
struct MEMORY {
   char buffer[N_SLOTS][SLOT_LEN];
   int head, tail;
   };

struct sembuf
   acquire = { 0, -1, SEM_UNDO},
   release = { 0, 1, SEM_UNDO};

enum {AVAIL_SLOTS, TO_CONSUME};
```

При вызове программы *parent* из командной строки нужно использовать 2 аргумента. Этими аргументами являются
максимальное время ожидания процесса (*sleep time*) во время выполнения для процесса-генератора и процесса-обработчика соответственно.

Задавая различные значения аргументов, можно легко моделировать взаимодействие генератора/обработчика, работающих *с разными скоростями*.

После создания и привязывания сегмента разделяемой памяти, в этот сегмент копируется содержимое структуры *MEMORY* , используя библиотечную функцию *memcpy* .

Определение *memcpy* имеет вид

```
void *memcpy(void *s1, const void *s2, size_t n);
```

и находится в файле */usr/include/string.h* .
Функция копирует *n* байт, начиная с адреса *s2*, в область, начинающуюся с *s1*, и возвращает *s1* .

Далее создаются переменные-семафоры и им присваиваются начальные значения: *AVAIL_SLOTS=6*, *TO_CONSUME=0*.

Затем порождаются процессы генератор (*producer*) и обработчик (*consumer)* сообщений.

Процессы получают при запуске единственный аргумент - время ожидания (*argv[1]* и *argv[2]* , соответственно).

Процесс-родитель ожидает завершения одного из процессов (генератора или обработчика).
Когда один из процессов завершается, его *ID* записывается в переменную *croaker.*
Затем процесс-родитель проверяет содержимое *croaker* , чтобы определить *ID* другого, оставшегося процесса.

Оставшийся процесс затем удаляется вызовом *kill()* ,
а сегмент разделяемой памяти
отсоединяется и удаляется.

```
/* Программа parent.cpp */
#include "local.h"
/*
 *  Родительский процесс
 */
main(int argc, char *argv[ ]) {

static
struct MEMORY    memory;
static ushort    start_val[2] = {N_SLOTS, 0};
int    semid, shmid, croaker;
char   *shmptr;
pid_t  p_id, c_id, pid = getpid( );
union  semun     arg;

memory.head = memory.tail = 0;

if (argc != 3 ) {
        fprintf(stderr, "%s producer_time consumer_time\n",
                        argv[0]);
        exit(-1);
}
/*
 *  Создание, привязка и инициализация сегмента разделяемой памяти
 */
if ((shmid=shmget((int)pid,sizeof(memory),
                IPC_CREAT | 0600 )) != -1){
        if ((shmptr=(char *)shmat(shmid, 0, 0)) == (char *) -1){
        perror("shmptr -- parent -- attach ");
        exit(1);
        }
        memcpy(shmptr, (char *)&memory, sizeof(memory));
} else {
        perror("shmid -- parent -- creation ");
        exit(2);
}
/*
 *  Создание и инициализация семафоров
 */
if ((semid=semget((int)pid, 2, IPC_CREAT | 0666)) != -1) {
        arg.array = start_val;
        if (semctl(semid, 0, SETALL, arg) == -1) {
        perror("semctl -- parent -- initialization");
        exit(3);
        }
```

```
} else {
        perror("semget -- parent -- creation ");
        exit(4);
}
/*
Порождение процесса producer
*/
if ( (p_id=fork( )) == -1) {
        perror("fork -- producer ");
        exit(5);
} else if ( p_id == 0) {
        execl( "producer", "producer", argv[1], (char *) 0);
        perror("execl -- producer ");
        exit(6);
}
/*
 *  Порождение процесса consumer
 */
if ( (c_id=fork()) == -1) {
        perror("fork -- consumer ");
        exit(7);
} else if (c_id == 0 ) {
        execl("consumer", "consumer", argv[2], (char *) 0);
        perror("execl -- consumer ");
        exit(8);
}
croaker = (int) wait( (int *) 0);                       /* Завершение по 1 */
kill( (croaker == pid ) ? c_id : p_id, SIGKILL);

shmdt( shmptr );                                        /* Отсоединение */
shmctl(shmid,IPC_RMID,(struct shmid_ds *) 0);       /* Удаление */
semctl(      semid, 0, IPC_RMID, arg);           /* arg  для Linux вместо 0 */
exit(0);
}
```

Процесс-генератор (*producer)* размещает двумерный массив, *source*, который содержит множество переменных *string*, используемых в дальнейшем для генерации случайных сообщений, записываемых в разделяемый сегмент памяти.
Для временного хранения сообщений создана переменная *local buffer.* Затем вычисляется значение *ID* процесса-родителя вызовом *getppid()*. Значение ID процесса-родителя используется в вызове *shmget()*. Это позволяет процессу-генератору сообщений обращаться к тому разделяемому сегменту памяти, который был создан процессом-родителем (в другом случае можно было бы передать идентификатор разделяемой памяти от процесса-родителя процессу-потомку через командную строку). Процесс-генератор получает доступ к разделяемому сегменту памяти и привязывает его.

Процесс использует локальный указатель *memptr* для привязки разделяемой памяти и для адресации разделяемого сегмента памяти.

Процесс-генератор затем получает доступ к множеству переменных-семафоров, снова используя *ID* родительского процесса, как аргумент вызова *semget()*.
После того, как это выполнено, определяется время *sleep* и вычисляется максимальное число сообщений, которые необходимо сгенерировать.
Затем программа циклически выполняет следующие шаги.
Освобождает *local_buffer*, заполняя его *null*-ами.
Генерируется случайным образом сообщение и записывается в *local_buffer*.
Процесс-генератор затем вычисляет переменную-семафор *AVAIL_SLOTS*.
Когда процесс получает значение семафора, сообщение из *local_buffer* копируется в область разделяемой памяти со смещением, равным *memory->tail*. Сообщение выводится на экран.
Содержимое *memory->tail* увеличивается таким образом, чтобы эта переменная указывала на следующую область хранения.
Семафор *TO_CONSUME* увеличивается каждый раз, когда поступает новое сообщение.
Затем процесс-генератор ждёт не больше *sleep_limit* секунд и продолжает выполнение цикла. Процесс-генератор завершит работу, когда будут сгенерированы все сообщения и записаны в разделяемый сегмент памяти, или когда получит сигнал завершить работу (например от процесса-родителя).

```
/* Программа producer.cpp */
#include "local.h"
/*
 *  Процесс producer
 */
main(int argc, char *argv[]) {

        static char *source[ROWS][COLS] = {
        {"A", "The", "One"},
        {" red"," polka_dot"," yellow"},
        {" spider", "dump truck", " tree"},
        {" broke"," ran"," fell"},
        {" down"," away"," out"}
        };
        static char  local_buffer[SLOT_LEN];
        int             i,r,c,sleep_limit,semid,shmid;
        pid_t           ppid = getppid( );
        char            *shmptr;
        struct MEMORY    *memptr;

        if ( argc != 2) {
                fprintf(stderr, "%s sleep_time", argv[0]);
                exit(-1);
        };
```

```
        /*
         *  Получение доступа, подсоединение ссылка на разделяемую
память
         */
        if ((shmid=shmget((int) ppid, 0, 0)) != -1 ){
                if ((shmptr=(char *)shmat(shmid, (char *)0,0))==(char *) -1){
                        perror("shmat -- producer -- attach ");
                        exit(1);
                }
                memptr = (struct MEMORY *) shmptr;
        } else {
                perror("shmget -- producer -- access ");
                exit(2);
        }
        /*
         *  Доступ к семафорам
         */
        if ( (semid=semget((int) ppid, 2, 0)) == -1 ) {
                perror("semget -- producer -- access ");
                exit(3);
        }
        sleep_limit = atoi(argv[1]) % 20;
        i = 20 - sleep_limit;
        srand((unsigned) getpid());
        while( i-- ) {
                memset(local_buffer, '\0',sizeof(local_buffer));
                for (r = 0;r < ROWS; ++r) { /* Генерация случайных
                                                                символов */
                        c = rand() % COLS;
                        strcat(local_buffer, source[r][c]);
                }
                acquire.sem_num = AVAIL_SLOTS;
                if (semop(semid, &acquire, 1) == -1) {
                        perror ("semop -- producer -- acquire ");
                        exit(4);
                }
                strcpy(memptr->buffer[memptr->tail], local_buffer);
                printf("P: [%d] %s.\n", memptr->tail,
                        memptr->buffer[memptr->tail]);
                memptr->tail = (memptr->tail + 1) % N_SLOTS ;
                release.sem_num = TO_CONSUME;
                if (semop(semid, &release, 1) == -1) {
                        perror("semop -- producer -- release ");
                        exit(5);
                }
                sleep( rand() %sleep_limit + 1);
        }
        exit(0);
}
```

Процесс-обработчик ( *consumer)* имеет права доступа к разделяемому сегменту памяти через семафор *TO_CONSUME*.
Если его значение больше нуля, то есть сообщения,
которые обработчик может прочесть.
Он копирует сообщение в *local_buffer* из разделяемого сегмента со смещением *memory->head* .
Затем содержимое *local_buffer* выводится на экран.
Значение *memory->head* увеличивается таким образом,
чтобы эта переменная указывала на следующую область хранения.
Переменная-семафор *AVAIL_SLOTS* используется процессом-обработчиком.
Из вывода программы видно, что если процесс-родитель получил в командной строке значения, позволяющие процессу-генератору работать быстрее, чем обработчик,
разделяемая область может оказаться полностью заполненной.
Когда это происходит, генератор должен блокироваться и
ожидать считывания сообщения обработчиком события.
Только после того, как обработчик считал сообщение,
слот освобождается и генератор сохраняет следующее сообщение.

```
/* Программа consumer.cpp */
#include "local.h"
/*
 *  Процесс  consumer
 */
main(int argc, char *argv[]) {

static char   local_buffer[SLOT_LEN];
int           i,sleep_limit,semid, shmid;
pid_t         ppid = getppid( );
char          *shmptr;
struct MEMORY     *memptr;

if ( argc != 2 ) {
        fprintf(stderr, "%s sleep_time", argv[0]);
        exit(-1);
};
/*
 *  Получение доступа, подсоединение и ссылка на разделяемую память
 */
if ((shmid=shmget((int) ppid, 0, 0)) != -1 ) {
        if( (shmptr=(char *)shmat(shmid,(char *)0,0)) == (char *) -1) {
                perror("shmat -- consumer -- attach ");
                exit(1);
        };
        memptr = (struct MEMORY *) shmptr;
} else {
        perror("shmget -- consumer -- access ");
        exit(2);
}
```

```
/*
 * Доступ к семафорам
 */
if( (semid=semget((int)ppid, 2, 0)) == -1 ) {
	perror("semget -- consumer -- access ");
	exit(3);
};
sleep_limit = atoi(argv[1]) % 20;
i = 20 - sleep_limit;
srand((unsigned)getpid());
while( i ) {
	acquire.sem_num = TO_CONSUME;
	if (semop(semid,&acquire, 1) == -1) {
		perror("semop -- consumer -- acquire ");
		exit(4);
	}
	memset(local_buffer,'\0',sizeof(local_buffer));
	strcpy(local_buffer,memptr->buffer[memptr->head]);
	printf("C: [%d] %s.\n",memptr->head,local_buffer);
	memptr->head = (memptr->head +1) % N_SLOTS;
	release.sem_num = AVAIL_SLOTS;
	if (semop(semid, &release, 1) == -1 ) {
		perror("semop -- consumer -- release ");
		exit(5);
	}
	sleep( rand() % sleep_limit + 1);
}
exit(0);
}
```