ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лекция 09 — Механизмы межпроцессного взаимодействия System V

Преподаватель: Поденок Леонид Петрович, 505a-5

+375 17 293 8039 (505a-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2022

2023.04.11

Оглавление

Механизмы межпроцессного взаимодействия System V

ОС UNIX поддерживает три типа средств межпроцессной связи (InterProcess Communication):
- очереди сообщений (msg);
- наборы семафоров (sem);
- совместно используемые сегменты памяти (shm).

```
#include <sys/msg.h>
#include <sys/sem.h>
#include <sys/shm.h>
```

Средства межпроцессной связи ОС UNIX имеют много общих черт.

Прежде чем использовать какой-либо из механизмов IPC процесс должен попросить систему организовать необходимые для этого программные механизмы.

```
msgget(), msgctl(), msgrcv(), mcgsnd()
semget(), semctl(), semop()
shmget(), shmctl(), shmat(), shmdt()
```

Обратившись к одному из этих вызовов, процесс становится *создателем* и *владельцем* некоторого средства IPC.

Кроме того, процесс должен указать первоначальные права на выполнение операций для всех процессов, включая себя.

Впоследствии владелец/создатель может уступить право владения или изменить права на операции при помощи системного вызова *****ctl()**, однако на протяжении всего времени существования средства обмена сообщениями создатель остается создателем.

Другие процессы, обладающие соответствующими правами, для выполнения различных управляющих действий также могут использовать системные вызовы *****ctl()**.

Итак, в парадигме IPC UNIX есть:

- создатель;
- владелец;
- пользователь.

Получение ключа для доступа к механизмам IPC

ftok — преобразовывает имя файла и идентификатор проекта в ключ для системных вызовов.

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

key_t ftok(const char *pathname, // маршрут к файлу
           int         proj_id); // идентификатор проекта
```

Для создания ключа с типом **key_t**, используемого для работы с **msgget(2)**, **semget(2)**, и **shmget(2)** используется файл с именем **pathname**, которое должно указывать на существующий файл и к нему должен быть доступ (право на чтение каталога, где располагается файл).

Младшие 8 бит **proj_id** должны быть отличны от нуля. Обычно при вызове этой функции в качестве **proj_id** передается символ ASCII, именно поэтому поведение функции считается не определенным в случае, если **proj_id** равен нулю.

Не гарантируется, что возвращаемый ключ **key_t** будет уникален. Обычно производится объединение указанного байта **proj_id** с младшими 16 битами номера **i-node** и младшими 8 битам номера устройства в 32-битный результат. Поэтому могут возникать конфликты, между файлами на разных устройствах (например, между файлами на **/dev/hda1** и файлами на **/dev/sda1)**.

Возвращаемое значение будет разным, если одновременно существующие файлы или идентификаторы проекта различаются.

В случае удачного завершения вызова возвращается значение созданного ключа **key_t**.

Возвращаемое значение одинаково для всех имен, указывающих на один и тот же файл при одинаковом значении **proj_id**. При ошибке возвращается **-1** а в **errno** код ошибки.

Права доступа к ресурсу

Для каждого ресурса система использует общую структуру типа **`struct ipc_perm`**, хранящую необходимую информацию о правах для проведения IPC-операции.

Структура **`ipc_perm`** включает следующие поля:

```
struct ipc_perm {          // В <sys/ipc.h>
    uid_t          cuid;   // ID пользователя создателя
    gid_t          cgid;   // ID группы создателя
    uid_t          uid;    // ID пользователя владельца
    gid_t          gid;    // ID группы владельца
    unsigned short mode;   // права для чтения-записи
};
```

Поле **`mode`** из структуры **`ipc_perm`** определяет в нижних 9 битах права доступа к ресурсу для вызвавшего системный вызов IPC процесса. Права определены следующим образом:

0400 Чтение пользователем
0200 Запись пользователем
0040 Чтение группой
0020 Запись группой
0004 Чтение остальными
0002 Запись остальными

Биты 0100, 0010 и 0001 (биты запуска) системой не используется. Кроме того, «запись» для набора семафоров на самом деле означает «изменение».

Тот же системный заголовочный файл определяет следующие символические константы:

IPC_CREAT	Создать ресурс, если ключ не существует
IPC_EXCL	Завершиться ошибкой, если ключ существует
IPC_NOWAIT	Ошибка, если запрос должен ждать
IPC_RMID	Удалить ресурс
IPC_SET	Установить параметры ресурса
IPC_STAT	Получить параметры ресурса
IPC_PRIVATE	**Частный (приватный) ключ**

Следует отметить, что **IPC_PRIVATE** является типом **key_t**, в то время как остальные символические константы являются флагами и могут быть объединены с помощью логического **ИЛИ** в переменную типа **int**.

Очереди сообщений

В системе очередь сообщений уникально идентифицируется положительным целым (**msqid**) и имеет связанную с ней структуру данных **struct msqid_ds**, определенную в **<sys/msg.h>** и содержащую следующие поля:

```
struct msqid_ds {
    struct ipc_perm msg_perm;    // права доступа
    msgqnum_t       msg_qnum;    // текущее количество сообщений в очереди
    msglen_t        msg_qbytes;  // максимально допустимое кол-во байт в очереди
    pid_t           msg_lspid;   // PID последнего вызова msgsnd(2)
    pid_t           msg_lrpid;   // PID последнего вызова msgrcv(2)
    time_t          msg_stime;   // время последнего msgsnd(2)
    time_t          msg_rtime;   // время последнего msgrcv(2)
    time_t          msg_ctime;   // последнее время изменения
};
```

msg_perm — структура **ipc_perm** определяет права доступа к очереди сообщений;
msg_qnum — число сообщений, находящихся в данный момент в очереди сообщений;
msg_qbytes — максимальная длина сообщения в байтах, разрешенная в очереди сообщений;
msg_lspid — ID процесса, выполнившего последний системный вызов **msgsnd(2)**;
msg_lrpid — ID процесса, выполнившего последний системный вызов **msgrcv(2)**;
msg_stime — время последнего вызова **msgsnd(2)**;
msg_rtime — время последнего вызова **msgrcv(2)**;
msg_ctime — время последнего системного вызова, изменившего поля структуры **msgid_ds**.

Наборы семафоров

Набор семафоров уникально идентифицируется положительным целым (**semid**) и имеет связанную структуру типа **struct semid_ds**, определенную в **<sys/sem.h>**, содержащую след. поля:

```
struct semid_ds {
    struct ipc_perm msg_perm;    // права доступа
    time_t          sem_otime;   // время последней операции
    time_t          sem_ctime;   // время последнего изменения
    unsigned long   sem_nsems;   // число семафоров в наборе
};
```

sem_perm - структура **ipc_perm**, определяющая права доступа к набору семафоров;
sem_otime - время последнего системного вызова **semop(2)**;
sem_ctime - время последнего системного вызова **semctl(2)**, который изменил поле указанной структуры или один из семафоров, принадлежащих набору;
sem_nsems - число семафоров в наборе. Каждый семафор идентифицируется неотрицательным целым числом от **0** до **sem_nsems-1**.

Семафор является структурой данных типа **struct sem**, содержащие следующие поля:

```
struct sem {     // определена в ядре
    int semval;  // значение семафора
    int sempid;  // PID последнего изменившего процесса
};
```

semval — значение семафора, неотрицательное целое.
sempid — PID последнего процесса, изменившего значение данного семафора.

Сегменты общей памяти

Сегмент общей памяти уникально идентифицируется положительным целым (**shmid**) и имеет связанную структуру данных **struct shmid_ds**, определённую в **<sys/shm.h>** и содержащую следующие поля:

```
struct shmid_ds {
    struct ipc_perm msg_perm;    // права доступа
    size_t          shm_segsz;   // размер сегмента
    pid_t           shm_cpid;    // PID создателя
    pid_t           shm_lpid;    // PID последней операции
    shmatt_t        shm_nattch;  // число текущих подключений
    time_t          shm_atime;   // время последнего подключения
    time_t          shm_dtime;   // время последнего отключения
    time_t          shm_ctime;   // время последнего изменения
};
```

shm_perm - структура ipc_perm, описывающая права доступа к сегменту общей памяти.
shm_segsz - размер в байтах сегмента общей памяти.
shm_cpid - ID процесса, создавшего сегмент общей памяти.
shm_lpid - ID последнего процесса, выполнившего системный вызов **shmat(2)** или **shmdt(2)**.
shm_nattch - количество текущих подключений для данного сегмента общей памяти.
shm_atime - время последнего системного вызова **shmat(2)**.
shm_dtime - время последнего системного вызова **shmdt(2)**.
shm_ctime - время последнего системного вызова **shmctl(2)**, изменившего **shmid_ds**.

Пространства имён IPC

Пространства имён IPC изолируют определённые ресурсы IPC, а именно IPC-объекты System V и очереди сообщений POSIX. Общая характеристика этих механизмов IPC в том, что объекты IPC распознаются механизмами не как пути файловой системы.

Каждое пространство имён IPC имеет свой набор идентификаторов System V IPC и *свою файловую систему* для очередей сообщений POSIX. Объекты, созданные в пространстве имён IPC, видимы всем другим процессам, которые являются членами этого пространства имён, и невидимы процессам из других пространств имён IPC.

Следующие интерфейсы **/proc** отличаются в каждом пространстве имён IPC:

- интерфейсы очереди сообщений POSIX в **/proc/sys/fs/mqueue**.

- IPC-интерфейсы System V в **/proc/sys/kernel**, а именно: **msgmax**, **msgmnb**, **msgmni**, **sem**, **shmall**, **shmmax**, **shmmni** и **shm_rmid_forced**.

- IPC-интерфейсы System V в **/proc/sysvipc**.

```
$ ls -l /proc/sysvipc
-r--r--r--. 1 root root 0 апр 10 12:12 msg
-r--r--r--. 1 root root 0 апр 10 12:12 sem
-r--r--r--. 1 root root 0 апр 10 12:12 shm
```

При уничтожении пространства имён IPC (т. е., когда завершается последний процесс из этого пространства имён), все объекты IPC из пространства имён автоматически уничтожаются.

Для использования пространств имён IPC требуется, чтобы ядро было собрано с параметром **CONFIG_IPC_NS**.

Очереди сообщений

Очереди сообщений как средство межпроцессной связи дают возможность процессам взаимодействовать, обмениваясь данными. Данные передаются между процессами дискретными порциями, называемыми *сообщениями*.

Процессы, использующие этот тип межпроцессной связи, могут выполнять две операции:

- послать сообщение (send message);
- принять сообщение (receive message).

Процесс, прежде чем послать или принять какое-либо сообщение, должен попросить систему организовать необходимые для этого программные механизмы.

Процесс делает это при помощи системного вызова **`msgget()`**.

Обратившись к нему, процесс становится *создателем* и *владельцем* некоторого средства обмена сообщениями. Кроме того, процесс должен указать первоначальные права на выполнение операций для всех процессов, включая себя.

Впоследствии владелец/создатель может уступить право владения или изменить права на операции при помощи системного вызова **`msgctl()`**, однако на протяжении всего времени существования средства обмена сообщениями создатель остается создателем.

Другие процессы, обладающие соответствующими правами, для выполнения различных управляющих действий также могут использовать системный вызов **`msgctl()`**. Итак, в парадигме очередей сообщений UNIX есть:

- создатель;
- владелец;
- пользователь.

Процессы, имеющие права на операции и пытающиеся послать или принять сообщение, могут приостанавливаться, если выполнение операции не оказалось успешным.

В частности это означает, что процесс, пытающийся послать сообщение, может ожидать, пока процесс-получатель не будет готов, или, наоборот, получатель может ждать отправителя.

Если указано, что процесс в таких ситуациях должен приостанавливаться, говорят о выполнении над сообщением операции с блокировкой (блокирующая операция).

Если приостанавливать процесс нельзя, говорят, что над сообщением выполняется операция без блокировки.

Процесс, выполняющий операцию с блокировкой, может быть приостановлен до тех пор, пока не будет удовлетворено одно из условий:

1) операция завершилась успешно;

2) процесс получил сигнал;

3) очередь сообщений ликвидирована.

Пользоваться этими возможностями обмена сообщениями процессам позволяют *системные вызовы*.

Вызывающий процесс передает системному вызову аргументы, а системный вызов выполняет (успешно или нет) свою функцию.

Если системный вызов завершается успешно, он выполняет то, что от него требуется, и возвращает некоторую содержательную информацию.

В противном случае процессу возвращается значение **`-1`**, известное как признак ошибки, а внешней переменной **`errno`** присваивается код ошибки.

Использование очередей сообщений

Перед тем, как посылать или принимать сообщения, должны быть созданы очередь сообщений с уникальным идентификатором и ассоциированная с ней структура данных.

Идентификатор очереди сообщений (**msqid**) используется для обращений к очереди сообщений и ассоциированной с ней структуре данных.

Реально в очереди сообщений хранятся не сами сообщения, а их дескрипторы (описатели), имеющие следующую структуру:

```
/* Structure of record for one message inside the kernel.
   The type `struct msg' is opaque.  */
struct msg {                 //
    struct msg *msg_next; // Указатель на следующее сообщение
    long        msg_type; // Тип сообщения
    short       msg_ts;   // Размер текста сообщения
    short       msg_spot; // Адрес текста сообщения
};
```

Приведенное определение находится в заголовке **<sys/msg.h>**.

В процессе подключения данного заголовка может подключаться несколько вложенных заголовков, в которых непосредственно находится указанная здесь и далее информация.

С каждым уникальным идентификатором очереди сообщений ассоциирована одна структура данных — **msqid_ds**, которая содержит (условно) следующую информацию:

```
struct msqid_ds {
    struct ipc_perm msg_perm; // Структура прав на выполнение операций

    struct msg *msg_first;    // Указатель на первое сообщение в очереди
    struct msg *msg_last;     // Указатель на последнее сообщение в очереди

    ushort msg_cbytes;        // Текущее число байт в очереди
    ushort msg_qnum;          // Число сообщений в очереди
    ushort msg_qbytes;        // Максимально допустимое число  байт в очереди

    ushort msg_lspid;         // Идентификатор последнего отправителя
    ushort msg_lrpid;         // Идентификатор последнего получателя

    time_t msg_stime;         // Время последнего отправления
    time_t msg_rtime;         // Время последнего получения
    time_t msg_ctime;         // Время последнего изменения
};
```

Это (или аналогичное) определение также находится во включаемом файле **<sys/msg.h>**.

Поле **msg_perm** данной структуры использует в качестве шаблона структуру **ipc_perm**, которая задает права на операции с сообщениями.

Структура прав на операции с сообщениями определяется так:

```
struct ipc_perm {
    key_t  key;  // Ключ, передаваемый в msgget()
    ushort uid;  // Эффективный UID владельца
    ushort gid;  // Эффективный GID владельца
    ushort cuid; // Эффективный UID создателя очереди
    ushort cgid; // Эффективный GID создателя очереди
    ushort mode; // Права на чтение/запись
    ushort seq;  // Последовательность номеров используемых слотов
};
```

Данное определение находится во включаемом файле **<sys/ipc.h>**, общем для всех средств межпроцессной связи — сообщений, семафоров и совместно используемой памяти.

Системный вызов **msgget** принимает два параметра — уникальный ключ **key** и некоторый набор флагов.

IPC_CREAT	Создать очередь сообщений, если ключ не существует
IPC_EXCL	Завершиться ошибкой, если ключ существует
IPC_PRIVATE	Частный (приватный) ключ
IPC_NOWAIT	Ошибка, если запрос должен ждать

Если в аргументе **msgflg** системного вызова **msgget** установлен только флаг **IPC_CREAT**, выполняется одно из двух действий:

- порождается новый идентификатор **msqid** и создаются ассоциированные с ним очередь сообщений и структура данных;

- возвращается существующий идентификатор **msqid**, с которым уже ассоциированы очередь сообщений и структура данных.

Действие определяется по значению аргумента **key**. Если еще не существует идентификатора **msqid** со значением ключа **key**, выполняется первое действие, то есть для данного ключа выделяется новый уникальный идентификатор и создаются ассоциированные с ним очередь сообщений и структура данных (при условии, что не будет превышен соответствующий системный лимит).

Если идентификатор **msqid** со специфицированным значением ключа **key** уже существует, выполняется второе действие, то есть возвращается ассоциированный идентификатор.

Если необходимо трактовать возвращение существующего идентификатора как ошибку, в передаваемом системному вызову аргументе **msgflg** нужно установить флаг **IPC_EXCL**.

Кроме того, можно указать ключ **key** со значением **IPC_PRIVATE**.

Если указан такой «личный» ключ, для него обязательно выделяется новый уникальный идентификатор очереди и создаются ассоциированные с ним очередь сообщений и структура данных (при условии, что это не приведет к превышению системного лимита).

В этом случае при выполнении утилиты **ipcs**[1] поле KEY для подобного идентификатора **msqid** из соображений секретности будет содержать нули.

При выполнении первого действия (порождение) процесс, вызвавший **msgget**, становится владельцем/создателем очереди сообщений — соответственно этому инициализируется ассоциированная структура данных.

Владелец очереди может быть изменен, однако процесс-создатель всегда остается создателем. При создании очереди сообщений определяются также начальные права на выполнение операций над ней.

После того, как созданы очередь сообщений с уникальным идентификатором и ассоциированная с ней структура данных, можно использовать системные вызовы семейства **msgop** (операции над очередями сообщений) и **msgctl** (управление очередями сообщений).

---

1) Показывает информацию о межпроцессных механизмах в системе

Операции

Операции заключаются в посылке и приеме сообщений.

Для каждой из этих операций предусмотрен системный вызов, **`msgsnd()`** и **`msgrcv()`** соответственно.

Для управления очередями сообщений используется системный вызов **`msgctl()`**.

Этот вызов позволяет выполнять следующие управляющие действия:

- опросить содержимое структуры данных, ассоциированной с идентификатором очереди сообщений **`msqid`**;

- изменить права на выполнение операций над очередью сообщений;

- изменить максимально допустимое число байт (**`msg_qbytes`**) в очереди сообщений, определяемой идентификатором **`msqid`**;

- удалить из системы идентификатор **`msqid`**, ликвидировать очередь сообщений и ассоциированную с ней структуру данных.

Создание очередей сообщений

Для создания используется системный вызов **msgget**.

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

int msgget(key_t key, int msgflg);
```

Тип **key_t** описан во включаемом файле **<sys/types.h>** при помощи **typedef** как целый тип.

Целочисленное значение, возвращаемое в случае успешного завершения системного вызова, есть идентификатор очереди сообщений (**msqid**).

В случае неудачи результат равен -1.

Новый идентификатор **msqid**, очередь сообщений и ассоциированная с ней структура данных выделяются в каждом из двух случаев:

- значение ключа **key** равно **IPC_PRIVATE;**
- ключ **key** еще не имеет ассоциированного с ним идентификатора очереди сообщений и выражение (**msgflg & IPC_CREAT**) истинно.

IPC_CREAT	Создать, если ключ не существует
IPC_EXCL	Завершиться ошибкой, если ключ существует
IPC_PRIVATE	Частный (приватный) ключ
IPC_NOWAIT	Ошибка, если запрос должен ждать

Целое число, передаваемое в качестве аргумента **`msgflg`**, удобно рассматривать как восьмеричное. Оно задает права на выполнение операций и флаги.

Права на выполнение операций есть права на чтение из очереди и запись в нее (то есть на прием/посылку сообщений) для владельца, членов группы и прочих пользователей. В следующей таблице сведены возможные элементарные права и соответствующие им восьмеричные значения:

Права на операции	**Восьмеричное значение**
Чтение для владельца	0400
Запись для владельца	0200
Чтение для группы	0040
Запись для группы	0020
Чтение для остальных	0004
Запись для остальных	0002

В каждом конкретном случае нужная комбинация прав задается как результат побитного ИЛИ значений, соответствующих элементарным правам. Полная аналогия с правами доступа к файлам.

Флаги определены во включаемом файле **<sys/ipc.h>**. В следующей таблице сведены мнемонические имена флагов и соответствующие им восьмеричные

Флаг	Восьмеричное значение
IPC_CREAT	0001000
IPC_EXCL	0002000

Значение аргумента **msgflg** в целом является, следовательно, результатом побитного **ИЛИ** (операция | в языке C) прав на выполнение операций и флагов, например:

```
msqid = msgget(key, (IPC_CREAT | 0644));
msqid = msgget(key, (IPC_CREAT | IPC_EXCL | 0400));
```

Системный вызов вида

```
msqid = msgget(IPC_PRIVATE, msgflg);
```

приведет к попытке выделения нового идентификатора очереди сообщений и ассоциированной информации независимо от значения аргумента **msgflg**.

Попытка может быть неудачной только из-за превышения системного лимита на общее число очередей сообщений, задаваемого настраиваемым параметром **MSGMNI**.

При использовании флага **IPC_EXCL** в сочетании с **IPC_CREAT** системный вызов **msgget** завершается неудачей в том и только в том случае, если с указанным ключом **key** уже ассоциирован идентификатор.

Флаг **IPC_EXCL** необходим, чтобы предотвратить ситуацию, когда процесс полагает, что получил новый (уникальный) идентификатор очереди сообщений, в то время как это не так.

Иными словами, когда используются и **IPC_CREAT** и **IPC_EXCL**, при успешном завершении системного вызова обязательно возвращается новый идентификатор **msqid**.

В справочной статье по **msgget** описывается начальное значение ассоциированной структуры данных, формируемое при успешном завершении системного вызова. Там же содержится перечень условий, приводящих к ошибкам, и соответствующих им мнемонических имен для значений переменной **errno**.

Управление очередями сообщений

Управление очередями сообщений выполняется с помощью системного вызова **msgctl()**.

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

int msgctl(int msqid,              // идентификатор очереди сообщений
           int cmd,                // управляющее действие
           struct msqid_ds *buf); // пользовательская структура
```

При успешном завершении результат равен нулю.

В случае неудачи возвращается -1.

В качестве аргумента **msqid** должен выступать идентификатор очереди сообщений, предварительно полученный при помощи системного вызова **msgget**.

Управляющее действие определяется значением аргумента **cmd**.

Допустимых значений три:

IPC_STAT — поместить информацию о состоянии очереди, содержащуюся в структуре данных, ассоциированной с идентификатором **msqid**, в пользовательскую структуру, на которую указывает аргумент **buf**.

IPC_SET — в структуре данных, ассоциированной с идентификатором **msqid**, переустановить значения действующих идентификаторов пользователя и группы, прав на операции, максимально допустимого числа байт в очереди.

IPC_RMID — удалить из системы идентификатор **msqid**, ликвидировать очередь сообщений и ассоциированную с ней структуру данных.

Чтобы выполнить управляющее действие **IPC_SET** или **IPC_RMID**, процесс должен иметь действующий идентификатор пользователя, равный либо идентификаторам создателя или владельца очереди, либо идентификатору суперпользователя.

Чтобы выполнить действие **IPC_STAT**, достаточно права на чтение.

Операции над очередями сообщений

Для выполнения операций над очередями сообщений используются системные вызовы **msgsnd()** и **msgrcv()**. В **man msgop** синтаксис упомянутых системных вызовов описан так:

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

int     msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

Аргумент **msgp** представляет собой указатель на структуру, определяемую вызывающим как:

```
struct msgbuf {
    long mtype;       // тип сообщения, значение должно быть > 0
    char mtext[1];    // данные сообщения (char mtext[])
};
```

Член **mtext** является массивом (или другой структурой), размер которого определяется членом **msgsz** и является неотрицательным целым значением.

Разрешены сообщения нулевой длины (т.е. без члена **mtext**).

Поле **mtype** должно быть только положительным целым значением. Это значение используется процессом-получателем для выбора сообщения из очереди, используя **msgrcv()**.

Посылка сообщений

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

int msgsnd(int msqid,          // идентификатор очереди сообщ., полученный msgget()
           const void *msgp,   // тип посылаемого сообщения и его текст
           size_t msgsz,       // длина сообщения в байтах
           int msgflg);        // флаги операции
```

При успешном завершении системного вызова **msgsnd()** возвращается ноль.

В случае неудачи возвращается **-1**.

В качестве аргумента **msqid** должен выступать идентификатор очереди сообщений, предварительно полученный при помощи системного вызова **msgget()**.

msgp — указатель на структуру в области памяти пользователя, содержащую тип посылаемого сообщения и его текст.

msgsz — длина массива символов в структуре данных, указываемой аргументом **msgp**, то есть длину сообщения.

Максимально допустимый размер данного массива определяется системным параметром **MSGMAX**.

Значение поля **msg_qbytes** у ассоциированной структуры данных может быть уменьшено с предполагаемой по умолчанию величины **MSGMNB** при помощи управляющего действия **IPC_SET** системного вызова **msgctl**, однако впоследствии увеличить его может только суперпользователь.

Аргумент **msgflg** позволяет специфицировать выполнение над сообщением операцию с блокировкой — для этого флаг **IPC_NOWAIT** должен быть сброшен (**msgflg & IPC_NOWAIT == 0**).

Блокировка имеет место, если либо текущее число байт в очереди уже равно максимально допустимому значению для указанной очереди (то есть значению поля **msg_qbytes** или **MSGMNB**), либо общее число сообщений во всех очередях равно максимально допустимому системой (системный параметр **MSGTQL**).

Если в такой ситуации флаг **IPC_NOWAIT** установлен, системный вызов **msgsnd()** завершается неудачей и возвращает **-1**.

Прием сообщений

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

ssize_t msgrcv(int msqid,    // идентиф. очереди сообщений, полученный msgget()
               void *msgp,   // тип принимаемого сообщения и его текст
               size_t msgsz, // длина принимаемого сообщения в байтах !!!
               long msgtyp,  // фильтр (0 - все, >0 – конкретно, <0 – порог фильтра)
               int msgflg);  // флаги операции
```

При успешном завершении системного вызова **msgrcv()** результат равен числу принятых байт. В случае неудачи возвращается **-1**.

В качестве аргумента **msqid** должен выступать идентификатор очереди сообщений, предварительно полученный при помощи системного вызова **msgget()**.

msgp – указатель на структуру в области памяти пользователя, содержащую тип принимаемого сообщения и его текст.

msgsz – длина принимаемого сообщения. Можно указать, что в случае, если значение данного аргумента меньше, чем длина сообщения в массиве, должна возникать ошибка (**msgflg**).

msgtyp – используется для выбора из очереди первого сообщения определенного типа. Если значение аргумента равно нулю, запрашивается первое сообщение в очереди, если больше нуля – первое сообщение типа **msgtyp**, а если меньше нуля – первое сообщение наименьшего из типов, которые не превосходят абсолютной величины аргумента **msgtyp**.

Аргумент **msgflg** позволяет специфицировать выполнение над сообщением операции с блокировкой — для этого должен быть сброшен флаг **IPC_NOWAIT** (**msgflg & IPC_NOWAIT == 0**).

Блокировка имеет место, если в очереди сообщений нет сообщения с запрашиваемым типом (**msgtyp**).

Если флаг **IPC_NOWAIT** установлен и в очереди нет сообщения требуемого типа, системный вызов немедленно завершается неудачей.

Аргумент **msgflg** может также указывать, что системный вызов должен заканчиваться неудачей, если размер сообщения в очереди больше значения **msgsz**, для этого в данном аргументе должен быть сброшен флаг **MSG_NOERROR** (**msgflg & MSG_NOERROR == 0**).

Если флаг **MSG_NOERROR** установлен, сообщение обрезается до длины, указанной аргументом **msgsz**.

Для чтения первого сообщения в очереди с типом, отличным от **msgtyp**, используется флаг **MSG_EXCEPT** и **msgtyp** больше 0.

Пр ошибке вызова помимо возврата -1 устанавливается **errno**.

Семафоры System V

Семафоры System V (**semget(2)**, **semop(2)**, **semctl(2)**) — это классический (старый) программный интерфейс семафоров. Они позволяют взаимодействовать процессам, которые изменяют значения семафоров. Значение семафора — это целое число в диапазоне от 0 до 32767 (7FFF).

Поскольку во многих приложениях требуется более одного семафора, ОС UNIX System V предоставляет возможность создавать наборы семафоров. Их максимальный размер ограничен системным параметром **SEMMSL**.

Наборы семафоров создаются при помощи системного вызова **semget()**.

Процесс, выполнивший системный вызов **semget()**, становится владельцем/создателем набора семафоров. Он определяет, сколько будет семафоров в наборе, кроме того, он специфицирует первоначальные права на выполнение операций над набором для всех процессов, включая себя.

Процесс-создатель может уступить право собственности или изменить права на операции при помощи системного вызова **semctl()**, предназначенного для управления семафорами, однако на протяжении всего времени существования набора семафоров создатель остается создателем.

Другие процессы, обладающие соответствующими правами, для выполнения прочих управляющих действий также могут использовать системный вызов **semctl()**.

Над каждым семафором, принадлежащим набору, при помощи системного вызова **semop()** можно выполнить любую из трех операций:

- увеличить значение (требуется право на изменение);
- уменьшить значение (требуется право на изменение);
- дождаться обнуления. (достаточно права на чтение);

Чтобы увеличить значение семафора, системному вызову **semop()** следует передать требуемое число.

Системный вызов **semop()** оперирует не с отдельным семафором, а с набором семафоров, применяя к нему «массив операций». Массив содержит информацию о том, с какими семафорами нужно оперировать и каким образом.

Выполнение массива операций с точки зрения пользовательского процесса является неделимым действием. Это значит, во-первых, что если операции выполняются, то только все вместе и, во-вторых, что другой процесс не может получить доступ к промежуточному состоянию набора семафоров, когда часть операций из массива уже выполнилась, а другая часть еще не успела.

Операции могут сопровождаться флагами.

SEM_UNDO — операция выполняется в проверочном режиме, то есть требуется только узнать, можно ли успешно выполнить данную операцию.

IPC_NOWAIT — при сброшенном флаге (**IPC_NOWAIT == 0**) системный вызов **semop()** может быть приостановлен до тех пор, пока значение семафора, благодаря действиям другого процесса, не позволит успешно завершить операцию (ликвидация набора семафоров также приведет к завершению системного вызова).

Операционная система выполняет операции из массива по очереди, причем порядок не оговаривается.

Если очередная операция не может быть выполнена, то эффект предыдущих операций аннулируется. Если таковой оказалась операция с блокировкой, выполнение системного вызова приостанавливается. Если неудачу потерпела операция без блокировки, системный вызов немедленно завершается и возвращается значение -1 как признак ошибки. При этом внешней переменной **errno** присваивается код ошибки.

Использование семафоров

Перед тем как использовать семафоры (выполнять операции или управляющие действия), нужно создать набор семафоров с уникальным идентификатором и ассоциированной структурой данных. Уникальный идентификатор называется идентификатором множества семафоров (**semid**). Он используется для обращений к множеству и структуре данных.

С точки зрения реализации набор семафоров представляет собой массив структур. Каждая структура соответствует одному семафору и определяется (условно) следующим образом:

```
struct sem {          // Структура в ядре (скрыта)
    ushort semval;  // Значение семафора
    short  sempid;  // Идентификатор процесса, выполнявшего последнюю операцию
    ushort semncnt; // Число процессов, ожидающих увеличения значения семафора
    ushort semzcnt; // Число процессов, ожидающих обнуления значения семафора
};
```

Определение находится во включаемом файле **<sys/sem.h>**.

С каждым идентификатором набора семафоров ассоциирована структура данных, содержащая следующую информацию (поля указаны не все):

```
struct semid_ds {
    struct ipc_perm sem_perm; // Структура прав на выполнение операций
    struct sem *sem_base;     // Указатель на первый семафор в наборе (в ядре)
    ushort sem_nsems;         // Количество семафоров в наборе
    time_t sem_otime;         // Время последней операции
    time_t sem_ctime;         // Время последнего изменения
};
```

Это определение также находится во включаемом файле **<sys/sem.h>**.

Следует отметить, что поле **sem_perm** данной структуры использует в качестве шаблона структуру **ipc_perm**, общую для всех средств межпроцессной связи.

Системный вызов **semget()** аналогичен вызову **msgget().**

Вызов предназначен для получения нового или опроса существующего идентификатора, а нужное действие определяется значением аргумента **key**.

semget() терпит неудачу в тех же ситуациях, что и **msgget()**.

Единственное отличие этих вызовов друг от друга состоит в том, что при создании требуется посредством аргумента **nsems** указывать число семафоров в наборе.

После того как созданы набор семафоров с уникальным идентификатором и ассоциированная с ним структура данных, можно использовать системные вызовы **semop()** для операций над семафорами и **semctl()** для выполнения управляющих действий.

semget() — создание набора семафоров

Для создания набора семафоров служит системный вызов **semget()**.

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

int semget(key_t key,     // ключ
           int   nsems,   // число семафоров в наборе
           int   semflg); // флаги
```

Целочисленное значение, возвращаемое в случае успешного завершения, есть идентификатор набора семафоров (**semid**). В случае неудачи результат равен -1.

Смысл аргументов **key** и **semflg** тот же, что и у соответствующих аргументов системного вызова **msgget()**.

key — ключ.

Если значение **key** не равно **IPC_PRIVATE**, а **semflg** равно нулю, **key** можно использовать для получения идентификатора уже созданного набора семафоров.

Если значение **key** равно **IPC_PRIVATE** или с ключом **key** не связано ни одного существующего набора семафоров, а в **semflg** при этом задано **IPC_CREAT**, создаётся новый набор семафоров в количестве **nsems**.

Если в **semflg** одновременно указаны **IPC_CREAT** и **IPC_EXCL** и набор семафоров для **key** уже существует, то **semget()** завершается с ошибкой и **errno** будет присвоено значение **EEXIST** (такой же результат как с **O_CREAT | O_EXCL** у **open(2)**).

nsems задает число семафоров в наборе. Если запрашивается идентификатор существующего набора, значение **nsems** не должно превосходить числа семафоров в наборе.

Если целью вызова **semget()** является получение **semid** существующего набора, а не создание нового, то аргумент **nsems** может быть равен 0.

В противном случае **nsems** должен быть больше 0 и меньше или равен максимально допустимому количеству семафоров в наборе (**SEMMSL**).

Если набор семафоров уже существует, то проверяются права доступа.

Превышение системных параметров **SEMMNI**, **SEMMNS** и **SEMMSL** при попытке создать новый набор всегда ведет к неудачному завершению.

SEMMNI определяет максимально допустимое число уникальных идентификаторов наборов семафоров в системе;

SEMMNS определяет максимальное общее число семафоров в системе;

SEMMSL определяет максимально допустимое число семафоров в одном наборе.

semctl() — управление семафорами Sytem V

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

int semctl(int semid,  // идентификатор набора семафоров, выданный semget()
           int semnum, // номер семафора из набора
           int cmd,    // операция (команда)
           ...);       // опциональный параметр arg
```

Вызов **semctl()** выполняет операцию, определённую аргументом **cmd**, над набором семафоров System V, указанном в **semid**, или над семафором с номером **semnum** из этого набора.

semnum — номер семафора в множестве.

Семафоры нумеруются, начиная с 0.

Данный вызов имеет три или четыре аргумента, в зависимости от значения **cmd**.

Если аргументов четыре, то четвертый аргумент **arg** имеет тип **union semun**. В вызывающей программе это объединение должно быть определено следующим образом:

```
union semun {
    int               val;    // значение для SETVAL
    struct semid_ds *buf;    // буфер для IPC_STAT, IPC_SET
    unsigned short  *array;  // массив для GETALL, SETALL
    struct seminfo  *__buf;  // буфер для IPC_INFO (есть только в Linux) */
};
```

Структура данных **semid_ds** определена в **<sys/sem.h>** следующим образом:

```
struct semid_ds {
    struct ipc_perm sem_perm;  // владелец и права
    time_t          sem_otime; // время последней операции semop
    time_t          sem_ctime; // время последнего изменения
    unsigned long   sem_nsems; // кол-во семафоров в наборе
};
```

Структура **ipc_perm** определена следующим образом (значения полей устанавливаются с помощью **IPC_SET**):

```
struct ipc_perm {
    key_t          __key; // ключ, передаваемый в semget(2)
    uid_t          uid;   // эффективный UID владельца
    gid_t          gid;   // эффективный GID владельца
    uid_t          cuid;  // эффективный UID создателя
    gid_t          cgid;  // эффективный GID создателя
    unsigned short mode;  // права
    unsigned short __seq; // порядковый номер
};
```

Возможные значения аргумента `cmd`

Операции, общие для всех IPC-механизмов

`IPC_STAT` — поместить информацию о состоянии множества семафоров, содержащуюся в структуре данных, ассоциированной с идентификатором **`semid`**, в пользовательскую структуру, на которую указывает **`arg.buf`**.

Копирует информацию из структуры данных ядра, связанной с **`semid`**, в структуру **`semid_ds`**, по адресу **`arg.buf`**. Аргумент **`semnum`** игнорируется. Нужны права на чтение набора семафоров.

`IPC_SET` — в структуре данных, ассоциированной с идентификатором **`semid`**, переустановить значения действующих идентификаторов пользователя и группы, а также прав на операции.

Записывает значения некоторых полей структуры **`semid_ds`**, на которую указывает **`arg.buf`**, в структуру данных ядра, связанную с этим набором семафоров.

Обновляются следующие поля структуры: **`sem_perm.uid`**, **`sem_perm.gid`** и (младшие 9 значащих битов) **`sem_perm.mode`**. При этом также обновляется поле **`sem_ctime`**.

Эффективный **`UID`** вызывающего процесса должен совпадать с идентификатором владельца набора семафоров **`sem_perm.uid`** или его создателя **`sem_perm.cuid`**, либо вызывающий должен иметь расширенные права. Аргумент **`semnum`** игнорируется.

`IPC_RMID` — удалить из системы идентификатор **`semid`**, ликвидировать множество семафоров и ассоциированную с ним структуру данных.

Немедленно удаляет набор семафоров, пробуждая все процессы, заблокированные в вызове **`semop(2)`**, при этом возвращается сообщение об ошибке, а **`errno`** присваивается значение **`EIDRM`**).

Эффективный UID вызывающего процесса должен совпадать с UID создателя или владельца набора, либо вызывающий должен иметь расширенные права. Аргумент **`semnum`** игнорируется.

Операции, специфические именно для семафоров

```
struct sem {        // Структура в ядре (скрыта)
    ushort semval;  // Значение семафора
    short  sempid;  // Идентификатор процесса, выполнявшего последнюю операцию
    ushort semncnt; // Число процессов, ожидающих увеличения значения семафора
    ushort semzcnt; // Число процессов, ожидающих обнуления значения семафора
};
```

GETALL / **GETVAL** — прочитать значения всех семафоров в наборе/конкретного с номером **semnum** и поместить их в массив **arg.array**. Возвращают текущее значение **semval**. Аргумент **semnum** в случае **GETALL** игнорируется. Нужны права на чтение набора семафоров.

GETNCNT — получить число процессов, ожидающих увеличения значение семафора, и выдать его в качестве результата. Возвращает значение **semncnt** для семафора с номером **semnum** из набора. (число процессов, ожидающих увеличения значения **semval** в семафора с номером **semnum**). Нужны права на чтение набора семафоров.

GETPID — получить идентификатор процесса, последним выполнявшего операцию над семафором, и выдать его в качестве результата. Возвращает значение **sempid** для семафора с номером **semnum** из набора. Это **PID** процесса, который последним выполнял операцию с этим семафором. Нужны права на чтение набора семафоров.

GETZCNT — получить число процессов, ожидающих обнуления значения семафора, и выдать его в качестве результата. Возвращает значение **semzcnt** для семафора с номером **semnum** (количество процессов, ожидающих, когда значение **semval** для семафора с номером **semnum** станет равным 0). Нужны права на чтение набора семафоров.

SETALL — установить значения всех семафоров множества равными значениям элементов массива, на который указывает **arg.array**.

Устанавливает значение **semval** всех семафоров набора, используя **arg.array** и изменяя также поле **sem_ctime** структуры **semid_ds**, связанной с набором.

Записи **undo** (**semop(2)**) очищаются для изменённых семафоров во всех процессах.

Если изменения значений семафоров приводят к отмене блокировки в вызовах **semop(2)**, выданных другими процессами, то эти процессы пробуждаются. Аргумент **semnum** игнорируется.

Вызывающему процессу нужны права на запись в набор семафоров.

SETVAL — установить значение семафора равным **arg.val**.

Устанавливает значение **semval** равным **arg.val** для **semnum**-го семафора из набора, изменяя также поле **sem_ctime** в структуре **semid_ds**, связанной с этим набором. Записи **undo** очищаются для изменённых семафоров во всех процессах.

Если изменения значений семафоров приводят к отмене блокировки в вызове **semop(2)** других процессов, то эти процессы пробуждаются. Аргумент **semnum** игнорируется.

Вызывающему процессу нужны права на запись в набор семафоров.

Результат системного вызова **semctl()** в случае успешного завершения зависит от выполняемого управляющего действия.

Как правило он равен 0, но действия **GETAAL**, **GETVAL**, **GETPID**, **GETNCNT** и **GETZCNT** являются исключениями.

При возникновении ошибки всегда возвращается -1.

Команды, которые есть только в Linux

IPC_INFO (есть только в Linux)

Возвращает параметры и информацию о системных ограничениях семафоров в структуре, указанной в **arg.__buf**. Данная структура имеет тип **seminfo**, который определён в **<sys/sem.h>**, если определён макрос тестирования свойств **_GNU_SOURCE**:

```
struct  seminfo {
    int semmap;  // количество записей в карте семафоров*
    int semmni;  // максимальное количество наборов семафоров
    int semmns;  // максимальное количество семафоров во всех наборах семафоров
    int semmnu;  // максимальное количество структур undo в системе*
    int semmsl;  // максимальное количество семафоров в наборе
    int semopm;  // максимальное количество операция для semop(2)
    int semume;  // максимальное количество записей undo на процесс*
    int semusz;  // размер struct sem_undo
    int semvmx;  // максимальное значение семафора
    int semaem;  // максимальное значение, которое может быть записано для
                 // регулирования семафора (SEM_UNDO)
};

* -- не используется в ядре
```

Значения **semmsl**, **semmns**, **semopm** и **semmni** можно изменить через **/proc/sys/kernel/sem**, (см. man proc(5)).

SEM_INFO (есть только в Linux)

Возвращает структуру **seminfo**, содержащую такую же информацию что и для **IPC_INFO**, за исключением того, что в следующих полях возвращается информация о системных ресурсах, потребляемых семафорами:

semusz — количество наборов семафоров, существующих в системе;

semaem — общее количество семафоров во всех наборах семафоров в системе.

SEM_STAT (есть только в Linux)

Возвращает структуру **semid_ds** как для **IPC_STAT**.

Аргумент **semid** в данном случае указывает не индекс семафора, а индекс во внутреннем массиве ядра, который хранит информацию о всех наборах семафоров в системе.

SEM_STAT_ANY (есть только в Linux, начиная с Linux 4.17)

Возвращает структуру **seminfo**, содержащую информацию как у **SEM_STAT**. Однако **sem_perm.mode** не проверяется на доступность чтения для **semid**, поэтому эту операцию может выполнять пользователь (как и любой пользователь, который может прочитать эту же информацию из **/proc/sysvipc/sem**).

Чтобы выполнить управляющее действие **IPC_SET** или **IPC_RMID**, процесс должен иметь действующий идентификатор пользователя, равный либо идентификаторам создателя или владельца очереди, либо идентификатору суперпользователя.

Для выполнения управляющих действий **SETVAL** и **SETALL** требуется право на изменение, а для выполнения остальных действий — право на чтение.

semop()/semtimedop() — операции над множествами семафоров

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

int semop(int                  semid,  // идентификатор набора семафоров
          struct sembuf       *sops,   // структура с параметрами операции
          size_t               nsops); //

// _GNU_SOURCE
int semtimedop(int             semid,
               struct sembuf *sops,
               size_t nsops,
               const struct timespec *timeout);
```

С каждым семафором в наборе семафоров System V связаны следующие значения:

```
unsigned short  semval;   // значение семафора
unsigned short  semzcnt;  // # ожидает ноль
unsigned short  semncnt;  // # ожидает увеличения
pid_t           sempid;   // PID процесса, выполнявшегося последним
```

Вызов **semop( )** производит операции над выбранными семафорами из набора семафоров **semid**.

```
int semop(int             semid,  // semid
          struct sembuf *sops,   // адрес первой структуры с операциями
          size_t          nsops); // количество структур в массиве
```

Каждый из **nsops** элементов в массиве, указанном в **sops**, является структурой, задающей операцию, которую требуется выполнить над отдельным семафором набора **semid**.

Элементы этой структуры имеют тип **struct sembuf**, который содержит поля:

```
struct sembuf {
    unsigned short sem_num;  // номер семафора
    short          sem_op;   // операция над семафором
    short          sem_flg;  // флаги операции
}
```

Флаги в **sem_flg** могут иметь следующие значения:

IPC_NOWAIT – если какая-либо операция, для которой задан этот флаг, не может быть успешно выполнена, системный вызов завершается неудачей, причем ни у одного из семафоров не будет изменено значение

SEM_UNDO – данный флаг задает проверочный режим выполнения операции – он предписывает аннулировать ее результат даже в случае успешного завершения системного вызова **semop( )**. Иными словами, блокировка всех операций (в том числе и тех, для которых задан флаг **SEM_UNDO**) выполняется обычным образом, но когда наконец все операции могут быть успешно выполнены, операции с флагом **SEM_UNDO** игнорируются.

Набор операций из sops выполняется в порядке появления в массиве и является атомарным.

То есть выполняются или все операции, или ни одной. Поведение системного вызова при обнаружении невозможности немедленного выполнения операций зависит от наличия флага **IPC_NOWAIT** в полях **sem_flg** отдельных операций, как это описано далее.

Каждая операция выполняется над семафором из набора с индексом **sem_num**, где первый семафор имеет номер 0. Есть три типа операций, различающихся значением **sem_op**:

sem_op > 0 — если значение **sem_op** является положительным целым числом, то оно добавляется к значению семафора **semval**.

Эта операция выполняется всегда и не переводит нить в режим ожидания. Вызывающий процесс должен иметь права на изменение набора семафоров.

Если при этом для операции стоит флаг **SEM_UNDO**, то система вычитает значение **sem_op** из значения коррекции (**semadj**) семафора.

sem_op == 0 — если значение **sem_op** равно нулю, то процесс должен иметь права на чтение набора семафоров.

Это операция «*ожидания нуля*» — если **semval** равно нулю, то операция может выполнится сразу.

В противном случае, если в поле семафора **sem_flg** указан флаг **IPC_NOWAIT**, то **semop()** завершается с ошибкой и **errno** присваивается значение **EAGAIN** и ни одна операция из **sops** не выполняется. Если же флаг **IPC_NOWAIT** сброшен **semzcnt** (счётчик нитей, ожидающих, пока значение семафора не сравняется с нулём) увеличивается на единицу, а нить переходит в режим ожидания пока не случится одно из:

- значение **semval** станет равным 0, тогда значение **semzcnt** уменьшается.

- набор семафоров удалится. В этом случае **semop()** завершается с ошибкой, а **errno** присваивается значение **EIDRM**.

- вызывающая нить получит сигнал. В этом случае значение **semncnt** уменьшается, **semop()** завершается с ошибкой, а **errno** присваивается значение **EINTR**.

sem_op < 0 — если значение **sem_op** меньше нуля, то процесс должен иметь права на изменение набора семафоров.

Если значение **semval** больше или равно абсолютному значению **sem_op**, то операция может выполнятся сразу, при этом абсолютное значение **sem_op** вычитается из **semval**.

Если для этой операции установлен флаг **SEM_UNDO**, система добавляет абсолютное значение **sem_op** к значению коррекции (**semadj**) семафора.

Если абсолютное значение **sem_op** больше **semval** и в **sem_flg** указан **IPC_NOWAIT** вызов **semop()** завершается с ошибкой, **errno** присваивается значение **EAGAIN** и ни одна операция из **sops** не выполняется.

Если **IPC_NOWAIT** сброшен, **semncnt** (счётчик нитей, ожидающих увеличения значения семафора) увеличивается на единицу, а нить переходит в режим ожидания пока не случится одно из:

- **semval** становится больше или равно абсолютному значению **sem_op** — в этом случае операция продолжается как описано выше.

- набор семафоров удалится из системы — в этом случае **semop()** завершается с ошибкой, а **errno** присваивается значение **EIDRM**.

- вызывающая нить получит сигнал — в этом случае значение **semncnt** уменьшается и **semop()** завершается с ошибкой, а **errno** присваивается значение **EINTR**.

При успешном выполнении значение **sempid** для каждого семафора, указанного в массиве, на который указывает **sops**, устанавливается равным идентификатору вызывающего процесса.

Также **semid_ds.sem_otime** присваивается значение текущего времени.

Значения **semval**, **sempid**, **semzcnt** и **semnct** семафора можно получить с помощью соответствующих вызовов **semctl(2)**.

Ограничения

SEMOPM — максимальное количество операций, разрешённых для одного вызова **semop()**. До версии Linux 3.19, значение по умолчанию было 3. Начиная с Linux 3.19, значение по умолчанию равно 500.

В Linux это ограничение можно прочитать и изменить через третье поле **/proc/sys/kernel/sem**. Замечание: это ограничение не должно превышать 1000, так как есть риск, что **semop(2)** завершится с ошибкой из-за фрагментации памяти ядра при выделении памяти при копировании массива **sops**.

SEMVMX — максимально допустимое значение **semval** — зависит от реализации (32767).

Реализация не накладывает существенных ограничений на максимальное значение (**SEMAEM**), на которое можно изменить значение семафора при выходе, максимальное количество системных структур откатываемых операций (**SEMMNU**) и максимальное количество элементов отката системных параметров на процесс.

Пример

В следующем фрагменте кода используется **semop( )** для атомарного ожидания момента, когда значение семафора 0 станет равным нулю и последующего увеличения значения семафора на единицу.

```
#include <sys/sem.h>
...
struct sembuf sops[2]; // массив операций для semop( )
int semid;

// код для установки semid не показан
...

sops[0].sem_num = 0;     // применяем к семафору 0
sops[0].sem_op  = 0;     // ждём значения, равного 0
sops[0].sem_flg = 0;

sops[1].sem_num = 0;     // применяем к семафору 0
sops[1].sem_op  = 1;     // увеличиваем значение на 1
sops[1].sem_flg = 0;

if (semop(semid, sops, 2) == -1) {
    perror("semop");
    exit(EXIT_FAILURE);
}
```

Пример создания semid

Следующий пример получает уникальный ключ семафора с помощью функции **ftok()**, а затем получает идентификатор семафора, связанного с этим ключом, с помощью функции **semget()** (первый вызов также проверяет, существует ли семафор).

Если семафор не существует, программа создает его (второй вызов **semget()**).

При создании семафора для процесса постановки в очередь программа пытается создать один семафор с правами чтения/записи для всех. При этом используется флаг **IPC_EXCL**, который приводит к сбою функции **semget()**, если семафор уже существует.

После создания семафора программа использует вызовы **semctl()** и **semop()** для его инициализации значениями в массиве **sbuf**.

Число процессов, которые могут выполняться одновременно без постановки в очередь, изначально устанавливается равным 2.

Последний вызов **semget()** создает идентификатор семафора, который позже может использоваться в программе.

Процессы, которые получают **semid** без его создания, проверяют, что **sem_otime** не равно нулю, чтобы быть уверенными, что процесс создания завершил инициализацию **semop()**.

Последний вызов **semop()** захватывает семафор и ожидает, пока он не освободится. Опция **SEM_UNDO** освобождает семафор при выходе из процесса, ожидает, пока одновременно не будет запущено менее двух процессов.

```
#include <stdio.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <errno.h>
#include <stdlib.h>
...
key_t          semkey;
int            semid;
struct sembuf sbuf;

union semun {                  // arg
    int               val;    //
    struct semid_ds *buf;     //
    unsigned short  *array; //
} arg;

struct semid_ds ds;          // сюда будем возвращать информацию из semctl()
...
// Получим уникальный ключ для семафора
if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
    perror("IPC error: ftok");
    exit(1);
}
```

```
// Получим идентификатор набора семафоров, связанного с этим ключом
if ((semid = semget(semkey, 0, 0)) == -1) { // проверим, существует ли семафор
    // Семафор не существует – создаем с правами 666
    if ((semid = semget(semkey, 1,              // просим один семафор
                        IPC_CREAT | IPC_EXCL | // создаем новый для экс. владения
                        S_IRUSR   | S_IWUSR |  // WR для UID
                        S_IRGRP   | S_IWGRP |  // WR для GID
                        S_IROTH   | S_IWOTH)   // WR для остальных
                        ) != -1) {
        // Если получили semid инициализируем семафор
        arg.val      = 0; // для команды SETVAL значение semval устанавливаем в 0
        sbuf.sem_num = 0; // семафор у нас один и индекс его 0
        sbuf.sem_op  = 2; // число процессов, к-рые могут выполняться одновр-енно
        sbuf.sem_flg = 0;
        if (semctl(semid, 0, SETVAL, arg) == -1 ||  // инициализируем в 0
            semop(semid, &sbuf, 1) == -1) {         // поднимаем до 2
            perror("IPC error: semop"); exit(1);
        }
    } else if (errno == EEXIST) { // если таки внезапно существует
        if ((semid = semget(semkey, 0, 0)) == -1) { // пытаемся получ. к нему доступ
            perror("IPC error 1: semget"); exit(1);
        }
        goto check_init;                            // и, получив, идем на проверку
    } else {
        perror("IPC error 2: semget"); exit(1);
    }
} else {
```

```
    // Семафор существует, проверим, завершил ли semid инициализацию
    // Приложение здесь может в цикле ждать вместо выхода
    check_init:
    arg.buf = &ds; // сюда будем возвращать информацию из semctl()
    if (semctl(semid, 0, IPC_STAT, arg) < 0) {
        perror("IPC error 3: semctl");
        exit(1);
    }
    if (ds.sem_otime == 0) { // если инициализация завершена, здесь будет не ноль
        perror("IPC error 4: semctl");
        exit(1);
    }
}
...
sbuf.sem_num = 0;        // у нас один семафор и его индекс 0
sbuf.sem_op  = -1;       // вычитаем 1 из semval
sbuf.sem_flg = SEM_UNDO;
if (semop(semid, &sbuf, 1) == -1) {
    perror("IPC Error: semop");
    exit(1);
}
```

Семафоры POSIX

Обзор

Семафоры POSIX предоставляют более простой и продуманный интерфейс чем семафоры System V, с другой стороны, семафоры POSIX не так широко распространены (особенно в старых системах), по сравнению с семафорами System V.

Семафоры POSIX позволяют синхронизировать свою работу как процессам, так и нитям.

Семафор представляет собой целое число, значение которого никогда не будет меньше нуля.

Над семафорами выполняются две операции — увеличение значения семафора на единицу **`sem_post(3)`** и уменьшение значения семафора на единицу **`sem_wait(3)`**.

Если значение семафора равно нулю, то операция **`sem_wait(3)`** блокирует работу до тех пор, пока значение не станет больше нуля.

Есть два вида семафоров POSIX — именованные семафоры и безымянные семафоры.

При компоновке следует указывать опцию **`-pthread`**.

Именованные семафоры

Именованные семафоры отличают по именам вида **/имя** — строка (с **null** в конце) длиной до **NAME_MAX-4** (т. е., 251) символов, состоящая из начальной косой черты и одного или нескольких символов (символ косой черты не допускается).

Два процесса могут работать с одним семафором указав его имя в **sem_open(3)**.

Функция **sem_open(3)** создаёт новый именованный семафор или открывает существующий. После открытия семафора с ним можно работать посредством **sem_post(3)** и **sem_wait(3)**.

Когда процесс закончил использовать семафор, его можно закрыть с помощью **sem_close(3)**. Когда все процессы закончили использовать семафор, его можно удалить из системы с помощью **sem_unlink(3)**.

Именованные семафоры POSIX располагаются в ядре. Пока семафор не удалён с помощью **sem_unlink(3)**, он остаётся в системе до её выключения.

Безымянные семафоры (семафоры в памяти)

Безымянные семафоры не имеют имени. Семафор размещается в области памяти, которая доступна нескольким нитям (общий семафор для нитей) или процессам (общий семафор для процессов).

Общий семафор для нитей размещается в области памяти, которая доступна из нитей процесса, например в глобальной переменной.

Общий семафор для процессов должен размещаться в области общей памяти (например, в сегменте общей памяти System V, созданной с помощью **shmget(2)**, или в объекте общей памяти POSIX, созданном с помощью **shm_open(3)**).

Перед началом использования безымянный семафор должны быть проинициализирован с помощью **sem_init(3)**. После этого с ним можно работать через **sem_post(3)** и **sem_wait(3)**. Если семафор больше не нужен, то семафор нужно уничтожить с помощью **sem_destroy(3)**, причем до освобождения выделенной для него памяти.

sem_init() — инициализирует безымянный семафор

```
#include <semaphore.h>

int sem_init(sem_t         *sem,     // адрес семафора
             int            pshared, // доступность из разных процессов
             unsigned int   value);  // начальное значение
```

Функция **sem_init()** инициализирует безымянный семафор по адресу, указанному в **sem**.

value – начальное значение семафора.

pshared – будет ли данный семафор доступен в разных процессах или только в нитях.

Если значение **pshared** равно 0, то семафор будет коллективно использоваться в нитях одного процесса, и должен располагаться по адресу, который доступен из всех нитей (например, глобальная переменная или переменная, динамически выделенная из кучи).

Если значение **pshared** не равно нулю, то семафор будет коллективно использоваться несколькими процессами, и должен располагаться в области общей памяти (**shm_open(3)**, **mmap(2)** и **shmget(2)**).

Поскольку потомок, создаваемый **fork(2)**, наследует отображение памяти родителя, то ему также доступен и семафор.

Любой процесс, имеющий доступ к области общей памяти, может обращаться к семафору с помощью **sem_post(3)**, **sem_wait(3)** и т. п. функций.

Инициализация семафора, который уже был инициализирован, приводит к непредсказуемым результатам.

Возвращаемое значение

При успешном выполнении **sem_init()** возвращается 0.

При ошибке значение семафора не изменяется, возвращается **-1**, а в **errno** указывается причина ошибки.

Ошибки

EINVAL значение **value** превышает **SEM_VALUE_MAX**.

ENOSYS значение **pshared** не равно нулю, то система не поддерживает семафоры, коллективно используемые процессами.

sem_destroy — уничтожает безымянный семафор

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

Функция **sem_destroy( )** уничтожает безымянный семафор, расположенный по адресу **sem**.

С помощью sem_destroy( ) должны уничтожаться только семафоры, которые были инициализированы с помощью sem_init(3).

Уничтожение семафора, заблокированного другим процессом или нитью **в sem_wait(3)**, приводит к непредсказуемым последствиям.

Использование уничтоженного семафора (не инициализированного повторно с помощью sem_init(3)) приводит к непредсказуемым результатам.

Возвращаемое значение

При успешном выполнении **sem_destroy( )** возвращается 0.

При ошибке значение семафора не изменяется, возвращается **-1**, а в **errno** указывается причина ошибки.

Ошибки

EINVAL значение **sem** не является корректным для семафора.

sem_open() — инициализирует и открывает именованный семафор

```
#include <fcntl.h>           // константы O_*
#include <sys/stat.h>        // константы для mode
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag); // имя и флаги
sem_t *sem_open(const char *name, int oflag,  // имя и флаги
                mode_t mode,                  //
                unsigned int value);          //
```

Связывается при указании параметра **-pthread**.

Функция **sem_open()** создаёт новый семафор POSIX или открывает существующий семафор. Семафору присваивается имя **name**.

В аргументе **oflag** задаются флаги, которые управляют работой вызова (определения значений флагов можно получить подключив **<fcntl.h>**).

O_CREAT – семафор создаётся, если ещё не существует.

Владельцем (ID пользователя) семафора устанавливается эффективный ID пользователя вызывающего процесса.

Владельцем группы (ID группы) устанавливается эффективный ID группы вызывающего процесса.

Если в **oflag** указаны **O_CREAT** и **O_EXCL** одновременно и семафор с заданным **name** уже существует, то возвращается ошибка.

Если в **oflag** указано **O_CREAT**, то должны быть заданы ещё два аргумента.

В аргументе **mode** задаются права для нового семафора, подобно **open(2)** (символические определения бит прав (режим доступа) можно получить подключив **<sys/stat.h>**).

Настройки прав маскируются маской процесса (**umask(2)**).

Права чтения и записи должны быть заданы для каждого класса пользователей, которым нужен доступ к семафору.

В аргументе **value** задаётся начальное значение нового семафора.

Если указан **O_CREAT** и семафор с заданным **name** существует, то **mode** и **value** игнорируются.

Возвращаемое значение

При успешном выполнении **sem_open()** возвращает адрес нового семафора.

Этот адрес используется при вызове других функций, работающих с семафорами.

При ошибке **sem_open()** возвращает **SEM_FAILED**, а в **errno** записывается номер ошибки.

Ошибки

EACCES семафор существует, но вызывающий не имеет прав для его открытия.

EEXIST в **oflag** указаны **O_CREAT** и **O_EXCL**, но семафор **name** уже существует.

EINVAL значение value было больше **SEM_VALUE_MAX**.

EINVAL в **name** есть только «/», и нет других символов.

EMFILE было достигнуто ограничение по количеству открытых файловых дескрипторов на процесс.

ENAMETOOLONG слишком длинное значение аргумента name.

ENFILE достигнуто максимальное количество открытых файлов в системе.

ENOENT в **oflag** не указан **O_CREAT** и семафор **name** не существует; или указан **O_CREAT**, но **name** указан в некорректной форме.

ENOMEM Недостаточно памяти.

sem_close() — закрывает именованный семафор

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

Функция **sem_close()** закрывает именованный семафор, на который указывает **sem**, позволяя освободить все ресурсы, которые система выделила под семафор вызывающему процессу.

Возвращаемое значение

При успешном выполнении **sem_close()** возвращается 0.

При ошибке значение семафора не изменяется, возвращается **-1**, а в **errno** указывается причина ошибки.

Ошибки

EINVAL значение **sem** для семафора не является корректным.

sem_unlink() — удаляет именованный семафор

```
#include <semaphore.h>

int sem_unlink(const char *name);
```

Функция **sem_unlink()** удаляет именованный семафор, на который ссылается **name**.
Имя семафора удаляется немедленно.
Семафор уничтожается после того, как все остальные процессы, в которых он открыт, закроют его.

Возвращаемое значение

При успешном выполнении **sem_unlink()** возвращается 0.
При ошибке значение семафора не изменяется, возвращается **-1**, а в **errno** указывается причина ошибки.

Ошибки

EACCES вызывающий не имеет прав для удаления этого семафора.
ENAMETOOLONG слишком длинное значение аргумента name.
ENOENT семафор с указанным name отсутствует.

sem_wait(), sem_trywait(), sem_timedwait() — блокирует семафор

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);

int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

`sem_timedwait(): _POSIX_C_SOURCE >= 200112L (#define _POSIX_C_SOURCE 200112L)`

Функция **`sem_wait()`** уменьшает (блокирует) семафор, на который указывает **`sem`**.

Если значение семафор больше нуля, то выполняется уменьшение и функция сразу возвращает управление.

Если значение семафора равно нулю, то вызов блокируется до тех пор, пока не станет возможным выполнить уменьшение (т. е., значение семафора не станет больше нуля), или пока не вызовется обработчик сигнала.

Функция **`sem_trywait()`** подобна **`sem_wait()`**, за исключением того, что если уменьшение нельзя выполнить сразу, то вызов не блокируется, а завершается с ошибкой (**`errno`** становится равным **`EAGAIN`**).

Функция **sem_timedwait()** подобна **sem_wait()**, за исключением того, что в **abs_timeout** задаётся ограничение по количеству времени, на которое вызов должен заблокироваться, если уменьшение невозможно выполнить сразу.

Аргумент **abs_timeout** указывает на структуру, в которой задаётся абсолютное время ожидания в секундах и наносекундах, начиная с эпохи, 1970-01-01 00:00:00 +0000 (UTC).

Эта структура определена следующим образом:

```
struct timespec {
    time_t tv_sec;  // секунды
    long   tv_nsec; // наносекунды [0 .. 999999999]
};
```

Если на момент вызова время ожидания уже истекло и семафор нельзя заблокировать сразу, то **sem_timedwait()** завершается с ошибкой просрочки (**errno** становится равным **ETIMEDOUT**).

Если операцию можно выполнить сразу, то **sem_timedwait()** никогда не завершится с ошибкой просрочки, независимо от значения **abs_timeout**.

Кроме того, в этом случае корректность **abs_timeout** даже не проверяется.

Возвращаемое значение

При успешном выполнении все функции возвращают 0.

При ошибке значение семафора не изменяется, возвращается **-1**, а в **errno** указывается причина ошибки.

Ошибки

EINTR вызов был прерван обработчиком сигнала.
EINVAL значение **sem** не является корректным для семафора.

В **sem_trywait()** может возникать следующая дополнительная ошибка:

EAGAIN операция не может быть выполнена без блокировки (т. е., значение семафора равно нулю).

В **sem_timedwait()** дополнительно могут возникать следующие ошибки:

EINVAL значение **abs_timeout.tv_nsecs** меньше 0, или больше или равно 1000 миллионов.
ETIMEDOUT истёк период ожидания в вызове раньше возможности блокировки семафора.

sem_post() — разблокирует семафор

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Функция **sem_post()** увеличивает (разблокирует) семафор, на который указывает **sem**.

Если значение семафора после этого становится больше нуля, то другой процесс или нить заблокированная в вызове **sem_wait(3)**, проснётся и заблокирует семафор.

Возвращаемое значение

При успешном выполнении **sem_post()** возвращается 0.

При ошибке значение семафора остаётся неизменным, возвращается **-1**, а в **errno** содержится код ошибки.

Ошибки

EINVAL значение **sem** не является корректным для семафора.

EOVERFLOW превышено максимально допустимое значение для семафора.

Пример использования функций семафоров POSIX

Программа работает с безымянным семафором. Она ожидает два аргумента командной строки. В первом аргументе задаётся значение в секундах, которое используется в будильнике для генерации сигнала **SIGALRM**. Обработчик выполняет **sem_post(3)** для увеличения семафора, которого ждёт в **main()** вызов **sem_t imedwait()**.

Во втором аргументе задаётся период ожидания в секундах для **sem_timedwait()**.

В двух разных запусках программы происходит следующее:

```
$ ./a.out 2 3
About to call sem_timedwait()
sem_post() из обработчика
sem_timedwait() выполнена успешно
$ ./a.out 2 1
About to call sem_timedwait()
истекло время ожидания sem_timedwait()
```

Исходный код программы

```
#include <unistd.h>     // STDOUT_FILENO
#include <stdio.h>
#include <stdlib.h>     // exit()
#include <semaphore.h>
#include <time.h>
#include <assert.h>
#include <errno.h>
#include <signal.h>
```

```
sem_t sem;

#define handle_error(msg) do { perror(msg); exit(EXIT_FAILURE); } while (0)

static void handler(int sig) {
    write(STDOUT_FILENO, "sem_post() из обработчика\n", 24);
    if (sem_post(&sem) == -1) {
        write(STDERR_FILENO, "ошибка sem_post()\n", 18);
        _exit(EXIT_FAILURE);
}   }

int main(int argc, char *argv[]) {

    struct sigaction sa; //
    struct timespec  ts; //
    int              s;

    if (argc != 3) {
        fprintf(stderr, "Использование: %s <alarm-secs> <wait-secs>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (sem_init(&sem, 0, 0) == -1) { // только для нитей, val == 0
        handle_error("sem_init()");
    }
    // установка обработчика SIGALRM; таймер будильника в argv[1]
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
```

```
    if (sigaction(SIGALRM, &sa, NULL) == -1) {
        handle_error("sigaction");
    }

    alarm(atoi(argv[1])); // !!! реально это может быть и не число !!!

    // относительный интервал как текущее время + кол-во секунд из argv[2]
    if (clock_gettime(CLOCK_REALTIME, &ts) == -1) { // _POSIX_C_SOURCE >= 199309L
        handle_error("clock_gettime()");
    }
    ts.tv_sec += atoi(argv[2]);

    printf("в main() вызывается sem_timedwait()\n");
    while ((s = sem_timedwait(&sem, &ts)) == -1 && errno == EINTR) {
        continue; // перезапускаем, если прервано обработчиком
    }

    // проверяем  что произошло
    if (s == -1) {
        if (errno == ETIMEDOUT)
            printf("истекло время ожидания sem_timedwait()\n");
        else
            perror("sem_timedwait()");
    } else {
        printf("sem_timedwait() выполнена успешно\n");
    }
    exit((s == 0) ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

sem_getvalue() — возвращает значение семафора

```
#include <semaphore.h>

int sem_getvalue(sem_t *sem, int *sval);
```

Функция **sem_getvalue()** помещает текущее значение семафора, заданного в **sem**, в виде целого, на которое указывает **sval**.

Если один или более процессов или нитей заблокированы в ожидании блокировки семафора с помощью **sem_wait(3)**, то в этом случае в POSIX.1 разрешено возвращать два варианта значения **sval** – 0 или отрицательное число, чьё абсолютное значение равно количеству процессов и нитей заблокированных в **sem_wait(3)**. В Linux используется первый вариант.

Возвращаемое значение

При успешном выполнении **sem_getvalue()** возвращается 0.

При ошибке значение семафора не изменяется, возвращается **-1**, а в **errno** указывается причина ошибки.

Ошибки

EINVAL значение **sem** не является корректным для семафора.