Лабораторная работа №10
«Процессы. Разделяемая память»
Лабораторная работа №11
«Процессы. Семафоры»
Лабораторная работа №10, 11 add
«Процессы. Отображаемые в память файлы»

На данных занятиях следует продолжить изучение основных средств, позволяющих осуществлять взаимодействие между процессами — *IPC* (*межпроцессное взаимодействие*, *interprocess communication*). Рассмотрим средства, предназначенные как для обмена информацией — *разделяемая память* (*shared memory*), *отображаемые в память файлы* (*memory mapped files*), так и для синхронизации процессов — *семафоры* (*semaphores*).

Основные задания

Задание №1

Напишите две программы, обменивающиеся информацией при помощи разделяемой памяти. Синхронизируйте их совместную работу с помощью сигналов.

Одна программа при запуске получает требуемый размер буфера, создает сегмент разделяемой памяти соответствующего размера, а затем создает процесс-потомок и загружает в него бинарный образ второй программы. Первая программа получает у пользователя количество вводимых данных, считывает их и помещает в сегмент общей памяти. Затем сообщает об этом второму процессу с помощью сигнала (выберите для этого подходящий сигнал) и «засыпает» - ждет ответа второго процесса. Второй процесс «просыпается» получив сигнал от первого процесса, вычисляет сумму записанных в сегмент общей памяти чисел и также записывает ее в сегмент общей памяти, посылает сигнал первому процессу и засыпает. Первый процесс выводит полученную сумму в стандартный поток вывода. Процессы попеременно работают таким образом, пока пользователь не прекратит этот процесс. Для этого следует ввести признак прекращения работы с клавиатуры. Первый процесс посылает сигнал о завершении работы второму процессу, удаляет сегмент общей памяти и сам завершается.

Напишите ОДНУ версию программы (самостоятельно выберите вариант программы из предложенных ниже):

a) идентификатор общей области памяти передается второму процессу с помощью аргументов командной строки;
b) идентификатор общей области памяти передается второму процессу с помощью дополнительной информации сигнала, инициализирующего работу;
c) процессы синхронизируют свою работу с помощью ранее рассмотренных средств синхронизации потоков, размещенных в общей области памяти (из рассмотренных средств в общей области памяти можно разместить мьютексы, условные переменные, блокировки чтения / записи). Для этого варианта средства синхронизации должны быть настроены для использования в разных процессах. В случае использования мьютекса для синхронизации работы приложения можно предусмотреть средства «*восстановления*» мьютекса в случае, если захвативший его процесс преждевременно закончит работу. Для этого посмотрите *man*-страницы для функций

`pthread_mutex_getrobust()`, `pthread_mutex_setrobust()` и `pthread_mutex_consistent()`.

Задание №2

Модифицируйте программы из **Задания №1**, синхронизировав их работу при помощи семафоров. Решите задачу в следующей модификации: есть два процесса — серверный, который создает общие области памяти, семафоры и ожидает данные для проведения вычислений, и клиентский, который запускается после серверного независимо от него, находит общую область памяти и семафоры, получает данные от пользователя, забирает вычисленное значение и выводит его на экран. Клиент и сервер должны договориться о правилах создания ключей для средств межпроцессного взаимодействия. Нужно продумать механизм остановки работы сервера — чтобы средства взаимодействия процессов были удалены из системы.

Задание №3 (Дополнительное)

Организуйте взаимодействие двух процессов с помощью отображаемых в память файлов. Модифицируйте соответствующим образом программы **Задания №1**.

Рекомендации по выполнению

Общие сведения о System V IPC

Кроме сигналов существуют и другие механизмы передачи информации между процессами. Часть этих механизмов, впервые появившихся в *UNIX System V* и впоследствии перешедших во все современные версии операционной системы *UNIX*, получила общее название *System V IPC* (*IPC – interprocess communications*). В группу *System V IPC* входят: *очереди сообщений*, *разделяемая память* и *семафоры*. Эти средства организации взаимодействия процессов обладают схожим интерфейсом для выполнения своих основных операций. На данном занятии мы рассмотрим такие механизмы, как *разделяемая память* и *семафоры*.

Пространство имен. Функция ftok()

Все средства связи из *System V IPC* являются средствами связи с непрямой адресацией. Для организации взаимодействия неродственных процессов с помощью средства связи с непрямой адресацией необходимо, чтобы это средство связи имело имя.

Множество всех возможных имен для объектов какого-либо вида принято называть *пространством имен* соответствующего вида объектов. Для всех объектов из *System V IPC* таким пространством имен является множество значений целочисленного типа данных – `key_t` – ключа. Программисту не разрешено напрямую присваивать значение ключа, это значение задается опосредованно: через комбинацию имени какого-либо файла, уже существующего в файловой системе, и небольшого целого числа – например, номера экземпляра средства связи. Получение значения ключа из этих двух компонентов осуществляется функцией `ftok()`.

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *pathname, char proj);
```

Функция `ftok` служит для преобразования имени существующего файла и

небольшого целого числа, например, порядкового номера экземпляра средств связи, в ключ *System V IPC*.

Параметр `pathname` должен являться указателем на имя существующего файла, доступного для процесса, вызывающего функцию. Параметр `proj` – это небольшое целое число, характеризующее экземпляр средства связи.

В случае невозможности генерации ключа функция возвращает отрицательное значение, в противном случае она возвращает значение сгенерированного ключа. Тип данных `key_t` обычно представляет собой *32*-битовое целое.

Следует подчеркнуть три важных момента, связанных с использованием имени файла для получения ключа:

- необходимо указывать имя файла, который уже существует в файловой системе и для которого процесс имеет право доступа на чтение;
- указанный файл должен сохранять свое положение на диске до тех пор, пока все процессы, участвующие во взаимодействии, не получат ключ *System V IPC*;
- задание имени файла, как одного из компонентов для получения ключа, ни в коем случае не означает, что информация, передаваемая с помощью ассоциированного средства связи, будет располагаться в этом файле. Информация будет храниться внутри адресного пространства операционной системы, а заданное имя файла лишь позволяет различным процессам генерировать идентичные ключи.

Дескрипторы System V IPC

Ядро операционной системы хранит информацию обо всех средствах *System V IPC*, используемых в системе, вне контекста пользовательских процессов. При создании нового средства связи или получении доступа к уже существующему процесс получает неотрицательное целое число – *дескриптор* (*идентификатор*) этого средства связи, которое однозначно идентифицирует его во всей вычислительной системе. Этот дескриптор должен передаваться в качестве параметра всем системным вызовам, осуществляющим дальнейшие операции над соответствующим средством *System V IPC*.

Разделяемая память

Рассмотрим наиболее простое средство межпроцессной коммуникации *System V IPC* — *разделяемую память* (*shared memory*). Операционная система может позволить нескольким процессам совместно использовать некоторую область адресного пространства.

Все средства связи *System V IPC* требуют предварительных инициализирующих действий для организации взаимодействия процессов.

Создание области разделяемой памяти

Для создания области разделяемой памяти с определенным ключом или доступа по ключу к уже существующей области применяется системный вызов `shmget()`. Существует два варианта его использования для создания новой области разделяемой памяти.

- *Первый способ.* В качестве значения ключа системному вызову поставляется значение, сформированное функцией `ftok()` для некоторого имени файла и номера экземпляра области разделяемой памяти. В качестве флагов поставляется комбинация прав доступа к создаваемому сегменту и флага `IPC_CREAT`. Если сегмент для данного ключа еще не существует, то система будет пытаться создать его с указанными правами доступа. Если же он уже существовал, то мы просто получим его дескриптор. Возможно добавление к этой комбинации флагов флага `IPC_EXCL`. Этот флаг гарантирует нормальное завершение системного вызова только в том случае,

если сегмент действительно был создан (т. е. ранее он не существовал), если же сегмент существовал, то системный вызов завершится с ошибкой, и значение системной переменной `errno`, описанной в файле `errno.h`, будет установлено в `EEXIST`.

- *Второй способ*. В качестве значения ключа указывается специальное значение `IPC_PRIVATE`. Использование значения `IPC_PRIVATE` всегда приводит к попытке создания нового сегмента разделяемой памяти с заданными правами доступа и с ключом, который не совпадает со значением ключа ни одного из уже существующих сегментов и который не может быть получен с помощью функции `ftok()` ни при одной комбинации ее параметров. Наличие флагов `IPC_CREAT` и `IPC_EXCL` в этом случае игнорируется.

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

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

Системный вызов `shmget` предназначен для выполнения операции доступа к сегменту разделяемой памяти и, в случае его успешного завершения, возвращает дескриптор *System V IPC* для этого сегмента (целое неотрицательное число, однозначно характеризующее сегмент внутри вычислительной системы и использующееся в дальнейшем для других операций с ним).

Параметр `key` является ключом *System V IPC* для сегмента, т. е. фактически его именем из пространства имен *System V IPC*. В качестве значения этого параметра может использоваться значение ключа, полученное с помощью функции `ftok()`, или специальное значение `IPC_PRIVATE`. Использование значения `IPC_PRIVATE` всегда приводит к попытке создания нового сегмента разделяемой памяти с ключом,который не совпадает со значением ключа ни одного из уже существующих сегментов и который не может быть получен с помощью функции `ftok()` ни при одной комбинации ее параметров.

Параметр `size` определяет размер создаваемого или уже существующего сегмента в байтах. Если сегмент с указанным ключом уже существует, но его размер не совпадает с указанным в параметре `size`, констатируется возникновение ошибки.

Параметр `shmflg` – *флаги* – играет роль только при создании нового сегмента разделяемой памяти и определяет права различных пользователей при доступе к сегменту, а также необходимость создания нового сегмента и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции *побитовое или* – " `|` ") следующих предопределенных значений и восьмеричных прав доступа:

- `IPC_CREAT` – если сегмента для указанного ключа не существует, он должен быть создан;
- `IPC_EXCL` – применяется совместно с флагом `IPC_CREAT`. При совместном их использовании и существовании сегмента с указанным ключом, доступ к сегменту не производится и констатируется ошибочная ситуация, при этом переменная `errno`, описанная в файле `<errno.h>`, примет значение `EEXIST`;
- `0400` – разрешено чтение для пользователя, создавшего сегмент;
- `0200` – разрешена запись для пользователя, создавшего сегмент;
- `0040` – разрешено чтение для группы пользователя, создавшего сегмент;
- `0020` – разрешена запись для группы пользователя, создавшего сегмент;

- `0004` – разрешено чтение для всех остальных пользователей;
- `0002` – разрешена запись для всех остальных пользователей.

Системный вызов возвращает значение дескриптора *System V IPC* для сегмента разделяемой памяти при нормальном завершении и значение `-1` при возникновении ошибки.

Получение доступа к разделяемой памяти

Доступ к созданной области разделяемой памяти в дальнейшем обеспечивается ее дескриптором, который вернет системный вызов `shmget()`. Доступ к уже существующей области также может осуществляться двумя способами:

- Если мы знаем ее ключ, то, используя вызов `shmget()`, можем получить ее дескриптор. В этом случае нельзя указывать в качестве составной части флагов флаг `IPC_EXCL`, а значение ключа, естественно, не может быть `IPC_PRIVATE`. Права доступа игнорируются, а размер области должен совпадать с размером, указанным при ее создании.
- Либо мы можем воспользоваться тем, что дескриптор *System V IPC* действителен в рамках всей операционной системы, и передать его значение от процесса, создавшего разделяемую память, текущему процессу. Отметим, что при создании разделяемой памяти с помощью значения `IPC_PRIVATE` – это единственно возможный способ.

После получения дескриптора необходимо включить область разделяемой памяти в адресное пространство текущего процесса. Это осуществляется с помощью системного вызова `shmat()`. При нормальном завершении он вернет адрес разделяемой памяти в адресном пространстве текущего процесса. Дальнейший доступ к этой памяти осуществляется с помощью обычных средств языка программирования.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Системный вызов `shmat` предназначен, кроме всего прочего, для включения области разделяемой памяти в адресное пространство текущего процесса. Для полного описания обращайтесь к *Manual*.

Параметр `shmid` является дескриптором *System V IPC* для сегмента разделяемой памяти, т. е. значением, которое вернул системный вызов `shmget()` при создании сегмента или при его поиске по ключу.

В качестве параметра `shmaddr` в рамках мы всегда будем передавать значение `NULL`, позволяя операционной системе самой разместить разделяемую память в адресном пространстве нашего процесса.

Параметр `shmflg` у нас будет принимать только два значения: `0` – для осуществления операций чтения и записи над сегментом и `SHM_RDONLY` – если мы хотим только читать из него. При этом процесс должен иметь соответствующие права доступа к сегменту.

Системный вызов возвращает адрес сегмента разделяемой памяти в адресном пространстве процесса при нормальном завершении и значение `-1` при возникновении ошибки.

Отключение совместного доступа к общей области памяти

После окончания использования разделяемой памяти процесс может уменьшить размер своего адресного пространства, исключив из него эту область с помощью системного вызова `shmdt()`. В качестве параметра системный вызов `shmdt()` требует указать адрес начала области разделяемой памяти в адресном пространстве процесса, т. е. значение, которое вернул системный вызов `shmat()`, поэтому данное значение следует сохранять на протяжении всего времени использования разделяемой памяти.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt(const void *shmaddr);
```

Системный вызов `shmdt` предназначен для исключения области разделяемой памяти из адресного пространства текущего процесса.

Параметр `shmaddr` является адресом сегмента разделяемой памяти,т. е. значением, которое вернул системный вызов `shmat()`.

Системный вызов возвращает значение `0` при нормальном завершении и значение `-1` при возникновении ошибки.

Команды ipcs и ipcrm

Созданная область разделяемой памяти сохраняется в операционной системе даже тогда, когда нет ни одного процесса, включающего ее в свое адресное пространство. Без специальных указаний она автоматически не удаляется после завершения созданного или использующего ее процесса. Конечно, можно создавать для каждого нового сеанса работы новый сегмент разделяемой памяти, но количество ресурсов в системе не безгранично. Следует удалять неиспользуемые ресурсы *System V IPC*. Это можно сделать как с помощью команд операционной системы, так и с помощью системных вызовов. Для того чтобы удалять ресурсы *System V IPC* из командной строки, следует использовать две команды, `ipcs` и `ipcrm`.

Команда `ipcs` выдает информацию обо всех средствах *System V IPC*, существующих в системе, для которых пользователь обладает правами на чтение: областях разделяемой памяти, семафорах и очередях сообщений.

```
ipcs [-asmq] [-tclup]
ipcs [-smq] -i id
ipcs -h
```

Команда `ipcs` предназначена для получения информации о средствах *System V IPC*, к которым пользователь имеет право доступа на чтение.

Опция `-i` позволяет указать идентификатор ресурсов. Будет выдаваться только информация для ресурсов, имеющих этот идентификатор.

Виды *IPC* ресурсов могут быть заданы с помощью следующих опций:

- `-s` для семафоров;
- `-m` для сегментов разделяемой памяти;
- `-q` для очередей сообщений;
- `-a` для всех ресурсов (по умолчанию).

Опции [-tclup] используются для изменения состава выходной информации. По умолчанию для каждого средства выводятся его ключ, идентификатор *IPC*, идентификатор владельца, права доступа и ряд других характеристик. Применение опций позволяет вывести:

- -t времена совершения последних операций над средствами *IPC*;
- -p идентификаторы процесса, создавшего ресурс, и процесса, совершившего над ним последнюю операцию;
- -c идентификаторы пользователя и группы для создателя ресурса и его собственника;
- -l системные ограничения для средств *System V IPC*;
- -u общее состояние *IPC* ресурсов в системе.

Опция -h используется для получения краткой справочной информации.

Из всей выводимой информации нас будут интересовать только *IPC* идентификаторы для созданных нами средств. Эти идентификаторы будут использоваться в команде ipcrm, позволяющей удалить необходимый ресурс из системы. Для удаления сегмента разделяемой памяти эта команда имеет вид:

```
ipcrm -m <IPC идентификатор>
```

Следует удалять созданные сегменты разделяемой памяти из операционной системы, особенно во время отладки приложения.

```
ipcrm [-m | -q | -s] id
```

Команда ipcrm предназначена для удаления ресурса *System V IPC* из операционной системы. Параметр id задает *IPC* идентификатор для удаляемого ресурса, параметр -m используется для сегментов разделяемой памяти, параметр -q – для очередей сообщений, параметр -s – для семафоров.

Для более полного изучения особенностей применения команд ipcs и ipcrm следует обратиться к справочным страницам man.

Освобождения общей области памяти

Для того, чтобы удалить область разделяемой памяти из системы можно воспользоваться системным вызовом shmctl(). Этот системный вызов позволяет полностью ликвидировать область разделяемой памяти в операционной системе по заданному дескриптору средства *IPC*, если, конечно, у вас хватает для этого полномочий. Системный вызов shmctl() позволяет выполнять и другие действия над сегментом разделяемой памяти, но их мы рассматривать не будет.

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

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

Системный вызов shmctl предназначен для получения информации об области разделяемой памяти, изменения ее атрибутов и удаления из системы. Данное описание не является полным описанием системного вызова. Для изучения полного описания обращайтесь к *Manual*.

Мы будем пользоваться системным вызовом `shmctl` только для удаления области разделяемой памяти из системы. Параметр `shmid` является дескриптором *System V IPC* для сегмента разделяемой памяти, т. е. значением, которое вернул системный вызов `shmget()` при создании сегмента или при его поиске по ключу.

В качестве параметра `cmd` мы, в основном, всегда будем передавать значение `IPC_RMID` – команду для удаления сегмента разделяемой памяти с заданным идентификатором. Параметр `buf` в этом случае не используется, поэтому мы всегда будем подставлять туда значение `NULL`.

Системный вызов возвращает значение `0` при нормальном завершении и значение `-1` при возникновении ошибки.

Разделяемая память и системные вызовы fork(), exec() и функция exit()

При выполнении системного вызова `fork()` все области разделяемой памяти, размещенные в адресном пространстве процесса, наследуются порожденным процессом.

При выполнении системных вызовов `exec()` и функции `exit()` все области разделяемой памяти, размещенные в адресном пространстве процесса, исключаются из его адресного пространства, но продолжают существовать в операционной системе.

Пример

Создадим две программы. Первая программа создает сегмент общей памяти. После создания и подключения сегмента программа выводит в стандартный поток вывода его идентификатор, записывает в этот сегмент общей памяти данные и ожидает, пока пользователь на нажмет `<Enter>`. Эта задержка позволит другому процессу подключить общий сегмент памяти и прочитать оттуда данные.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/shm.h>

#define SHMEM_SIZE 4096
#define SH_MESSAGE "Hello World!\n"

int main(void) {
    int shm_id;
    char *shm_buf;
    int shm_size;
    struct shmid_ds ds;

    shm_id = shmget(IPC_PRIVATE, SHMEM_SIZE,
                    IPC_CREAT | IPC_EXCL | 0600);
    if (shm_id == -1) {
        fprintf(stderr, "shmget() error\n");
        return EXIT_FAILURE;
    }

    shm_buf = (char *) shmat(shm_id, NULL, 0);
    if (shm_buf == (char *) -1) {
        fprintf(stderr, "shmat() error\n");
```

```
        return EXIT_SUCCESS;
    }

    shmctl(shm_id, IPC_STAT, &ds);

    shm_size = ds.shm_segsz;
    if (shm_size < strlen(SH_MESSAGE)) {
        fprintf(stderr, "error: segsize=%d\n", shm_size);
        return EXIT_SUCCESS;
    }

    strcpy(shm_buf, SH_MESSAGE);

    printf("ID: %d\n", shm_id);
    printf("Press <Enter> to exit...");
    fgetc(stdin);

    shmdt(shm_buf);
    shmctl(shm_id, IPC_RMID, NULL);

    return EXIT_SUCCESS;
}
```

Откомпилируйте, запустите программу. Пока не готова вторая программа просмотрите список совместно используемых сегментов памяти с их ключами и идентификаторами (команда `ipcs -m`). Завершите программу и удалите незакрытый сегмент общей памяти с помощью команды `ipcrm -m shmid`. Еще раз посмотрите список доступных сегментов общей памяти.

Напишите вторую программу, которая будет считывать информацию, записанную в общей памяти первым процессом. Так как у процессов нет договоренности о выделении общего ключа для сегмента памяти, то можно организовать взаимодействие процессов, передав второй программе-клиенту идентификатор сегмента с помощью аргумента командной строки.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>

int main(int argc, char ** argv) {
    int shm_id;
    char *shm_buf;

    if (argc < 2) {
        fprintf(stderr, "Too few arguments\n");
        exit(EXIT_FAILURE);
    }

    shm_id = atoi(argv[1]);
    shm_buf = (char *) shmat(shm_id, 0, 0);
    if (shm_buf == (char *) -1) {
        fprintf(stderr, "shmat() error\n");
```

```
        return EXIT_SUCCESS;
    }

    printf("Message: %s\n", shm_buf);
    shmdt(shm_buf);

    return EXIT_SUCCESS;
}
```

Проверьте совместную работу этих двух программ. Для проверки лучше использовать два терминальных окна — в каждом окне будет работать своя программа. Посмотрите состояние общей памяти перед работой, во время работы и после работы программ.

Семафоры для синхронизации процессов

Очень часто бывает необходимо синхронизировать работу процессов для их корректного взаимодействия через разделяемую память. Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали семафоры, концепцию которых описал *Дейкстра* (*Dijkstra*) в 1965 году. Следует отметить, что набор операций над семафорами *System V IPC* отличается от классического набора операций `{P, V}`, предложенного *Дейкстрой*. Он включает три операции:

- `A(S, n)` – увеличить значение семафора `S` на величину `n`;
- `D(S, n)` – пока значение семафора `S < n`, процесс блокируется. Далее `S = S - n`;
- `Z(S)` – процесс блокируется до тех пор, пока значение семафора `S` не станет равным `0`.

Изначально все *IPC*-семафоры инициируются нулевым значением.

Классической операции `P(S)` соответствует операция `D(S,1)`, а классической операции `V(S)` соответствует операция `A(S,1)`. Аналогом ненулевой инициализации семафоров *Дейкстры* значением `n` может служить выполнение операции `A(S,n)` сразу после создания семафора `S`, с обеспечением атомарности создания семафора и ее выполнения посредством другого семафора. Таким образом, классические семафоры реализуются через семафоры *System V IPC*. Обратное не является верным. Используя операции `P(S)` и `V(S)`, мы не сумеем реализовать операцию `Z(S)` .

IPC-семафоры являются средством связи с непрямой адресацией, требуют инициализации для организации взаимодействия процессов и специальных действий для освобождения системных ресурсов по его окончании. Пространством имен *IPC*-семафоров является множество значений ключа, генерируемых с помощью функции `ftok()`. Для совершения операций над семафорами системным вызовам в качестве параметра передаются *IPC*- дескрипторы семафоров, однозначно идентифицирующих их во всей вычислительной системе, а вся информация о семафорах располагается в адресном пространстве ядра операционной системы. Это позволяет организовывать через семафоры взаимодействие процессов, даже не находящихся в системе одновременно.

Создание массива семафоров

В целях экономии системных ресурсов операционная система *Linux* позволяет создавать не по одному семафору для каждого конкретного значения ключа, а связывать с ключом целый массив семафоров (в *Linux* – до `500` семафоров в массиве, хотя это количество

может быть уменьшено системным администратором). Для создания массива семафоров, ассоциированного с определенным ключом, или доступа по ключу к уже существующему массиву используется системный вызов `semget()`, являющийся аналогом системного вызова `shmget()` для разделяемой памяти. Этот вызов возвращает значение *IPC*-дескриптора для массива массива семафоров. При этом применяются способы создания и доступа, аналогичные способам для разделяемой памяти. Вновь созданные семафоры инициируются нулевым значением.

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

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

Системный вызов `semget` предназначен для выполнения операции доступа к массиву *IPC*-семафоров и, в случае ее успешного завершения, возвращает дескриптор *System V IPC* для этого массива (целое неотрицательное число, однозначно характеризующее массив семафоров внутри вычислительной системы и использующееся в дальнейшем для других операций с ним).

Параметр `key` является ключом *System V IPC* для массива семафоров, т. е. фактически его именем из пространства имен *System V IPC*. В качестве значения этого параметра может использоваться значение ключа, полученное с помощью функции `ftok()`, или специальное значение `IPC_PRIVATE`. Использование значения `IPC_PRIVATE` всегда приводит к попытке создания нового массива семафоров с ключом, который не совпадает со значением ключа ни одного из уже существующих массивов и не может быть получен с помощью функции `ftok()` ни при одной комбинации ее параметров.

Параметр `nsems` определяет количество семафоров в создаваемом или уже существующем массиве. В случае, если массив с указанным ключом уже имеется, но его размер не совпадает с указанным в параметре `nsems`, констатируется возникновение ошибки.

Параметр `semflg` – *флаги* – играет роль только при создании нового массива семафоров и определяет права различных пользователей при доступе к массиву, а также необходимость создания нового массива и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции *побитовое или* – "`|`") следующих предопределенных значений и восьмеричных прав доступа:

- `IPC_CREAT` — если массива для указанного ключа не существует, он должен быть создан
- `IPC_EXCL` — применяется совместно с флагом `IPC_CREAT`. При совместном их использовании и существовании массива с указанным ключом, доступ к массиву не производится и констатируется ошибка, при этом переменная `errno`, описанная в файле `<errno.h>`, примет значение `EEXIST`
- `0400` — разрешено чтение для пользователя, создавшего массив
- `0200` — разрешена запись для пользователя, создавшего массив
- `0040` — разрешено чтение для группы пользователя, создавшего массив
- `0020` — разрешена запись для группы пользователя, создавшего массив
- `0004` — разрешено чтение для всех остальных пользователей
- `0002` — разрешена запись для всех остальных пользователей

Вновь созданные семафоры инициируются нулевым значением.

Системный вызов возвращает значение дескриптора *System V IPC* для массива семафоров при нормальном завершении и значение `-1` при возникновении ошибки.

Выполнение операций над семафорами

Для выполнения операций `A`, `D` и `Z` над семафорами из массива используется системный вызов `semop()`, обладающий довольно сложной семантикой. Это достаточно сложный метод. Его можно применять не только для выполнения всех трех операций, но еще и для нескольких семафоров в массиве *IPC*-семафоров одновременно. Для правильного использования этого вызова необходимо выполнить следующие действия:

1. Определиться, для каких семафоров из массива предстоит выполнить операции. Необходимо иметь в виду, что все операции реально совершаются только перед успешным возвращением из системного вызова, т.е. если вы хотите выполнить операции `A(S1,5)` и `Z(S2)` в одном вызове и оказалось, что `S2 != 0`, то значение семафора `S1` не будет изменено до тех пор, пока значение `S2` не станет равным `0`. Порядок выполнения операций в случае, когда процесс не переходит в состояние ожидание, не определен. Так, например, при одновременном выполнении операций `A(S1,1)` и `D(S2,1)` в случае `S2 > 1` неизвестно, что произойдет раньше – уменьшится значение семафора `S2` или увеличится значение семафора `S1`. Если порядок выполнения важен, то лучше применить несколько вызовов вместо одного.
2. После того как вы определились с количеством семафоров и совершаемыми операциями, необходимо завести в программе массив из элементов типа `struct sembuf` с размерностью, равной определенному количеству семафоров (если операция совершается только над одним семафором, можно, естественно, обойтись просто переменной). Каждый элемент этого массива будет соответствовать операции над одним семафором.
3. Заполнить элементы массива. В поле `sem_flg` каждого элемента нужно занести значение `0` (другие значения флагов мы рассматривать не будем). В поля `sem_num` и `sem_op` следует занести номера семафоров в массиве *IPC* семафоров и соответствующие коды операций. Семафоры нумеруются, начиная с `0`. Если у вас в массиве всего один семафор, то он будет иметь номер `0`. Операции кодируются так:
   - для выполнения операции `A(S,n)` значение поля `sem_op` должно быть равно `n`;
   - для выполнения операции `D(S,n)` значение поля `sem_op` должно быть равно `–n`;
   - для выполнения операции `Z(S)` значение поля `sem_op` должно быть равно `0`.
4. В качестве второго параметра системного вызова `semop()` указать адрес заполненного массива, а в качестве третьего параметра – ранее определенное количество семафоров, над которыми совершаются операции.

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

int semop(int semid, struct sembuf *sops, int nsops);
```

Системный вызов `semop` предназначен для выполнения операций `A`, `D` и `Z`. Это не полное описание системного вызова. Для полного описания обращайтесь к *Manual*.

Параметр `semid` является дескриптором *System V IPC* для набора семафоров, т. е. значением, которое вернул системный вызов `semget()` при создании набора семафоров или

при его поиске по ключу.

Каждый из `nsops` элементов массива, на который указывает параметр `sops`, определяет операцию, которая должна быть совершена над каким-либо семафором из массива *IPC* семафоров, и имеет тип структуры `struct sembuf`, в которую входят следующие переменные:

- `short sem_num` — номер семафора в массиве *IPC* семафоров (нумерация начинается с `0`);
- `short sem_op` — выполняемая операция;
- `short sem_flg` — флаги для выполнения операции. Мы всегда будем считать эту переменную равной `0`.

Значение элемента структуры `sem_op` определяется следующим образом:

- для выполнения операции `A(S,n)` значение должно быть равно `n`;
- для выполнения операции `D(S,n)` значение должно быть равно `-n`;
- для выполнения операции `Z(S)` значение должно быть равно `0`.

Семантика системного вызова подразумевает, что все операции будут в реальности выполнены над семафорами только перед успешным возвращением из системного вызова. Если при выполнении операций `D` или `Z` процесс перешел в состояние ожидания, то он может быть выведен из этого состояния при возникновении следующих особых ситуаций:

- массив семафоров был удален из системы;
- процесс получил сигнал, который должен быть обработан.

В этом случае происходит возврат из системного вызова с констатацией ошибочной ситуации.

Системный вызов возвращает значение `0` при нормальном завершении и значение `-1` при возникновении ошибки.

Удаление набора семафоров

Массив семафоров может продолжать существовать в системе и после завершения использовавших его процессов, а семафоры будут сохранять свое значение. Это может привести к некорректному поведению программ, предполагающих, что семафоры были только что созданы и, следовательно, имеют нулевое значение. Необходимо удалять семафоры из системы перед запуском или программ или перед их завершением. Для удаления семафоров можно воспользоваться рассмотренными ранее командами `ipcs` и `ipcrm`. Команда `ipcrm` в этом случае должна иметь вид

```
ipcrm -s <IPC идентификатор>
```

Для этой же цели мы можем применять системный вызов `semctl()`, который умеет выполнять и другие операции над массивом семафоров, но их мы рассматривать не будем.

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

int semctl(int semid, int semnum, int cmd, union semun arg);
```

Системный вызов `semctl` предназначен для получения информации о массиве *IPC* семафоров, изменения его атрибутов и удаления его из системы. Данное описание не является полным описанием этого системного вызова. Для изучения полного описания обращайтесь к *Manual*.

Мы будем применять системный вызов `semctl` только для удаления массива семафоров из системы. Параметр `semid` является дескриптором *System V IPC* для массива семафоров, т. е. значением, которое вернул системный вызов `semget()` при создании массива или при его поиске по ключу.

В качестве параметра `cmd` мы всегда будем передавать значение `IPC_RMID` – команду для удаления сегмента разделяемой памяти с заданным идентификатором. Параметры `semnum` и `arg` для этой команды не используются, поэтому мы всегда будем подставлять вместо них значение `0`.

Если какие-либо процессы находились в состоянии ожидания для семафоров из удаляемого массива при выполнении системного вызова `semop()`, то они будут разблокированы и вернутся из вызова `semop()` с индикацией ошибки.

Системный вызов возвращает значение `0` при нормальном завершении и значение `-1` при возникновении ошибки.

Пример

Использует семафоры для совместной работы двух программ предыдущего примера.

Измененная версия первой программы:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

#define SHMEM_SIZE 4096
#define SH_MESSAGE "Hello, World!\n"

#define SEM_KEY 2015
#define SHM_KEY 2015

union semnum {
    int val;
    struct semid_ds *buf;
    unsigned short * array;
} sem_arg;


int main(void) {
    int shm_id, sem_id;
    char * shm_buf;
    int shm_size;
    struct shmid_ds ds;
    struct sembuf sb[1];
    unsigned short sem_vals[1];
```

```
shm_id = shmget(SHM_KEY, SHMEM_SIZE,
                IPC_CREAT | IPC_EXCL | 0600);
if (shm_id == -1) {
    fprintf(stderr, "shmget() error\n");
    return EXIT_FAILURE;
}

sem_id = semget(SEM_KEY, 1,
                0600 | IPC_CREAT | IPC_EXCL);
if (sem_id == -1) {
    fprintf(stderr, "semget() error\n");
    return EXIT_FAILURE;
}

printf("Semaphore: %d\n", sem_id);
sem_vals[0] = 1;
sem_arg.array = sem_vals;

if (semctl(sem_id, 0, SETALL, sem_arg) == -1) {
    fprintf(stderr, "semctl() error\n");
    return EXIT_FAILURE;
}

shm_buf = (char *) shmat(shm_id, NULL, 0);
if (shm_buf == (char *) -1) {
    fprintf(stderr, "shmat() error\n");
    return EXIT_FAILURE;
}

shmctl(shm_id, IPC_STAT, &ds);
shm_size = ds.shm_segsz;
if (shm_size < strlen(SH_MESSAGE)) {
    fprintf(stderr, "error: segsize=%d\n", shm_size);
    return EXIT_FAILURE;
}

strcpy(shm_buf, SH_MESSAGE);

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

sb[0].sem_num = 0;
sb[0].sem_flg = SEM_UNDO;

sb[0].sem_op = -1;
semop(sem_id, sb, 1);

sb[0].sem_op = -1;
semop(sem_id, sb, 1);

semctl(sem_id, 1, IPC_RMID, sem_arg);
shmdt(shm_buf);
```

```
    shmctl(shm_id, IPC_RMID, NULL);

    return EXIT_SUCCESS;
}
```

Измененная версия второй программы:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

#define SEM_KEY 2015
#define SHM_KEY 2015


int main(int argc, char ** argv) {
    int shm_id, sem_id;
    char * shm_buf;
    struct sembuf sb[1];

    shm_id = shmget(SHM_KEY, 1, 0600);
    if (shm_id == -1) {
        fprintf(stderr, "shmget() error\n");
        return EXIT_FAILURE;
    }

    sem_id = semget(SEM_KEY, 1, 0600);
    if (sem_id == -1) {
        fprintf(stderr, "semget() error\n");
        return EXIT_FAILURE;
    }

    shm_buf = (char *) shmat(shm_id, 0, 0);
    if (shm_buf == (char *) -1) {
        fprintf(stderr, "shmat() error\n");
        return EXIT_FAILURE;
    }

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

    sb[0].sem_num = 0;
    sb[0].sem_flg = SEM_UNDO;

    sb[0].sem_op = 1;
    semop(sem_id, sb, 1);

    shmdt(shm_buf);
```

```
    return EXIT_SUCCESS;
}
```

Посмотрите на состояние семафоров до, во время и после работы этих двух программ.

Отображаемые в память файлы

С появлением понятия виртуальной памяти, когда физические размеры памяти перестали играть роль сдерживающего фактора в развитии вычислительных систем, стало возможным отображать файлы непосредственно в адресное пространство процессов. То есть, появилась возможность работать с файлами как с обычной памятью, заменив выполнение базовых операций над ними с помощью системных вызовов на использование операций обычных языков программирования. Файлы, содержимое которых отображается непосредственно в адресное пространство процессов, получили название файлов, *отображаемых в память* (*memory mapped файлов*). Такое отображение может быть осуществлено не только для всего файла в целом, но и для некоторой его части.

С точки зрения программиста работа с такими файлами выглядит следующим образом:

- Отображение файла из пространства имен в адресное пространство процесса происходит в два этапа: сначала выполняется отображение в дисковое пространство, а уже затем из дискового пространства в адресное. Поэтому вначале файл необходимо открыть, используя обычный системный вызов `open()`.
- Вторым этапом является отображение файла целиком или частично из дискового пространства в адресное пространство процесса. Для этого используется системный вызов `mmap()`. Файл после этого можно закрыть, выполнив системный вызов `close()`, так как необходимая информация о расположении файла на диске уже сохранена в других структурах данных при вызове `mmap()`.

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>
void *mmap(void *start, size_t length,
        int prot, int flags, int fd, off_t offset);
```

Системный вызов `mmap` предназначен для отображения предварительно открытого файла (например, с помощью системного вызова `open()`) в адресное пространство вычислительной системы. После его выполнения файл может быть закрыт (например, системным вызовом `close()`), что никак не повлияет на дальнейшую работу с отображенным файлом.

Это не полное описание системного вызова. Для получения полной информации обращайтесь к *Manual*.

Параметр `fd` является файловым дескриптором для файла, который мы хотим отобразить в адресное пространство (т.е. значением, которое вернул системный вызов `open()`).

Ненулевое значение параметра `start` может использоваться только очень квалифицированными системными программистами, поэтому мы пока будем всегда полагать его равным значению `NULL`, позволяя операционной системе самой выбрать начало области адресного пространства, в которую будет отображен файл.

В память будет отображаться часть файла, начиная с позиции внутри его, заданной значением параметра `offset` – смещение от начала файла в байтах, и длиной, равной

значению параметра `length` (естественно, тоже в байтах). Значение параметра `length` может и превышать реальную длину от позиции `offset` до конца существующего файла. На поведении системного вызова это никак не отразится, но в дальнейшем при попытке доступа к ячейкам памяти, лежащим вне границ реального файла, возникнет сигнал `SIGBUS` (реакция на него по умолчанию – прекращение процесса с образованием *core* файла).

Параметр `flags` определяет способ отображения файла в адресное пространство. Мы будем использовать только два его возможных значения: `MAP_SHARED` и `MAP_PRIVATE`. Если в качестве его значения выбрано `MAP_SHARED`, то полученное отображение файла впоследствии будет использоваться и другими процессами, вызвавшими `mmap` для этого файла с аналогичными значениями параметров, а все изменения, сделанные в отображенном файле, будут сохранены во вторичной памяти. Если в качестве значения параметра `flags` указано `MAP_PRIVATE`, то процесс получает отображение файла в свое монопольное распоряжение, но все изменения в нем не могут быть занесены во вторичную память (т.е., не могут быть сохранены).

Параметр `prot` определяет разрешенные операции над областью памяти, в которую будет отображен файл. В качестве его значения мы будем использовать значения `PROT_READ` (*разрешено чтение*), `PROT_WRITE` (*разрешена запись*) или их комбинацию через операцию "*побитовое или*" – "`|`". Необходимо отметить две существенные особенности системного вызова, связанные с этим параметром:

1. Значение параметра `prot` не может быть шире, чем операции над файлом, заявленные при его открытии в параметре `flags` системного вызова `open()`. Например, нельзя открыть файл только для чтения, а при его отображении в память использовать значение `prot = PROT_READ | PROT_WRITE`.
2. В результате особенностей операционной системе *Linux* некоторых версий попытка записать в отображение файла, открытое только для записи, более 32-х байт одновременно может приводить к ошибке (возникает сигнал о нарушении защиты памяти).

При нормальном завершении системный вызов возвращает начальный адрес области памяти, в которую отображен файл (или его часть), при возникновении ошибки – специальное значение `MAP_FAILED`.

- После этого с содержимым файла можно работать, как с содержимым обычной области памяти.
- По окончании работы с содержимым файла, необходимо освободить дополнительно выделенную процессу область памяти, предварительно синхронизировав содержимое файла на диске с содержимым этой области (если это необходимо). Такие действия выполняет системный вызов `munmap()`.

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>

int munmap (void *start, size_t length);
```

Системный вызов `munmap` служит для прекращения отображения *memory mapped файла* в адресное пространство вычислительной системы. Если при системном вызове `mmap()` было задано значение параметра `flags`, равное `MAP_SHARED`, и в отображении файла была разрешена операция записи (в параметре `prot` использовалось значение `PROT_WRITE`), то `munmap` синхронизирует содержимое отображения с содержимым файла

во вторичной памяти. После его выполнения области памяти, использовавшиеся для отображения файла, становятся недоступны текущему процессу.

Параметр `start` является адресом начала области памяти, выделенной для отображения файла, т.е. значением, которое вернул системный вызов `mmap()`.

Параметр `length` определяет ее длину, и его значение должно совпадать со значением соответствующего параметра в системном вызове `mmap()`.

При нормальном завершении системный вызов возвращает значение `0`, при возникновении ошибки – значение `-1`.

Пример

Рассмотрим пример двух программ, обменивающихся информацией с помощью отображаемых в память файлов (также с использованием семафоров). Первая программа:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/sem.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>

#define SEM_KEY 2015
#define MM_FILENAME "message"

union semnum {
    int val;
    struct semid_ds * buf;
    unsigned short * array;
} sem_arg;


int main(int argc, char **argv) {
    int fd, sem_id, msg_len;
    struct sembuf sb[1];
    unsigned short sem_vals[1];
    char *mbuf;

    if (argc < 2) {
        fprintf(stderr, "Too few arguments\n");
        return EXIT_FAILURE;
    }

    msg_len = strlen(argv[1]);
    fd = open(MM_FILENAME, O_RDWR | O_CREAT | O_TRUNC, 0600);
    if (fd == -1) {
        fprintf(stderr, "Cannot open file\n");
        return EXIT_FAILURE;
    }

    mbuf = (char *) malloc(msg_len);
```

```
    if (mbuf == NULL) {
        fprintf(stderr, "malloc error\n");
        return EXIT_FAILURE;
    }

    mbuf = memset(mbuf, 0, msg_len);
    write(fd, mbuf, msg_len);
    lseek(fd, 0, SEEK_SET);
    free(mbuf);

    mbuf = mmap(0, msg_len, PROT_WRITE | PROT_READ,
                MAP_SHARED, fd, 0);
    if (mbuf == MAP_FAILED) {
        fprintf(stderr, "mmap() error\n");
        return EXIT_FAILURE;
    }
    close(fd);
    strncpy(mbuf, argv[1], msg_len);
    msync(mbuf, msg_len, MS_SYNC);

    sem_id = semget(SEM_KEY, 1,
                0600 | IPC_CREAT | IPC_EXCL);
    if (sem_id == -1) {
        fprintf(stderr, "semget() error\n");
        return EXIT_FAILURE;
    }

    sem_vals[0] = 1;
    sem_arg.array = sem_vals;

    if (semctl(sem_id, 0, SETALL, sem_arg) == -1) {
        fprintf(stderr, "semctl() error\n");
        return EXIT_FAILURE;
    }

    sb[0].sem_num = 0;
    sb[0].sem_flg = SEM_UNDO;

    sb[0].sem_op = -1;
    semop(sem_id, sb, 1);

    sb[0].sem_op = -1;
    semop(sem_id, sb, 1);

    semctl(sem_id, 1, IPC_RMID, sem_arg);
    munmap(mbuf, msg_len);
    unlink(MM_FILENAME);

    return EXIT_SUCCESS;
}
```

Вторая программа:

```
#include <stdio.h>
#include <sys/sem.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

#define SEM_KEY 2015
#define MM_FILENAME "message"

int main(void) {
    int fd, sem_id, buf_len;
    struct sembuf sb[1];
    char *mbuf;
    struct stat st;

    sem_id = semget(SEM_KEY, 1, 0600);
    if (sem_id == -1) {
        fprintf(stderr, "semget() error\n");
        return EXIT_FAILURE;
    }

    fd = open(MM_FILENAME, O_RDONLY);
    if (fd == -1) {
        fprintf(stderr, "Cannot open file\n");
        return EXIT_FAILURE;
    }

    if (fstat(fd, &st) == -1) {
        fprintf(stderr, "fstat() error\n");
        return EXIT_FAILURE;
    }

    buf_len = st.st_size;

    mbuf = mmap(0, buf_len, PROT_READ,
                MAP_SHARED, fd, 0);
    if (mbuf == MAP_FAILED)    {
        fprintf(stderr, "mmap() error\n");
        return EXIT_FAILURE;
    }

    close(fd);
    write(1, mbuf, buf_len);
    printf("\n");
    munmap(mbuf, buf_len);

    sb[0].sem_num = 0;
    sb[0].sem_flg = SEM_UNDO;
```

```
    sb[0].sem_op = 1;
    semop(sem_id, sb, 1);

    return EXIT_SUCCESS;
}
```