

[КАК СТАТЬ АВТОРОМ](#)

bsergik 18 июня 2011 в 17:18

Знакомство с межпроцессным взаимодействием на Linux

Программирование*

Межпроцессное взаимодействие (*Inter-process communication (IPC)*) – это набор методов для обмена данными между потоками процессов. Процессы могут быть запущены как на одном и том же компьютере, так и на разных, соединенных сетью. IPC бывают нескольких типов: «сигнал», «сокет», «семафор», «файл», «сообщение»...

В данной статье я хочу рассмотреть всего 3 типа IPC:

1. именованный канал
2. разделенная память
3. семафор

Отступление: данная статья является учебной и рассчитана на людей, только еще вступающих на путь системного программирования. Ее главный замысел — познакомиться с различными способами взаимодействия между процессами на POSIX-совместимой ОС.

Именованный канал

Для передачи сообщений можно использовать механизмы сокетов, каналов, D-bus и другие технологии. Про сокеты на каждом углу можно почитать, а про D-bus отдельную статью написать. Поэтому я решил остановиться на малоизвестных технологиях отвечающих стандартам POSIX и

привести рабочие примеры.

Рассмотрим передачу сообщений по именованным каналам. Схематично передача выглядит так:



Для создания именованных каналов будем использовать функцию, `mkfifo()`:

```
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

Функция создает специальный FIFO файл с именем `pathname`, а параметр `mode` задает права доступа к файлу.

Примечание: `mode` используется в сочетании с текущим значением `umask` следующим образом: $(mode \& \sim umask)$. Результатом этой операции и будет новое значение `umask` для создаваемого нами файла. По этой причине мы используем `0777 (S_IRWXO | S_IRWXG | S_IRWXU)`, чтобы не затирать ни один бит текущей маски.

Как только файл создан, любой процесс может открыть этот файл для чтения или записи также, как открывает обычный файл. Однако, для корректного использования файла, необходимо открыть его одновременно двумя процессами/потоками, одним для получение данных (чтение файла), другим на передачу (запись в файл).

В случае успешного создания FIFO файла, **mkfifo()** возвращает 0 (нуль). В случае каких либо ошибок, функция возвращает -1 и выставляет код ошибки в переменную **errno**.

Типичные ошибки, которые могут возникнуть во время создания канала:

Все потоки Разработка Администрирование Дизайн Менеджмент Маркетинг Научпоп 🔍

- *ENOENT* – не существует какой-либо директории, упомянутой в **pathname**, либо является битой ссылкой
- *ENOSPC* – нет места для создания нового файла
- *ENOTDIR* – одна из директорий, упомянутых в **pathname**, на самом деле не является таковой
- *EROFS* – попытка создать FIFO файл на файловой системе «только-на-чтение»

Чтение и запись в созданный файл производится с помощью функций **read()** и **write()**.

Пример

mkfifo.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>
```

```
#define NAMEDPIPE_NAME "/tmp/my_named_pipe"
#define BUFSIZE      50
```

```
int main (int argc, char ** argv) {
    int fd, len;
    char buf[BUFSIZE];

    if ( mkfifo(NAMEDPIPE_NAME, 0777) ) {
        perror("mkfifo").
```

◆ +77 👁 171K 📖 319 ➡

```
printf("%s is created\n", NAMEDPIPE_NAME);
```

```
if ( (fd = open(NAMEDPIPE_NAME, O_RDONLY)) <= 0 ) {
    perror("open");
    return 1;
}
```

```
printf("%s is opened\n", NAMEDPIPE_NAME);
```

```
do {
    memset(buf, '\0', BUFSIZE);
    if ( (len = read(fd, buf, BUFSIZE-1)) <= 0 ) {
        perror("read");
        close(fd);
        remove(NAMEDPIPE_NAME);
        return 0;
    }
    printf("Incomming message (%d): %s\n", len, buf);
} while ( 1 );
```

```
}
```

[скачать]

Мы открываем файл только для чтения (*O_RDONLY*). И могли бы использовать *O_NONBLOCK* модификатор, предназначенный специально для FIFO файлов, чтобы не ждать когда с другой стороны файл откроют для записи. Но в приведенном коде такой способ неудобен.

Компилируем программу, затем запускаем ее:

```
$ gcc -o mkfifo mkfifo.c  
$ ./mkfifo
```

В соседнем терминальном окне выполняем:

```
$ echo 'Hello, my named pipe!' > /tmp/my_named_pipe
```

В результате мы увидим следующий вывод от программы:

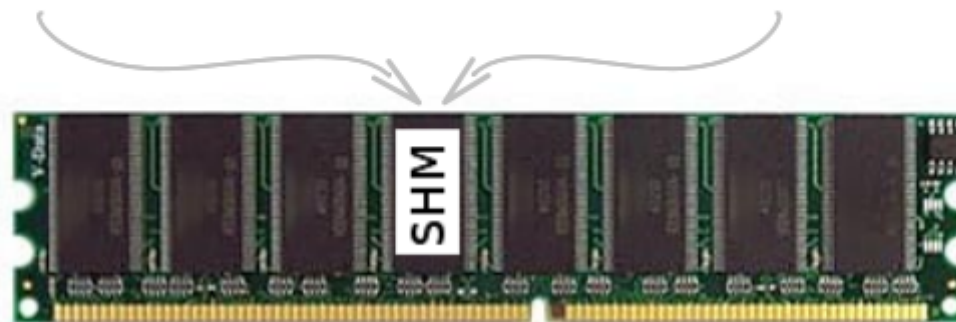
```
$ ./mkfifo  
/tmp/my_named_pipe is created  
/tmp/my_named_pipe is opened  
Incomming message (22): Hello, my named pipe!  
read: Success
```

Разделяемая память

Следующий тип межпроцессного взаимодействия – разделяемая память (*shared memory*). Схематично изобразим ее как некую именованную область в памяти, к которой обращаются одновременно два процесса:

Процесс №1
обращается к
разделенной
памяти по
имени "SHM"

Процесс №2
обращается к
разделенной
памяти по
имени "SHM"



Для выделения разделяемой памяти будем использовать POSIX функцию `shm_open()`:

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

int shm_open(const char *name, int oflag, mode_t mode);
```

Функция возвращает файловый дескриптор, который связан с объектом памяти. Этот дескриптор в дальнейшем можно использовать другими функциями (к примеру, `mmap()` или `mprotect()`).

Целостность объекта памяти сохраняется, включая все данные связанные с ним, до тех пор пока объект не отсоединен/удален (`shm_unlink()`). Это означает, что любой процесс может получить доступ к нашему объекту памяти (если он знает его имя) до тех пор, пока явно в одном из процессов мы не вызовем `shm_unlink()`.

Переменная **oflag** является побитовым «ИЛИ» следующих флагов:

- *O_RDONLY* – открыть только с правами на чтение
- *O_RDWR* – открыть с правами на чтение и запись
- *O_CREAT* – если объект уже существует, то от флага никакого эффекта. Иначе, объект создается и для него выставляются права доступа в соответствии с `mode`.
- *O_EXCL* – установка этого флага в сочетании с *O_CREAT* приведет к возврату функцией `shm_open` ошибки, если сегмент общей памяти уже существует.

Как задается значение параметра **mode** подробно описано в предыдущем параграфе «передача сообщений».

После создания общего объекта памяти, мы задаем размер разделяемой памяти вызовом **ftruncate()**. На входе у функции файловый дескриптор нашего объекта и необходимый нам размер.

Пример

Следующий код демонстрирует создание, изменение и удаление разделяемой памяти. Так же показывается как после создания разделяемой памяти, программа выходит, но при следующем же запуске мы можем получить к ней доступ, пока не выполнен **shm_unlink()**.

shm_open.c

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

```
#include <string.h>

#define SHARED_MEMORY_OBJECT_NAME "my_shared_memory"
#define SHARED_MEMORY_OBJECT_SIZE 50
#define SHM_CREATE 1
#define SHM_PRINT 3
#define SHM_CLOSE 4

void usage(const char * s) {
    printf("Usage: %s <create|write|read|unlink> ['text']\n", s);
}

int main (int argc, char ** argv) {
    int shm, len, cmd, mode = 0;
    char *addr;

    if ( argc < 2 ) {
        usage(argv[0]);
        return 1;
    }

    if ( (!strcmp(argv[1], "create") || !strcmp(argv[1], "write")) && (argc == 3) )
        len = strlen(argv[2]);
        len = (len <= SHARED_MEMORY_OBJECT_SIZE) ? len : SHARED_MEMORY_OBJECT_SIZE;
        mode = O_CREAT;
        cmd = SHM_CREATE;
    } else if ( ! strcmp(argv[1], "print" ) ) {
        cmd = SHM_PRINT;
    } else if ( ! strcmp(argv[1], "unlink" ) ) {
        cmd = SHM_CLOSE;
    } else {
```



```
usage(argv[0]);
return 1;
}

if ( (shm = shm_open(SHARED_MEMORY_OBJECT_NAME, mode|O_RDWR, 0777)) == -1 ) {
    perror("shm_open");
    return 1;
}

if ( cmd == SHM_CREATE ) {
    if ( ftruncate(shm, SHARED_MEMORY_OBJECT_SIZE+1) == -1 ) {
        perror("ftruncate");
        return 1;
    }
}

addr = mmap(0, SHARED_MEMORY_OBJECT_SIZE+1, PROT_WRITE|PROT_READ, MAP_SHARED, s
if ( addr == (char*)-1 ) {
    perror("mmap");
    return 1;
}

switch ( cmd ) {
case SHM_CREATE:
    memcpy(addr, argv[2], len);
    addr[len] = '\0';
    printf("Shared memory filled in. You may run '%s print' to see value.\n", e
    break;
case SHM_PRINT:
    printf("Got from shared memory: %s\n", addr);
    break;
```

```
}

munmap(addr, SHARED_MEMORY_OBJECT_SIZE);
close(shm);

if ( cmd == SHM_CLOSE ) {
    shm_unlink(SHARED_MEMORY_OBJECT_NAME);
}

return 0;
}
```

[\[скачать\]](#)

После создания объекта памяти мы установили нужный нам размер shared memory вызовом **ftruncate()**. Затем мы получили доступ к разделяемой памяти при помощи **mmap()**. (Вообще говоря, даже с помощью самого вызова **mmap()** можно создать разделяемую память. Но отличие вызова **shm_open()** в том, что память будет оставаться выделенной до момента удаления или перезагрузки компьютера.)

Компилировать код на этот раз нужно с опцией **-lrt**:

```
$ gcc -o shm_open -lrt shm_open.c
```

Смотрим что получилось:

```
$ ./shm_open create 'Hello, my shared memory!'
Shared memory filled in. You may run './shm_open print' to see value.
$ ./shm_open print
```

```
Got from shared memory: Hello, my shared memory!  
$ ./shm_open create 'Hello!'  
Shared memory filled in. You may run './shm_open print' to see value.  
$ ./shm_open print  
Got from shared memory: Hello!  
$ ./shm_open close  
$ ./shm_open print  
shm_open: No such file or directory
```

Аргумент «create» в нашей программе мы используем как для создания разделенной памяти, так и для изменения ее содержимого.

Зная имя объекта памяти, мы можем менять содержимое разделяемой памяти. Но стоит нам вызвать **shm_unlink()**, как память перестает быть нам доступна и **shm_open()** без параметра *O_CREATE* возвращает ошибку «No such file or directory».

Семафор

Семафор – самый часто употребляемый метод для синхронизации потоков и для контролирования одновременного доступа множеством потоков/процессов к общей памяти (к примеру, глобальной переменной). Взаимодействие между процессами в случае с семафорами заключается в том, что процессы работают с одним и тем же набором данных и корректируют свое поведение в зависимости от этих данных.

Есть два типа семафоров:

1. семафор со счетчиком (counting semaphore), определяющий лимит ресурсов для процессов, получающих доступ к ним

2. бинарный семафор (binary semaphore), имеющий два состояния «0» или «1» (чаще: «занят» или «не занят»)

Рассмотрим оба типа семафоров.

Семафор со счетчиком

Смысл семафора со счетчиком в том, чтобы дать доступ к какому-то ресурсу только определенному количеству процессов. Остальные будут ждать в очереди, когда ресурс освободится.

Итак, для реализации семафоров будем использовать POSIX функцию **sem_open()**:

```
#include <semaphore.h>

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

В функцию для создания семафора мы передаем имя семафора, построенное по определенным правилам и управляющие флаги. Таким образом у нас получится именованный семафор. Имя семафора строится следующим образом: в начале идет символ "/" (косая черта), а следом латинские символы. Символ «косая черта» при этом больше не должен применяться. Длина имени семафора может быть вплоть до 251 знака.

Если нам необходимо создать семафор, то передается управляющий флаг *O_CREATE*. Чтобы начать использовать уже существующий семафор, то **oflag** равняется нулю. Если вместе с флагом *O_CREATE* передать флаг *O_EXCL*, то функция **sem_open()** вернет ошибку, в случае если семафор с указанным именем уже существует.

Параметр **mode** задает права доступа таким же образом, как это объяснено в предыдущих главах.

А переменной **value** инициализируется начальное значение семафора. Оба параметра **mode** и **value** игнорируются в случае, когда семафор с указанным именем уже существует, а **sem_open()** вызван вместе с флагом *O_CREATE*.

Для быстрого открытия существующего семафора используем конструкцию:

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag);
```

, где указываются только имя семафора и управляющий флаг.

Пример семафора со счетчиком

Рассмотрим пример использования семафора для синхронизации процессов. В нашем примере один процесс увеличивает значение семафора и ждет, когда второй сбросит его, чтобы продолжить дальнейшее выполнение.

sem_open.c

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

#define SEMAPHORE_NAME "/my_named_semaphore"

int main(int argc, char ** argv) {
    sem_t *sem;

    if ( argc == 2 ) {
```

```
printf("Dropping semaphore...\n");
if ( (sem = sem_open(SEMAPHORE_NAME, 0)) == SEM_FAILED ) {
    perror("sem_open");
    return 1;
}
sem_post(sem);
perror("sem_post");
printf("Semaphore dropped.\n");
return 0;
}

if ( (sem = sem_open(SEMAPHORE_NAME, O_CREAT, 0777, 0)) == SEM_FAILED ) {
    perror("sem_open");
    return 1;
}

printf("Semaphore is taken.\nWaiting for it to be dropped.\n");
if (sem_wait(sem) < 0 )
    perror("sem_wait");
if ( sem_close(sem) < 0 )
    perror("sem_close");

return 0;
}
```

[скачать]

В одной консоли запускаем:

```
$ ./sem_open
Semaphore is taken.
Waiting for it to be dropped.      <-- здесь процесс в ожидании другого процесса
```

```
sem_wait: Success  
sem_close: Success
```

В соседней консоли запускаем:

```
$ ./sem_open 1  
Dropping semaphore...  
sem_post: Success  
Semaphore dropped.
```

Бинарный семафор

Вместо бинарного семафора, для которого так же используется функция `sem_open`, я рассмотрю гораздо чаще употребляемый семафор, называемый «мьютекс» (`mutex`).

Мьютекс по существу является тем же самым, чем является бинарный семафор (т.е. семафор с двумя состояниями: «занят» и «не занят»). Но термин «`mutex`» чаще используется чтобы описать схему, которая предохраняет два процесса от одновременного использования общих данных/переменных. В то время как термин «бинарный семафор» чаще употребляется для описания конструкции, которая ограничивает доступ к одному ресурсу. То есть бинарный семафор используют там, где один процесс «занимает» семафор, а другой его «освобождает». В то время как мьютекс освобождается тем же процессом/поток, который занял его.

Без мьютекса не обойтись в написании, к примеру базы данных, к которой доступ могут иметь множество клиентов.

Для использования мьютекса необходимо вызвать функцию `pthread_mutex_init()`:

```
#include <pthread.h>
```

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr
```

Функция инициализирует мьютекс (переменную **mutex**) атрибутом **mutexattr**. Если **mutexattr** равен *NULL*, то мьютекс инициализируется значением по умолчанию. В случае успешного выполнения функции (код возврата 0), мьютекс считается инициализированным и «свободным».

Типичные ошибки, которые могут возникнуть:

- *EAGAIN* – недостаточно необходимых ресурсов (кроме памяти) для инициализации мьютекса
- *ENOMEM* – недостаточно памяти
- *EPERM* – нет прав для выполнения операции
- *EBUSY* – попытка инициализировать мьютекс, который уже был инициализирован, но не уничтожен
- *EINVAL* – значение **mutexattr** не валидно

Чтобы занять или освободить мьютекс, используем функции:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_trylock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Функция **pthread_mutex_lock()**, если **mutex** еще не занят, то занимает его, становится его обладателем и сразу же выходит. Если мьютекс занят, то блокирует дальнейшее выполнение процесса и ждет освобождения мьютекса.

Функция `pthread_mutex_trylock()` идентична по поведению функции `pthread_mutex_lock()`, с одним исключением – она не блокирует процесс, если **mutex** занят, а возвращает *EBUSY* код.

Функция `pthread_mutex_unlock()` освобождает занятый мьютекс.

Коды возврата для `pthread_mutex_lock()`:

- `EINVAL` – mutex неправильно инициализирован
- `EDEADLK` – мьютекс уже занят текущим процессом

Коды возврата для `pthread_mutex_trylock()`:

- `EBUSY` – мьютекс уже занят
- `EINVAL` – мьютекс неправильно инициализирован

Коды возврата для `pthread_mutex_unlock()`:

- `EINVAL` – мьютекс неправильно инициализирован
- `EPERM` – вызывающий процесс не является обладателем мьютекса

Пример mutex

mutex.c

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <errno.h>

static int counter; // shared resource
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
void incr_counter(void *p) {
    do {
        usleep(10); // Let's have a time slice between mutex locks
        pthread_mutex_lock(&mutex);
        counter++;
        printf("%d\n", counter);
        sleep(1);
        pthread_mutex_unlock(&mutex);
    } while ( 1 );
}

void reset_counter(void *p) {
    char buf[10];
    int num = 0;
    int rc;
    pthread_mutex_lock(&mutex); // block mutex just to show message
    printf("Enter the number and press 'Enter' to initialize the counter with new v
sleep(3);
    pthread_mutex_unlock(&mutex); // unblock blocked mutex so another thread may wc
    do {
        if ( gets(buf) != buf ) return; // NO fool-protection ! Risk of overflow !
        num = atoi(buf);
        if ( (rc = pthread_mutex_trylock(&mutex)) == EBUSY ) {
            printf("Mutex is already locked by another process.\nLet's lock mutex t
            pthread_mutex_lock(&mutex);
        } else if ( rc == 0 ) {
            printf("WOW! You are on time! Congratulation!\n");
        } else {
            printf("Error: %d\n", rc);
            return;
        }
    }
```

```
        counter = num;
        printf("New value for counter is %d\n", counter);
        pthread_mutex_unlock(&mutex);
    } while ( 1 );
}

int main(int argc, char ** argv) {
    pthread_t thread_1;
    pthread_t thread_2;
    counter = 0;

    pthread_create(&thread_1, NULL, (void *)&incr_counter, NULL);
    pthread_create(&thread_2, NULL, (void *)&reset_counter, NULL);

    pthread_join(thread_2, NULL);
    return 0;
}
```

[\[скачать\]](#)

Данный пример демонстрирует совместный доступ двух потоков к общей переменной. Один поток (первый поток) в автоматическом режиме постоянно увеличивает переменную **counter** на единицу, при этом занимая эту переменную на целую секунду. Этот первый поток дает второму доступ к переменной **count** только на 10 миллисекунд, затем снова занимает ее на секунду. Во втором потоке предлагается ввести новое значение для переменной с терминала.

Если бы мы не использовали технологию «мьютекс», то какое значение было бы в глобальной переменной, при одновременном доступе двух потоков, нам не известно. Так же во время запуска становится очевидна разница между **pthread_mutex_lock()** и **pthread_mutex_trylock()**.

Компилировать код нужно с дополнительным параметром **-lpthread**:

```
$ gcc -o mutex -lpthread mutex.c
```

Запускаем и меняем значение переменной просто вводя новое значение в терминальном окне:

```
$ ./mutex
Enter the number and press 'Enter' to initialize the counter with new value anytime
1
2
3
30 <Enter>      <--- новое значение переменной
Mutex is already locked by another process.
Let's lock mutex using pthread_mutex_lock().
New value for counter is 30
31
32
33
1 <Enter>      <--- новое значение переменной
Mutex is already locked by another process.
Let's lock mutex using pthread_mutex_lock().
New value for counter is 1
2
3
```

Вместо заключения

В следующих статьях я хочу рассмотреть технологии d-bus и RPC. Если есть интерес, дайте

Знать.

Спасибо.

UPD: Обновил 3-ю главу про семафоры. Добавил подглаву про мьютекс.

Теги: linux, posix, ipc, программирование

Хабы: Программирование

Редакторский дайджест



Присылаем лучшие статьи раз в месяц

Электронная почта



42

Карма

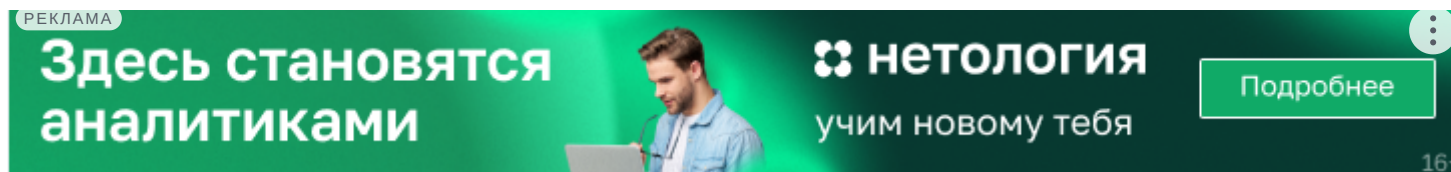
0

Рейтинг

Сергей Балабанов @bsergik

Пользователь

Реклама



Комментарии 22



o  **nagato** 18.06.2011 в 17:32

Сейчас вспомнил, когда сдавал экзамен, мне задали вопрос: «Представим, что между двумя парами процессов инициализированы `pipe` и `fifo` соответственно. Какое может возникнуть отличие в их работе, если они взаимодействуют одинаково?»
На вопрос не ответил, а когда ушел, забыл спросить ответ. Может, кто-нибудь знает ответ на этот вопрос?

◆ +1 Ответить  ...


o  **bsergik** 18.06.2011 в 17:43 ^

У `pipe` нет имени и он создается для единичного использования. Пайп создается в рамках одного процесса и `pipe` могут использовать только наследники этого процесса. Самый простой пример – «`cat some_file | grep some_name`». Здесь пайп создается шеллом, в котором запущена команда.
В отличие от `pipe`, `fifo` создается именованным и выглядит как обычный файл в системе Linux, с некоторыми оговорками. Доступ к `fifo` могут иметь кто угодно, кто имеет соответствующие привилегии.

Вот пример `fifo`:

```
prwxr-x--x 1 serge users 0 Jun 18 13:43 my_named_pipe
```

◆ +2 Ответить  ...

o  **nagato** 18.06.2011 в 17:46 ^

Нет, это я знаю, вопрос был в предположении, что и `pipe` и `fifo` уже были корректно созданы и открыты.

◆ +1 Ответить

○  **shadoof** 18.06.2011 в 22:08 ^

Разница такая: время жизни `pipe` не превышает время жизни породившего его процесса. `fifo` – его еще называют именованным каналом. FIFO нужно удалять при необходимости, т. к. цикл его существования не привязан к времени жизни процесса.

◆ +2 Ответить

НЛО прилетело и опубликовало эту надпись здесь

○  **redchrom** 18.06.2011 в 19:59 ^

В Linux `shm_open()` тупо открывает файл в `/dev/shm`, следовательно `ls`. Кстати чтобы избежать утечки ресурсов при работе с `memory mapped files`, дескрипторы можно закрывать, а на файлы делать `unlink` как только сделали маппинг – на файлы есть счётчик ссылок. Заметил, что некоторые вообще дескриптор на замапленный файл передают через `unix socket`, чтобы `unlink` можно было сделать сразу же после `open`.

◆ 0 Ответить

○  **Krass** 18.06.2011 в 20:52

Эх, почему-то всегда интересные статьи по программированию появляются уже после экзамена =) все равно спасибо, продолжайте дальше!

◆ 0 Ответить

Antigluk 18.06.2011 в 21:16

А с чем связано, что имя семафора должно начинаться со слеша?

0 Ответить

bsergik 19.06.2011 в 03:45

Сходу могу только сказать, что необходимость слеша в начале имени семафора продиктована требованиями POSIX стандарта. Почему это так, я сейчас не могу сказать. Обещаю поискать больше информации по этому поводу и ответить на ваш вопрос.

0 Ответить

bsergik 19.06.2011 в 17:02

Вобщем вот что я накопал для вашего вопроса:

1. Стандарт POSIX утверждает что если имя нашего семафора начинается со слеша, то любой вызов `sem_open` с таким именем в любой процедуре должен возвращать дескриптор одного и того же семафора. Проще говоря вызов `sem_open("/semaphore",...)` в любой процедуре вернет дескриптор одного и того же семафора. Тот же стандарт пишет, что если слеша в начале имени нет, то поведение стандартом не описывается, а ложится на плечи разработчиков `sem_open`. Проще говоря, что делать с именем семафора без слеша в начале «решает» конкретная реализация `sem_open`.
2. Я не поленился глянуть в реализацию `sem_open` в стандартной библиотеке `glibc`. Так вот тамошняя реализация функции допускает отсутствие слеша. Более того, `sem_open` не пользуется слешом в начале имени, а смело его пропускает вот таким наглым образом:

```
sem_t * sem_open (const char *name, int oflag, ...) {
```

...

```
/* Construct the filename. */  
while (name[0] == '/')  
++name;
```

...

3. Ответ на ваш вопрос: имя семафора должно начинаться со слеша, скорее всего, потому, что

семафор, как и разделенная память, должен создаваться в файловой системе ОС. Выглядеть это должно примерно так: `/dev/sem/my_semaphore_name`. Именно так реализовано в системе QNX, которая, к слову сказать, является полностью POSIX совместимой, в отличие от Linux.

 +2 Ответить  ...

○  **Antigluk** 19.06.2011 в 17:58 ^

Спасибо большое за подробный анализ

 0 Ответить  ...

○  **Semy** 19.06.2011 в 04:41

Тема про семафоры по-моему раскрыта не полно. А вообще, хорошо, спасибо.
Про d-bus читаю с удовольствием.

 0 Ответить  ...

○  **bsergik** 19.06.2011 в 23:38 ^

Расписал чуть подробнее. Добавил подглаву про mutex.

 0 Ответить  ...

○  **seriyPS** 19.06.2011 в 08:02

Про Dbus интересно было бы. Использовал его пару раз из Python и из консоли через qdbus, но не сильно вникал в подробности. В частности интересно – Dbus где-то кроме десктопа актуален? Как у него с ограничением прав доступа?


А в статье хотелось бы еще видеть рекомендации что в каких случаях лучше применять...

 +1 Ответить  ... **qmax** 20.06.2011 в 00:53

а как связан описанный тут механизм shm с загрузкой со библиотек?

 0 Ответить  ... **redchrom** 20.06.2011 в 18:52 ^

Библиотеки также мапятся через mmap (с PROT_READ|PROT_EXEC и MAP_SHARED) так что механизм тот-же.

 0 Ответить  ... **qmax** 20.06.2011 в 19:24 ^

тоесть, ld-linux, окромя манипуляциями с таблицами ссылок, никакой особой подземной магии не юзает?

 0 Ответить  ... **redchrom** 20.06.2011 в 19:30 ^

Не, ну там куча логики но в том что касается shared memory всё просто – open и mmap, а там уже ядру решать.


 0 Ответить  ... **Зомбак** 13.07.2011 в 14:13

Хорошая статья, спасибо!

Давно хотел посмотреть про разделяемую память и тут на тебе. Спасибо.


 0 [УТВЕРДИТЬ](#)  ... **Zom6ak** 14.07.2011 в 20:56

Хочу кое-что заметить/спросить. В случае разделяемой памяти – зачем использовать mmap/memmap, если можно просто работать с дескриптором? Я вот тут экспериментировал и оказалось, что можно просто вызывать read/write для такого дескриптора и всё будет отлично работать. Так вроде даже проще...

 0 [Ответить](#)  ... **bsergik** 16.07.2011 в 23:36 ^

Последовательно действий именно такая:

- 1) идентифицировать разделяемую область неким символическим именем, к примеру "/mySharedMem"
- 2) с помощью shm_open открыть файловый дескриптор на только что идентифицированной области
- 3) ftruncate'ом задать необходимый размер сегмента
- 4) с помощью mmap мапировать этот сегмент в свое адресное пространство

 0 [Ответить](#)  ... **Levhav** 14.11.2017 в 07:46

Подскажите пожалуйста как отличить именованный канал от обычного файла?

Реализовал работу с каналами точно так же как в примере с mkfifo.c

► [Цитата кода из статьи](#)

в итоге всё работает хорошо. Но если канал существует то он используется а не удаляется и создаётся заново. И тоже всё работает.

Но если вместо канала создан обычный файл то `mkfifo` не вернёт ошибку, а код чтения из этого файла будет выполняться в бесконечном цикле без остановки.

0 Ответить

Только полноправные пользователи могут оставлять комментарии. [Войдите, пожалуйста.](#)

ПОХОЖИЕ ПУБЛИКАЦИИ

18 января в 20:00

Кунг-фу стиля Linux: автоматическое генерирование заголовочных файлов

+31 4.7K 45 10 +10

8 августа 2021 в 13:54

Google: команде безопасности ядра Linux не хватает примерно сотни инженеров

+12 9.7K 9 45 +45

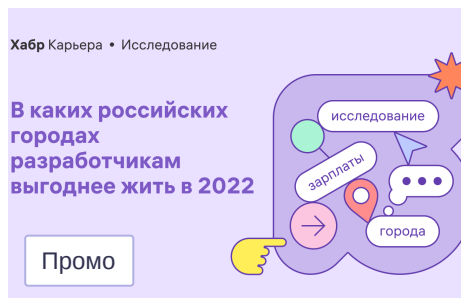
7 февраля 2018 в 22:17

Расширение и использование Linux Crypto API

+36 11K 87 16 +16



Хотите рассказать о себе в наших социальных сетях?




В каких городах России разработчикам выгоднее жить




Шифр как предчувствие: древние коды и квантовые компьютеры

КУРСЫ

 Программирование на Python для детей

19 июня 2022 • 14 700 ₽ • GeekBrains

 Linux (Ubuntu). Уровень 2. Программирование в Linux на C

30 мая 2022 • 32 990 ₽ • Специалист.ру

 Разработка драйверов устройств в Linux

30 мая 2022 • 36 990 ₽ • Специалист.ру

 Основы администрирования Linux

30 мая 2022 • 29 600 ₽ • Сетевая Академия ЛАНИТ


 Расширенное администрирование ОС Astra Linux Special Edition

30 мая 2022 • 32 000 ₽ • АИС

Больше курсов на Хабр Карьере

ЛУЧШИЕ ПУБЛИКАЦИИ ЗА СУТКИ

вчера в 20:00

Как я открыл компанию по созданию детских наборов электроники +73  6.3K  61  18 +18


вчера в 16:00

Используем клиентский процессор по максимуму. Часть 2: SIMD + мультипоточность +47  2.3K  39  5 +5


вчера в 20:51

Генерация лабиринтов: алгоритм Эллера +26  3.5K  81  2 +2

вчера в 19:13

Ленточные накопители: фантастические твари мира архивирования +26  5.6K  26  48 +48

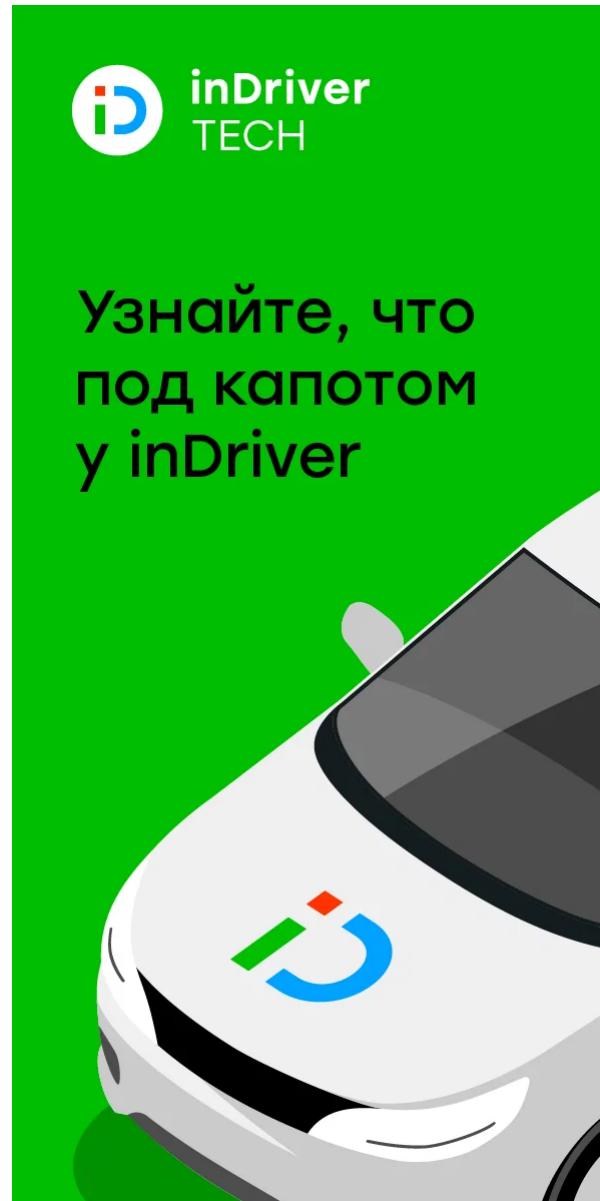
вчера в 18:02

Что можно поменять в дёснах (правда, цена операции — 2 месяца смузи-диеты) +25  5K  31  14 +14

Конвейер для 3D-моделей лиц и full-body-двойников: как снимаем и обрабатываем данные

Турбо

Реклама



ЧИТАЮТ СЕЙЧАС

«Тинькофф Банк» начал автоматически без согласия оформлять рассрочки после просмотра сторис в мобильном приложении

 52K  179 **+179**

Подробный отзыв о Яндекс Практикуме: за что хвалить и ругать

 89K  125 **+125**

DuckDuckGo в рамках сделки с Microsoft не блокирует в своём браузере трекеры LinkedIn или Bing

 4.5K  15 **+15**

Вышел Linux 5.18

 4.3K  2 **+2**

Асинхронный python без головной боли

 2.7K  4 **+4**

Как мы научились запускать 10-часовые UI-тесты за 5 минут в условиях 30 релизов в день

Турбо

Реклама



Ваш аккаунт

[Войти](#)

[Регистрация](#)

Разделы

[Публикации](#)

[Новости](#)

[Хабы](#)

[Компании](#)

[Авторы](#)

[Песочница](#)

Информация

[Устройство сайта](#)

[Для авторов](#)

[Для компаний](#)

[Документы](#)

[Соглашение](#)

[Конфиденциальность](#)

Услуги

[Корпоративный блог](#)

[Медийная реклама](#)

[Нативные проекты](#)

[Мегапроекты](#)



Настройка языка

Техническая поддержка

Вернуться на старую версию

© 2006–2022, Habr