Объекты SVID IPC

Андрей Боровский

Я должен отметить,
что основная цель
компьютерных наук,
– устранение неразберихи,
так и не была достигнута
Эсгар Дейкстра

Мы продолжаем изучение механизмов взаимодействия между процессами в Linux. Каналы различных типов, рассмотренные в предыдущей статье, существовали в Unix практически с самого начала. Позже к ним были добавлены и другие механизмы межпроцессного взаимодействия. Мы остановимся на трех механизмах, которые появились в Unix System V и были описаны в *System V Interface Definition* (SVID) – сообщениях, разделяемой памяти и семафорах. В настоящее время эти механизмы поддерживаются почти всеми Unix-системами (очереди сообщений не поддерживаются в Mac OS X 10.3 [1]).

Интерфейсы трех механизмов SVID IPC основаны на общих принципах. Для того, чтобы разные процессы могли получить доступ к одному объекту системы, они должны «договориться» об идентификации этого объекта. Роль идентификатора для всех объектов System V IPC выполняет ключ – число, уникальное в пределах подсистемы System V IPC. Для того, чтобы использовать один и тот же объект, программы должны использовать один и тот же ключ. Для каждого объекта IPC предусмотрены специальные функции чтения и записи, а также управляющая функция.

Сообщения

Механизм сообщений Linux похож на механизм сообщений, используемый в графических многооконных средах. Сообщения накапливаются в очередях и могут изыматься из очередей последовательно или в произвольном порядке. Каждая группа процессов может создать одну или несколько очередей для обмена сообщениями. Сообщение определяется как «последовательность байтов, передаваемая от одного процесса другому». Система сообщений SVID обладает следующими свойствами:

- Возможность накопления сообщений в очереди. Приложения, использующие сообщения для обмена данными, создают свою собственную очередь сообщений, которая может (и должна) быть удалена приложением-владельцем в момент завершения его работы.
- Возможность произвольного выбора сообщений из очереди на основе назначенных им идентификаторов. Эта возможность позволяет организовать приоритетную обработку сообщений, а также идентифицировать сообщения, посылаемые разными приложениями, участвующими в обмене данными.
- Произвольная структура и размер сообщения.

Последний пункт требует уточнения. Максимальный размер сообщения и максимальное количество сообщений в очереди ограничены, причем не существует единого для всех Unix-систем способа определить эти ограничения. В Linux максимальная длина сообщения в байтах задана константой MSGMAX, определенной в фале <linux/msg.h>, а максимальное число сообщений - константой MSGMNG из того же файла. На платформе IA32 размер сообщения не может превышать 8 килобайт, а длина очереди – 16384 (16K) сообщений.

Структура данных, использующаяся для передачи сообщений, может быть определена следующим образом:

```
struct msgp {
  long mtype;
  ... // Любые другие поля
};
```

Поле mtype является единственным обязательным полем в этой структуре. В поле mtype хранится произвольный идентификатор сообщения, который может интерпретироваться как тип передаваемых данных. Кроме поля mtype структура данных сообщения может содержать любое количество других полей любых типов.

В качестве примера использования очередей сообщений рассмотрим совместную работу двух программ – клиента и сервера. Исходные тексты вы найдете на диске (файлы msgcli.c и msgserv.c соответственно). Для того чтобы программы могли обмениваться сообщениями, они должны использовать один и тот же формат сообщений и идентификатор очереди. Эти данные, общие для клиента и сервера, удобно вынести в отдельный заголовочный файл (мы назовем его msgtypes.h). Наша структура данных выглядит так:

```
#define MAXLEN 512

struct msg_t {
  long mtype;
  int snd_pid;
  char body[MAXLEN];
};
```

Помимо поля mtype мы вводим поле snd_pid, которое будет содержать идентификатор процесса-отправителя сообщения, и поле body, которое предназначено для текста сообщения. Мы могли бы определить несколько разных структур для сообщений разных типов. Значение поля mtype указывало бы, с какой структурой мы имеем дело. Поле mtype может быть не только идентификатором типа сообщения. С его помощью можно указать, например, приоритет сообщения. Используя функцию произвольной выборки сообщений, приложение может считывать в первую очередь сообщения с более высоким приоритетом.

Помимо структуры сообщения нам следует определить ключ очереди. Для получения уникального ключа можно использовать функцию ftok(3), однако руководство по работе с функциями SVID рекомендует выбирать значения самостоятельно, поэтому в нашем примере мы определим ключ как константу в файле msgtypes.h:

```
#define KEY 1174
```

Маловероятно, что в системе уже существует другая очередь сообщений с ключом 1174. В принципе, программа, создающая объект IPC, может узнать, существует ли уже такой объект (см. ниже использование флага IPC_EXCL), однако толку от этого не много (разве что сообщить пользователю, что, поскольку ключ занят, работать программа не сможет). Допустим, процесс установил, что объект с указанным идентификатором существует, но что ему делать? Процесс может выбрать другой идентификатор из какого-нибудь пула, однако о новом идентификаторе нужно как-то оповестить другие процессы. Для оповещения можно использовать именованные каналы, для которых, в свою очередь, необходим уникальный идентификатор... Уникальность идентификатора файловых каналов основана на уникальности имен файловой системы (имеются в виду полные имена, начиная с корневого слеша). Функция ftok(), которую мы рассмотрим ниже, тоже пытается генерировать идентификаторы, основываясь на уникальности имен файловой системы.

Рассмотрим теперь работу сервера. Сервер получает сообщение, переданное клиентом, распечатывает сообщение на экране терминала, возвращает клиенту сообщение “Ok!”, ждет подтверждения, что клиент получил ответ, затем удаляет очередь и завершает работу. Программу-сервер следует запустить до запуска программы-клиента.

Текст программы (msgserv.c), как всегда, начинается с заголовочных файлов. Все типы, константы и функции, использующиеся при работе с сообщениями, становятся доступны при включении в текст программы файлов <sys/ipc.h> и <sys/msg.h>. Очередь сообщений создается при помощи функции msgget(2):

```
msgid = msgget(KEY, 0666 | IPC_CREAT);
```

Первый параметр msgget() – ключ, гарантирующий уникальность очереди. Ключ очереди представлен числом, поэтому его можно спутать с другим числом – идентификатором очереди, который присваивает система. Помните, что ключ нужен только для открытия очереди, а для работы с ней используется идентификатор. Второй параметр функции msgget() представляет собой комбинацию маски прав доступа к создаваемой очереди (аналогичной маске прав доступа к именованным каналам) и нескольких дополнительных флагов: кажется, программист, писавший функции SVID IPC, сильно экономил на параметрах. Флаг IPC_CREAT указывает, что в результате вызова msgget() должна быть создана новая очередь. При установке флага IPC_EXCL, функция msgget() вернет сообщение об ошибке, если очередь с указанным ключом уже существует. В случае успеха msgget() возвращает положительное значение – идентификатор созданной очереди.

Передача и получение сообщений выполняется при помощи функций msgsnd(2) и msgrcv(2) соответственно. Первым параметром обеих функций является идентификатор очереди, возвращенный функцией msgget(). Во втором параметре передается размер структуры сообщения. Как было сказано выше, программа, читающая сообщения из очереди, должна указать размер сообщения, соответствующий ожидаемому идентификатору и может читать сообщения разного размера (речь идет о ситуации, когда программа ждет сообщений определенного типа). На диске есть пример polymsgserv/polymsgcli, демонстрирующий этот подход. Третьим параметром функции msgrcv() является идентификатор сообщения. Если значение этого параметра больше нуля, из очереди будет извлечено сообщение с соответствующим значением поля mtype. Если этот параметр равен нулю, из очереди будет извлечено первое по порядку сообщение, а если параметр отрицательный, из очереди будет извлечено первое сообщение, чей идентификатор меньше либо равен абсолютному значению параметра. Последний параметр в функциях msgsnd() и msgrcv() позволяет задать дополнительные флаги. Обычно функция, читающая сообщение из очереди, приостанавливает выполнение программы до тех пор, пока не будет извлечено сообщение ожидаемого типа. Именно так работает эта функция в наших программах. При указании флага IPC_NOWAIT функция msgrcv() вернет сообщение об ошибке, если на момент ее вызова в очереди отсутствует подходящее сообщение.

В нашем примере сервер и клиент используют разные идентификаторы для посылаемых сообщений. Это сделано для того, чтобы программа, вызывающая последовательно msgsnd() и msgrcv(), не извлекала из очереди свои собственные сообщения. Программа-сервер записывает в очередь сообщения со значением mtype, равным 1, а считывает – со значением, равным 2 (у программы-клиента все наоборот).

Для удаления очереди используется функция msgctl(2), которая, как и все функции *ctl(), может выполнять множество разных действий (например, получение данных о состоянии очереди). Первый параметр этой функции, это, как всегда, идентификатор очереди, второй параметр – команда (IPC_STAT, IPC_SET

или IPC_RMID). Третий параметр используется в вызовах-запросах (то есть, когда второй параметр равен IPC_STAT), а также для конфигурации очереди (команда IPC_SET). В нем передается указатель на структуру msgid_ds, поля которой содержат значения различных параметров очереди. Функция возвращает статус выполнения команды. Вызов

```
msgctl(msgid, IPC_RMID, 0);
```

удаляет очередь с идентификатором msgid.

Рассмотрим теперь программу-клиент. Первым делом программа-клиент должна получить идентификатор очереди. Для этого используется функция msgget() с тем же ключом очереди, что и у сервера, с маской прав доступа, но без дополнительных флагов. В этом варианте функция возвращает идентификатор уже существующей очереди с заданным ключом или -1, если такой очереди не существует:

```
msgid = msgget(KEY, 0666);
if (msgid == -1) {
   printf("Server is not running!\n", msgid);
   return EXIT_FAILURE;
}
```

Далее клиент считывает строку, вводимую пользователем, формирует сообщение, записывая в поле mtype значение 2, отправляет сообщение, и ждет ответ сервера – сообщение с идентификатором 1. Скомпилируйте обе программы (можете просто скомандовать make msgdemo), запустите сначала сервер, потом, в другом окне терминала, клиент. Напечатайте в окне клиента строку и нажмите ввод.

Разделяемая память

Спецификация SVID описывает интерфейс для работы с разделяемыми блоками памяти. Разделяемые блоки памяти представляют собой область памяти, отображенную адресное пространство нескольких процессов. Если один процесс запишет данные в разделяемую область, другой процесс может считывать их оттуда как из собственной области глобальной памяти. Мы продемонстрируем использование разделяемой памяти на уже знакомом примере клиент-сервер. Как и в случае с сообщениями, нам нужно описать общие структуры данных для клиента и сервера в заголовочном файле (на диске это файл shmemtypes.h):

```
#define FTOK_FILE "./shmemserv"

#define MAXLEN 512

struct memory_block {
  int server_lock;
  int client_lock;
  int turn;
  int readlast;
  char string[MAXLEN];
};
```

Механизм разделяемой памяти не налагает никаких ограничений на структуру блока памяти – мы определяем структуру memory_block, исходя исключительно из наших собственных потребностей. Первые четыре поля структуры memory_block – служебные, они нужны для реализации модифицированного алгоритма Петерсона, о котором будет сказано ниже. Последнее поле предназначено собственно для передачи данных. В нашем заголовочном файле мы не определяем ключ для идентификации разделяемого блока, а используем имя исполнимого файла сервера для генерации ключа с помощью ftok(). Естественно, этот метод сработает только если сервер будет скомпилирован под именем shmemserv. Рассмотрим исходный код, инициализирующий сервер:

```
key_t key;
int shmid;
struct memory_block * mblock;
key = ftok(FTOK_FILE, 1);
shmid = shmget(key, sizeof(struct memory_block), 0666 | IPC_CREAT);
mblock = (struct memory_block *) shmat(shmid, 0, 0);
```

Как уже отмечалось, ftok() генерирует ключ, используя в качестве «затравки» имя файла, в нашем случае - имя исполнимого файла сервера. Использование имени самой программы для генерации ключа до некоторой степени гарантирует уникальность ключа.

Функции для работы с разделяемой памятью объявлены в файлах sys/ipc.h sys/shm.h. Разделяемый блок памяти выделяется при помощи функции shmget(2), которой передаются три параметра. В первом параметре мы передаем ключ, идентифицирующий выделяемый блок памяти. Второй параметр позволяет указать размер блока в байтах. В третьем параметре передается маска прав доступа и флаги, аналогичные флагам msgget(). Функция shmget() возвращает идентификатор выделенного блока памяти (его не следует путать с указателем на блок). Для того чтобы получить указатель на созданный блок разделяемой памяти, этот блок нужно отобразить в локальное адресное пространство процесса. Отображение блока разделяемой памяти в локальное адресное пространство выполняет функция shmat(2). У этой функции тоже три параметра. Первый параметр, это идентификатор, возвращенный функцией shmget(). Во втором параметре передается *желательный* начальный адрес для отображения разделяемого блока в локальном адресном пространстве. Функция shmat() постарается отобразить разделяемый блок в локальное пространство, начиная с указанного адреса, но успешный результат не гарантирован. Если во втором параметре shmat() передать нулевое значение, функция сама выберет начальный адрес области отображения. Значение желательного адреса должно быть выравнено по границе страничных областей. Можно также не выравнивать адрес, но передать в третьем параметре функции флаг SHM_RND, и тогда функция сама скорректирует значение адреса. Среди дополнительных флагов, которые можно передать в третьем параметре, отметим флаг SHM_RDONLY, который присваивает отображаемой области статус «только для чтения».

При успешном выполнении функция shmat() возвращает указатель на начало области отображения, с которым мы можем работать как с обычным указателем на выделенный блок памяти. Для того чтобы понять дальнейшую работу сервера, следует иметь в виду, что сами по себе объекты разделяемой памяти не предоставляют никаких средств синхронизации доступа, так что нам приходится самим позаботиться об этих средствах. Для синхронизации работы клиента и сервера, и разграничения доступа мы используем упомянутый уже алгоритм Петерсона [2], который позволяет разграничить доступ к блоку разделяемой памяти, используя неатомарные операции.

Неатомарность спин-блокировок

Простейший алгоритм разделения доступа (который сразу приходит в голову) можно описать схематически с помощью следующей конструкции:

```
while (spin_lock == TRUE);
spin_lock = TRUE;
... // Доступ к разделяемому ресурсу
spin_lock = FALSE;
```

Проблема заключается в том, что этот алгоритм простых спин-блокировок (spin locks) не только прожорлив, но и не гарантирует надежного разграничения

доступа. Допустим, что один процесс начал проверять значение переменной spin_lock, дождался того момента, когда оно станет равно FALSE и переходит к строке

```
spin_lock = TRUE;
```

В многозадачной и системе (особенно на нескольких процессорах) другой процесс, ожидающий доступа к разделяемому ресурсу, может «вклиниться» в тот момент, когда первый процесс уже проверил, но еще не изменил значение spin_lock. Второй процесс проверит значение, все еще равное FALSE и в результате оба процесса окажутся в критической области и получат доступ к разделяемому ресурсу. Описанная проблема вызвана тем, что операция «проверить значение – изменить значение» неатомарна, то есть, ее выполнение может быть прервано другим процессом. Первый алгоритм, гарантирующий синхронизацию при использовании неатомарных операций, придумал Т. Деккер, а применил Э. Дейкстра в 1965 году. В 1981 году Г. Петерсон предложил более простой алгоритм.

Функция shmdt(2) удаляет область отображения в локальном адресном пространстве (но не удаляет блок разделяемой памяти). Блок разделяемой памяти удаляется вызовом

```
shmctl(shmid, IPC_RMID, 0);
```

который и по форме, и по сути подобен приведенному выше вызову msgctl(). Клиент получает доступ к блоку разделяемой памяти с помощью вызовов

```
shmid = shmget(key, sizeof(struct memory_block), 0666);
if (shmid == -1) {
  printf("Server is not running!\n");
  return EXIT_FAILURE;
}
mblock = (struct memory_block *) shmat(shmid, 0, 0);
```

При этом мы проверяем, запущен ли сервер. Далее клиент читает информацию, записанную в разделяемый блок сервером, считывает строку с терминала и передает строку серверу, используя механизм спин-блокировок. Скомпилируйте обе программы (скомандовав make shmemdemo), запустите сервер, затем клиент и набирайте строки в окне клиента. Работа программ завершится когда вы нажмете q и затем Ввод.

Поработав с программами, вы, конечно, обратите внимание на медлительность, с которой сервер отвечает клиенту. Кроме того, вы можете заметить существенный рост потребления ресурсов процессора при работе программ. Виной всему спин-блокировки, которые используются в алгоритме Петерсона. Именно из-за спин-блокировок процессор проводит значительную часть времени в цикле непрерывного опроса значения переменной. Сам алгоритм Петерсона сегодня можно найти только в учебниках по разработке операционных систем, хотя и современные ОС его практически не используют. Вместо этого ОС создают объекты синхронизации, контролирующие доступ к критическим секциям с помощью специальных атомарных операций процессора, и предоставляют пользовательским программам доступ к этим объектам. Именно такими объектами являются рассмотренные далее семафоры.

Семафоры

Семафоры широко используются как средство синхронизации потоков и процессов. В Unix-системах реализованы три типа семафоров – семафоры System V, семафоры POSIX и семафоры в разделяемой памяти. Поскольку статья посвящена System V IPC, мы рассмотрим семафоры System V. Подробное описание всех трех типов семафоров можно найти в [3].

Состояние семафора определяется значением некоторой внутренней переменной и переданным ему параметром. В зависимости от этих значений семафор либо приостанавливает выполнение обратившегося к нему потока (до тех пор, пока другой поток не переведет семафор в другое состояние), либо изменяет значение внутренней переменной, разрешив потоку дальнейшее выполнение. Следующая функция иллюстрирует логику работы семафора в зависимости от значения переменной состояния (semvalue) и управляющей переменной sem_op.

```
void semaphore (int sem_op)
{
   static int semvalue; // Внутренняя переменная
   if (sem_op != 0) {
     if (sem_op < 0)  while (semvalue < ABS(sem_op));
     semvalue += sem_op;
   }
   else while (semvalue != 0);
}
```

Отрицательное значение sem_op соответствует операции проверки доступности ресурса и вызывает приостановку потока, если доступ к ресурсу заблокирован. Положительное значение сигнализирует о высвобождении ресурса. Приведенная выше функция semaphore() описывает поведение многозначного семафора (general semaphore). Именно такие семафоры используются в System V IPC.

Перепишем клиент и сервер из предыдущего примера, заменив спин-блокировки семафорами (на диске вы найдете исходный текст сервера в файле semserv.c, а исходный текст клиента – в файле semcli.c. Все идентификаторы, связанные с семаформаи, определены в файле <sys/sem.h>. Программа-сервер создает семафоры с помощью вызова

```
semid = semget(key, 2, 0666|IPC_CREAT);
```

Функция semget(2) похожа на msgget() и shmget(), но у нее есть дополнительный параметр – количество создаваемых семафоров. Дело в том, что многим процессам, использующим семафоры, требуется более одного семафора (тогда как блоки разделяемой памяти и очереди сообщений обычно существуют в единственном экземпляре). Определение уникального ключа для каждого из нескольких семафоров затруднительно, поэтому функция semget() позволяет создавать несколько семафоров сразу. Нашему приложению понадобится два семафора (которые мы и создаем с помощью semget()). Нам понадобилось бы три семафора, если бы... Впрочем, это уже совсем другая история. Первый семафор указывает, должен ли сервер читать запись, сделанную клиентом, второй - должен ли клиент читать запись, сделанную сервером. В случае успешного завершения функция semget() возвращает идентификатор нового массива семафоров.

Для определения состояния семафора используется структура sembuf. В структуре sembuf определено много полей (конкретный набор зависит от версии Unix-системы), из которых обязательными являются три:

- short sem_num – номер семафора (в массиве), над которым выполняется операция (нумерация начинается с нуля).
- short sem_op – число, изменяющее состояние семафора.
- short sem_flg – дополнительные флаги.

Как и в приведенном выше схематическом примере работы семафора, отрицательное значение sem_op соответствует операции проверки доступности ресурса и вызывает приостановку процесса, если ресурс недоступен. Положительное значение заставляет семафор высвободить ресурс (или приблизиться к этому). Указатель на массив структур sembuf (по одной структуре на семафор) передается как второй параметр функции semop(2), которая либо изменяет состояние семафора, либо приостанавливает вызывавший поток. Первый

параметр этой функции – идентификатор, возвращенный semget(). В третьем параметре semop() передается число записей в массиве sembuf. Вот как, например, мы указываем, что клиент может записывать данные в разделяемую область:

```
buf[1].sem_op = 1;
semop(semid, (struct sembuf*) &buf[1], 1);
```

А эти строки приостановят выполнение сервера до тех пор, пока клиент не изменит значение первого семафора:

```
buf[0].sem_op = -1;
semop(semid, (struct sembuf*) &buf, 1);
```

Получая разрешение на доступ к разделяемой области, процесс производит чтение/запись, разрешает доступ другому процессу, запрещает доступ себе и приостанавливается. Удаление семафора выполняет с помощью функции semctl(), в которой, кроме прочего, нужно указывать число семафоров:

```
semctl(semid, 2, IPC_RMID);
```

Скомпилируйте сервер под именем semserv, а клиент под именем semcli, (или командуйте make semdemo) и запустите клиент и сервер. Вы увидите, что обмен данными выполняется гораздо быстрее, а процессор загружается гораздо меньше, чем в случае использования спин-блокировок.

При всем богатстве выбора средств взаимодействия между процессами в Unix/Linux, самыми популярными средствами были и остаются сокеты. Ими мы и займемся в следующий раз.

Литература

1. W. R. Stevens, S. A. Rago, **Advanced Programming in the UNIX® Environment: Second Edition,** Addison Wesley Professional, 2005
2. Таненбаум Э. С., Вудхалл А. С., **Операционные системы: разработка и реализация.** - СПб.: Питер, 2005
3. Стивенс У., **UNIX: Взаимодействие процессов.** - СПб.: Питер, 2003