

Обзор методов межпроцессного взаимодействия в Linux

Наумов Д.А., доц. каф. КТ

Операционные системы и системное программное обеспечение,
2019

Содержание лекции

- 1 Обзор методов межпроцессного взаимодействия в Linux
 - Общие сведения о межпроцессном взаимодействии в Linux
 - Локальные методы межпроцессного взаимодействия
 - Удаленное межпроцессное взаимодействие
- 2 Сигналы
- 3 Использование общей памяти
- 4 Использование общих файлов
- 5 Каналы
- 6 Именованные каналы FIFO

В основе межпроцессного взаимодействия (IPC, InterProcess Communication) лежит обмен данными между работающими процессами.

Межпроцессное взаимодействие в Linux можно классифицировать по трем критериям:

- По широте охвата взаимодействие бывает локальным и удаленным. Локальное взаимодействие подразделяется на родственное и неродственное.
- По направлению передачи данных межпроцессное взаимодействие бывает однонаправленным и двунаправленным.
- По характеру доступа взаимодействие бывает открытым и закрытым.

Закрытое взаимодействие

взаимодействие осуществляется только между двумя процессами.

Открытое взаимодействие

какой-нибудь другой процесс может присоединиться к обмену данными

Локальное взаимодействие

отвечает за обмен данными между процессами, которые работают в одной Linux-системе. Если обмен данными осуществляется между родительским и дочерним процессами, то такое взаимодействие называется родственным.

Удаленное взаимодействие

это обмен данными между двумя процессами, которые работают в разных системах.

Однонаправленное межпроцессное взаимодействие

характеризуется тем, что один процесс является отправителем данных, другой — приемником этих данных.

Двунаправленное взаимодействие

позволяет процессам общаться на равных, но в этом случае процессы должны "договориться" об очередности и синхронизации приема и передачи информации.

Существование различных способов межпроцессного взаимодействия обусловлено тем, что при их реализации возникают две проблемы:

- 1 *Проблема синхронизации.* Процессы работают независимо друг от друга, в ходе обмена данными между процессами должна учитываться эта "независимость".
- 2 *Проблема безопасности.* Если один процесс направляет данные другому процессу, то возникает опасность того, что потенциальный злоумышленник может перехватить эти данные. Проблема усложняется еще и тем, что злоумышленник может "подсунуть" процессу-приемнику собственные данные, если последние не защищены. Чем шире зона охвата межпроцессного взаимодействия, тем более уязвимым (с точки зрения безопасности) оно является.

- Простейший способ межпроцессного взаимодействия — локальное родственное двунаправленное закрытое взаимодействие родителя и потомка.
- Родительский процесс может передавать дочернему некоторые данные посредством аргументов через одну из функций семейства `exec()`.
- В свою очередь дочерний процесс при нормальном завершении сообщает родителю свой код возврата.

Пример: kinsfolk-child1.c

```
$ gcc -o kinsfolk-child1 kinsfolk-child1.c
$ ./kinsfolk-child1 2007
$ echo $?
0
$ ./kinsfolk-child1 2008
$ echo $?
1
$ ./kinsfolk-child1 2300
$ echo $?
0
$ ./kinsfolk-child1 2400
$ echo $?
1
```

Пример: kinsfolk-parent.c

```
$ gcc -o kinsfolk-parent1 kinsfolk-parent1.c
$ ./kinsfolk-parent1 2000
2000: leap year
$ ./kinsfolk-parent1 2007
2007: not leap year
```


Удаленное межпроцессное взаимодействие

Сокет

комплексное понятие, которое условно можно назвать "точкой соединения процессов"

- Unix-сокеты для локального взаимодействия процессов;
- Интернет-сокеты для удаленного взаимодействия процессов.

В программах сокеты фигурируют в виде файловых дескрипторов, над которыми (во многих случаях) можно осуществлять обычные операции чтения-записи (`read()`, `write()` и т. д.).

- При взаимодействии посредством сокетов процессы рассматриваются по схеме "клиент — сервер".
- Процесс-сервер устанавливает "правила общения" и предлагает всем желающим межпроцессное взаимодействие.

Понятие сигнала

Сигнал

сообщение, которое один процесс-отправитель посылает другому процессу или самому себе.

- В распоряжении процессов находится стандартный набор сигналов, заранее определенных ядром Linux.
- Каждый сигнал имеет свой уникальный номер, а также символическую константу, соответствующую этому номеру.

Процесс-получатель может отреагировать на сигнал одним из следующих трех способов:

- **Принятие сигнала.** Обычно это приводит к немедленному завершению процесса.
- **Игнорирование сигнала.** Процесс может установить для себя политику игнорирования определенных сигналов.
- **Перехватывание и обработка сигнала.** Процесс может задать собственное поведение в отношении конкретного сигнала.

```
$ kill -s SIGNAL PID
```

Здесь SIGNAL — это символическая константа посылаемого сигнала, PID — идентификатор процесса-получателя сигнала.

```
$ yes > /dev/null &
```

```
[1] 5852
```

```
$ ps
```

```
PID TTY TIME CMD
```

```
5481 pts/2 00:00:00 bash
```

```
5852 pts/2 00:00:04 yes
```

```
5853 pts/2 00:00:00 ps
```

```
$ kill -s SIGINT 5852
```

```
$ ps
```

```
PID TTY TIME CMD
```

```
5481 pts/2 00:00:00 bash
```

```
5854 pts/2 00:00:00 ps
```

```
[1]+ Interrupt yes >/dev/null
```

Отправка сигнала `kill()`

Для отправки сигнала предусмотрен системный вызов `kill()`, который объявлен в заголовочном файле `signal.h` следующим образом:

```
int kill (pid_t PID, int SIGNAL);
```

Этот системный вызов посылает процессу с идентификатором `PID` сигнал `SIGNAL`. Возвращаемое значение — 0 (при успешном завершении) или -1 (в случае ошибки).

Программа `kill1.c`

Если в аргументе `PID` системного вызова `kill()` указать текущий идентификатор, то процесс пошлет сигнал сам себе.

Программа `kill2.c`

Обработка сигнала `sigaction()`

Системный вызов `sigaction()` позволяет задавать поведение процесса по отношению к конкретным сигналам:

```
int sigaction (int SIGNAL, const struct sigaction * ACTION,  
              struct sigaction * OLDACTION);
```

Данный системный вызов устанавливает политику реагирования процесса на сигнал `SIGNAL`. Политика определяется указателем `ACTION` на структуру типа `sigaction`. При удачном завершении функции по адресу `OLDACTION` заносится прежняя политика в отношении сигнала.

Структура `sigaction` содержит различные поля:

- `sa_handler` — адрес функции-обработчика сигнала;
- `sa_flags` — набор флагов. Для простой обработки сигнала достаточно обнулить это поле;
- `sa_mask` — маска сигналов. Список сигналов, которые будут заблокированы во время работы функции-обработчика.

Пример sigaction1.c

Системный вызов `sigaction()` позволяет задавать поведение процесса по отношению к конкретным сигналам:

```
$ gcc -o sigaction1 sigaction1.c
$ ./sigaction1
signal...
signal...
signal...
signal...
```

Обратите внимание, что каждый раз при нажатии клавиш `Ctrl+C` процесс реагирует мгновенно. При использовании сигналов всегда следует помнить, что на момент получения сигнала программа может выполнять что-то важное.

```
$ ps -e | grep sigaction1
4883 pts/1 00:44:27 sigaction1
$ kill 4883
```

Сигналы и многозадачность

Если во время выполнения функции-обработчика процесс получает еще один сигнал, программа может начать вести себя неадекватно.

- включить в обработчик сигнала минимальное число инструкций. Лучше, если это будет одна инструкция, которая просто устанавливает флаг, регистрирующий получение сигнала.
- программа может при необходимости проверять значение этого флага и реагировать соответствующим образом.
- применяется специальный тип данных `sig_atomic_t` - обычное целое число, но ядро Linux гарантирует, что математические операции над таким числом являются атомарными и не могут быть прерваны.

Пример 1: `sigaction2.c`

Пример 2: `kinsfolk-child2.c`, `kinsfolk-parent2.c`

Использование общей памяти

Реализация межпроцессного взаимодействия посредством совместно используемой памяти:

- ❶ один из процессов выделяет некоторый объем памяти, который называется сегментом. К общему сегменту памяти привязаны два числа:
 - Идентификатор сегмента — служит для доступа к сегменту внутри процесса.
 - Ключ сегмента — идентифицирует сегмент для других процессов. Ключ может быть выделен автоматически во время создания сегмента. Процессы могут также заранее договориться о применении определенного статического ключа.
- ❷ после выделения совместно используемого сегмента другие процессы могут задействовать его.

Выделение общей памяти

Выделение общей памяти осуществляется при помощи системного вызова `shmget()`, который объявлен в заголовочном файле `sys/shm.h` следующим образом:

```
int shmget (key_t KEY, size_t SIZE, int FLAGS);
```

- **KEY** — это ключ сегмента. Если взаимодействующие программы не договорились заранее о применении статического ключа, то в данном поле можно указать константу `IPC_PRIVATE`, которая иницирует динамическое выделение ключа.
- **SIZE** — размер сегмента. Здесь следует учитывать, что данный аргумент является запрашиваемым числом байтов для сегмента. Реально выделенное число байтов может отличаться от `SIZE`.
- Аргумент **FLAGS** определяет права доступа к совместно используемому сегменту, а также дополнительные флаги.

Активизация совместного доступа

Процесс может получить доступ к созданному сегменту двумя способами:

- Использовать идентификатор сегмента, полученный от другого процесса.
- Вызвать `shmget()`, применяя для этого заранее известный ключ.

Для работы с общим сегментом памяти нужно, чтобы каждый из взаимодействующих процессов обратился к системному вызову `shmat()`

```
void * shmat (int ID, void * ADDRESS, int FLAGS);
```

Данный системный вызов возвращает адрес совместно используемого сегмента памяти.

- ID — это идентификатор сегмента.
- ADDRESS - адрес, по которому будет доступен общий сегмент памяти.
- FLAGS - дополнительные флаги.

Отключение совместного доступа

Системный вызов `shmdt()` вызывается в каждом процессе после того, как тот закончил работу с общей памятью.

```
int shmdt (void * ADDRESS);
```

- `ADDRESS` — это адрес совместно используемой памяти, который возвращает `shmat()` при подключении сегмента.

Системный вызов `shmctl()` позволяет осуществлять различные операции над общим сегментом памяти.

```
int shmctl (int ID, int COMMAND, struct shmid_ds * DESC);
```

- `ID` — это идентификатор сегмента.
- `COMMAND` — команда, которую требуется выполнить в отношении сегмента общей памяти с идентификатором `ID`. Часто употребляются две команды: `IPC_STAT` (получить данные о сегменте) и `IPC_RMID` (удалить сегмент).
- `DESC` — указатель на структуру, в которую (для команды `IPC_STAT`) заносятся данные о сегменте.

Пример

shm1-owner.c

- Программа создает общий сегмент с использованием "динамического ключа" (IPC_PRIVATE).
- После успешного создания и подключения сегмента программа выводит на экран его идентификатор, заносит в сегмент данные и останавливается до тех пор, пока пользователь не нажмет клавишу Enter.
- Эта задержка позволяет другому процессу подключить общий сегмент памяти и прочитать оттуда данные.

shm1-user.c

- Поскольку процессы "не договорились" заранее о выделении общего ключа для сегмента памяти, то единственным адекватным способом взаимодействия будет непосредственная передача программе-клиенту идентификатора сегмента через аргумент.

Пример

В первом окне запускаем процесс-сервер:

```
$ gcc -o shm1-owner shm1-owner.c  
$ ./shm1-owner  
ID: 31391750  
Press <Enter> to exit...
```

Теперь, не нажимая клавиши <Enter>, переходим в другое терминальное окно и запускаем процесс-клиент:

```
$ gcc -o shm1-user shm1-user.c  
$ ./shm1-user 31391750  
Message: Hello World!
```

Пример

Наберите в клиентском окне команду `ipcs -m`. Эта команда выводит на экран список совместно используемых сегментов памяти с их ключами и идентификаторами.

```
$ ipcs -m
----- Shared Memory Segments -----
key shmid owner perms bytes nattch status
0x00005d8b 13565953 root 777 316 1
0x00000000 13795330 nn 600 393216 2 dest
0x00000000 13828099 nn 600 393216 2 dest
0x00000000 29589508 nn 666 112320 1 dest
0x00000000 14811141 nn 777 393216 2 dest
0x00000000 31391750 nn 600 4096 1
```

Пример

Теперь нажмите клавишу Enter в исходном терминале и вызовите `ipcs -m` еще раз:

```
$ ipcs -m
----- Shared Memory Segments -----
key shmid owner perms bytes nattch status
0x00005d8b 13565953 root 777 316 1
0x00000000 13795330 nn 600 393216 2 dest
0x00000000 13828099 nn 600 393216 2 dest
0x00000000 29589508 nn 666 112320 1 dest
0x00000000 14811141 nn 777 393216 2 dest
```

Использование общих файлов

Системный вызов `mmap()` позволяет частично или целиком отображать в оперативной памяти содержимое файла.

```
void * mmap (void * ADDRESS, size_t LEN, int PROT,  
            int FLAGS, int FD, off_t OFFSET);
```

- ADDRESS - адрес, по которому будет отображаться файл. Если указать здесь NULL, то адрес будет выбран автоматически.
- LEN - размер отображаемой области.
- PROT - уровень защиты отображаемой области. Этот аргумент может состоять из побитовой дизъюнкции флагов PROT_READ (чтение), PROT_WRITE (запись) и PROT_EXEC (выполнение).
- FLAGS - при реализации межпроцессного взаимодействия этот аргумент обычно принимает значение MAP_SHARED.
- FD — дескриптор открытого файла, который будет отображаться в памяти.
- OFFSET — смещение, от которого будет отображаться файл.

Использование общих файлов

Системный вызов `munmap()` освобождает отображаемую область памяти.

```
int munmap (void * ADDRESS, size_t LEN);
```

- ADDRESS - буфер отображаемой памяти, который будет освобожден.
- LEN - размер отображаемой области.

Пример: `mmap1.c`

```
$ gcc -o mmap1 mmap1.c
$ echo "" > myfile
$ echo -n LINUX >> myfile
$ ./mmap1 myfile
$ cat myfile
XUNIL
```

Использование общих файлов

- Данные файла, отображаемые в памяти, на самом деле существуют отдельно от самого файла.
- Если отображаемая область изменяется, то сброс данных файла на носитель осуществляется только при вызове `munmap()`.
- Однако использование отображаемых в памяти файлов при реализации межпроцессного взаимодействия зачастую требует осуществить сброс данных на носитель (синхронизацию) без отключения отображаемой области.

Это позволяет делать системный вызов `msync()`

```
int msync (void * ADDRESS, size_t LEN, int FLAGS);
```

- `ADDRESS` - буфер отображаемой памяти, который будет освобожден.
- `LEN` - размер отображаемой области.

Создание канала: `pipe()`

- Каналы осуществляют в Linux родственное локальное межпроцессное взаимодействие.
- Интерфейс канала представляет собой два связанных файловых дескриптора, один из которых предназначен для записи данных, другой — для чтения.
- Возможность родственного межпроцессного взаимодействия через канал обусловлена тем, что дочерние процессы наследуют от родителей открытые файловые дескрипторы.

Для создания канала служит системный вызов `pipe()`:

```
int pipe (int PFDS[2]);
```

При успешном завершении системного вызова `pipe()` в текущем процессе появляется канал

- запись в который осуществляется через дескриптор `PFDS[0]`,
- чтение - через `PFDS[1]`.

Создание канала: `pipe()`

- Данные, передаваемые через канал, практически не нуждаются в разделении доступа.
- Канал автоматически блокирует читающий или пишущий процесс, когда это необходимо.
- Если читающий процесс запрашивает больше данных, чем есть в настоящий момент в канале, то процесс блокируется до тех пор, пока в канале не появятся новые данные.

Заккрытие "концов" канала осуществляется при помощи системного вызова `close()`

```
int pipe (int PFDS[2]);
```

- закрытие дескриптора происходит только тогда, когда все процессы, разделяющие этот дескриптор, вызовут `close()`.

Создание канала: pipe()

Пример: pipe1-parent.c, pipe1-src.c, pipe1-dst.c

```
$ gcc -o pipe1-parent pipe1-parent.c
```

```
$ gcc -o pipe1-src pipe1-src.c
```

```
$ gcc -o pipe1-dst pipe1-dst.c
```

```
$ ./pipe1-parent
```

```
Wait please.....
```

```
Hello World
```

Перенаправление ввода-вывода: dup2()

Системный вызов dup2() позволяет перенаправлять ввод-вывод.

```
int dup2 (int FD1, int FD2);
```

Системный вызов dup2() перенаправляет ввод-вывод с дескриптора FD2 на дескриптор FD1.

Пример: dup01.c. Программа выводит сообщение "Hello World!" посредством функции printf(). Но системный вызов dup2() перенаправил стандартный вывод (дескриптор 1) в файл. Поэтому после работы программы текст "Hello World!" окажется не на экране терминала, а в файле myfile:

```
$ gcc -o dup01 dup01.c
$ ./dup01
$ cat myfile
Hello World!
```

Перенаправление ввода-вывода: dup2()

Пример: dup02.c.

Программа осуществляет перенаправление стандартного ввода: при попытке прочитать что-нибудь из стандартного ввода программа выведет на экран содержимое файла myfile:

```
$ gcc -o dup02 dup02.c
$ echo "Hello World" > myfile
$ ./dup02
Hello World
```

Перенаправление ввода-вывода: dup2()

Аргументами `dup2()` могут быть дескрипторы канала. Это позволяет налаживать взаимодействие любых процессов, которые используют в своей работе консольный ввод-вывод.

Пример: `pipe2.c`.

- программа принимает два аргумента: имя каталога и произвольную строку.
- программа вызывает команду `ls` для первого аргумента и `grep -i` для второго.
- при этом стандартный вывод `ls` связывается со стандартным вводом `grep`.
- В итоге на экран выводятся все элементы указанного каталога, в которых содержится строка (без учета регистра символов) из второго аргумента.

```
$ gcc -o pipe2 pipe2.c
```

```
$ ./pipe2 / bin
```

```
bin
```


Именованные каналы FIFO

- Именованные каналы FIFO работают аналогично обычным.
- Особенность FIFO в том, что они представлены файлами специального типа и "видны" в файловой системе.
- Работа с этими файлами осуществляется обычным образом при помощи системных вызовов `open()`, `close()`, `read()`, `write()` и т. д. Это, позволяет использовать их вместо обычных файлов.
- Через FIFO могут взаимодействовать процессы, не являющиеся ближайшими родственниками.

Именованные каналы создаются командой `mkfifo` и удаляются командой `rm`:

```
$ mkfifo myfifo
$ ls -l myfifo
prw-r--r-- 1 nn nn 0 2011-05-11 09:57 myfifo
$ rm myfifo
```

В расширенном выводе программы `ls` каналы FIFO обозначаются

Именованные каналы FIFO

Работать с FIFO можно даже без написания специальных программ. Проведем небольшой эксперимент:

```
$ mkfifo myfifo  
$ cat myfifo
```

Поскольку канал FIFO пуст, то программа cat оказалась заблокированной. Для снятия блокировки откроем новое окно терминала и введем следующую команду:

```
$ echo "Hello World" > myfifo
```

В первом окне будет написано сообщение "Hello World" и программа cat завершится.

Именованные каналы FIFO

Для создания именованных каналов FIFO предусмотрен системный вызов `mkfifo()`

```
int mkfifo (const char * FILENAME, mode_t MODE);
```

- FILENAME - это имя файла,
- MODE - права доступа.

Создание именованного канала FIFO, имя которого передается через аргумент:

```
if (mkfifo (argv[1], 0640) == -1) {  
    fprintf (stderr, "Can't make fifo");  
    return 1;  
}
```

Каналы FIFO функционируют аналогично обычным каналам.

- Если канал опустел, то читающий процесс блокируется до тех пор, пока не будет прочитано заданное количество данных или пишущий процесс не вызовет `close()`.
- И наоборот, пишущий процесс "засыпает" если данные не успевают считываться из FIFO.

Пример: `fifo2-server.c`, `fifo2-client.c`

Если запустить "сервер" то он создаст FIFO и "замрет" в ожидании поступления данных:

```
$ gcc -o fifo2-server fifo2-server.c  
$ ./fifo2-server
```

Запустим теперь в другом терминальном окне клиентскую программу:

```
$ gcc -o fifo2-client fifo2-client.c  
$ ./fifo2-client Hello
```

Серверный процесс тут же "проснется" и выведет сообщение "Hello" и завершится.