

8. Введение в межпроцессное взаимодействие

Наличие в Unix-системах простых и эффективных средств взаимодействия между процессами оказало программирование в Unix не менее важное влияние, чем представление объектов системы в виде файлов. Благодаря межпроцессному взаимодействию (*Inter-Process Communication, IPC*) разработчик (и пользователь) может разбить решение сложной задачи на несколько простых операций, каждая из которых доверяется отдельной небольшой программе. Последовательная обработка одной задачи несколькими простыми программами очень похожа на конвейерное производство (среди многих значений английского *pipeline* есть и <конвейер>, но в этом разделе мы для перевода слова *pipe* будем пользоваться принятым в отечественной литературе термином <канал>. Альтернативой конвейерному подходу являются большие монолитные пакеты, построенные по принципу <все в одном>. Использование набора простых утилит для решения одной сложной задачи требует несколько большей квалификации со стороны пользователя, но взамен предоставляет гибкость, не достижимую при использовании монолитных <монстров>. Наборы утилит, использующих открытые протоколы IPC, легко наращивать и модифицировать. Разбиение сложных задач на сравнительно небольшие подзадачи также позволяет снизить количество ошибок, допускаемых программистами. Помимо всего этого у IPC есть еще одно важное преимущество. Программы, использующие IPC, могут <общаться> друг с другом практически также эффективно, как и с пользователем, в результате чего появляется возможность автоматизировать выполнение сложных задач. Могущество скриптовых языков Unix и Linux во многом основано на возможностях IPC.

Закон Брукса

Фредерик Брукс, автор книги <Мифический человеко-месяц>, высказал предположение (известное как <закон Брукса>), что количество ошибок в проекте должно быть пропорционально квадрату числа участников проекта, тогда как объем полезной работы при увеличении числа участников проекта растет линейно. Закон Брукса означал, что на определенном этапе развития проекта любая попытка привлечь к разработке новых программистов приводит лавинообразному росту числа ошибок (а значит все больше и больше работы требуется на их выявление и исправление). Если бы закон Брукса выполнялся, то для всех проектов существовал бы верхний порог сложности, при попытке превысить который КПД разработки начинал бы стремительно падать. Что же касается открытой модели разработки ПО, то она, с точки зрения закона Брукса, была бы невозможна в принципе. Для того, чтобы понять, в чем Ф. Брукс ошибался, следует рассмотреть исходные посылки его рассуждений. Закон Брукса основан двух предположениях (а) ошибки чаще возникают на стыке элементов проекта, выполняемых разными разработчиками (соответственно, чем больше таких <швов>, тем больше ошибок); (б) модель взаимодействия разработчиков представляет собой полный граф (то есть, каждый разработчик взаимодействует со всеми остальными участниками проекта), число ребер которого пропорционально квадрату числа вершин. Ни то, ни другое утверждение, вообще говоря, неверно. В частности, при программировании <канальных> утилит всем участникам проекта нет необходимости непосредственно контактировать между собой. Каждая группа разработчиков должна следовать только фиксированному протоколу обмена данными между программами, так что в этом случае число ошибок подчиняется линейной, а не квадратичной, зависимости.

В этом разделе мы ограничимся рассмотрением IPC с помощью каналов различных типов. С

точки зрения программиста работа, программ в канале, организованном с помощью символа "|", выглядит очень просто. Данные со стандартного потока вывода одной программы перенаправляются на стандартный поток ввода другой программы, чей стандартный поток вывода может быть также перенаправлен. Но как быть в том случае, если необходимо использовать канал внутри самой программы?

8.1. Неименованные каналы

Самый распространенный вариант внутри-программного использования каналов: программа запускает другую программу и считывает данные, которые та выводит в свой стандартный поток вывода. С помощью этого трюка разработчик может использовать в своей программе функциональность другой программы, не вмешиваясь во внутренние детали ее работы. Для решения этой задачи мы воспользуемся функциями `popen(3)` и `pclose(3)`. Формально эти функции подобны функциям `fork(3)` и `fclose(3)`. Функция `popen()` запускает внешнюю программу и возвращает вызвавшему ее приложению указатель на структуру `FILE`, связанный либо со стандартным потоком ввода, либо со стандартным потоком вывода запущенного процесса. Первый параметр функции `popen()` - строка, содержащая команду, запускающую внешнюю программу. Второй параметр определяет, какой из стандартных потоков (вывода или ввода) будет возвращен. Аргумент "w" соответствует потоку ввода запускаемой программы, в этом случае приложение, вызвавшее `popen()`, записывает данные в поток. Аргумент "r" соответствует потоку вывода. Функция `pclose()` служит для завершения работы с внешним приложением и закрытием канала. Программа `makelog` выполняет команду оболочки, переданную ей в качестве параметра и записывает данные, выводимые этой командой, одновременно на стандартный терминал и в файл `log.txt` (аналогичными функциями обладает стандартная команда `tee`). Например, если скомпилировать программу:

```
gcc makelog.c -o makelog
```

```
а затем скомандовать  
makelog "ls -al"
```

на экране терминала будут распечатаны данные, выводимые командой оболочки `ls -al`, а в рабочей директории программы `makelog` будет создан файл `log.txt`, содержащий те же данные. Кавычки вокруг команды оболочки нужны для того, чтобы программа `makelog` получала строку вызова команды как один параметр командной строки.

Изюминка программы `makelog` заключается в использовании функции `popen()`. Рассмотрим фрагмент исходного текста программы:

```
f = popen(argv[1], "r");
```

Эта операция очень похожа на открытие обычного файла для чтения. Переменная `f` имеет тип `FILE *`, но в параметре `argv[1]` функции `popen` передается не имя файла, а команда на запуск программы или команды оболочки, например, `"ls -al"`. Если вызов `popen()` был успешен, мы можем считывать данные, выводимые запущенной командой, с помощью обычной функции `fread(3)`:

```
fread(buf, 1, BUF_SIZE, f)
```

Особенность функции `popen()` заключается в том, что эта функция не возвращает `NULL`, даже если переданная ей команда не является корректной. Самый простой способ обнаружить ошибку в этой ситуации - попытаться прочесть данные из потока вывода. Если в потоке вывода нет данных (`fread()` возвращает значение 0), значит произошла ошибка. Для вывода данных, прочитанных с помощью `fread()`, на терминал мы используем функцию `write()` с указанием дескриптора стандартного потока вывода:

```
write(1, buf, len);
```

Параллельно эти же данные записываются в файл на диске. По окончании чтения данных открытый канал нужно закрыть:

```
pclose(f);
```

Следует иметь в виду, что `pclose()` вернет управление вызывающему потоку только после того как запущенное с помощью `pipe()` приложение завершит свою работу.

В заключение отметим еще одну особенность функции `pipe()`. Для выполнения переданной ей команды `pipe()` сперва запускает собственный экземпляр оболочки, что с одной стороны хорошо, а с другой - не очень. Хорошо это потому, что при вызове `pipe()` автоматически выполняются внутренние операции оболочки (такие как обработка шаблонов имен файлов), используются переменные окружения типа `PATH` и `HOME` и т.п. Отрицательная сторона подхода, применяемого `pipe()`, связана с дополнительными накладными расходами на запуск процесса оболочки в том случае, когда для выполнения команды оболочка не нужна.

Для обмена данными с внешним приложением функция `pipe()` использует каналы неявным образом. В своих программах мы можем использовать каналы и непосредственно. Наиболее распространенный тип каналов, - неименованные однонаправленные каналы (anonymous pipes), создаваемые функцией `pipe(2)`. На уровне интерфейса программирования такой канал представляется двумя дескрипторами файлов, один из которых служит для чтения данных, а другой - для записи. Каналы не поддерживают произвольный доступ, т. е. данные могут считываться только в том же порядке, в котором они записывались. Неименованные каналы используются преимущественно вместе с функцией `fork(2)` и служат для обмена данными между родительским и дочерним процессами. Для организации подобного обмена данными, сначала, с помощью функции `pipe()`, создается канал. Функции `pipe()` передается единственный параметр - массив типа `int`, состоящий из двух элементов. В первом элементе массива функция возвращает дескриптор файла, служащий для чтения данных из канала (выход канала), а во втором - дескриптор для записи (вход). Затем, с помощью функции `fork()` процесс <раздваивается>. Дочерний процесс наследует от родительского процесса оба дескриптора, открытых с помощью `pipe()`, но, также как и родительский процесс, он должен использовать только один из дескрипторов. Направление передачи данных между родительским и дочерним процессом определяется тем, какой дескриптор будет использоваться родительским процессом, а какой - дочерним. Продемонстрируем изложенное на простом примере программы `pipes.c`, использующей функции `pipe()` и `fork()`.

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
int main (int argc, char * argv[])
{ int pipedes[2];
  pid_t pid;
  pipe(pipedes);
  pid = fork();
  if ( pid > 0 ) {
    char *str = "String passed via pipe\n";
    close(pipedes[0]);
    write(pipedes[1], (void *) str, strlen(str) + 1);
    close(pipedes[1]);
  } else {
    char buf[1024];
    int len;
    close(pipedes[1]);
    while ((len = read(pipedes[0], buf, 1024)) != 0)
      write(2, buf, len);
    close(pipedes[0]);
  }
```

```
}  
return 0;  
}
```

Оба дескриптора канала хранятся в переменной `pipedes`. После вызова `fork()` процесс раздваивается и родительский процесс (тот, в котором `fork()` вернула ненулевое значение, равное, кстати, PID дочернего процесса) закрывает дескриптор, открытый для чтения, и записывает данные в канал, используя дескриптор, открытый для записи (`pipedes[1]`). Дочерний процесс (в котором `fork()` вернула 0) первым делом закрывает дескриптор, открытый для записи, и затем считывает данные из канала, используя дескриптор, открытый для чтения (`pipedes[0]`). Назначение дескрипторов легко запомнить, сопоставив их с аббревиатурой I/O (первый дескриптор - для чтения (input), второй - для записи (output)). Стандарт POSIX предписывает, чтобы каждый процесс, получивший оба канальных дескриптора, закрывал тот дескриптор, который ему не нужен, перед тем, как начать работу с другим дескриптором, и хотя в системе Linux этим требованием можно пренебречь, лучше все же придерживаться строгих правил. В нашем примере нам не нужно беспокоиться о синхронизации передачи данных, поскольку ядро системы выполнит всю трудную работу за нас. Но в жизни встречаются и не столь тривиальные случаи. Например, ничто не мешает нам создать несколько дочерних процессов с помощью нескольких вызовов `fork()`. Все эти процессы могут использовать один и тот же канал, при условии, что каждый процесс использует только один из дескрипторов `pipedes`, согласно его назначению. В этой ситуации нам пришлось бы выполнять синхронизацию передачи данных явным образом.

Как канал передает данные

Для передачи данных по каналу используются специальные объекты ядра системы, называемые буферами каналов (pipe buffers). Даже если предыдущая запись заполнила буфер не полностью, повторная запись данных в буфер становится возможной только после того, как прежде записанные данные будут прочитаны. Это означает, что если разные процессы, пишущие данные в один и тот же канал, передают данные блоками, размеры которых не превышают объем буферов, данные из блоков разных процессов не будут перемешиваться между собой. Использование этой особенности каналов существенно упрощает синхронизацию передачи данных. Узнать размер буфера можно с помощью вызова функции

`fpathconf(pipedes, _PC_PIPE_BUF)`

где `pipedes` - дескриптор канала. На архитектуре IA32 размер буфера составляет 4 килобайта. Начиная с ядра 2.6.11, каждый канал может использовать до 16 буферов, что существенно повышает производительность каналов.

Познакомившись с неименованными каналами, мы можем самостоятельно реализовать аналог функции `open()` без <дополнительных расходов> (то есть, без запуска процесса оболочки). Напишем небольшую программу, которая запускает утилиту `netstat`, читает данные, выводимые этой утилитой, и выводит их на экран. Если бы мы использовали для этой цели функцию `open()`, то получили бы доступ к потоку вывода `netstat` с помощью

`popen("netstat", "r");`

и скопировали данные на экран. Этот способ прост, но не эффективен. Мы напишем другую программу (файл `printns.c`). Структура этой программы та же, что и в предыдущем примере, только теперь родительский процесс читает данные с помощью канала. Самое интересное

происходит в дочернем процессе, где выполняется последовательность функций:

```
close(pipedes[0]);  
dup2(pipedes[1], 1);  
execve("/bin/netstat", NULL, NULL);
```

С помощью функции `dup2(2)` мы перенаправляем стандартный поток вывода дочернего процесса (дескриптор стандартного потока вывода равен 1) в канал, используя дескриптор `pipdes[1]`, открытый для записи. Далее с помощью функции `execve(2)` мы заменяем образ дочернего процесса процессом `netstat` (обратите внимание, что поскольку в нашем распоряжении нет оболочки с ее переменной окружения `PATH`, путь к исполняемому файлу `netstat` нужно указывать полностью). В результате родительский процесс может читать стандартный вывод `netstat` через поток, связанный с дескриптором `pipdes[0]` (и никакой оболочки!).

8.2. Именованные каналы

Хотя в приведенном выше примере неименованные каналы используются только для передачи данных между процессами, связанными <родственными узами>, существует возможность использовать их и для передачи данных между совершенно разными процессами. Для этого нужно организовать передачу дескрипторов канала между неродственными процессами, однако, передача дескрипторов стороннему процессу носит скорее характер трюка (или <хака>), и мы на ней останавливаться не будем. Для передачи данных между неродственными процессами мы воспользуемся механизмом именованных каналов (named pipes), который позволяет каждому процессу получить свой, <законный> дескриптор канала. Передача данных в этих каналах (как, впрочем, и в однонаправленных неименованных каналах) подчиняется принципу FIFO (первым записан - первым прочитан), поэтому в англоязычной литературе иногда можно встретить названия FIFO pipes или просто FIFOs. Именованные каналы отличаются от неименованных наличием имени, то есть идентификатора канала, потенциально видимого всем процессам системы. Для идентификации именованного канала создается файл специального типа pipe. Это еще один представитель семейства виртуальных файлов Unix, не предназначенных для хранения данных (размер файла канала всегда равен нулю). Файлы именованных каналов являются элементами VFS, как и обычные файлы Linux, и для них действуют те же правила контроля доступа. Файлы именованных каналов создаются функцией `mkfifo(3)`. Первый параметр этой функции - строка, в которой передается имя файла, идентифицирующего канал, второй параметр - маска прав доступа к файлу. Функция `mkfifo()` создает канал и файл соответствующего типа. Если указанный файл канала уже существует, `mkfifo()` возвращает -1, (переменная `errno` принимает значение `EEXIST`). После создания файла канала процессы, участвующие в обмене данными, должны открыть этот файл либо для записи, либо для чтения. После закрытия файла канала, файл (и канал) продолжают существовать. Для того, чтобы закрыть сам канал, нужно удалить его файл, например с помощью последовательных вызовов `unlink(2)`.

Рассмотрим работу именованного канала на примере простой системы клиент-сервер. Программа-сервер создает канал и передает в него текст, вводимый пользователем с клавиатуры. Программа-клиент читает текст и выводит его на терминал. Программы из этого примера можно рассматривать как упрощенный вариант системы мгновенного обмена сообщениями между пользователями многопользовательской ОС. Исходный текст программы-сервера хранится в файле `typeserver.c`. Вызов функции `mkfifo()` создает файл-идентификатор канала в рабочей директории программы:

```
mkfifo(FIFO_NAME, 0600);
```

где `FIFO_NAME` - макрос, задающий имя файла канала (в нашем случае - `"/fifofile"`).

В качестве маски доступа мы используем восьмеричное значение 0600, разрешающее процессу с аналогичными реквизитами пользователя чтение и запись (можно было бы использовать маску 0666, но на мы на всякий случай воздержимся от упоминания Числа Зверя, пусть даже восьмеричного, в нашей программе). Для краткости мы не проверяем значение, возвращенное `mkfifo()`, на предмет ошибок. В результате вызова `mkfifo()` с заданными параметрами в рабочей директории программы должен появиться специальный файл `fifofile`. Далее в программе-сервере мы просто открываем созданный файл для записи:

```
f = fopen(FIFO_NAME, "w");
```

Считывание данных, вводимых пользователем, выполняется с помощью `getchar()`, а с помощью функции `fputc()` данные передаются в канал. Работа сервера завершается, когда пользователь вводит символ "q". Исходный текст программы-клиента можно найти в файле `typesclient.c`. Клиент открывает файл `fifofile` для чтения как обычный файл:

```
f = fopen(FIFO_NAME, "r");
```

Символы, передаваемые по каналу, считываются с помощью функции `fgetc()` и выводятся на экран терминала с помощью `putchar()`. Каждый раз, когда пользователь сервера нажимает ввод, функция `fflush()`, вызываемая сервером (см. файл `typeserver.c`), выполняет принудительную очистку буферов канала, в результате чего клиент считывает все переданные символы. Получение символа "q" завершает работу клиента.

Скомпилируйте программы `typeserver.c` и `typesclient.c` в одной директории. Запустите сначала сервер, потом клиент в разных окнах терминала. Печатайте текст в окне сервера. После каждого нажатия клавиши [Enter] клиент должен отображать строку, напечатанную на сервере.

Для создания файла FIFO можно воспользоваться также функцией `mknod(2)`, предназначенной для создания специальных файлов различных типов (FIFO, сокеты, файлы устройств и обычные файлы для хранения данных). В нашем случае вместо

```
mkfifo(fname, 0600);
```

можно было бы написать

```
mknod(fname, S_IFIFO, 0);
```

Одной из сильных сторон Unix/Linux IPC является возможность организовывать взаимодействие между программами, которые не только ничего не знают друг о друге, но и используют разные механизмы ввода/вывода. Сравним нашу программу `typesclient` и команду `ls`. Казалось бы, между ними нет ничего общего - `typesclient` получает данные, используя именованный канал, а `ls` выводит содержимое директории в стандартный поток вывода. Однако, мы можем организовать передачу данных от `ls` к `typesclient` с помощью всего лишь пары команд `bash`! В директории программы `typesclient` дайте команду:

```
mknod fifofile p
```

Эта команда создаст файл канала `fifofile` также, как это сделала бы программа `typeserver`. Запустите программу `typesclient`, а затем в другом окне терминала дайте команду, наподобие

```
ls -al > /path/fifofile
```

где `/path/fifofile` - путь к файлу FIFO. В результате, программа `typesclient` распечатает содержимое соответствующей директории. Главное, чтобы в потоке данных не встретился символ "q", завершающий ее работу.

Каналы представляют собой простое и удобное средство передачи данных, которое, однако,

подходит не во всех ситуациях. Например, с помощью каналов не очень удобно организовывать обмен асинхронными сообщениями между процессами.

Рекомендуемая литература:

D. P. Bovet, M. Cesati, Understanding the Linux Kernel, 3rd Edition, O'Reilly, 2005

W. R. Stevens, S. A. Rago, Advanced Programming in the UNIX® Environment: Second Edition, Addison Wesley Professional, 2005

Стивенс У., UNIX: Взаимодействие процессов. - СПб.: Питер, 2003