Тема 7. Взаимодействие процессов

Главные вопросы, которые мы обсудим, представлены на СЛАЙДЕ 1. По ходу лекции даются примеры основных коммуникационных системных вызовов.

10.1 Принципы и механизмы взаимодействия процессов

В одном из предыдущих разделов лекционного курса рассматривалась процедура создания процесса, и рассказывалось о том, как родительский процесс может получить статус завершения своего потомка. Это самая простая форма взаимодействия дочерних процессов, но не всегда самая эффективная. Известные нам к текущему моменту механизмы не позволяют контролировать выполняющийся процесс или обращаться к внешнему, независимому процессу.

Между собой могут общаться не только родительский и дочерний процессы, но также неродственные процессы в одной системе и даже расположенные на разных компьютерах.

Взаимодействие процессов – это механизм обмена данными между процессами. Очевидно, что когда Web-браузер запрашивает Web-страницу у сервера, тот должен в ответе выслать HTML-данные. Обычно используются сокеты, работающие через удаленное соединение. Еще один пример взаимодействия можно продемонстрировать. Например, можно вызвать команду:

```
$ ls | lpr
```

чтобы вывести на печать список файлов в каталоге. Оболочка (у нас – *Bash*) создает два отдельных процесса – *lp* и *lpr* и соединяет их **конвейером** (или **каналом**). Канал представляется символом вертикальной черты. Конвейер – это однонаправленный способ передачи данных от одного процесса к другому. Процесс *ls* записывает данные в конвейер, а процесс *lpr* считывает данные из него.

Таким образом, существуют разные способы взаимодействия процессов (СЛАЙД 2).

- Совместно используемая память, когда процессы могут читать и писать данные в рамках заданной области памяти.
- Отображаемая память, которая похожа на совместно используемую, но организуется связь с файлами.
- Конвейеры, которые позволяют последовательно передавать данные от одного процесса к другому.
- FIFO-файлы. В отличие от конвейеров с ними работают несвязанные процессы, поскольку у такого файла есть имя в ФС, и к нему может обращаться любой процесс.
- Сокеты, которые соединяют несвязанные процессы, работающие на разных компьютерах (далее будет использоваться термин «хост») и др.

Различия между способами взаимодействия определяются следующими критериями (СЛАЙД 3).

- Ограничено ли взаимодействие рамками связанных процессов (есть общий предок) или соединяются процессы, которые выполняются в одной ФС или на разных хостах.
- Ограничен ли процесс *readonly*- или *writeonly*-операциями.

- Число взаимодействующих процессов.
- Синхронизируются ли взаимодействующие процессы (например, должен ли процесс-писатель перейти в режим ожидания при заполнении системных буферов).

10.2 Совместно используемая память

Это, очевидно, простейший способ коммуникации процессов, который выглядит так, как если бы два или более процессов вызвали функцию распределения памяти и получили указатели на один и тот же сегмент памяти. Когда какой-то процесс изменяет содержимое памяти, это отражается на других процессах.

Это также быстрейший способ взаимодействия, т.к. процесс обращается к общей памяти, как к своей, с той же скоростью. По сути никаких обращений к ядру ОС или к системным вызовам здесь не нужно. Также как не выполняется ненужное копирование данных.

Недостаток – ядро не синхронизирует доступ процессов к общей памяти. Это – забота программиста. К примеру, нельзя читать данные из совместно используемых сегментов, пока в них осуществляется запись. Равно, как не должны два процесса одновременно писать в один и тот же сегмент. Стандартная стратегия заключается в использовании семафоров, о которых речь пойдет дальше. Однако, в этой части лекционного раздела мы пренебрегаем синхронизацией.

Итак, при совместном использовании области памяти один процесс должен сперва выделить память. Затем остальные процессы, которые хотят получить доступ к ней, должны «подключить» требуемый сегмент памяти. По окончании работы каждый процесс должен «отключить» сегмент. Оставшийся последним процесс должен освободить память.

ОС Linux использует понятие виртуальной памяти, которая процессу выделяется страницами. У каждого процесса есть таблица страниц, в которой устанавливается соответствие между виртуальными и физическими адресами памяти. За процессами закреплены свои адреса, однако разными процессам разрешается ссылаться на одни и те же страницы. Это и есть **разделяемая память**. СЛАЙД 4.

Далее отметим, что при выделении сегмента разделяемой памяти создаются страницы виртуальной памяти. Эта операция должны выполняться только один раз, т.к. остальные процессы будут обращаться к этому же сегменту. Если поступает запрос на создание существующего сегмента, то страницы не создаются, а возвращаются идентификаторы уже выделенных страниц. Для разделения сегмента процесс подключает его, при этом создаются ссылки на его страницы. По окончании работы с сегментом адресные ссылки удаляются. Когда все процессы закончили работу с сегментом, то один и только один из процессов освобождает страницы виртуальной памяти.

Размер разделяемого (совместно используемого) сегмента кратен **размеру страницы** виртуальной памяти. В Linux эта величина равна 4 Кб, но проверить это значение можно вызовом функции *getpagesize()*.

Для использования функций работы с сегментами должны подключаться файлы *sys/shm.h* и *sys/stat.h*.

Выделение сегментов памяти

Операция выделения совместной памяти выполняется функцией *shmget()*. Первый аргумент – целочисленный ключ, которым идентифицируется сегмент. Если несвязанные процессы хотят получить доступ к одному и тому же сегменту, то они указывают одинаковый ключ. Очевидно, что посторонним процессам никто не мешает выбрать тот же ключ сегмента, и это приведет к системному конфликту. Специальным значением ключа *IPC_PRIVATE* можно гарантировать, что создается совершенно новый сегмент.

Второй аргумент – размер сегмента (в байтах), который затем округляется, чтобы быть кратным размеру страницы виртуальной памяти. СЛАЙД 5.

Третий аргумент – набор битовых флагов. Наиболее важные из них (СЛАЙД 6):

- *IPC_CREAT* – указывает на создание нового сегмента, которому присваивается заданный ключ.

- *IPC_EXCL* – всегда используется с *IPC_CREAT* и вынуждает функцию *shmget()* создать новый совместно используемый сегмент памяти либо вернуть идентификатор существующего сегмента, если такой ключ есть в системе. Если такая ситуация возникает, а данный флаг не указан, то возвращается идентификатор существующего сегмента без создания нового.

- **Флаги режима** из 9 флагов, задающих права доступа для владельца, группы и остальных пользователей. Биты выполнения игнорируются. К примеру, *S_IRUSR* и *S_IWUSR* предоставляют владельцу сегмента право чтения-записи, а *S_IROTH* предоставляет право остальным пользователям только читать сегмент.

Так можно создать новый разделяемый сегмент доступный для чтения-записи только его владельцу:

```
int segId = shmget(shmKey, getpagesize(), IPC_CREAT | S_IRUSR | S_IWUSR);
```

Если такой сегмент существует, то проверяются права доступ к нему. Если не существует – он создается и возвращается идентификатор сегмента.

Подключение и отключение сегментов памяти

Для подключения используется функция *shmat()*. Первый аргумент – идентификатор сегмента. Второй – указатель на место в адресном пространстве процесса, где создается привязка на совместно используемую память. Можно задать *NULL* – ОС выберет первый доступный адрес. Третий аргумент – флаги:

- *SHM_RND* – адрес во втором аргументе должен округляться до числа, кратного размеру страницы. Если не указать, то придется самостоятельно решать вопросы выравнивания сегмента по границе страницы.

- *SHM_RDONLY* – сегмент доступен только для чтения.

При успешном завершении функция вернет адрес подключенного сегмента. Если мы создадим дочерний процесс, то он унаследует этот адрес, и сам впоследствии может отключить сегмент.

Для отключения используется *shmdt()*. Ей передается адрес, возвращенный функцией *shmat()*. Если текущий процесс последний, кто ссылается на сегмент, то он удаляется из памяти. Функции *exit()* и *exec()* **отключают** сегменты автоматически. СЛАЙД 7.

Контроль и освобождение разделяемых сегментов

Функция *shmctl()* возвращает информацию о совместно используемом сегменте, а также может модифицировать его. Первый аргумент – идентификатор сегмента.

Для получения информации о сегменте в качестве второго аргумента передается *IPC_STAT*, а третий аргумент – указатель на структуру *shmid_ds*.

Для удаления сегмента в качестве второго аргумента передается *IPC_RMID*, а третий аргумент – *NULL*. Сегмент удаляется, когда последний подключивший его процесс отключает сегмент. СЛАЙД 8.

Каждый разделяемый сегмент должен освобождаться явным образом с помощью данной функции, чтобы не превысить системный лимит на общее количество таких сегментов. Функции *exit()* и *exec()* **не освобождают** сегменты автоматически.

Могут выполняться и другие операции над разделяемыми сегментами. С описанием этих операций можно ознакомиться в справочной системе и/или литературе.

Пример кода, в котором совместно используется память, приведен на СЛАЙДЕ 9.

К сожалению, Linux не гарантирует монопольный доступ к сегменту, даже если он был задан как *IPC_PRIVATE*. Чтобы несколько процессов могли совместно работать с общим сегментом, они должны как-то договориться о выборе одинакового ключа.

10.3 Семафоры

Итак, процессы должны каким-то образом координировать свое выполнение при совместном доступе к памяти. В Linux поддерживается так называемые SystemV-семафоры (далее – семафоры). Они используются, выделяются и освобождаются аналогично совместно используемым областям памяти. На практике обычно используется один семафор, но они, тем не менее, работают группами. Мы опишем в этой части системные вызовы, реализующие двоичный семафор.

Для использования функций работы с семафорами должны подключаться файлы *sys/ipc.h*, *sys/sem.h*, *sys/types.h*.

Выделение и освобождение семафоров

Эти две операции выполняются функциями *semget()* и *semctl()*. Первый аргумент функции *semget()* – ключ-идентификатор группы семафоров. Второй – количество семафоров в группе. Третий – флаги прав доступа. Функция возвращает идентификатор группы. Если задан ключ, принадлежащий уже существующей группе, то будет возвращен ее идентификатор. В этом случае второй аргумент может быть равен 0.

Семафоры остаются «жить» даже по окончании всех процессов, которые их использовали. В связи с этим, чтобы ОС не исчерпала лимит семафоров, последний процесс явно удаляет группу. Для этого вызывается функция *semctl()* с идентификатором группы, числом семафоров в группе, флагом *IPC_RMID* и любым значением (оно будет проигнорировано) типа *union semun* в качестве аргументов. Значение идентификатора пользователя для процесса, вызвавшего функцию, должно совпадать с аналогичным значением процесса, который создал группу семафоров. Удаляемая группа семафоров немедленно освобождается. СЛАЙД 10.

В листинге программы, приведенном на СЛАЙДЕ 11, показано использование функций для работы с двоичным семафором.

Инициализация семафоров

Эта операция отличается от выделения семафора. Чтобы проинициализировать семафор, вызывается функция *semctl()* с нулевым значением второго аргумента и *SETALL* – третьего. Четвертый аргумент должен иметь тип *union semun* , поле *array* которого указывает на массив значений типа *unsigned short int*. Каждое значение инициализирует один семафор из группы.

В листинге программы, приведенном на СЛАЙДЕ 12, показана инициализация двоичного семафора.

Ожидание и установка семафоров

Мы помним, что семафор, по сути, является неотрицательным счетчиком и поддерживает операции установки и ожидания семафора, которые в Linux реализуются системным вызовом *semop()*. Первый аргумент – идентификатор группы, второй аргумент – массив значений *struct sembuf*, задающих выполняемые операции. Третий – размер этого массива.

Поля структуры *sembuf*:

- *sem_num* – номер в группе.

- *sem_op* – число, обозначающее конкретную операцию. Если положительное число – оно добавляется к счетчику семафора. Если отрицательное число – модуль этого числа вычитается из счетчика. Операции, приводящие к отрицательному значению счетчика, блокируются до тех пор, пока значение не станет достаточно большим. Если это число, равное 0, то операция блокируется до тех пор, пока значение счетчика не станет равным нулю.

- *sem_flg* – значение флага. Если он равен *IPC_NOWAIT*, то отключается блокировка

операции. Т.е. если запрашиваемая операция может привести к блокированию, то функция *semop()* завершится с кодом ошибки. Если флаг равен *SEM_UNDO*, то ОС автоматически отменит заданную операцию по завершению процесса.

В листинге программы, приведенном на СЛАЙДЕ 14, показаны операции ожидания и установки двоичного семафора.

Флаг *SEM_UNDO* позволяет решить проблему, которая может возникать после завершения процесса, у которого есть ресурсы, связанные с семафором. Аварийно завершился процесс или нормально, значение счетчика будет автоматически корректироваться, отменяя эффект операции над семафором.

10.4 Отображаемые на память файлы

Благодаря механизму отображаемой памяти процессы получают возможность общаться друг с другом посредством совместно используемого файла. Схематически это можно представить как совместный доступ к именованному сегменту памяти.

При отображении файла в память формируется связь между файлом и памятью процесса. ОС разбивает файл на страничные блоки, копирует их в страницы виртуальной памяти, чтобы они стали доступны в адресном пространстве процесса. Так процесс сможет обращаться к содержимому файла как к обычной памяти. При записи данных в соответствующую область памяти содержимое файла будет меняться. Это ускоряет доступ к файлам.

Функция *mmap()* предназначена для отображения обычного файла в памяти процесса (СЛАЙД 15).

```
#include <sys/mman.h>
void* mmap(void* start // адрес начала отображаемой области
  , size_t length      // длина отображаемой области (в байтах)
  , int prot           // степень защиты отображаемых адресов
  , int flags          // флаги
  , int fd             // дескриптор открытого файла
  , off_t offset       // смещение от начала файла
);
```

Если в первом аргументе задать NULL, то ОС выберет первый доступный адрес. Третий аргумент задает степень защиты диапазона отображаемых адресов. Он может содержать объединение битовых констант *PROT_READ*, *PROT_WRITE* и *PROT_EXEC*, соответствующих разрешению на чтение, запись и выполнение. Дополнительные флаги, задаваемые в четвертом аргументе (СЛАЙД 16):

- *MAP_FIXED*. При наличии этого флага ОС Linux использует значение первого аргумента как точный адрес размещения отображаемого файла. Этот адрес должен соответствовать началу страницы.

- *MAP_PRIVATE*. Изменения, вносимые в отображаемую память, записываются не в присоединенный файл, а в частную копию файла, принадлежащую процессу. Другие процессы не узнают об этих изменениях. Данный режим не совместим с режимом *MAP_SHARED*.

- *MAP_SHARED*. Изменения, вносимые в отображаемую память, немедленно фиксируются в файле, минуя буфер. Этот режим используется при организации взаимодействия процессов.

С помощью шестого аргумента можно перенести в память весь файл или только его часть, корректируя так, как нужно, начальное смещение и длину отображаемой области.

При успешном завершении функция возвращает указатель на начало области памяти. В противном случае возвращается флаг *MAP_FAILED*.

По окончании работы с отображаемым файлом его необходимо освободить с помощью функции *munmap()*.

```
#include <sys/mman.h>
int munmap(void* start // начальный адрес
```

```
  , size_t length      // длина отображаемой области
);
```

ОС при завершении программы автоматически освобождает отображаемые области. СЛАЙД 17.

На СЛАЙДАХ 18 и 19 показаны листинги программ, которые записывают и читают содержимое файлов, отображаемых в память. Программа-писатель записывает случайно сгенерированное число в файл, а программа-читатель получает это число без использование системного вызова *read*, удваивает и записывает результат.

Совместный доступ к файлу

Процессы могут взаимодействовать друг с другом через области отображаемой памяти, связанные с одним и тем же файлом. Если в функции *mmap()* указать флаг *MAP_SHARED*, то все данные, заносимые в отображаемую память, будут немедленно записываться в файл, т.е. становиться видимыми другими процессам. При отсутствии этого флага ОС может осуществлять предварительную буферизацию записываемых данных.

Функция (СЛАЙД 20) переноса содержимого буфера в дисковый файл *msync()*:

```
#include <sys/mman.h>
int msync(mem_addr    //адрес начала отображаемой области
  , mem_length    //длина отображаемой области
  , MS_SYNC | MS_INVALIDATE
);
```

Третий параметр может содержать следующие флаги.

- *MS_ASYNC*. Операция обновления ставится в очередь планировщика и будет выполнена, но не обязательно до того, как функция завершится.
- *MS_SYNC*. Операция обновления выполняется немедленно. До ее завершения функция блокируется. Флаги *MS_ASYNC* и *MS_SYNC* нельзя указывать одновременно.
- *MS_INVALIDATE*. Все остальные отображаемые области помечаются как недействительные и подлежащие обновлению.

Например, обновить файл, область отображения которого начинается с адреса *memAddr* и имеет длину *memLength*, можно так:

```
msync(memAddr, memLength, MS_SYNC | MS_INVALIDATE);
```

При работе с отображаемыми файлами следует придерживаться заданного порядка во избежание конкуренции за ресурс. Эту проблему можно решить, например, созданием семафора, позволяющего только одному процессу обращаться к отображаемой памяти в определенный момент времени.

10.5 Конвейеры

Конвейер – это коммуникационное устройство, допускающее однонаправленное взаимодействие. Данные записываются на «входном» конце конвейера и читаются – на «выходном». Они являются последовательными устройствами: данные всегда читаются в том порядке, в котором они были записаны. Обычно используются как средство связи между потоками одного процесса или между родительским и дочерним процессами.

В командной оболочке *Bash* конвейер создается оператором | (вертикальная черта). Показанная выше команда *$ ls | lpr* заставляет интерпретатор команд запустить два дочерних процесса один для *ls*, другой – для *lpr*.

Так же интерпретатор формирует конвейер, соединяющий стандартный выходной поток процесса *ls* со стандартным входным потоком процесса *lpr*. Таким образом, имена файлов, полученные программой *ls*, посылаются программе печати в том порядке, в котором они отображались бы на терминале.

Очевидно, что информационная емкость конвейера ограничена. Если пишущий процесс помещает данные в конвейер быстрее, чем читающий их извлекает, то буфер конвейера переполняется. Значит, процесс-писатель заблокируется до тех пор, пока буфер не освободится. Обратное тоже верно: если читатель обращается к конвейеру, в который еще не поступили данные, то он будет заблокирован в ожидании данных. Таким образом, конвейер автоматически синхронизирует оба процесса.

Создание конвейеров

Конвейер создается функцией *pipe()*, которой передается массив из двух целых чисел. В нулевом его элементе функция сохраняет дескриптор файла, соответствующего выходному концу конвейера, а в первом элементе – дескриптор файла, соответствующего входному концу.

На СЛАЙДЕ 21 приведена заготовка кода для создания конвейера. После этого данные, записываемые в файл *write_fd*, могут быть прочитаны из файла *read_fd*.

```
int pipe_fds[2];
int read_fd;
int write_fd;

pipe(pipe_fds);
read_fd = pipe_fds[0];
write_fd = pipe_fds[1];
```

Взаимодействие родительского и дочернего процессов

Функция *pipe()* создает два файловых дескриптора, которые действительны только в текущем процессе и его потомках. Их нельзя передавать стороннему процессу. Дочерний процесс получает копии дескрипторов после завершения функции *fork()*.

В коде программы, приведенном на СЛАЙДАХ 22 и 23, родительский процесс записывает в конвейер строку, а потомок читает ее. С помощью функции *fdopen()* файловые дескрипторы приводятся к типу *FILE**. В результате появляется возможность использовать высокоуровневые средства ввода-вывода – функции *printf()* и *fgets()*.

Сначала в этой программе объявляется массив *fds* из двух целых чисел. Функция *pipe()* создает конвейер и помещает в массив дескрипторы обоих концов конвейера. Затем функция *fork()* порождает дочерний процесс. После закрытия выходного конца родитель начинает в конвейер записывать строки. Дочерний процесс начинает выступать в качестве читателя после закрытия входного конца конвейера.

В функции *writer()* родитель принудительно выталкивает буфер конвейера с помощью вызова функции *fflush()*. Без этого строка рискует остаться в буфере и оказаться в конвейере только после завершения родительского процесса.

Возвращаясь к нашему примеру для вывода списка файлов на печать, функция *fork()* вызывается два раза: один – для дочернего процесса *ls*, второй – для дочернего процесса *lpr*. Оба процесса наследуют копии дескрипторов конвейера, поэтому могут взаимодействовать друг с другом. Как организовать соединение двух несвязанных процессов, мы покажем далее в параграфе «FIFO-файлы».

Перенаправление стандартных потоков ввода, вывода и ошибок

Часто требуется создать дочерний процесс и сделать один из концов конвейера стандартным входным или выходным потоком. В этом случае можно воспользоваться функцией *dup2()*, которая делает один файловый дескриптор равным другому. Достаточно легко *STDIN* связывается с определенным файловым дескриптором.

```
dup2(fd, STDIN_FILENO);
```

Константа *STDIN_FILENO* представляет собой дескриптор файла, соответствующего стандартному потоку ввода. Приведенный выше код закрывает входной поток, а затем

открывает его под видом файла *fd*. Дескрипторы 0 и *fd* будут после этого указывать на одну и ту же позицию в файле и иметь одинаковый набор флагов состояния. Иначе говоря, мы получили взаимозаменяемые дескрипторы.

Приведенный на СЛАЙДАХ 24 и 25 код с помощью *dup2()* соединяет выходной конец конвейера со входом команды *sort*.

После создания конвейера функцией *fork()* делится на два процесса. Родитель записывает в конвейер различные строки, а потомок соединяет выходной конец конвейера со своим входным потоком, после чего запускает команду *sort*.

Функции popen() и pclose()

Конвейеры часто используются для передачи данных программе, выполняющейся как подпроцесс, или приема от нее данных. Специально для этого предназначены функции *popen()* и *pclose()*. Они устраняют необходимость в функциях *pipe()*, *dup2()*, *exec()* и *fdopen()*. Чтобы в этом убедиться, достаточно взглянуть на СЛАЙД 26 и сравнить с кодом со СЛАЙДОВ 24 и 25.

Функция *popen()* создает дочерний процесс, в нем начинает выполняться команда *sort*. Этот вызов заменяет функции *pipe()*, *fork()*, *dup2()* и *execlp()*. Второй аргумент указывает на то, что активный процесс хочет осуществлять запись в дочерний процесс. Функция *popen()* возвращает указатель на один из концов конвейера, а второй конец соединяется со стандартным входным потоком дочернего процесса. Функция *pclose()* закрывает входной поток дочернего процесса, дожидается его завершения и возвращает код статуса.

Первый аргумент функции *popen()* является командой *Bash*, выполняемой в контексте процесса */bin/sh*. Он просматривает содержимое переменной окружения *PATH*, чтобы определить, где искать команду. Если второй аргумент равен “r”, то функция возвращает указатель на стандартный выходной поток дочернего процесса, после чего программа сможет читать из него данные. Если второй аргумент “w”, то функция возвращает указатель на стандартный входной поток дочернего процесса, после чего программа сможет писать в него данные. В случае ошибки возвращается *NULL*.

Функция *pclose()* закрывает поток, указатель на который был возвращен функцией *popen()*, и дожидается завершения дочернего процесса.

10.6 FIFO-файлы

Файл FIFO – это конвейер, у которого есть имя в ФС. Любой процесс может открыть и закрыть такой файл. Процессы на противоположных концах конвейера не обязаны быть родственниками. Файлы FIFO еще называют **именованными конвейерами**.

Такой файл создается командой *mkfifo*. Полное имя файла указывается в командной строке.

Файл FIFO можно программно создать с помощью функции *mkfifo()*. Первым аргументом является путь к файлу. Второй аргумент – права доступа к файлу со стороны владельца, его группы и всех остальных. Поскольку у любого конвейера есть две стороны, то права доступа должны учитывать оба случая. Если конвейер не может быть по каким-то причинам создан, функция вернет значение -1. Для работы с функцией нужно подключить файлы <*sys/types*.h> и <*sys/stat.h*>. СЛАЙД 27.

К FIFO-файлу можно обратиться как к обычному файлу. При организации межпроцессного взаимодействия одна программа открывает файл для записи, а другая – для чтения (СЛАЙД 28). Над FIFO-файлом можно выполнять низкоуровневые операции:

```
int fd = open(fifoPath, O_WRONLY);
write(fd, dataBuffer, dataLength);
close(fd);
```

Аналогично, над FIFO-файлом можно выполнять высокоуровневые операции:

```
FILE fifo = fopen(fifoPath, "r");
fscanf(fifo, "%s", dataBuffer);
fclose(fifo);
```

У FIFO-файла одновременно может быть несколько читающих и записывающих процессов. Входные потоки разбиваются на атомарные блоки, размер которых определяется константой *POSIX_PIPE_BUF* (512, 1024, 2048, 4096 кб – типичные значения). Если несколько программ будут параллельно осуществлять запись в файл, их блоки будут чередоваться. То же самое относится и к программам, осуществляющим одновременное чтение данных из файла.

10.7 Сокеты

Сокет – это устройство двунаправленного взаимодействия, которое предназначено для связи с другим процессом, выполняющемся на том же или на другом хосте. Сокеты используются в частности многими Internet-программами.

Например, с помощью программы *telnet* можно получить от Веб-сервера HTML-страницу, поскольку обе программы общаются по сети при помощи сокетов. Чтобы установить соединение с сервером www.sfu-kras.ru можно ввести команду

```
$ telnet www.sfu-kras.ru 80
```

Когда соединение будет установлено, можно ввести команду GET /. В результате через сокет будет послан запрос Веб-серверу, который в ответ вернет содержимое первой HTML-страницы и закроет соединение.

При создании сокета необходимо задать три параметра: тип взаимодействия, пространство имен и протокол.

Тип взаимодействия определяет способ интерпретации передаваемых данных и число подключений (абонентов). Данные, отправляемые через сокет, формируются в блоки, обычно называемые **пакетами**. Тип взаимодействия указывает на то, как обрабатываются пакеты, и как они передаются от отправителя к получателю. Возможны следующие варианты (СЛАЙД 29).

- При взаимодействии **с установлением соединения** гарантируется доставка пакетов именно в том порядке, в каком они были отправлены. Если пакеты теряются или приходят не в исходном порядке из-за сетевых проблем, то принимающая сторона автоматически запрашивает у отправителя повторную отправку данных. Адреса запрашиваются при установке соединения. Сокеты в этом случае называются **потоковыми**.

- При передаче **дейтаграмм** (**без установки соединения**) не гарантируется доставка и правильный порядок пакетов. Пакеты могут потеряться и приходить в произвольном порядке. Сокеты называются **дейтаграммными**.

Пространство имен сокета определяет способ записи адресов. Например, в локальном пространстве имен адреса – это обычные имена файлов. В пространстве Intenet адрес сокета состоит из IP-адреса хоста и номера порт. Благодаря последним можно различать сокеты, созданные на одном хосте.

Протокол определяет способ передачи данных. Основными семействами протоколов являются TCP/IP (ключевые сетевые протоколы, используемые в Intenet), ATM, а также локальный коммуникационный протокол UNIX. Не все комбинации типов взаимодействия, пространств имен и протоколов поддерживаются.

Системные вызовы для работы с сокетами

Сокеты являются более гибкими в управлении механизмами взаимодействия, нежели рассмотренные ранее.

При работе с сокетами используются следующие функции (СЛАЙД 30).
- *socket()* — создает сокет;
- *close()* — уничтожает сокет;
- *connect()* — устанавливает соединение между двумя сокетами;
- *bind()* — назначает серверному сокету адрес;
- *listen()* — переводит сокет в режим приема запросов на подключение;
- *accept()* — принимает запрос на подключение и создает новый сокет, который будет обслуживать данное соединение;
- *write()* — для отправеки данных;
- *read()* — для получения данных.

Сокеты представляются в программе файловыми дескрипторами.

Функции *socket()* и *close()* создают и уничтожают сокет соответственно. В первом случае необходимо задать три параметра: пространство имен, тип взаимодействия и протокол (СЛАЙД 31). Константы, определяющие пространство имен, начинаются с префикса AF. Например, константы *AF_LOCAL* и *AF_UNIX* соответствуют локальному пространству имен, а константа *AF_INET* — пространству имен Internet. Константы, определяющие тип взаимодействия, начинаются с префикса SOCK_. Сокетам, ориентированным на установку соединения, соответствует константа *SOCK_STREAM*, а дейтаграммным сокетам — константа *SOCK_DGRAM*.

Выбор протокола определяется связкой «пространство имен — тип взаимодействия». Поскольку для каждой такой пары, как правило, лучше всего подходит какой-то один протокол, в третьем параметре функции *socket()* обычно задается значение 0 (выбор по умолчанию). В случае успешного завершения функция *socket()* возвращает дескриптор сокета.

Чтение и запись данных через сокеты осуществляется с помощью обычных файловых функций, таких как *read()*, *write()* и т. д. По окончании работы с сокетом его необходимо закрыть с помощью функции *close()*.

Чтобы установить соединение между двумя сокетами, следует на стороне клиента вызвать функцию *connect()*, указав адрес серверного сокета. **Клиент** — это процесс, инициирующий соединение, а **сервер** — это процесс, ожидающий поступления запросов на подключение. В первом параметре функции *connect()* задается дескриптор клиентского сокета, во втором — адрес серверного сокета, в третьем — длина (в байтах) адресной структуры, на которую ссылается второй параметр. Формат адреса будет разным в зависимости от пространства имен.

При работе с сокетами можно применять те же самые функции, что и при работе файлами. Имеются также специальные функции *send()* и *recv()*, являющиеся альтернативой традиционным функции *write()* и *read()*.

Серверы

Жизненный цикл сервера можно представить так (СЛАЙД 32):
1) создание сокета, ориентированного на соединения, с помощью *socket()*;
2) назначение сокету адреса привязки функцией *bind()*;
3) перевод сокета в режим ожидания запросов с помощью *listen()*;
4) прием поступающих запросов на подключение функцией *accept()*;
5) чтение пакетов/дейтаграмм с помощью функций *read()* или *recv()*;
6) закрытие сокета функцией *close()*.

Данные не записываются и не читаются напрямую через серверный сокет. Вместо этого всякий раз, когда сервер принимает запрос на соединение, ОС создает отдельный сокет, используемый для передачи данных через это соединение.

Серверному сокету необходимо с помощью функции *bind()* назначить адрес, чтобы

клиент смог его найти. Первым аргументом функции является дескриптор сокета. Второй аргумент — это указатель на адресную структуру, формат которой будет зависеть от выбранного семейства адресов. Третий аргумент — это длина адресной структуры в байтах. После получения адреса сокет, ориентированный на соединения, должен вызвать функцию *listen()*, тем самым обозначив себя как сервер. Первым аргументом этой функции также является дескриптор сокета. Второй аргумент определяет, сколько запросов может находиться в очереди ожидания. Если очередь заполнена, все последующие запросы отвергаются. Этот аргумент задает не предельное число запросов, которое способен обработать сервер, а максимальное количество клиентов, которые могут находиться в режиме ожидания.

Сервер принимает от клиента запрос на подключение, вызывая функцию *accept()*. Первый ее аргумент — это дескриптор сокета. Второй аргумент указывает на адресную структуру, заполняемую адресом клиентского сокета. Третий аргумент содержит длину (в байтах) адресной структуры. Функция *accept()* создает новый сокет для обслуживания клиентского соединения и возвращает его дескриптор. Исходный серверный сокет продолжает принимать запросы от клиентов. Чтобы прочитать данные из сокета, не удалив их из входящей очереди, можно использовать функцию *recv()*. Она принимает те же аргументы, что и функция *read()*, а также дополнительный аргумент – *flags*. Флаг *MSG_PEEK* задает режим неразрушающего чтения, при котором прочитанные данные остаются в очереди (СЛАЙД 33).

Локальные сокеты

Сокеты, соединяющие процессы в пределах одного компьютера, работают в локальном пространстве имен (*AF_LOCAL* или *AF_UNIX*, это синонимы). Такие сокеты называются **локальными**, или **UNIX-сокетами**. Их адресами являются имена файлов, указываемые только при создании соединения.

Имя сокета задается в структуре типа *sockaddr_un* (из <*sys/un.h*> и <*unistd.h*>). В поле *sun_family* необходимо записать константу *AF_LOCAL*, указывающую на то, что адрес находится в локальном пространстве имен. Поле *sun_path* содержит полное имя файла и не может превышать 108 байтов. Длина структуры *sockaddr_un* вычисляется с помощью макроса *SUN_LEN*. Допускается любое имя файла, но процесс должен иметь право записи в каталог, где находится файл. При подключении к сокету процесс должен иметь право чтения файла. Только процессам, работающим в пределах одного хоста, разрешается взаимодействовать друг с другом посредством локальных сокетов (СЛАЙД 34).

При работе в локальном пространстве допускается только протокол с номером 0.

Локальный сокет является частью ФС, поэтому отображается командой *ls*. Если локальный сокет больше не нужен для работы, то его удаляют функцией *unlink()*.

Работу с локальными сокетами мы проиллюстрируем двумя программами. Первая — это сервер. Он создает локальный сокет и переходит в режим ожидания запросов на подключение. Приняв запрос, сервер читает сообщения из сокета и отображает их на экране, пока соединение не будет закрыто. Если поступает сообщение "*adios amigo*", то сервер удаляет сокет и завершает свою работу. Программа *socket-server* ожидает полное имя сокета в командной строке.

Код серверной части приведен на СЛАЙДАХ 35-36.

Клиентская программа подключается к локальному сокету и посылает сообщение. Имя сокета и текст сообщения задаются в командной строке. Код клиентской части приведен на СЛАЙДЕ 37.

Прежде, чем отправить текст сообщения, клиент записывает в сокет число, определяющее его длину в байтах. На противоположной стороне сервер выясняет для него длину сообщения и выделяет для него буфер соответствующего размера. После этого сервер считывает текст сообщения.

Internet-сокеты

LOCAL-сокеты (они же *UNIX*-сокеты) используются для организации взаимодействия двух процессов на одном хосте. Взаимодействие на разных хостах могут обеспечить **Internet-сокеты**.

Пространству имен Internet соответствует константа *AF_INET*. Internet-сокеты чаще всего используют протоколы стека TCP/IP. В этом стеке протокол IP отвечает за низкоуровневую доставку сообщений, осуществляя при необходимости их разбивку на пакеты и последующую компоновку. Доставка пакетов не гарантируется, поэтому они могут исчезать, приходить в неправильном порядке, дублироваться. Каждый хост в сети имеет свой IP-адрес. Протокол TCP функционирует поверх протокола IP и обеспечивает надежную доставку сообщений, ориентированную на установление соединения.

Адрес Internet-сокета состоит из двух частей: адреса хоста и номера порта. Эта информация хранится в структуре типа *sockaddr_in*. В поле *sin_family* необходимо указать константу *AF_INET*, тогда адрес будет принадлежать пространству имен Internet. В поле *sin_addr* хранится IP-адрес хоста в виде 32-битного целого числа. Благодаря номерам портов можно различать сокеты, создаваемые на одном хосте. В разных системах многобайтные значения могут храниться с разным порядком следования байтов, поэтому с помощью *htons()* необходимо преобразовать номер порта в число с *сетевым порядком следования байтов* (СЛАЙД 38)

Функция *gethostbyname()* преобразует адрес хоста из текстового представления во внутреннее 32-разрядное. Функция возвращает указатель на структуру типа *hostent*. IP-адрес находится в ее поле *h_addr*. Программа, чей код приведен на СЛАЙДАХ 39 и 40, иллюстрирует работу с Internet-сокетами. Она запрашивает начальную страницу у сервера, адрес которого указан в командной строке.

Программа извлекает имя Web-сервера из командной строки. Далее вызывается функция *gethostbyname()*, преобразующая имя в числовое представление. После этого программа подключает TCP-сокет к 80-му порту сервера. Веб-серверы общаются по протоколу HTTP, поэтому программа посылает команду GET, в ответ на которую сервер возвращает текст начальной страницы.

Например, чтобы получить начальную страницу сервера www.sfu-kras.ru, нужно ввести такую команду:

```
$ ./socket-internet www.sfu-kras.ru
```

Пары сокетов

Как указывалось ранее, функция *pipe()* создает два дескриптора для входного и выходного концов конвейера. Возможности конвейеров ограничены, т.к. с файловыми дескрипторами должны работать родственные процессы, а данные передаются только в одном направлении. Функция *socketpair()* создает два дескриптора для двух родственных сокетов, находящихся на одном хосте (СЛАЙД 41). С помощью дескрипторов можно организовать двунаправленное взаимодействие процессов.

Первые три параметра функции *socketpair()* такие же, как и у функции *socket()*, т.е. пространство имен (всегда *AF_LOCAL*), тип взаимодействия и протокол. Четвертый параметр – это массив из двух целых чисел, в которые записываются дескрипторы сокетов, подобно функции *pipe()*.

Литература и дополнительные источники к Теме 10

1. Delve into UNIX process creation - http://www.ibm.com/developerworks/aix/library/au-unixprocess.html
2. LXF83:Unix API - http://wiki.linuxformat.ru/index.php/LXF83:Unix_API
3. Лав, Р. Linux. Системное программирование/ Р.Лав. – СПб.: Питер, 2008. – 416 с.
4. Программирование сокетов в Linux - http://gzip.rsdn.ru/article/unix/sockets.xml

5. Протокол IP - http://www.citforum.ru/internet/tifamily/ipspec.shtml
6. Протокол TCP - http://www.citforum.ru/internet/tifamily/tcpspec.shtml
7. Протокол UDP - http://www.citforum.ru/internet/tifamily/udpspec.shtml
8. Развитие стека TCP/IP: протокол IPv.6 - http://www.citforum.ru/nets/ip/glava_9.shtml
9. Стек протоколов TCP/IP - http://www.agpu.net/fakult/ipimif/fpiit/kafinf/umk/el_lib/calc_system/lab_work_net/kulgin_3.htm
10. Руководство программиста для Linux - http://citforum.ru/operating_systems/linux_pg/lpg_03.shtml
11. byte order - http://encyclopedia2.thefreedictionary.com/byte+order