1 Средства межпроцессного взаимодействия

Поскольку адресные пространства каждого процесса изолированы друг от друга, система должна предоставлять процессам средства взаимодействия.

Простейшее взаимодействие можно организовать, используя файлы в файловой системе. Даже в этом случае необходимо предусмотреть средства взаимной блокировки процессов на случай, когда один или несколько процессов записывают или считывают данные из одного файла.

Предположим, что некоторый процесс должен увеличить целое число, хранящееся в файле, на 1. Простейший фрагмент программы может выглядеть следующим образом (для простоты отсутствует проверка ошибок):

```
fd = open(FILENAME, O_RDWR);
read(fd, &value, sizeof(value));
value++;
lseek(fd, 0L, SEEK_SET);
write(fd, &value, sizeof(value));
close(fd);
```

Однако, такая работа с разделяемым ресурсом (а в данном случае файл является разделяемым ресурсом) содержит серьёзнейший изъян. Предположим, что одновременно два процесса начали модификацию содержимого файла, хранящего число 3. Тогда в зависимости от некоторых случайных (то есть непредсказуемых заранее) факторов значение числа в файле может увеличиться на 1 или на 2.

Такая ошибка работы с разделяемым ресурсом получила в англоязычной литературе название “race condition”. Чтобы избежать её, процесс должен каким-то способом ограничить на время доступ других процессов к разделяемому ресурсу. Операционные системы POSIX предоставляют достаточно богатый выбор средств блокировки: эксклюзивное создание файлов, семафоры, блокировки файлов через `flock`, `lockf` или `fnctl`.

Кроме того предоставляются разнообразные средства межпроцессного обмена, такие как анонимные и именованные каналы, сигналы, разделяемая память и очереди сообщений, файлы, отображаемые в память, сокеты.

Некоторые из этих средств будут рассмотрены в следующих разделах.

1.1 Эксклюзивное создание файлов и файлы-замки

Простейший способ организовать работу с разделяемым ресурсом предполагает использование файлов-замков. Файл-замок — это специальный файл, существование которого в файловой системе означает, что разделяемый ресурс заблокирован.

Работа процесса с разделяемым ресурсом может тогда выглядеть следующим образом: вначале создаётся файл-замок. Если файл не существовал и был успешно создан, процесс может работать с разделяемым файлом, не опасаясь, что другой процесс начнёт в это же время параллельную работу с этим же файлом. После того, как процесс завершит работу с разделяемым ресурсом, файл-замок удаляется. Если же процесс не смог создать файл-замок, он должен приостановить своё выполнение на некоторое небольшое время, после чего снова попытаться создать файл-замок.

Для создания файла-замка в простейшем случае можно использовать флаг `O_EXCL` системного вызова `open`. Этот флаг гарантирует, что если системный вызов `open` завершился успешно, никакой другой процесс не сможет создать файл с использованием флага `O_EXCL` до тех пор, пока файл не будет удалён.

```
while ((fl=open(LOCKFILE,O_CREAT|O_EXCL|O_RDWR,0600))<0) {
  usleep(100000); /* задержка 0.1 секунды */
}
fd = open(FILENAME, O_RDWR);
read(fd, &value, sizeof(value));
value++;
lseek(fd, 0L, SEEK_SET);
write(fd, &value, sizeof(value));
close(fd);
close(fl);
unlink(LOCKFILE);
```

В полной программе ещё необходимо проверять, что эксклюзивное открытие завершилось с ошибкой EEXIST, потому что в противном случае ошибка в системном вызове open произошла не из-за того, что файл-замок уже существует.

К сожалению, эксклюзивное открытие файлов не работает, если файловая система на которой создаётся файл-замок, монтируется с другого компьютера (файловая система NFS). Формально, флаг O_EXCL поддерживается, но система не может гарантировать атомарность операции из-за особенностей протокола NFS.

Чтобы корректно создавать файлы-замки на таких дисках нужно использовать более сложную процедуру, состоящую из следующих шагов: создать в каталоге, в котором будет создан файл-замок, файл с уникальным именем (например, включающим в себя имя компьютера, идентификатор процесса и время создания); создать связь между этим файлом и файлом замка, причём результат работы системного вызова link нужно проигнорировать; с помощью системного вызова stat проверить, что число ссылок на файл увеличилось до 2. Только в этом случае создание файла-замка можно считать успешным. Пример фрагмента, устанавливающего замок, приведён ниже.

```
while (1) {
  if ((fdl = open(UNIQUEFILE, O_CREAT | O_RDWR, 0600)) < 0) {
    /* ошибка, нужно принять какие-то меры... */
  }
  close(fdl);
  link(UNIQUEFILE, LOCKFILE);
  if (stat(UNIQUEFILE, &statbuf) < 0) {
    /* ошибка, нужно принять какие-то меры... */
  }
  if (statbuf.st_nlink == 2) break;
  unlink(UNIQUEFILE);
  /* создание замка неуспешно, нужно подождать */
  usleep(100000);
}
unlink(UNIQUEFILE);
/* создание файла-замка успешно, можем идти дальше */
/* в конце нужно не забыть удалить файл-замок */
```

1.1.1 Достоинства и недостатки

Недостатки. Во-первых, создание файла-замка — операция достаточно сложная и занимающая достаточно много времени, особенно если файл создаётся на сетевом диске.

Во-вторых, отсутствует возможность приостановить выполнение процесса до тех пор, пока файл-замок не будет удалён. Поэтому процесс вынужден периодически пытаться создать файл-замок без гарантии, что в очередной раз операция завершится успешно. Как было сказано выше, если файловая система монтируется с другого компьютера, или даже когда файловая система экспортируется на другие компьютеры, NFS-сервер не может просто так удалять файлы, поскольку он не знает, используется ли этот файл каким-либо клиентом или нет. Поэтому очень частое создание и удаление файлов-замков может приводить к исчерпанию квоты пользователя на дисковое пространство и количество файлов.

Достоинства. Метод файлов-замков не требует, чтобы процессы, работающие с разделяемым ресурсом, находились в родственных отношениях. Процессы могут работать на разных компьютерах (при условии, что файл-замок создаётся корректно, как описано выше). Этот метод — единственный, который всегда работает в такой ситуации.

1.2 Анонимные каналы

Использовать временные файлы для передачи информации между двумя процессами во многих случаях слишком накладно.

Все операционные системы семейства UNIX предоставляют простейшее средство однонаправленной пересылки данных между процессами — анонимные каналы (часто их называют просто каналами — pipe).

Канал создаётся системным вызовом `pipe`.

```
#include <unistd.h>
int pipe(int filedes[2]);
```

При успешном завершении системный вызов возвращает 0, а при ошибке — `-1`, и переменная `errno` устанавливается в код ошибки. Неуспешное завершение `pipe` скорее всего означает, что переполнилась таблица открытых файлов у процесса либо у всей системы.

Системному вызову передаётся массив из двух элементов, в который этот системный вызов записывает номера двух файловых дескрипторов. Два файловых дескриптора связаны друг с другом. Данные записываются в файловый дескриптор `filedes[1]`. Записанные данные можно прочитать из файлового дескриптора `filedes[0]`.

Вообще, `filedes[1]` ссылается на начало канала, `filedes[0]` — на конец канала (если мы предполагаем, что данные «текут» от начала к концу). В результате создания нового процесса с помощью `fork`, либо копирований файлового дескриптора с помощью `dup`, `dup2`, на начало или конец канала может ссылаться несколько файловых дескрипторов у разных процессов. Начало канала (в которую ведётся запись) считается закрытым, когда закрыты все файловые дескрипторы, которые ссылаются на него. Аналогично, конец канала (чтение из канала) считается закрытым, когда закрыты все файловые дескрипторы, которые ссылаются на него. Чтобы установить момент, когда какой-то конец канала закрывается, ядро подсчитывает количество ссылок на канал.

После открытия канала читать и писать в него можно с помощью системных вызовов read и write. Можно связать с файловым дескриптором дескриптор потока `FILE*` с помощью функции `fdopen` и использовать высокоуровневые функции ввода/вывода. В последнем случае высокоуровневую буферизацию лучше всего отключить вызовом `setbuf(f, NULL);`, где `f` — дескриптор потока, связанного с каналом. Файловый дескриптор чтения может использоваться только для чтения из канала, а файловый дескриптор записи — только для записи в канал. Файловые дескрипторы канала не позволяют выполнять операцию `lseek`. Файловые дескрипторы канала закрываются обычным образом с

помощью системного вызова close.

При работе с каналом системные вызовы read и write работают с некоторыми особенностями. Если канал пуст и соответствующий файловый дескриптор не был переведён в неблокирующий режим, системный вызов read приостанавливает выполнение процесса до тех пор, пока в канале не появятся данные, либо начало канала не будет закрыто. Если начало канала закрыто, системный вызов возвращает значение 0. Когда в канале присутствуют данные, системный вызов завершается немедленно и возвращает только те данные, которые присутствуют в канале, то есть количество считанных байт может быть меньше размера буфера, переданного функции read.

Если при выполнении системного вызова write конец канала оказался закрытым, процесс, пытающийся выполнить write получает сигнал SIGPIPE. По умолчанию этот сигнал вызывает печать сообщения Broken pipe и завершение программы. Если сигнал SIGPIPE игнорируется, блокируется или обрабатывается процессом, write возвращает значение -1 с кодом ошибки EPIPE. Каналу в памяти ядра соответствует буфер ограниченного размера. Поэтому когда write пытается записать в канал больше данных, чем в нём остаётся места, процесс будет приостановлен до тех пор, пока в канале не появится достаточно свободного места. В канале не сохраняются границы сообщений, то есть если один процесс сделал write два раза, другой процесс прочитает сразу все данные из канала.

Операции write и read являются для каналов атомарными, если размер данных не превосходит размера внутреннего буфера канала. Этот размер можно узнать из константы PIPE_BUF, определённой в файле <limits.h>. Например, для ядра Linux размер буфера канала равен по умолчанию 4096 байтов. Атомарность понимается в том смысле, что никакой другой процесс в системе не может наблюдать момент, когда операция выполнилась частично. Операция происходит как бы мгновенно. Если объём записываемых данных превышает константу PIPE_BUF, операция write может не быть атомарной.

1.2.1 Использование каналов

Каналы можно использовать для организации конвейерного выполнения команд, когда стандартный вывод одной команды попадает на стандартный ввод другой команды. Например, если мы хотим организовать конвейер

```
ls -l | wc -l
```

то есть передать на вход команде wc -l результат работы команды ls -l, мы можем сделать это следующей программой.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main()
{
  int fd[2];
  int pid1, pid2;

  if (pipe(fd) < 0) { perror("pipe"); exit(1); }
  if ((pid1 = fork()) < 0) { perror("fork"); exit(1); }
  if (!pid1) {
```

```
    /* child 1: "ls -l" */
    dup2(fd[1], 1); close(fd[0]); close(fd[1]);
    execlp("ls", "ls", "-l", NULL);
    perror("execlp"); _exit(1);
  }
  if ((pid2 = fork()) < 0) { perror("fork"); exit(1); }
  if (!pid2) {
    /* child 2: "wc -l" */
    dup2(fd[0], 0); close(fd[0]); close(fd[1]);
    execlp("wc", "wc", "-l", NULL);
    perror("execlp"); _exit(1);
  }
  close(fd[0]); close(fd[1]);
  wait(NULL); wait(NULL);

  return 0;
}
```

1.2.2 Использование каналов для синхронизации

Использование каналов предоставляет процессам возможность синхронизовать свою работу. Процесс-читатель может быть приостановлен до тех пор, пока в канале не появятся данные. Процесс-писатель будет приостановлен, пока в канале не освободится место для данных.

Кроме того, каналы могут использоваться для блокировки доступа к разделяемому ресурсу, например, разделяемому файлу, как было описано в первом разделе. Предположим, что процесс, который желает выполнить операцию, требующую блокировки ресурса, должен получить «жетон». После выполнения критической операции процесс сдаёт жетон. Жетоном будет некоторое (произвольное) число, хранящееся в канале. Получение жетона соответствует операции `read`, возврат жетона — операции `write`.

```
/* получить жетон */
read(fdp[0], &dummy, sizeof(dummy));
/* выполнить операцию */
fd = open(FILENAME, O_RDWR);
read(fd, &value, sizeof(value));
value++;
lseek(fd, 0L, SEEK_SET);
write(fd, &value, sizeof(value));
close(fd);
/* сдать жетон */
write(fdp[1], &dummy, sizeof(dummy));
```

Естественно, в примере, приведённом выше, в канале можно хранить само модифицируемое число, но, если разделяемые данные имеют сложную структуру и требуют произвольного доступа, хранение данных в канале становится слишком сложным.

1.2.3 Достоинства и недостатки

Достоинства. Операции работы с каналами не требуют модификации файловой системы и поэтому выполняются быстрее. Процесс может быть приостановлен (при помощи `read`) до тех пор, пока не появятся данные (ресурс не будет освобождён).

Недостатки. Взаимодействующие процессы должны быть порождены одним процессом, который создаст для них канал. Выполняться они могут только на одном компьютере.

1.3 Именованные каналы

Именованные каналы также называются FIFO (first-in first-out) по дисциплине работы с ними. Именованные каналы отличаются от анонимных каналов тем, что именованные каналы имеют точку привязки в файловой системе — имя. Кроме открытия, работа с именованными каналами не отличается от работы с обычными анонимными каналами.

Перед использованием именованный канал должен быть создан с помощью вызова функции `mkfifo`.

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo ( const char *pathname, mode_t mode );
```

Здесь аргумент `pathname` задаёт имя создаваемого канала, а аргумент `mode` — права доступа на создаваемый именованный канал. Права доступа, как обычно, модифицируются маской прав открытия файла `umask` процесса. Если файл с таким именем существует, функция завершается с ошибкой `EEXIST`.

Как обычно, в случае успеха функция возвращает 0, а при неудачном завершении — `-1`, и в этом случае переменная `errno` будет установлена в код ошибки.

Для открытия именованного канала используется системный вызов `open`. Именованный канал может открываться только на чтение или только на запись, но не на чтение и запись одновременно. Естественно, процесс может открыть два файловых дескриптора: один на чтение, другой на запись. Процесс, открывающий именованный канал на запись, будет заблокирован до тех пор, пока не появится процесс, открывший именованный канал на чтение. То же самое верно и в обратную сторону.

Процесс, который собирается сам и читать из именованного канала, и писать в него, должен открывать канал специальным образом, чтобы избежать блокировки. Сначала процесс должен открыть канал на чтение в неблокирующемся режиме, после этого открыть канал на запись в нормальном режиме, и после сбросить неблокирующий режим на дескрипторе чтения из канала. Фрагмент программы, открывающей именованный канал, приведён ниже (все возможные ошибки игнорируются для краткости записи).

```
long flags;

/* открываем дескриптор чтения */
fdr = open(PIPE, O_RDONLY | O_NONBLOCK);
/* открываем дескриптор записи */
fdw = open(PIPE, O_WRONLY);
/* сбрасываем флаг неблокирующего доступа */
flags = fcntl(fdr, F_GETFL);
fcntl(fdr, F_SETFL & ~O_NONBLOCK)
```

Системные вызовы `read`, `write` и `close` работают точно так же, как и в случае обычных каналов. После того, как именованный канал стал ненужным, его нужно удалить из файловой системы с помощью системного вызова `unlink`.

1.3.1 Достоинства и недостатки

Достоинства. Взаимодействующие процессы не обязаны находиться в родственных отношениях. Хотя создание и открытие именованного канала требует некоторых операций с файловой системой, сам буфер обмена хранится в памяти ядра. Процесс может быть приостановлен до тех пор, пока в канале на появятся данные (или ресурс не будет освобождён).

Недостатки. Некоторые, в особенности устаревшие, системы не поддерживают именованные каналы. Именованные каналы неприменимы, когда процессы работают на разных компьютерах.

2 Функции завершения

Программа может зарегистрировать специальную функцию, которая будет вызвана, когда программа будет завершать своё выполнение по вызову `exit` или после возврата из функции `main`. Для этого используется вызов функции `atexit`.

```
#include <stdlib.h>

int atexit(void (*function)(void));
```

Если было зарегистрировано несколько функций, они будут вызваны в порядке, обратном порядку их регистрации.

Если программа работает с разделяемыми ресурсами (например, устанавливает файл-замок), рекомендуется регистрировать функции-обработчики завершения программы, которые будут освобождать все занятые программой ресурсы, которые не освобождаются системой автоматически (файлы-замки, жетоны, семафоры и пр.). Помимо этого программа ещё должна установить обработчики некоторых сигналов (о сигналах — на следующем занятии).

Если программа завершается системным вызовом `_exit`, обработчики завершения программы не вызываются.

3 Принудительное завершение работы процесса

Чтобы завершить процесс с заданным идентификатором процесса или группу процессов с заданным идентификатором группы процессов, нужно послать процессу или группе сигнал `SIGTERM`, как показано в следующем фрагменте программы:

```
#include <sys/types.h>
#include <signal.h>
kill(pid, SIGTERM);
```

В некоторых случаях, когда, например, сигнал `SIGTERM` игнорируется или обрабатывается неправильно, этот сигнал может не завершить процесс. Тогда процессу нужно послать сигнал `SIGKILL`:

```
kill(pid, SIGKILL);
```

Посылать сразу сигнал SIGKILL ни в коем случае нельзя, так как этот сигнал не даёт возможности прерываемому процессу выполнить завершающие действия, например, освободить ресурс. Между посылкой SIGTERM и SIGKILL должно пройти какое-то время (зависит от ситуации, например 1 секунда), которое отводится процессу на «добровольное» завершение.

После того, как сигнал был послан, необходимо прочитать статус завершения процесса с помощью какой-либо функции семейства wait.