Понятие о потоке ввода-вывода

Как уже упоминалось в лекции 4, среди всех категорий средств коммуникации наиболее употребительными являются каналы связи, обеспечивающие достаточно безопасное и достаточно информативное взаимодействие процессов.

Существует две модели передачи данных по каналам связи – поток ввода-вывода и сообщения. Из них более простой является потоковая модель, в которой операции передачи/приема информации вообще не интересуются содержимым того, что передается или принимается. Вся информация в канале связи рассматривается как непрерывный поток байт, не обладающий никакой внутренней структурой. Изучению механизмов, обеспечивающих потоковую передачу данных в операционной системе UNIX, и будет посвящен этот семинар.

Понятие о работе с файлами через системные вызовы и стандартную библиотеку ввода-вывода для языка С

Потоковая передача информации может осуществляться не только между процессами, но и между процессом и устройством ввода-вывода, например между процессом и диском, на котором данные представляются в виде файла. Поскольку понятие файла должно быть знакомо изучающим этот курс, а системные вызовы, использующиеся для потоковой работы с файлом, во многом соответствуют системным вызовам, применяемым для потокового общения процессов, мы начнём наше рассмотрение именно с механизма потокового обмена между процессом и файлом.

Из курса программирования на языке С вам известны функции работы с файлами из стандартной библиотеки ввода-вывода, такие как `fopen()`, `fread()`, `fwrite()`, `fprintf()`, `fscanf()`, `fgets()` и т. д. Эти функции входят как неотъемлемая часть в стандарт ANSI на язык С и позволяют программисту получать информацию из файла или записывать ее в файл при условии, что программист обладает определенными знаниями о содержимом передаваемых данных. Так, например, функция `fgets()` используется для вывода из файла последовательности символов, заканчивающейся символом '\n' – перевод каретки. Функция `fscanf()` производит вывод информации, соответствующей заданному формату, и т.д. С точки зрения потоковой модели операции, определяемые функциями стандартной библиотеки ввода-вывода, не являются потоковыми операциями, так как каждая из них требует наличия некоторой структуры передаваемых данных.

Для языка С++ все обстоит аналогично, только там вы используете объекты классов `ofstream` и `ifstream`, их методы и операции «>>» и «<<».

В операционной системе UNIX эти функции представляют собой надстройку – сервисный интерфейс – над системными вызовами, осуществляющими прямые потоковые операции обмена информацией между процессом и файлом и не

требующими никаких знаний о том, что она содержит. Чуть позже вы кратко познакомитесь с системными вызовами `open()`, `read()`, `write()` и `close()`, которые применяются для такого обмена, но сначала нам нужно ввести ещё одно понятие – понятие *файлового дескриптора* .

Файловый дескриптор

В лекции 3 говорилось, что информация о файлах, используемых процессом, входит в состав его системного контекста и хранится в его блоке управления – PCB. В операционной системе UNIX можно упрощённо полагать, что информация о файлах, с которыми процесс осуществляет операции потокового обмена, наряду с информацией о потоковых линиях связи, соединяющих процесс с другими процессами и устройствами ввода-вывода, хранится в некотором массиве, получившем название таблицы открытых файлов или таблицы файловых дескрипторов. Индекс элемента этого массива, соответствующий определённому потоку ввода-вывода, получил название файлового дескриптора для этого потока. Таким образом, файловый дескриптор представляет собой небольшое целое неотрицательное число, которое для текущего процесса в данный момент времени однозначно определяет некоторый действующий канал ввода-вывода. Некоторые файловые дескрипторы на этапе старта любой программы ассоциируются со стандартными потоками ввода-вывода. Так, например, файловый дескриптор 0 соответствует стандартному потоку ввода, файловый дескриптор 1 – стандартному потоку вывода, файловый дескриптор 2 – стандартному потоку для вывода ошибок. В нормальном интерактивном режиме работы стандартный поток ввода связывает процесс с клавиатурой, а стандартные потоки вывода и вывода ошибок – с текущим терминалом.

Более детально строение структур данных, содержащих информацию о потоках ввода-вывода, ассоциированных с процессом, мы будем рассматривать позже, при изучении организации файловых систем в UNIX.

Открытие файла. Системный вызов open()

Файловый дескриптор используется в качестве параметра, описывающего поток ввода-вывода, для системных вызовов, выполняющих операции над этим потоком. Поэтому прежде чем совершать операции чтения данных из файла и записи их в файл, мы должны поместить информацию о файле в таблицу открытых файлов и определить соответствующий файловый дескриптор. Для этого применяется процедура открытия файла, осуществляемая системным вызовом `open()`. С её описанием можно ознакомиться при помощи команды `man open`.

Системный вызов `open()` применяется для открытия как уже существующих файлов в файловой системе, так и для создания новых файлов с последующим их открытием. Поэтому он существует в двух формах.

Если вы собираетесь работать с уже существующим файлом, не предполагая, что файл нужно будет создавать, то прототип системного вызова `open()` выглядит следующим образом:

```
int open(char *filename, int flags);
```

Здесь первый параметр – это указатель на символьное имя файла (неважно полное или относительное), а второй параметр – это флаги для того, чтобы специфицировать операции, которые предполагается применять к файлу в дальнейшем. Из всего набора флагов для такой формы системного вызова `open()` нас, на текущем уровне знаний, будут интересовать только флаги `O_RDONLY`, `O_WRONLY`, `O_RDWR`. Эти три флага являются взаимоисключающими: хотя бы один из них должен быть применен и наличие одного из них не допускает наличия двух других. Они описывают набор операций, которые, при успешном открытии файла, будут разрешены над файлом в дальнейшем: только чтение, только запись, чтение и запись. Как вам известно из материалов семинаров 1–2, у каждого файла существуют атрибуты прав доступа для различных категорий пользователей. Если файл с заданным именем существует на диске, и права доступа к нему для пользователя, от имени которого работает текущий процесс, не противоречат запрошенному набору операций, то операционная система сканирует таблицу открытых файлов от ее начала к концу в поисках первого свободного элемента, заполняет его и возвращает индекс этого элемента в качестве файлового дескриптора открытого файла. Если файла на диске нет, не хватает прав или отсутствует свободное место в таблице открытых файлов, то констатируется возникновение ошибки.

Если же вы допускаете, что файл может отсутствовать и его нужно будет создавать, или требуете, чтобы файл изначально отсутствовал и был создан новый, то используется другая форма системного вызова с прототипом

```
int open(char *filename, int flags, int mode);
```

В случае, когда мы **допускаем**, что файл на диске может отсутствовать, и хотим, чтобы он был создан, один из трех перечисленных выше флагов должен использоваться в комбинации с флагом `O_CREAT`. Если файл существует, то все происходит по рассмотренному выше сценарию. Если файла нет, сначала выполняется создание файла с набором прав, указанным в параметре mode системного вызова. Проверка соответствия набора операций объявленным правам доступа может и не производиться (как, например, в ряде версий Linux).

В случае, когда мы **требуем**, чтобы файл на диске отсутствовал и был создан в момент открытия, флаг для набора операций должен использоваться в комбинации с флагами `O_CREAT` и `O_EXCL`.

Подробнее об операции открытия файла и ее месте среди набора всех файловых операций будет рассказываться в лекционной теме 11.

Системные вызовы read(), write(), close()

Для совершения потоковых операций чтения информации из файла и её записи в файл применяются системные вызовы `read()` и `write()`. С их описаниями можно ознакомиться при помощи команды `man read` и `man write` соответственно.

При работе с файлами информация записывается в файл или читается из файла, начиная с места, определяемого указателем текущей позиции в файле. Изначально, при использовании указанных выше флагов, указатель текущей позиции выставляется на начало файла. Значение указателя текущей позиции увеличивается на количество реально прочитанных или записанных байт после операций чтения или записи. При чтении информации из файла она не пропадает из него. Если системный вызов `read()` возвращает значение 0, то это означает, что файл прочитан до конца.

Мы сейчас не акцентируем внимание на понятии указателя текущей позиции в файле и взаимном влиянии значения этого указателя и поведения системных вызовов. Этот вопрос будет обсуждаться в дальнейшем на семинарах, посвященных файловой системе.

После завершения потоковых операций процесс должен выполнить операцию закрытия потока ввода-вывода, во время которой произойдет окончательный сброс буферов на линии связи, освободятся выделенные ресурсы операционной системы, и элемент таблицы открытых файлов, соответствующий файловому дескриптору, будет отмечен как свободный. За эти действия отвечает системный вызов `close()`. Надо отметить, что при завершении работы процесса (см. семинар 3–4) с помощью явного или неявного вызова функции `exit()` происходит автоматическое закрытие всех открытых потоков ввода-вывода.

Зачем проверять результат close()? Именно на этом этапе при записи в файл может произойти ошибка, связанная с нехваткой пространства на диске.

***Пример**: 05-0.c*

Маска создания файлов текущего процесса. Команда umask и системный вызов umask()

Посмотрите на права доступа различных категорий пользователей к появившемуся в результате прогона примера файлу. В программе в системном вызове `open()` мы требовали права доступа `0666`, то есть `read` и `write` для всех категорий пользователей, а в реальности наблюдаем права `0644`, то есть `rw-r--r--`. Как же так, дорогая редакция!? Дело в том, что на реальные права доступа различных категорий пользователей во время работы системных вызовов при создании объектов оказывает влияние так называемая маска создания файлов текущего процесса.

Маска создания файлов текущего процесса запрещает открытие некоторых прав доступа к создаваемым объектам в процессе независимо от того, что вы указали в

соответствующих параметрах системных вызовов. Изначально это маска выставляется при вашем входе в систему в командном интерпретаторе и передается по наследству всем порождаемым процессам. Выглядит она похоже на права доступа в восьмеричном виде, но проставленные в значение 1 биты запрещают установление этого права доступа.

Узнать маску создания файлов, установленную в командном интерпретаторе можно с помощью команды `umask`. С помощью этой же команды можно установить новое значение маски создания файлов. Для изменения маски создания файлов в текущем процессе и в дальнейшем в его детях нужно использовать системный вызов `umask()`.

***Пример**: 05-1.c*

Понятие о pipe. Системный вызов pipe()

Наиболее простым способом для передачи информации с помощью потоковой модели между различными процессами или даже внутри одного процесса в операционной системе UNIX является pipe (канал, труба, конвейер).

Важное отличие pip'а от файла заключается в том, что прочитанная информация немедленно удаляется из него и не может быть прочитана повторно.

Pipe можно представить себе в виде трубы ограниченной ёмкости, расположенной внутри адресного пространства операционной системы, доступ к входному и выходному отверстию которой осуществляется с помощью системных вызовов. В действительности pipe представляет собой область памяти, недоступную пользовательским процессам напрямую, зачастую организованную в виде кольцевого буфера (хотя существуют и другие виды организации). По буферу при операциях чтения и записи перемещаются два указателя, соответствующие входному и выходному потокам. При этом выходной указатель никогда не может перегнать входной и наоборот. Для создания нового экземпляра такого кольцевого буфера внутри операционной системы используется системный вызов `pipe()`.

В процессе работы системный вызов организует выделение области памяти под буфер и указатели и заносит информацию, соответствующую входному и выходному потокам данных, в два элемента таблицы открытых файлов, связывая тем самым с каждым pip'ом два файловых дескриптора. Для одного из них разрешена только операция чтения из pip'а, а для другого – только операция записи в pipe. Для выполнения этих операций мы можем использовать те же самые системные вызовы `read()` и `write()`, что и при работе с файлами. Естественно, по окончании использования входного или/и выходного потока данных, нужно закрыть соответствующий поток с помощью системного вызова `close()` для освобождения системных ресурсов. Необходимо отметить, что, когда все процессы, использующие pipe, закрывают все ассоциированные с ним файловые дескрипторы, операционная

система ликвидирует pipe. Таким образом, время существования pip’а в системе не может превышать время жизни процессов, работающих с ним.

***Пример**: 05-2.c*

Организация связи через pipe между процессом родителем и процессом-потомком. Наследование файловых дескрипторов при вызовах fork() и exec()

Понятно, что если бы всё достоинство pip'ов сводилось к замене функции копирования из памяти в память внутри одного процесса на пересылку информации через операционную систему, то овчинка не стоила бы выделки. Однако таблица открытых файлов наследуется процессом ребёнком при порождении нового процесса системным вызовом `fork()` и входит в состав неизменяемой части системного контекста процесса при системном вызове `exec()` (за исключением тех потоков данных, для файловых дескрипторов которых был специальными средствами выставлен признак, побуждающий операционную систему закрыть их при выполнении `exec()`, однако их рассмотрение выходит за рамки нашего курса). Это обстоятельство позволяет организовать передачу информации через pipe между родственными процессами, имеющими общего прародителя, создавшего pipe.

***Пример 3**: 05-3.c*

Pipe служит для организации однонаправленной или симплексной связи. Если бы в предыдущем примере мы попытались организовать через pipe двустороннюю связь, когда процесс-родитель пишет информацию в pipe, предполагая, что её получит процесс-ребёнок, а затем читает информацию из pipe, предполагая, что её записал порожденный процесс, то могла бы возникнуть ситуация, в которой процесс-родитель прочитал бы собственную информацию, а процесс-ребёнок не получил бы ничего. Для использования одного pipe в двух направлениях необходимы специальные средства синхронизации процессов, о которых речь идет в лекциях «Алгоритмы синхронизации» (лекция 5) и «Механизмы синхронизации» (лекция 6). Более простой способ организации двунаправленной связи между родственными процессами заключается в использовании двух pipe.

Необходимо отметить, что в некоторых UNIX-подобных системах (например, в Solaris 2) реализованы полностью дуплексные pip’ы. В таких системах для обоих файловых дескрипторов, ассоциированных с pipe, разрешены и операция чтения, и операция записи. Однако такое поведение не характерно для pip’ов и не является переносимым.

Задачи на семинар

***Задача 1 (5 баллов)**:*

*Измените программу **05-1.c** так, чтобы она читала содержимое файла, созданного исходной программой. Все лишние операторы, конструкции и флаги необходимо удалить.*

Данная задача не принимается, пока в ней присутствует что-то лишнее или отсутствует что-то необходимое. Сам входной файл в репозиторий класть не нужно, только исходный код.

***Задача 2 (5 баллов)**:*

*Модифицируйте программу **05-3.c** для организации двусторонней связи при помощи двух pipe. Каждый из процессов должен отправить и получить хотя бы по одному сообщению. Сообщения от разных процессов должны быть отличимы друг от друга.*

Использование библиотечных функций запрещено, можно использовать только системные вызовы, которые рассматривались на семинаре.

Домашнее задание

Дедлайн для вашей группы прописывается в таблице с результатами. Сдать задание нужно до наступления указанного дня, сдавать в указанную дату уже поздно. Проверка производится один раз, штрафы за недочеты выставляются сразу.

***Задача 1 (10 баллов)**:*

*Модифицируйте пример **05-3.c** для связи между собой двух родственных процессов, выполняющих две разные программы. В основной программе создается pipe, далее следует вызов fork() и в обоих процессах с помощью одной из функций семейства exec() запускается новая программа. Для того, чтобы процесс родитель мог общаться с процессом ребенком (ребенок пишет – родитель читает) после вызова exec(), необходимо передать в запускаемые программы номера соответствующих дескрипторов. Способ передачи - абсолютно любой на ваше усмотрение.*

При работе с потоками данных использование библиотечных функций запрещено, можно использовать только системные вызовы, которые рассматривались на семинаре.

Не забывайте обработку результатов системных вызовов и в частности, возвращения из exec в ту же программу.