Семинары 5-6. Организация взаимодействия процессов через pipe и FIFO в UNIX

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.

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 2 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 (семинары 13 и 14–15).

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() использует набор флагов для того, чтобы специфицировать операции, которые предполагается применять к файлу в дальнейшем или которые должны быть выполнены непосредственно в момент открытия файла. Из всего возможного набора флагов на текущем уровне знаний вас будут интересовать только флаги O_RDONLY, O_WRONLY, O_RDWR, O_CREAT и O_EXCL. Первые три флага являются взаимоисключающими: хотя бы один из них должен быть применён и наличие одного из них не допускает наличия двух других. Эти флаги 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.

В случае, когда мы **допускаем**, что файл на диске может отсутствовать, и хотим, чтобы он был создан, флаг для набора операций 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 параметрах системного вызова. Проверка соответствия набора операций объявленным правам доступа может и не производиться (как, например, в 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 её месте среди набора всех файловых операций будет рассказываться на лекции 7 «Файловая система с точки зрения пользователя». Работу системного вызова open() с флагами O_APPEND и O_TRUNC мы разберем на семинаре 13, посвящённом организации файловых систем в UNIX.

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.

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 семинаре 13.

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() (С его описанием можно ознакомиться при помощи команды man 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.

***Пример 1**: ex1.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(). Более подробное описание можно получить, выполнив команду man pipe.

***Пример 2**: ex2.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**: ex3.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.

Особенности поведения вызовов read() и write() для pip'а

Системные вызовы read() и write() имеют определённые особенности поведения при работе с pipe, связанные с его ограниченным размером, задержками в передаче данных и возможностью блокирования обменивающихся информацией процессов. Организация запрета блокирования этих вызовов для pipe выходит за рамки нашего курса.

Будьте внимательны при написании программ, обменивающихся большими объемами информации через pipe. Помните, что за один раз из pipe может прочитаться меньше информации, чем вы запрашивали, и за один раз в pipe может записаться меньше информации, чем вам хотелось бы. Проверяйте значения, возвращаемые вызовами!

Одна из особенностей поведения блокирующегося системного вызова read() связана с попыткой чтения из пустого pipe. Если есть процессы, у которых этот pipe открыт для записи, то системный вызов блокируется и ждёт появления информации. Если таких процессов нет, он вернёт значение 0 без блокировки процесса. Эта особенность приводит к **необходимости закрытия файлового дескриптора, ассоциированного с входным концом pipe, в процессе, который будет использовать pipe для чтения** (close(fd[1]) в процессе-ребёнке в программе **05-3.c**. Аналогичной особенностью поведения при отсутствии процессов, у которых pipe открыт для чтения, обладает и системный вызов write(), с чем связана **необходимость закрытия файлового дескриптора, ассоциированного с выходным концом pipe, в процессе, который будет использовать pipe для записи** и (close(fd[0]) в процессе-родителе в той же программе).

Понятие FIFO. Использование системного вызова mknod() для создания FIFO. Функция mkfifo()

Как вы выяснили, доступ к информации о расположении pip'а в операционной системе и его состоянии может быть осуществлён только через таблицу открытых файлов процесса, создавшего pipe, и через унаследованные от него таблицы открытых файлов процессов-потомков. Поэтому изложенный выше механизм обмена информацией через pipe справедлив лишь для родственных процессов, имеющих общего прародителя, инициировавшего системный вызов pipe(), или для таких процессов и самого прародителя и не может использоваться для потокового общения с другими процессами. В операционной системе UNIX существует возможность использования pip'а для взаимодействия других процессов, но её реализация достаточно сложна и лежит далеко за пределами наших занятий.

Для организации потокового взаимодействия любых процессов в операционной системе UNIX применяется средство связи, получившее название FIFO (от First Input First Output) или именованный pipe. FIFO во всём подобен pip'у, за одним исключением: данные о расположении FIFO в адресном пространстве ядра и его состоянии процессы могут получать не через родственные связи, а через файловую систему. Для этого при создании именованного pip'а на диске заводится файл специального типа, обращаясь к которому процессы могут получить интересующую их информацию. Для создания FIFO используется системный вызов mknod() или существующая в некоторых версиях UNIX функция mkfifo().

Следует отметить, что при их работе не происходит действительного выделения области адресного пространства операционной системы под именованный pipe, а только заводится файл-метка, существование которой позволяет осуществить реальную организацию FIFO в памяти при его открытии с помощью уже известного вам системного вызова open().

После открытия именованный pipe ведёт себя точно так же, как и неименованный. Для дальнейшей работы с ним применяются системные вызовы read(), write() и close(). Время существования FIFO в адресном пространстве ядра операционной системы, как и в случае с pip'ом, не может превышать время жизни последнего из использовавших его процессов. Когда все процессы, работающие с FIFO, закрывают все файловые дескрипторы, ассоциированные с ним, система освобождает ресурсы, выделенные под FIFO. Вся непрочитанная информация теряется. В то же время файл-метка остается на диске и может использоваться для новой реальной организации FIFO в дальнейшем.

Важно понимать, что файл типа FIFO не служит для размещения на диске информации, которая записывается в именованный pipe. Эта информация располагается внутри адресного пространства операционной системы, а файл является только меткой, создающей предпосылки для её размещения.

Не пытайтесь просмотреть содержимое этого файла с помощью Midnight Commander (mc)!!! Это приведет к его глубокому зависанию!

Особенности поведения вызова open() при открытии FIFO

Системные вызовы read() и write() при работе с FIFO имеют те же особенности поведения, что и при работе с pip'ом. Системный вызов open() при открытии FIFO также ведёт себя несколько иначе, чем при открытии других типов файлов, что связано с возможностью блокирования выполняющих его процессов. Если FIFO открывается только для чтения, и флаг O_NDELAY не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на запись. Если флаг O_NDELAY задан, то возвращается значение файлового дескриптора, ассоциированного с FIFO. Если FIFO открывается только для записи, и флаг O_NDELAY не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на чтение. Если флаг O_NDELAY задан, то констатируется возникновение ошибки и возвращается значение -1. Задание флага O_NDELAY в параметрах системного вызова open() приводит и к тому, что процессу, открывшему FIFO, запрещается блокировка при выполнении последующих операций чтения из этого потока данных и записи в него.

***Пример 4**: ex4.c*

В этой программе информацией между собой обмениваются процесс-родитель и процесс-ребёнок. Обратите внимание, что повторный запуск этой программы приведёт к ошибке при попытке создания FIFO, так как файл с заданным именем уже существует. Здесь нужно либо удалять его перед каждым прогоном программы с диска вручную, либо после первого запуска модифицировать исходный текст, исключив из него всё, связанное с системным вызовом mknod(). С системным вызовом, предназначенным для удаления файла при работе процесса, вы познакомитесь позже (на семинаре 13) при изучении файловых систем.