Семинар 4. Организация взаимодействия процессов в UNIX.

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 начнем наше рассмотрение именно с механизма потокового обмена между процессом и файлом.

Из курса программирования на языке С вам должны быть 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 ввода из файла последовательности символов, заканчивающейся символом '\n' - перевод каретки; функция fscanf() производит ввод 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 требуюшими никаких априорных знаний о том, что она содержит. Чуть позже мы кратко познакомимся с системными вызовами open(), read(), write() и close(), которые применяются для такого обмена, но сначала нам нужно ввести еще одно понятие - понятие файлового дескриптора.

Файловый дескриптор. Информация о файлах, используемых процессом, входит в состав его системного контекста и хранится в его блоке управления - PCB. В операционной системе UNIX можно упрощенно 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 определенному 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 текущий 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 соответсвует 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.

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():

```
#include <fcntl.h>
int open(char *path, int flags);
int open(char *path, int flags, int mode).
```

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 нас 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 употреблен 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 последующем: только чтение, только запись, чтение и запись. Как вам известно из материалов первого семинара, 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 тогда 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 - то сначала выполняется создание файла с набором прав, указанном 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.

Системные вызовы read(), write(), close(). Для совершения потоковых операций чтения информации из файла и ее записи в файл применяются системные вызовы read() и write(). 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.

После завершения потоковых операций, процесс должен выполнить операцию закрытия потока ввода-вывода, во время которой произойдет окончательный досброс 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 с помощью явного или неявного вызова функции exit() происходит автоматическое закрытие всех открытых потоков ввода-вывода.

```
#include <sys/types.h>
#include <unistd.h>
size_t read(int fd, void *addr, size_t nbytes);
size_t write(int fd, void *addr, size_t nbytes);
int close(int fd);
```

Задания:

1) Написать программу, которая создает файл и пишет в него “Hello, World!”
2) Написать программы, которая читает эту фразу из файла и выводит ее на экран.

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). Важным отличием pip'a от файла является то, что прочитанная информация немедленно удаляется из него и не может быть прочитана повторно. О pip'е можно думать, как о трубе ограниченной емкости, 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 многочисленные 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()*. При своей работе он 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 pipe'ом два файловых дескриптора. Для одного из них разрешена только операция чтения из pip'a, 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, что по окончании необходимости использования входного или/и выходного потока данных, нам необходимо закрыть соответствующий поток с помощью системного вызова *close()* для освобождения системных ресурсов. Необходимо отметить, что, когда все процессы, использующие pip'е, закрывают все ассоциированные с ним файловые дескрипторы, операционная система ликвидирует pipe. Таким образом, время существования pip'a в системе не может превышать время жизни процессов, работающих с ним.

Задания:

1) Написать программу для пересылки сообщения через pipe в рамках одного процесса.
2) Написать программу для пересылки сообщения через pipe между двумя родственными процессами.
3) Организовать двухстороннюю связь между двумя родственными процессами.

Понятие о FIFO. Использование системного вызова mknod для создания FIFO. Функция mkfifo. Как мы выяснили, доступ к информации о расположении pip'a в операционной системе и его состоянии может быть осуществлен 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. Откровенно говоря, в операционной системе UNIX существует возможность использования pip'а для взаимодействия других процессов, но ее 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 всем 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 узнать интересующую их информацию. Для создания FIFO применяется 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 которого позволяет осуществить реальную организацию FIFO в памяти при его открытии с помощью уже известного нам ситемного вызова *open()*. После открытия именованный pipe ведет 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 pip'а, не может превышать времени жизни последнего из использующих 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 жить на диске, и он может быть использован 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 ее размещения.

Особенности поведения вызова *open()* при открытии FIFO. Системные вызовы *read()* и *write()* при работе с FIFO имеют такие же особенности поведения, как и при работе с pip'ом. Системный вызов *open()* при открытии FIFO также ведет себя несколько

иначе, чем при открытии других типов файлов, что связано с возможностью блокирования выполняющих его процессов. Если FIFO открывается только для чтения, и не задан флаг O_NDELAY, 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 не задан флаг O_NDELAY, 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.

Задания:

1) Организовать передачу сообщения между двумя родственными процессами.
2) Организовать передачу сообщения между двумя неродственными процессами.