Особенности поведения вызовов read() и write() для pip’a

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.

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).

Дополнительное важное замечание: во многих операционных системах a la UNIX, в частности в куче версий Linux, существует известный bug при работе системных вызовов `read()` и `write()`, который пока разработчики Linux не собираются устранять. Дело в том, что эти системный вызовы работают не атомарно, а стало быть их работа в режиме разделения времени может быть прервана, и управление может получить другой процесс. Это может приводить к противоречиям со стандартом. По стандарту системный вызов `read()` не может прочитать информацию из pipe, большую чем размер буфера pip’a. Однако в реальности это не так.

Рассмотрим следующую модельную ситуацию. Пусть у нас есть pipe с буфером 32 Kb и два близкородственных процесса желающих поработать с этим pip’ом. Один из них собирается записать в pipe 256 Kb, а второй — прочитать 256 Kb. Пусть управление первым получил записывающий процесс. Он положит в pipe 32 Kb и заблокируется. Управление получит читающий процесс. Допустим после чтения 16 Kb у него закончится отведенное для работы время и управление отдадут записывающему процессу. Тот пропишет 16 Kb в свободное место буфера и снова заблокируется. Управление передаем читающему. Он читает очередные 16

Kb и у него забирают управление. И так далее. В результате читающий процесс может прочитать все запрошенные 256 Kb данных, поскольку в буфере до возврата из системного вызова `read()` все время появляется новая информация.

Системные вызовы read и write

Особенности поведения при работе с pipe, FIFO и socket с блокировкой

Системный вызов read

Ситуация	Поведение
Попытка прочитать меньше байт, чем есть в наличии в канале связи.	Читает требуемое количество байт и возвращает значение, соответствующее прочитанному количеству. Прочитанная информация удаляется из канала связи.
В канале связи находится меньше байт, чем затребовано, но не нулевое количество.	Читает все, что есть в канале связи, и возвращает значение, соответствующее прочитанному количеству. Прочитанная информация удаляется из канала связи.
Попытка читать из канала связи, в котором нет информации. Блокировка вызова разрешена.	Вызов блокируется до тех пор, пока не появится информация в канале связи и пока существует процесс, который может передать в него информацию. Если информация появилась, то процесс разблокируется, и поведение вызова определяется двумя предыдущими строками таблицы. Если в канал некому передать данные (нет ни одного процесса, у которого этот канал связи открыт для записи), то вызов возвращает значение `0`. Если канал связи полностью закрывается для записи во время блокировки читающего процесса, то процесс разблокируется, и системный вызов возвращает значение `0`.
Системный вызов write	
Ситуация	**Поведение**
Попытка записать в канал связи меньше байт, чем осталось до его заполнения.	Требуемое количество байт помещается в канал связи, возвращается записанное количество байт.
Попытка записать в канал связи больше байт, чем осталось до его заполнения. Блокировка вызова разрешена.	Вызов блокируется до тех пор, пока все данные не будут помещены в канал связи. Если размер буфера канала связи меньше, чем передаваемое количество информации, то вызов будет ждать, пока часть информации не будет считана из канала связи. Возвращается полностью записанное количество байт.
Попытка записи в канал связи, из которого некому больше читать, или полное закрытие канала на чтение во время блокировки системного вызова.	Если вызов был заблокирован, то он разблокируется. Процесс получает сигнал `SIGPIPE`. Если этот сигнал обрабатывается пользователем, то системный вызов вернет значение `-1` и установит переменную `errno` в значение `EPIPE`.

Необходимо отметить дополнительную особенность системного вызова `write` при работе с pip'ами и FIFO. Запись информации, размер которой не превышает размер буфера, должна осуществляться атомарно – одним подряд лежащим куском. Этим объясняется ряд блокировок и ошибок в предыдущем перечне.

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.

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**: 05-4.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()`.

Глобальная переменная errno

При возникновении ошибок системные вызовы, как правило, возвращают значение -1. Но при этом возникает потребность отличить одну ошибку от другой. Для этого используется целочисленная глобальная переменная `errno`, объявленная в заголовочном файле errno.h.

В описании системных вызовах описывается набор константных значений, которые присваиваются в переменную в случае возникновения конкретной ошибки. В случае, когда ошибки не происходит, значение, лежащее в errno никак не относится к предыдущему вызову.

Значения задаются макросом. Избранные примеры возможных значений для библиотечной функции `mkfifo()`:

EACCES - один из каталогов в пути не разрешает доступ на поиск или выполнение

EEXIST - файл с таким путем уже существует

ENAMETOOLONG - либо длина пути больше, чем PATH_MAX, либо имя файла имеет большую по сравнению с NAME_MAX длину. В системе GNU не существует предела общей длины имени файла, но некоторые файловые системы могут установить пределы длины данных компонентов.

ENOSPC - в каталоге или файловой системе недостаточно места для нового файла

ENOTDIR - компонент, указанный как каталог в пути, фактически не является каталогом

Задачи на семинар

***Задача 1 (5 баллов)**:*

Напишите на базе примера 05-4.c две программы, одна из которых пишет информацию в FIFO, а вторая – читает из него и выводит на экран, так чтобы между ними не было ярко выраженных родственных связей (т. е. чтобы ни одна из них не была потомком другой).

Обе программы должны уметь создавать fifo в случае его отсутствия и отличать сценарий, когда файл уже существует и можно продолжать корректное выполнение программы от сценария, когда произошла иная ошибка.

Примечание: для выполнения задач необходимо использовать системные вызовы, которые рассматривались на семинаре. Решения с другими библиотечными функциями не принимаются. Не забывайте проверять возвращаемые значения системных вызовов на предмет ошибок.

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)**:*

Самостоятельно изучите вопросы запрета блокирования системных вызовов read и write для pipe. При помощи этих вызовов определите точный размер pipe для вашей операционной системы и выведите его на экран. Решения, выполняющие эту задачу другим способом, приниматься не будут.

Программы, которые не завершают свою работу, не принимаются. Программа не должна печатать промежуточных ответов, только финальный. В случае вывода неверного ответа из оценки вычитается разница вашего ответа с правильным, 1 байт = 1 балл.

***Задача 1* (1 бонусный балл)**:*

Рубрика “проверь себя”: определите размер pipe для вашей операционной системы любым другим программным способом и выведите его все в той же программе вторым ответом.