Уточнение знаний об описании потоков, связывающих процесс с файлами

В материалах семинаров 5-6, посвященных потокам ввода-вывода, рассказывалось о хранении информации о файлах внутри адресного пространства процесса с помощью таблицы открытых файлов, о понятии файлового дескриптора, о необходимости введения для простоты работы операций открытия и закрытия файлов (системные вызовы `open()` и `close()`) и об операциях чтения и записи (системные вызовы `read()` и `write()`). Мы обещали вернуться к более подробному рассмотрению затронутых вопросов в текущих семинарах. Пора выполнять обещанное. Далее в этом разделе, если не будет оговорено особо, под словом «файл» будет подразумеваться регулярный файл.

Вся информация об атрибутах файла и его расположении на физическом носителе содержится в соответствующем файлу индексном узле и, возможно, в нескольких связанных с ним логических блоках. Для того чтобы при каждой операции над файлом не считывать эту информацию с физического носителя заново, представляется логичным, считав информацию один раз при первом обращении к файлу (системный вызов `open()`), хранить ее в адресном пространстве процесса или в части адресного пространства ядра, характеризующей данный процесс. Именно поэтому в лекции по теме 3 данные о файлах, используемых процессом, были отнесены к составу системного контекста процесса, содержащегося в его PCB.

С точки зрения пользовательского процесса каждый файл представляет собой линейный набор байт, снабженный *указателем текущей позиции* процесса в этом наборе. Все операции чтения из файла и записи в файл производятся в этом наборе с того места, на которое показывает указатель текущей позиции. По завершении операции чтения или записи указатель текущей позиции перемещается в конец прочитанного или записанного участка файла. Значение этого указателя является динамической характеристикой файла для использующего его процесса и также должно храниться в PCB.

На самом деле организация информации, описывающей открытые файлы в адресном пространстве ядра операционной системы UNIX, является более сложной.

Некоторые файлы могут использоваться одновременно несколькими процессами независимо друг от друга или совместно. Для того чтобы не хранить дубликаты информации об атрибутах таких файлов и их расположении на внешнем носителе для каждого процесса отдельно, такие данные обычно размещаются в адресном пространстве ядра операционной системы в единственном экземпляре, а доступ к ним процессы получают только при выполнении соответствующих системных вызовов для операций над файлами.

Независимое использование одного и того же файла несколькими процессами в операционной системе UNIX предполагает возможность для каждого процесса совершать операции чтения и записи в файл по своему усмотрению. При этом для корректной работы с информацией необходимо организовывать взаимоисключения для операций ввода-вывода. Совместное использование одного и того же файла в операционной системе UNIX возможно для близко родственных процессов, т. е. процессов, один из которых является потомком другого или которые имеют общего родителя. При совместном использовании файла процессы разделяют некоторые данные, необходимые для работы с файлом, в частности, указатель текущей позиции. Операции чтения или записи, выполненные в одном процессе, изменяют значение указателя текущей позиции во всех близко родственных процессах, одновременно использующих этот файл.

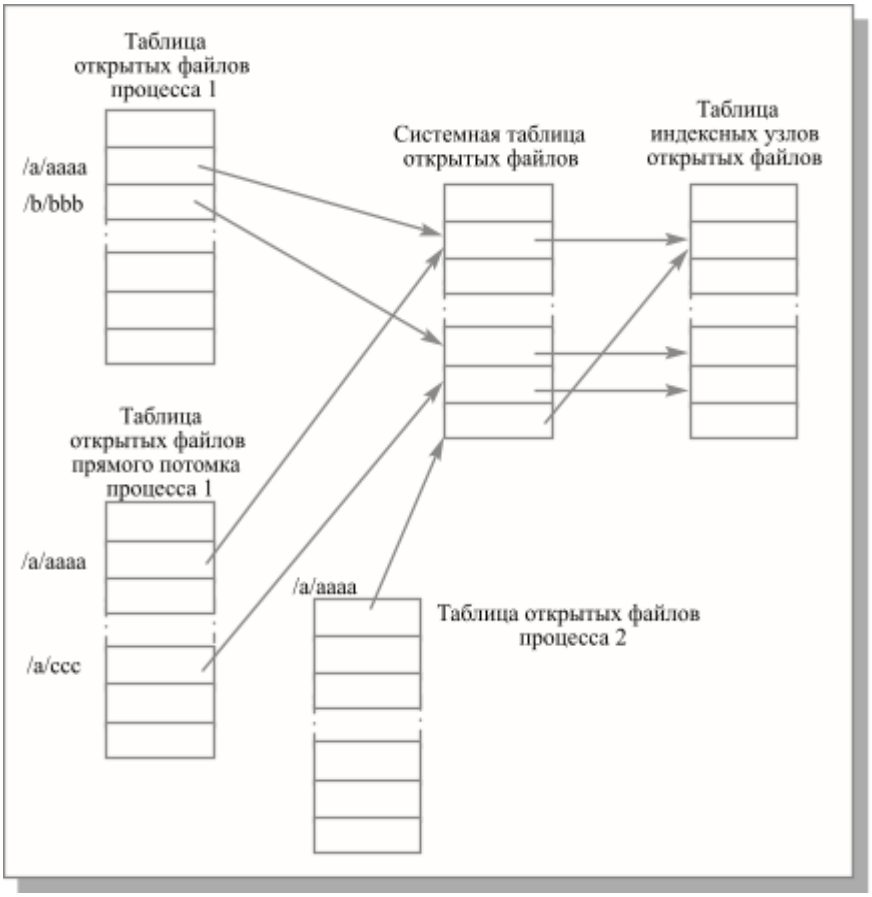
Это означает, в частности, что если из файла совместного использования что-то прочитает родитель, а потом ребенок, то ребенок будет читать информацию с того места, на котором остановился родитель.

Как мы видим, вся информация о файле, необходимая процессу для работы с ним, может быть разбита на три части:

- данные, специфичные для этого процесса;
- данные, общие для близко родственных процессов, совместно использующих файл, например, указатель текущей позиции;
- данные, являющиеся общими для всех процессов, использующих файл, – атрибуты и расположение файла.

Естественно, что для хранения этой информации применяются три различные связанные структуры данных, лежащие, как правило, в адресном пространстве ядра операционной системы, – *таблица открытых файлов процесса*, *системная таблица открытых файлов* и *таблица индексных узлов открытых файлов*. Для доступа к этой информации в управляющем блоке процесса заводится таблица открытых файлов, каждый непустой элемент которой содержит ссылку на соответствующий элемент системной таблицы открытых файлов, содержащей данные, необходимые для совместного использования файла близко родственными процессами. Из системной таблицы открытых файлов мы, в свою очередь, можем по ссылке добраться до общих данных о файле, содержащихся в таблице индексных узлов открытых файлов (см. рис. 3). Только таблица открытых файлов процесса входит в состав его PCB и, соответственно, наследуется при рождении нового процесса. Индекс элемента в этой таблице (небольшое целое неотрицательное число) или файловый дескриптор является той величиной, характеризующей файл, которой может оперировать процесс при работе на уровне пользователя. В эту же таблицу открытых файлов помещаются и ссылки на данные, описывающие другие потоки ввода-вывода, такие как pipe и FIFO (об этом уже упоминалось в семинарах 5-6). Как мы увидим позже (в материалах семинаров 21–22, посвященных сетевому программированию), эта же таблица будет

использоваться и для размещения ссылок на структуры данных, необходимых для передачи информации от процесса к процессу по сети.



Стандартные операции над логическими файлами в UNIX

Единственными логическими файлами в UNIX, не связанными с организацией коллекции файлов (логической файловой системой), являются регулярные файлы. Все они, с точки зрения операционной системы, являются файлами прямого доступа. Мы знаем из лекций по теме 11, что базовыми операциями над файлами прямого доступа являются операции read, write, truncate и seek. Операциям read и write соответствуют системные вызовы `read()` и `write()`, которые мы рассмотрели раньше на семинарах 5-6. Операции truncate соответствует системный вызов `ftruncate()`, описание которого вы можете найти в UNIX Manual, с помощью команды man. В отличие от базовой операции системный вызов `ftruncate()` позволяет не только уменьшить размер файла, но и увеличить его размер. При этом, если размер файла мы уменьшаем, то вся информация в конце файла, не влезающая в новый размер, будет потеряна. Если же размер файла мы увеличиваем, то это будет выглядеть так, как будто мы дополнили его до недостающего размера нулевыми байтами. Операции seek соответствует системный вызов `lseek()`.

Основными операциями над логическими файлами, не связанными с организацией коллекции файлов (логической файловой системой), являются операции создания файла, удаления файла, чтения атрибутов файла и записи атрибутов файла.

Для создания файла, как мы знаем, можно использовать системный вызов `open()` с опцией `O_CREAT`, либо системный вызов `create()` — смотрите UNIX Manual. Для удаления файла предназначен системный вызов `unlink()` — опять же смотрите UNIX Manual.

Возможность записи атрибутов файла для рядовых пользователей весьма ограничена. Пожалуй, единственное, что может сделать хозяин файла — это сменить хозяина, группу хозяев или права доступа к файлу для разных категорий пользователей с помощью команд командного интерпретатора `chown`, `chgrp`, `chmod` — см. семинары 1-2. А вот узнать атрибуты файла (и не только регулярного, а и, забегая вперед, директорий, файлов типа «связь» и других типов файлов, несвязанных с организацией логической файловой системы) можно с помощью семейства системных вызовов `stat()`, `fstat()`, `lstat()`.

Системные вызовы `stat()`, `fstat()` и `lstat()` служат для получения информации об атрибутах файла.

Системный вызов `stat()` читает информацию об атрибутах файла, на имя которого указывает параметр `filename`, и заполняет ими структуру, расположенную по адресу `buf`. Заметим, что имя файла должно быть полным, либо должно строиться относительно той директории, которая является текущей для процесса, совершившего вызов. Если имя файла относится к файлу типа «связь», то читается информация (рекурсивно!) об атрибутах файла, на который указывает символическая связь.

Системный вызов `lstat()` идентичен системному вызову `stat()` за одним исключением: если имя файла относится к файлу типа «связь», то читается информация о самом файле типа «связь».

Системный вызов `fstat()` идентичен системному вызову stat, только файл задается не именем, а своим файловым дескриптором (естественно, файл к этому моменту должен быть открыт).

Для системных вызовов `stat()` и `lstat()` процессу не нужны никакие права доступа к указанному файлу, но могут понадобиться права для поиска во всех директориях, входящих в специфицированное имя файла.

Структура `stat` в различных версиях UNIX может быть описана по-разному. В Linux она содержит:

- устройство, на котором расположен файл
- номер индексного узла для файла
- тип файла и права доступа к нему
- счетчик числа жестких связей
- идентификатор пользователя владельца
- идентификатор группы владельца
- тип устройства для специальных файлов устройств
- размер файла в байтах (если определен для данного типа файлов)
- размер блока для файловой системы
- число выделенных блоков
- время последнего доступа к файлу
- время последней модификации файла
- время создания файла

Тип файла можно определить соответствующим макросом.

Права находятся в младших 9 битах соответствующего поля структуры, аналогично тому, как они задавались в маске файлов процесса.

Специальные функции для работы с содержимым директорий

Стандартные системные вызовы `open()`, `read()` и `close()` не могут помочь программисту изучить содержимое файла типа «директория». Директорию можно открыть и получить файловый дескриптор, но прочитать ее не получится. Для анализа содержимого директорий используется набор функций из стандартной библиотеки языка C.

С точки зрения программиста в этом интерфейсе директория представляется как файл последовательного доступа, над которым можно совершать операции чтения очередной записи и позиционирования на начале файла. Перед выполнением этих операций директорию необходимо открыть, а после окончания – закрыть. Для открытия директории используется функция `opendir()`, которая подготавливает почву для совершения операций и позиционирует нас на начале файла. Чтение очередной записи из директории осуществляет функция `readdir()`, одновременно позиционируя нас на начале следующей записи (если она, конечно, существует). Для операции нового позиционирования на начале директории (если вдруг понадобится) применяется функция `rewinddir()`. После окончания работы с директорией ее необходимо закрыть с помощью функции `closedir()`.

Эти функции работают с типом данных `struct dirent`, описывающей одну запись в директории. Поля этой записи сильно варьируются от одной файловой системы к другой, но одно из полей, которое и будет нас интересовать, всегда

присутствует в ней. Это поле `char d_name[]` неопределенной длины, не превышающей значения `NAME_MAX+1`, которое содержит символьное имя файла, завершающееся символом конца строки.

Внимание! В структуре `struct dirent` есть поле `d_type` для типа файлов. Его нельзя использовать для определения типа файла в UNIX, так как такой код непереносим! Функция `readdir()` — это стандартная функция языка С, не связанная с конкретной ОС. Поэтому если вы напишете рабочую программу для ОС Linux, а затем запустите ее:

- **на другой системе UNIX,**
- **или на той же ОС, но на другой файловой системе,**
- **или все там же, но используете при сборке другую стандартную библиотеку языка С;**

вы можете внезапно обнаружить, что `readdir()` стала различать только два типа файлов: регулярный файл и директория.

Специальные функции для работы с содержимым директорий

Как мы могли убедиться на прошлом семинаре, стандартные системные вызовы `open()`, `read()` и `close()` не могут помочь изучить и содержимое файла типа «связь». Для получения содержимого файлов используется функция `readlink()`. Так как файл содержит всего одну запись, являющуюся строкой, то работа с `readlink()` скорее напоминает работу с системным вызовом `read()`, чем с `readdir()`.

Системный вызов `readlink()` служит для чтения информации, находящейся непосредственно в самом файле типа «связь» с полным или относительным именем pathname. Содержимое файла заносится в буфер `buf`, имеющий размер `bufsize`. При этом признак конца строки (нулевой байт) в конец содержимого не добавляется. Если содержимое файла типа «связь» по размеру превышает значение, указанное в параметре `bufsize`, при помещении содержимого в буфер `buf` происходит его «молчаливое» усечение до заданного значения.

Задачи на семинар

Подзадачу можно дописывать в течение недели без штрафа, коэффициенты за основную задачу повышены: 1, 1 и 0.7 вместо 1, 0.7 и 0.5 соответственно.

***Задача 1 (15 баллов)**:*

Напишите программу - аналог команды ls. Имя директории передается в программу, как аргумент командной строки, если аргумент отсутствует, выбирается текущая директория. Для всех файлов, входящих в данную директорию, напечатайте имена и их типы. Для файлов, которые являются ссылкой дополнительно напечатайте, на какой файл они ссылаются.

Обязательно проверьте работу программы на директории /dev и сравните результаты ее работы с результатом работы команды «ls –al /dev».

***Подзадача 1 (до 5 бонусных баллов)**:*

Снабдите вашу программу текстовым файлом README и максимально приблизьте выдачу вашей программы к выдаче ls -al. За каждую дополнительную колонку: права доступа для некоторой категории пользователей, размер файла в байтах и так далее, начисляется дополнительные 0.5 баллов, при условии, что смысл колонки отражен в README.