Лабораторная работа IV

Работа с файлово-каталожной системой в ОС Linux

Методические материалы к заданию

Подробное описание возможностей и синтаксиса команд можно ознакомиться в документации, доступной по команде `man <команда>`.

Основные команды для работы с файлами и каталогами

- **`cd`** – смена каталога;
- **`cp`** – копирование файлов;
- **`ls`** – выводит список файлов (в том числе каталогов) в указанном каталоге;
- **`file`** – выводит тип указанного файла;
- **`find`** – поиск файлов;
- **`ln`** – создание ссылок;
- **`mkdir`** – создание каталога;
- **`mv`** – перемещение файла или каталога;
- **`pwd`** – вывод имени текущего каталога;

- **rm** – удаление файла;
- **rmdir** – удаление каталога;
- **cat** – слияние и вывод файлов;

Ссылки на файлы

В Linux существует два вида ссылок, обычно называемых жесткие ссылки и символьные, или “мягкие” ссылки. Жесткая ссылка является собственно символьным именем какого-либо файла – записью в соответствующем каталоге со ссылкой на индексный дескриптор этого файла. Таким образом, файл может иметь одновременно несколько символьных имен, в том числе в различных каталогах. Файл будет удален с диска только тогда, когда будет удалено последнее из его символьных имен. Нет такого понятия, как “настоящее” имя: все символьные имена одного файла имеют одинаковый статус. Мягкая ссылка (или символьная ссылка, или `symlink`) принципиально отличается от жесткой ссылки: она является специальным файлом (с отдельным индексным дескриптором), который содержит полный путь к другому файлу. Таким образом, мягкая ссылка может указывать на файлы, которые находятся на других файловых системах, и не нуждается в наличии того файла, на который она указывает. Когда происходит попытка доступа к файлу, ядро операционной системы заменяет ссылку на тот путь, который она содержит. Однако команда `rm` удаляет саму ссылку, а не файл, на который она указывает. Для чтения состояния символьной ссылки, а также имени файла, на который она указывает, используется команда `readlink`. Полное имя файла может задаваться как с использованием абсолютного пути, например, `/home/user/file`, так и с помощью относительного пути – пути, заданного относительно текущего каталога. Это особенно часто применяется в скриптах. Для этого в каждом каталоге есть два служебных каталога: `..` – указывает на родительский каталог `.` – указывает на текущий каталог Например, команда `cd` .. позволит перейти на уровень выше, а команда `cd` . ничего не изменит. Другой пример: команда `./script.bash` запускает скрипт именно из текущего каталога. Наконец, если мы находимся в домашнем каталоге пользователя `user`, то путь к файлу `./../../home/user/file` будет соответствовать пути к файлу в домашнем каталоге, как и описанный выше пример абсолютного пути. Для того, чтобы перейти к корню файловой системы можно использовать команду `cd /` Для обозначения домашнего каталога активного пользователя можно использовать символ ~. Тогда запись `cd ~` будет эквивалентна записи `cd $HOME`.

Условия заданий

В рамках данной лабораторной работы можно предоставить реализацию всех скриптов как на Bash, так и на языке Си. *При выборе второго варианта вы можете претендовать на 150% от максимальной стоимости задания.*

Все скрипты должны обрабатывать любые сценарии, соответствующие заданию, в том числе некорректный ввод параметров пользователем, использование имен файлов, содержащих необычные, но не запрещенные для использования в именах файлов символы, различные последовательности запусков разработанных скриптов и других действий пользователя в файловой системе. Все возникающие ошибки при выполнении скриптов, в том числе возникающие при выполнении отдельных утилит операционной системы, должны обрабатываться, и содержательные сообщения о них должны выводиться пользователю командами разрабатываемого скрипта.

1. Скрипт **`rmtrash`**
   a) Скрипту передается один параметр – имя файла в текущем каталоге вызова скрипта.
   b) Скрипт проверяет, создан ли скрытый каталог `trash` в домашнем каталоге пользователя. Если он не создан – создает его.
   c) После этого скрипт создает в этом каталоге жесткую ссылку на переданный файл с уникальным именем (например, присваивает каждой новой ссылке имя, соответствующее следующему натуральному числу) и удаляет файл в текущем каталоге.
   d) Затем в скрытый файл `trash.log` в домашнем каталоге пользователя помещается запись, содержащая полный исходный путь к удаленному файлу и имя созданной жесткой ссылки.

2. Скрипт **`untrash`**
   a) Скрипту передается один параметр – имя файла, который нужно восстановить (без полного пути – только имя).
   b) Скрипт по файлу `trash.log` должен найти все записи, содержащие в качестве имени файла переданный параметр, и выводить по одному на экран полные имена таких файлов с запросом подтверждения.
   c) Если пользователь отвечает на подтверждение положительно, то предпринимается попытка восстановить файл по указанному полному пути (создать в соответствующем каталоге жесткую ссылку на файл из trash и удалить соответствующий файл из trash). Если каталога, указанного в полном пути к файлу, уже не существует, то файл восстанавливается в домашний каталог пользователя с выводом соответствующего сообщения.
   d) В качестве опции передается флаг политики восстановления файла в случае конфликта имен:
      - `-i` [`--ignore`] – скрипт ничего не делает, а лишь выводит сообщение о невозможности восстановить файл. Это опция выбирается по умолчанию.
      - `-u` [`--unique`] – скрипт приписывает к востанавливаемому файлу актуальный номер версии (`file.txt` -> `file(1).txt`). Расширение файла должно сохраниться.
      - `-o` [`--overwrite`] – скрипт восстанавливает файл поверх существующего.

3. Скрипт **`backup`**
   a) Скрипт создаст в `/home/user/` каталог с именем `Backup-YYYY-MM-DD`, где `YYYY-MM-DD` – дата запуска скрипта, если в `/home/user/` нет каталога с именем, соответствующим дате, отстоящей от текущей менее чем на 7 дней. Если в `/home/user/` уже есть «действующий» каталог резервного копирования (созданный не ранее 7 дней от даты запуска скрипта), то новый каталог не создается. Для определения текущей даты можно воспользоваться командой `date`.
   b) Если новый каталог был создан, то скрипт скопирует в этот каталог все файлы из каталога `/home/user/source/` (для тестирования скрипта создайте такую директорию и набор файлов в ней). После этого скрипт выведет в режиме дополнения в файл `/home/user/backup-report` следующую информацию: строка со сведениями о создании нового каталога с резервными копиями с указанием его имени и даты создания; список файлов из `/home/user/source/`, которые были скопированы в этот каталог.
   c) Если каталог не был создан (есть «действующий» каталог резервного копирования), то скрипт должен скопировать в него все файлы из `/home/user/source/` по следующим правилам: если файла с таким именем в каталоге резервного копирования нет, то он копируется из `/home/user/source`. Если файл с таким именем есть, то его размер сравнивается с размером одноименного файла в действующем каталоге резервного

копирования. Если размеры совпадают, файл не копируется. Если размеры отличаются, то файл копируется с автоматическим созданием версионной копии, таким образом, в действующем каталоге резервного копирования появляются обе версии файла (уже имеющийся файл переименовывается путем добавления дополнительного расширения «`.YYYY-MM-DD`» (дата запуска скрипта), а скопированный сохраняет имя). После окончания копирования в файл `/home/user/backup-report` выводится строка о внесении изменений в действующий каталог резервного копирования с указанием его имени и даты внесения изменений, затем строки, содержащие имена добавленных файлов с новыми именами, а затем строки с именами добавленных файлов с существовавшими в действующем каталоге резервного копирования именами с указанием через пробел нового имени, присвоенного предыдущей версии этого файла.

4. Скрипт **`upback`**

   Скрипт должен скопировать в каталог `/home/user/restore/` все файлы из актуального на данный момент каталога резервного копирования (имеющего в имени наиболее свежую дату), за исключением файлов с предыдущими версиями.

Правила оформления и написания программ на C

Программы должны удовлетворять следующим требованиям:

- В случае, если это требуется по ТЗ, корректно обрабатывать ошибки при взаимодействии с внешним миром: ошибки ввода-вывода, некорректный пользовательский ввод и прочее. Если это произошло – необходимо вывести сообщение об ошибке (на английском языке) и завершить исполнение с ненулевым кодом возврата.
- Программа никогда не должна падать. Падение – признак ошибок в реализации.
- Программа не должна содержать лишних сущностей: закомментированных больших участков кода, неиспользуемых переменных и функций и тому подобное. Это засоряет код и увеличивает время проверки.

При успешном выполнении программа возвращает код `0`. Если же что-то пошло не так, то она сообщает о проблеме через ненулевой код возврата и сообщение об ошибке в поток вывода ошибок `stderr`.

За что можно потерять баллы к критерии качества кода

Кроме правильности результата будет учитываться скорость работы программы. То есть, если проверяющий не дождался за разумное время завершения работы программы, то тест будет считаться не пройденным.

В программе можно использовать стандартные библиотеки и заголовочные файлы, а также те, которые относятся к POSIX/GNU C библиотеке. Например, `<bits/stdc++.h>` таковым не является и его использование влечёт за собой потерю баллов.

Если программа использует функции, которые явно не объявлены в файле с исходным кодом (например, тип `size_t` без подключения `<stdlib.h>` и пр.), то за это также будут снижаться баллы (даже если у вас всё работает).

Обратите внимание: обёртка Bash на Си — это не самурайский путь, вам необходимо использовать GNU C библиотеку при выполнении лабораторной работы. Все сдачи начиная с этого момента на Си с обёрткой Bash *не будут засчитываться* никаким образом.