Лабораторная 1. "Редактор *vi*/*vim*. Простейшие скрипты *shell*"

Теоретическая часть

Практическая часть

Общая постановка задачи: С помощью *vi*/*vim* создать скрипт для *shell*, обеспечивающий получение заданным образом организованной выходной информации. Результаты выполнения записываются в файл (наиболее универсальный способ их сохранения). Скрипт сделать исполняемым, выполнить, проанализировать результат. Используются перенаправление ввода вывода, внешние улилиты и фильтры, при необходимости -- переменные *shell*.

1. Список процессов: имя пользователя, одна или более строк списка его процессов, общее число процессов.

2. Количество файлов в заданном каталоге: имя каталога, число файлов, дата.

3. Список подкаталогов: имя каталога, одна или несколько строк списка подкаталогов (если найдутся).

(2., 3.) Имя анализируемого каталога может быть задано аргументом командной строки.

4. Шестнадцатиричный (или восьмеричный) дамп заданного файла: имя файла, строки дампа, имя пользователя, дата (в качестве подписи).

5. "Самодокументирующийся скрипт": форматирует собственный текст, снабжает заголовком и подписью по аналогии с 4.

6. Статистика текущего сеанса: имя пользователя, текущее время, дата, текущий каталог, число процессов в системе.

7. Замер времени выполнения команды: имя команды/процесса, время выполнения. Команда или имя внешней программы задаются в командной строке скрипта. Сам скрипт такой программой быть не может (что сильно упрощает реализацию!)

Лабораторная 2. "Более сложные скрипты *shell*"

Теоретическая часть

Практическая часть.

1. Получение скриптом времени выполнения заданной команды, в т.ч. самого себя (т.е. *shell*-а, исполняющего данный скрипт).

2. Генератор случайного числа $x = k*x \bmod m$ (вместо мультипликативного алгоритма можно выбрать другой, желательно тоже целочисленный). Инициализация от текущего времени.

Обновление генератора через фиксированные промежутки времени (напр., 10 сек.). Сгенерированное число записывается в файл. Учесть возможность одновременной работы нескольких таких генераторов.

3. Аналогичный генератор случайных чисел, но сгенерированные значения записываются в переменную. Учесть доступность переменной после запуска скрипта.

4. Решение квадратного уравнения. Коэффициенты задаются в командной строке, при отсутствии считываются с *stdin*.

5. Расширение команды *kill*: аргумент -- имя процесса.

6. Расширение команды *kill*: аргумент -- имя пользователя-владельца.

7. Расширение команды *kill*: аргумент -- имя терминала, где выполняется процесс.

(5.-7. Упрощенный вариант -- только для одного процесса, найденного первым. Более сложный -- для всех удовлетворяющих критерию.)

8. Интерактивная "диалоговая" игра.

Лабораторная 3 "Изучение потокового редактора *sed* и регулярных выражений"

Теоретическая часть

Практическая часть

Общая постановка задачи: создать скрипт *sed* (в отдельном файле, передаваемом *sed*) или скрипт *shell*, вызывающий *sed* с необходимыми параметрами. Используются регулярные выражения. Команд для *sed* может быть несколько. Допускается привлекать простейшие внешние фильтры, например, *wc*.

1. Аналог утилиты (фильтра) *head*, но игнорирует пустые строки (не пропускает их и не подсчитывает как обработанные). Параметры командной строки учитываются, хотя бы число строк.

2. Аналогично, но расширяется функциональность утилиты (фильтра) *tail*.

3. "Расширение" артиклей в тексте комментариями вида: `a` --> `Article 'a'`, для `an` и `the` -- аналогично. Учесть регистр первой буквы артикля.

(Вместо артиклей можно брать любые другие характерные слова текста.)

4. Сокращения в тексте: замены `you` --> `u`, `for` --> `4`, `to` --> `2` и т.п. Регистр букв учитывается, когда это возможно.

5. Замена числительных: `one` --> `1`, `two` --> `2` и т.д. Учесть порядковые числительные: `first` --> `1st` (или `1-st`, если так больше нравится), `second` --> `2nd`, `third` --> `3rd`, `fourth` --> `4th` и т.д. Нечувствительность к регистру букв во входном тексте.

6. Обратное преобразование (расшифровка) числительных.

(С целью упрощения можно ограничиваться однословными числительными, т.е. не превосходящими 12)

7. Замена строчных букв на заглавные в начале предложений, т.е. после точки, не находящейся внутри, например, числа.

8. Подсчет числа предложений в тексте. *(Для упрощения можно считать, что больше 3 предложений в строке не бывает.)*

9. Найти и вывести слова, содержащие все 5 гласных (`a`, `e`, `i`, `o`, `u`) в любом порядке.

(Аналогичное задание формулируется для других заданных букв или цифр. Общее число контролируемых символов заранее определено, оно не менее 4, но не более 10)

10. "Фильтр базара" -- замена фиксированным маркером слов, объявленных "запрещенными" (словарь "запрещенных" слов невелик).

Лабораторная 4. "Совместное использование средств командной строки Unix"

Теоретическая часть

Практическая часть

Общая постановка задачи: Реализовать поиск файлов с обходом дерева каталогов. Над найденными файлами выполняются заданные действия. (*Похоже на утилиту find, но способ поиска и действия фиксированные.*) Задача формулируется как составная: "общая" и "специальная" части.

Общая часть задания (поиск):

1. Поиск файла по имени, образец поиска может быть регулярным выражением (имена проверяются на соответствие регулярному выражению). Образец передается как аргумент командной строки. Поиск начинается от текущего директория.

2. Поиск файла по имени, образец поиска задан списком (имена проверяются на совпадение с любым из этого списка). Способ передачи списка -- ряд аргументов в командной строке. Поиск начинается от текущего директория.

Специальная часть задания (действия над найденными файлами):

1. Подсчет суммарного размера файлов, их количества, среднего размера (итогового).

Размеры в байтах, если большая -- в килобайтах (дробная).

2. Подсчет общего и среднего количества строк в файлах. Вывод больших чисел форматированный для удобства чтения (например, по группировка цифр по 3 с разделителем между группами).

3. Выделение в файле "чистых" числовых строк (т.е. таких, которые могут быть преобразованы в числа, возможно, вещественные), преобразование, подсчет суммы полученных таким образом чисел.

4. Для файлов с заданным заголовком (первой строкой) вывод их содержимого в виде листинга: построчно, строки пронумерованы. Заголовок считается фиксированным.

5. Подсчет суммы значений байтов каждого файла и общей по всем файлам (подсчет контрольной суммы файлов). Вместо суммирования можно использовать другую доступную функцию.

6. Аналогично предыдущему заданию, но контрольная сумма вычисляется не по байтам, а по словам (*усложненный вариант*).

Лабораторная 5. "Основы программирования на *C* под Unix"

Теоретическая часть

Практическая часть

Задания выполняются с использованием как библиотек, так и, возможно, вызовов внешних программ. Ограничений на создание временных файлов не налагается, но желательно этого избегать. Для управления обработкой проекта используется утилита *make* (для которой нужно написать соответствующий *makefile*).

1. Программа-фильтр -- инверсия порядка байт в потоке (первый -- последний, второй -- предпоследний, и т.д.).
2. Программа-фильтр -- инверсия порядка строк потока (предполагается, что поток -- текст, состоящий из отдельных строк). Длину строк можно считать ограниченной некоторой константой.
3. Программа-фильтр -- инверсия порядка символов в каждой строке потока, порядок самих строк не изменяется. Длину строк можно считать ограниченной некоторой константой.
4. Проверка и коррекция текста по словарю. Проверка производится на совпадение, без анализа словоформ; корректировать достаточно одиночные ошибки. Программа оформляется как фильтр, словарь -- внешний файл, его размер заранее не известен.
5. Проверка и "цензура" текста: удаление (замена фиксированным шаблоном) "запрещенных" слов, заданных словарем. Проверка на совпадение, без анализа словоформ. Программа оформляется как фильтр, словарь -- внешний файл, его размер заранее не известен.
6. Программа -- "записная книжка". Режим работы -- командная строка (программа

вызывается на 1 операцию, задаваемую аргументами командной строки, после его исполнения управление возвращается в *shell*. База данных -- текстовый файл, записи включают несколько полей (формат выбирается произвольно). Операции: добавление, удаление, поиск (по различным полям) и отображение, изменение записи.

7. Программа -- "записная книжка". Аналогично, но режим работы интерактивный (диалоговый): ввод команды, исполнение, ожидание следующей.
8. Программа-фильтр -- преобразование символов в комбинации азбуки Морзе. Непреобразуемые символы отбрасываются.
9. Программа-фильтр -- преобразование комбинаций азбуки Морзе в печатные символы.
10. Программа -- криптографический фильтр: шифрование и дешифрование потока. Криптоалгоритм выбирается произвольно (можно простейшие). Способ передачи пароля и других параметров шифрования выбирается произвольно. (*В случае достаточно сложного криптоалгоритма задание раздивается на 2: прямое и обратное преобразование.*).

Примечание: Более сложный вариант трех последних программ с использованием средств межпроцессного взаимодействия и сетевых протоколов имеется в соответствующих заданиях в весеннем семестре.

Лабораторная 6. "Основы управления процессами в Unix. Программы-демоны."

Теоретическая часть

Практическая часть

Формулирка задания сейчас общая, без разбивки на варианты.

Создание программы-демона, которая "умеет" обрабатывать произвольные заданные сигналы. Обработка заключается в протоколировании сигнала (записи в файл номера сигнала, его символического имени и, например, времени его получения). Список контролируемых сигналов задается файлом конфигурации, который считывается демоном при старте и по сигналу `SIGHUP` (этот сигнал также может быть среди протоколируемых).

Лабораторная 7. "Взаимодействие процессов Unix. Использование именованных (*pipe*) и неименованных (*FIFO*) каналов."

Теоретическая часть

Возможность/невозможность открытия FIFO в зависимости от наличия открытых дескрипторов на него.

Практическая часть

Общая постановка задачи: Написать программы (отдельные) или программу, порождающую в ходе выполнения дочение процессы; реализовать передачу данных

между процессами, используя каналы (именованные или неименованные в зависимости от варианта); отладить многопроцессный программный комплекс. Процессы взаимодействуют друг с другом по принципу "клиент-сервер".

1. Процесс-родитель порождает 3 дочерних процесса, каждому из которых он предоставляет собственный неименованный канал (pipe). Каждый из порожденных процессов (клиентов) передает родительскому (серверу) текстовые строки, например вводимые с консоли. Родительский процесс отображает присылаемые строки в порядке их поступления.
   Примечание. Необходимо обеспечить передачу каждому порожденному процессу дескрипторов, соответствующих именоо его "персональному" каналу, и одноврменно возможность для родительского процесса контролировать все каналы.
2. Программа, поведение которой определяются именем исполняемого файла, из которого порожден процесс. Если имя *server*, то то процесс выполняет функции сервера: создает именованный канал (FIFO), принимает из него сообщения, выделяет из сообщений 2 числа *x* и *y* и возвращает в канал ответное сообщение, содержащее сумму *x+y* ("канальный сервер-калькулятор на одно действие"). Если имя файла другое, процесс выполняет функции клиента: вводит 2 числа с консоли, формирует сообщение, передает его в канал, ожидает ответного сообщения, отображает его на консоль.
   Примечание. Необходимо правильно организовать прием и передачу данных, чтобы минимизировать вероятность "подхвата" фрагментов, относящихся к разным сообщениям. Имя файла, из которого был порожен процесс, передается ему в виде первого по счету аргумента командной строки -- параметра *argv[0]* функции *main()* (программы на языке *C*); перед проверкой имя файла должно быть освобождено от возможно присутствующего пути.
3. Программа, автоматически определяющая свою функцию -- клиент или сервер -- по наличию коммуникационного ресурса: каждый экземпляр процесса производит поиск именованного канала (FIFO) с заранее известным именем. Если FIFO не найден, процесс создает его и начинает функционировать как сервер. Если FIFO найден, процесс открывает его и начинает функционировать как клиент. Функции сервера: занести в FIFO текущее время (формат можно выбирать произвольно) и ежесекундно его обновлять. Функции клиента: считывать текущее значения времени из FIFO и выводить его на консоль (однократно при запуске или в цикле).
   Примечание. Со стороны клиента можно предусмотреть "неразрушающее" чтение -- например, немедленный возврат прочитанного значения в канал. Кроме того, клиент, вероятно, должен адекватно относиться к пустоте канала (ожидание при чтении из FIFO). В свою очередь, сервер обязан следить за постоянным наличием в канале одного и только одного (желательно правильного) значения текущего времени. Для этого требуется постоянно проверять FIFO на пустоту, а по истечении периода смены показаний удалять старое значение и записывать новое; одновременно сервер не должен блокироваться, если FIFO в этот момент окажется пустым. Вместо отдельной программы-клиена в процессе отладки можно использовать стандартные средства вывода содержимого файла (*cat*, копирование в */dev/tty*); чтение в этом

случае однозначно "разрушающее".

4. Программа -- генератор паролей. Аналогично предыдущему заданию, процесс автоматически определяет свою функцию: клиент или сервер. Сервер записывает в канал FIFO очередной сгенерированныйим пароль (алгоритм генерации и проверки паролей может быть выбиран произвольно) и ожидает опустошения канала, затем записывает новый пароль, и т.д. Клиент просто считывает пароль из FIFO и использует его (выводит на консоль), чтение обычное "разрушающее", т.к. клиент не заинтересован в повторном использовании этого же пароля другим процессом.
   Примечание. В целом аналогично предыдущему заданию. Необходимость обеспечить для сервера контроль текущего состояния канала и реагирование на его опустошение. В качестве усложняющего элемента можно потребовать от сервера периодическую смену пароля, находящегося в FIFO, если клиенты долго не забирают его оттуда.

Лабораторная 8. "Взаимодействие процессов Unix. Использование объектов System V IPC: *semaphore*, *messages*, *shared memory*."

Теоретическая часть

Идентификация объектов, генерация ключа, группы функций. Набор объектов, называемых *System V IPC*, призван комплексно решить задачи, возникающие при взаимодействии процессов и включает 3 вида объектов:

- *Semaphore* -- семафор, используемый в первую очередь для синхронизации и защиты критических секций, отличается от классического тем, что является не только многозначным, но и векторным, то есть объединяет целую группу счетчиков-семафоров.
- *Messages queue* -- сообщения, точнее очередь сообщений, пригодных для передачи небольших порций данных в том числе и между процессами, а также для синхронизации. Сообщение характеризуется его типом, который представляется целым числом. Очередь обеспечивает выборку сообщений с группировкой по типу, что позволяет реализовать приоритеты сообщений
- *Shared memory* -- разделяемая память, блок данных, который может быть отображен на адресное пространство более чем одного процесса одновременно. Это позволяет наиболее эффективно реализовать передачу значительных объемов информации между процессами, а также совместную их обработку, но одновременно может требовать дополнительных мер по защите от конфликтов (например, с помощью семафоров).

Всем экземплярам объектов System V IPC сопоставлены целочисленные (тип переменной *int* ?? *unsigned int* ?? *unsigned short*) идентификаторы, относящиеся к трем независимым пространствам идентификаторов: соответственно семафоров, очередей сообщений и разделяемый блоков памяти, и не пересекающиеся с файловыми дескрипторами или любыми другими идентификаторами в системе.
Объекты System V IPC не имеют имен, проблема их глобальной (между процессами, в том числе и не родственными друг другу) видимости решается использованием имен в

файловой системе и специального промежуточного элемента -- *ключа*.

```
#include
Определенную проблему составляет обнаружение IPC-объектов процессами,
изна-чально не связанными друг с другом. В UNIX процесс манипулирования
объектом состоит, как минимум, из двух стадий: получение идентификатора
объекта, определяемого специаль-ным ключом, и затем его использование.
Вызовы подключения объектов однотипны и имеют вид: ...get(key_t <ключ>,
[<прочие_параметры>,] int <флаги>), где пре-фикс имени вызова определяет, с
каким объектом он оперирует, например, semget - сема-форы, shmget -
разделяемая память, и т.д. Вызовы возвращает неотрицательный идентифи-катор
объекта или  1 при неуспешном завершении.
Ключ - целочисленное значение, идентифицирующее IPC-объект (в том числе еще
не созданный), но не служащее для обращений к нему. Одинаковые значениями
ключа ссыла-ются на один и тот же объект, однако повторное его создание с
тем же ключом даст, скорее всего, новый идентификатор. Ключ описывается
библиотечным типом key_t, в данной сис-теме совпадающим с int. Один из
рекомендованных руководством способов генерации ключа состоит в
использовании функции ftok, аргументами которой являются:
- строка -- имя реально существующего файла;
- целое значение -- например, порядковый номер, условный код, идентификатор
процесса (<i.pid< i="">) и т.д.
Флаги формируются наложением по ИЛИ маски прав доступа и маски режимов
от-крытия. Маска прав доступа занимает младшие 9 разрядов и аналогична
применяется к эле-ментам файловой системы: комбинация вида "rwx-rwx-rwx",
задаваемая числовым значе-ние; биты "x" (права выполнения) нулевые. Наиболее
значимыми комбинациями флагов мас-ки режима открытия являются:
- IPC_CREAT - присоединение существующего объекта, если он не существует, он
создается заново;
- IPC_EXCL | IPC_CREAT - обязательно создание нового объекта, попытка
при-соединить существующие приводит к ошибке;
- 0 - присоединение существующего объекта, ошибка если не найден.
Следует помнить, что существование IPC-объекта не зависит от состояния
создавшего его процесса, поэтому объект сохраняется до его явного удаления
либо перезапуска системы. Бесконтрольное оставление неиспользуемых объектов
может привести к нерациональному расходу системных ресурсов.
Вызовы, отвечающие за использование созданных и подключенных объектов,
инди-видуальны для различных их типов, однако имеют близкие имена и форматы
вызовов.
</i.pid<>
```

Практическая часть

Общая постановка задачи: Написать программы (программу, порождающую копии -- дочение процессы), взаимодействующие друг с другом посредством объектов System V IPC и образующие таким образом единый программный комплекс, реализующий функции в соответствии с вариантами задания.

1. Копирование файла двумя параллельными процессами: один осуществляет чтение из файла-источника, второй -- запись в файл-приемник. Используются 2 независимых буфера: для чтения и для записи, которые меняются местами (передаются между процессами) по мере соответственно заполнения и опустошения. В качестве буферов используются блоки разделяемой памяти

(*shared memory*), для синхронизации процессов и управления доступом к буферам -- семафоры (*semaphore*).
Примечание. Различие скоростей чтения и записи может приводить к простоям процессов, что необходимо учесть: заполненный "буфер чтения" можно передать как "буфер записи" в распоряжение "процесса записи" независимо от его состояния, но "процесс чтения" все равно вынужден будет ждать освобождения бывшего "буфера записи", чтобы использовать его как "буфер чтения". Представляет интерес оценка эффекта от распараллеливания процессов, но заметной она будет лишь на больших размерах файлов и меделенных устройствах.

2. Усложненный вариант предыдущего задания: вместо 2 попеременно используемых буферов в блоках разделяемой памяти создается очередь блоков, что придает системе большую гибкость и большие возможности по выравниванию скоростей чтения и записи.
3. Передача сообщений через разделяемую память: процесс-сервер последовательно извлекает сообщения из очереди, созданной в разделяемой памяти и выводит их в файл; процессы-клиенты вводят сообщения (текстовые) с консоли и передают в очередь. Используются объекты: буфер (очередь) для хранения передаваемых сообщений -- блок разделемой памяти (*shared memory*); семафоры (*semaphore*) для синхронизации и управления очередью
Примечание. Представляет интерес эксперимент с использованием в качестве передаваемых сообщений объектов -- сообщений SystemV IPC (*messages*): требуется ли размещать данные таких сообщений именно в разделяемой памяти.
4. Конвейерная обработка. В блоке разделяемой памяти (64 Кбайт), разбитой на подблоки (4 Кбайта каждый) 3 процесса выполняют последовательную обработку подблоков:
    - 1 -- генерирует случайное содержимое очередного подблока;
    - 2 -- сортирует содержимое подблока (алгоритм может быть задан произвольным образом);
    - 3 -- выводит содержимое подблоков по мере готовности.

    Примечание. Для управления выделением и обработкой подблоков в *shared memory* потребуются семафоры (*semaphore*), наборы которых должны описывать все подблоки.

5. Параллельная обработка. Начальные условия аналогичны предыдущему заданию, но распредление функций процессов иное:
    - 1 -- генерирует случайное содержимое очередного свободного подблока;
    - 2 и 3 -- реализуют сортировку блока различными алгоритмами.

    Процессы 2 и 3 конкурируют за возможность отсортировать заполненные блоки. Все процессы ведут статистику -- количество обработанных блоков -- для оценки сравнительной эффективности сортировки.
    Примечание. Считается, что сортировкой обработка блока заканчивается, но с целью отладки может быть удобно выводить их содержимое на консоль.

6. Моделирование задачи об обедающих философах. Модель имитационная, в ней участвуют процессы: "философы" 1..5 и "слуга" ("диспетчер"), а также объекты, соответствующие "блюду" и "вилкам" 1..5. Функционирование каждого процесса-"философа" сводится к осуществлению "еды": процесс пытается последовательно завладеть двумя "вилками", с помощью которых может начать использовать "блюдо" (длительность пребывания в этом состоянии -- случайная величина), по окончании "еды" он так же последовательно освобождает обе "вилки"; при этом "философы" не "общаются" друг с другом и, следовательно, не могут "договориться" о согласованных действиях. Функции "слуги" могут быть разнообразны: от простой регистрации событий и ведения статистики модели до предупреждения и разрешения конфликтов.
   Возможны подварианты модели:
   - фиксированный порядок захвата "вилок" "философами";
   - случайный выбор "вилки";
   - бесконечное ожидание "вилки";
   - тайм-аут ожидания (по истечении заданного времени "философ" обязан освободить уже взятую вилку);
   - различные законы (параметры) распредления времени "еды";
   - участие слуги в управлении взаимодействием.

   Подварианты задания:

   - сбор и затем вывод статистики модели, останов сеанса по времени;
   - обнаружение тупика и прекращение сеанса моделирования;
   - предотвращение тупика силами "слуги".

   Примечание. В данном случае для задачи не существенны сами объекты-ресурсы (критические и некритические), поэтому программная модель может не включать их. Достаточно только объектов, отвечающих за синхронизацию и управление доступом, т.е. семафоров (либо иных, используемых для этой цели).

Лабораторная 9. "Взаимодействие процессов Unix. Использование сокетов."

Теоретическая часть

Понятие сокета, характеристики, адресация, использование, группы функций, основные функции.

Практическая часть

Общая постановка задачи: Создать программу-сервер, реализующие протокол реальной сетевой службы или собственный; как правило, серверы выполняются в режиме демона. При необходимости создаются также вспомогательные программы, например, клиент для тестирования и демонстрации работоспособности сервера.
Примечание. В качестве "универсального" клиента для протокола TCP можно

использовать программу *telnet*.

7. "Сервер морзянки" -- прямое преобразование. Программа-демон, поддерживает соединения TCP, преобразует принимаемый текст в знаки азбуки Морзе. Номер базового порта TCP задается как параметр командной строки
   Примечание. В качестве знаком азбуки Морзе используются традиционные символы '.' (точка) и '–' (тире). Внутри группы, соответствующей одной букве исходного текста, знаки записываются подряд (без пробелов). Группы отделяются друг от друга пробелами. Последовательности групп, соответствующие словам исходного текста, отделяются друг от друга двойными пробелами. Регистр букв игнорируется. Нераспознанные символы исходного текста заменяются пробелами.
8. "Сервер морзянки" -- обратное преобразование. Аналогично предыдущему, но выполняется преобразование из знаков азбуки Морзе в обычный текст.
   Примечание. Правила преобразования в основном соответствуют описанным в предыдущем задании. Нераспознанные комбинации знаков игнорируются либо заменяются пробелами. Представляет интерес исследование способности преобразования к "самосинхронизации" после ошибок в знаках азбуки Морзе.
9. Сервис *date* (*daytime*). Программа-демон, реализующая сервер указанной службы с поддержкой протоколов TCP и UDP (порт номер 13): в ответ на обращение выдает текущие дату и время. Подробности см. в *RFC867*
10. Сервис *echo*. Программа-демон, реализующая сервер указанной службы с поддержкой протоколов TCP и UDP (порт номер 7): любое полученное сообщение возвращается отправителю без изменений. Подробности см. в *RFC347*
11. Сервис *systat*. Программа-демон, реализующая сервер указанной службы с поддержкой протоколов TCP и UDP (порт номер 11): в ответ на обращение выдает текущий список активных пользователей системы. Подробности см. в *RFC866*
12. Сервис *time*. Программа-демон, реализующая сервер указанной службы с поддержкой протоколов TCP и UDP (порт номер 37): в ответ на обращение выдает текущее время в секундах, прошедших с 00:00 1.01.1900. Подробности см. в *RFC868*

Лабораторная 10. "pthread: реализовать простейшей многопоточной программы"

Создать программу-сервер, реализующие протокол реальной сетевой службы или собственный; программа должна выполняться в режиме демона. Сервер должен обслуживать параллельную обработку нескольких клиентов.

1. Реализация простейшего HTTP-сервера, который отдаёт статические ресурсы
2. Реализация простейшего FTP-сервера, реализующего возможность параллельного входа нескольких анонимных пользователей
3. Реализация простейшего *многопоточного* HTTP-паука: программе-пауку передается начальный URL. По этому URL программа скачивает страничку, которую затем анализирует и вычленяет другие URL, встречающиеся в

документе. Затем программа запрашивает вновь полученные URL в отдельных потоках. Максимальное количество потоков должно быть ограничено аргументом командной строки. В конце работы программа выдает статистику: количество скаченных байтов и средняя скорость скачивания.

4. Реализация простейшего *многопоточного* FTP-паука: аналогично предыдущему, только осуществляется выкачивание всего FTP-сайта