Лабораторные работы по курсу АКОС

За каждую из трёх лабораторных работ можно получить оценку из диапазона [0, 1]. Далее оценки за лабораторные работы складываются с некоторыми весами, образуя общую оценку за лабораторные. К ней добавится оценка за контрольные, что в итоге как-то трансформируется в вашу оценку на зачёте. В случае, если я буду сомневаться в том, какую оценку вам ставить, я буду спрашивать на зачёте дополнительные вопросы.

Оценка лабораторных работ производится следующим образом: если коммит, в котором программа проходит хотя бы один разумный тест, сделан до дедлайна, штрафов за просрок нет. Просрок на 0-2 недели штрафуется 0.25 баллами, просрок на 2-4 недели отнимает у вас 0.5 балла, далее вы можете получить максимум 0.25 баллов за лабораторную. Обратите внимание, что я не штрафую вас, если не успею вовремя проверить работу. Важно, чтобы вы коммитили код, который работает хотя бы приблизительно, и, желательно, почаще.

Кроме штрафов за дедлайны вы можете получить штраф за недоделанное задание (если времени на его переделывание уже не останется или вы больше не хотите ничего менять). Это будет какой-то множитель от 0 до 1, на который я умножу вашу оценку за лабораторную, в зависимости от степени недоделанности работы.

Инструкция по работе с репозиторием

Для того, чтобы начать работать с git-репозиторием, склонируйте его к себе. Это делается выполнением следующей команды в командной строке:

```
git clone ssh://<login>@hill.cs.msu.ru/home/<login>/repos <name_of_your_repository>
```

<login> - ваш логин (его можно найти в этом документе: http://bit.ly/2loLntq)

<name_of_your_repository> - название директории, в которой ваш репозиторий будет храниться у вас на машине

Вам предложат ввести пароль от вашей учётной записи на hill.cs.msu.ru. Вводите его (он будет разослан лично каждому в письме). Чтобы не вводить пароль каждый раз при использовании репозитория, создайте себе пару ssh-ключей по этой инструкции:

```
https://www.digitalocean.com/community/tutorials/how-to-set-up-ssh-keys--2
```

Лектор постарался и написал для вас небольшую инструкцию по работе с git, которую вы найдёте внутри вашей новой папки с репозиторием. Можете её удалить, если она вам не нужна.

Каждое задание нужно оформлять в виде отдельной папки в вашем репозитории. Они должны называться lab_1, lab_2 и lab_3. Внутри каждой папки нужно будет сделать Makefile или CMakeLists.txt, чтобы любую лабораторную можно было легко и без проблем скомпилировать.

Попробуйте проделать все эти шаги и закоммитить что-нибудь тестовое, например, удалить папку task_0, которая там по дефолту присутствует. Если всё получится, я увижу ваши изменения у себя через небольшой промежуток времени и отмечу это галочкой в табличке здесь: http://bit.ly/2loLntq

Style guide

- Не коммитить бинарники в репозиторий! Для этого добавьте их в файл .gitignore в корне репозитория, чтобы при коммите git их игнорировал. Также добавьте в .gitignore следующие строки:

```
build
CMakeFiles
```

- Файл comments.txt - для проверяющего! Вам его трогать не нужно, иначе очень неудобно вспоминать, какие вопросы остались по заданию в предыдущий раз. Историю изменений проверяющий всё равно сможет увидеть через git, но это менее удобно, так что, пожалуйста, экономьте время проверяющего.
- Отступ - четыре пробела
- Скобочки - как в этом примере:

```
int some_func(int a, char* b) {
    if (a > 6 && (*b < 42 || some_func(a, b))) {
        do_something_crazy(a, b);
    } else {
        do_nothing_at_all();
    }
```

- Блоки, состоящие из одной команды, всё равно нужно отбивать скобочками:

```
if (something) {
    single_command;
}
```

- Бинарные арифметические операции отбиваются пробелами с обоих сторон, унарные пишутся вместе с операндом, и & в указателях и адресах ставятся рядом с типом, а не рядом с переменной:

```
a + b

if (!c) {
    ...
}

const char* a;

struct sockaddr& b;
```

- Циклы for, while отделяются от условия цикла пробелом:

```
for (int i = 0; i < 42; ++i) {
    ...
}
```

- Глобальные переменные мы решительно отринем, товарищи, как идеологически вредные. Согласен закрыть на них глаза только в том случае, если без них действительно громоздко и неудобно, и если всё остальное сделано хорошо и без ошибок.

Lab 1: Unix Utilites

Дедлайн: 12 марта

Описание: воспроизвести по одной утилите из каждой выделенной группы. Перед написанием кода рекомендуется прочитать man-страницу по соответствующей утилите. Список утилит для реализации (три числа от 1 до 5, номер утилиты из первой, второй и третьей группы соответственно) лежит в этой таблице: https://yadi.sk/i/7sMxRoco3Dhxqk

1) ls, cp, mv, realpath, mkdir – базовые утилиты Unix,

2) lsof, ps, pidof, top, w – процессы,

3) paste, sort, grep, wc, uniq - работа с текстом.

Вместо выпавшего вам задания из группы "процессы"можно реализовать альтернативное задание - программу procinfo. Про неё можно почитать в man-странице, реализовывать нужно основной режим, без аргументов.

Опции, которые нужно реализовать в программах (отсутствие каких-либо опций будет штрафоваться):

- Навигация
  - ls: no_args, -l, -a, -la. Выводить содержимое произвольного числа директорий, указанных в параметрах.
  - cp: file-to-file, file-to-dir, dir-to-dir (-R) (-R только если останется время, сначала сделать первые два режима). Уметь копировать произвольное число файлов в директорию, если она идёт последним аргументом. Уметь копировать бинарные файлы.
  - mv: file-to-file, file-to-dir, dir-to-dir. Уметь перемещать произвольное число файлов в директорию, если она идёт последним аргументом. Уметь перемещать бинарные файлы.
  - realpath: follow symlinks
  - mkdir: no_args, -p
- Процессы
  - lsof: only main mode (no_args)
  - ps: all processes
  - pidof: main mode
  - top: выводить следующие поля (интерактивность делать не нужно!)
    - PID – Process Id
    - PPID – Parent Process Pid
    - USER – User Name
    - NICE – Nice value
    - %CPU – Процент от потребления CPU с момента запуска top
    - TIME – Суммарное время, которое процесс исполнялся с момента запуска
    - %MEM – Потребляемая процессом память (в процентах от общей)
    - STAT – Process Status

    Вот это должно вам помочь: https://yadi.sk/i/8U1HhluV3EtnFy
- Работа с текстом
  - w: main mode
  - paste: custom delimiter (-d) with TAB as default, files in argv
  - sort: lexicographic, reverse, file or std input
  - grep: -i, -v, file or std input
  - wc: no_args, -l, -w, -c, combination
  - uniq: no_args, -c

Lab 2: Multi-threaded server and client for text chat

Дедлайн: 30 апреля

Описание: Необходимо написать серверное и клиентское приложение для текстового чата. Протокол обмена сообщениями приведён ниже.

0) TCP, UTF-8, Big-endian.

1) Сервер пушит сообщения клиентам, соединение висит постоянно

2) Формат сообщения:

   [тип сообщения](1 byte)[длина сообщения](4 bytes)[тело сообщения](длина_сообщения bytes)

   [тело_сообщения] = [длина первой строки](4 bytes)[первая строка] ... [длина последней строки](4 bytes)[последняя строка]

   Числа - это тоже строки, записываются в виде последовательности байт (big-endian). Число 0x12345678 будет записано как 0x00 0x00 0x00 0x04 0x12 0x34 0x56 0x78.

   Время - UNIX time with microseconds (8 bytes), 4 byte for seconds, 4 byte for microseconds. (gettimeofday)

3) типы сообщений:

   3.1) Общие типы сообщений

   * **s** (status) Тело сообщения - строка со статусом (int, 4 bytes)
     0 - OK
     1 - сообщение неизвестного типа
     2 - незалогиненный пользователь
     3 - ошибка аутентификации
     4 - ошибка регистрации
     5 - ошибка доступа
     6 - invalid message

   3.2) Сообщения "client → server"

   * **r** (regular)
     Тело сообщения: одна строка с сообщением
     Ответ от сервера: regular-сообщение, содержащее серверный timestamp, логин и то же самое сообщение, или статусное сообщение с ошибкой
   * **i** (login)
     Тело сообщения: две строки – имя пользователя, пароль
     Ответ от сервера: статусное сообщение

   * **o** (logout)
     Тело сообщения: пустое
     Ответ от сервера: пустой
   * **h** (history)
     Тело сообщения: строка с количеством запрашиваемых последних сообщений из чата (только regular)
     Ответ сервера: несколько h-сообщений или статусное сообщение. При запросе более 50 сообщений присылаются только последние 50 сообщений.
   * **l** (list) – запросить список пользователей, которые сейчас онлайн
     Тело сообщения: пустое
     Ответ сервера: список пользователей или статусное сообщение

* **k** (kick)
  Тело сообщения: две строки – user_id участника, причина удаления из чата
  Ответ сервера: статусное сообщение

3.3) Сообщения "server → client"

* **r** (regular) – текстовое сообщение от одного из участников чата
  Тело сообщения: три строки – timestamp, login, сообщение
* **m** (message) – произвольное сообщение от сервера
  Тело сообщения: две строки – timestamp, произвольная строка
* **h** (history)
  Тело сообщения: три строки – timestamp, login, сообщение
* **l** (list) – список участников чата, которые сейчас онлайн
  Тело сообщения: последовательность пар строк ($user\_id_i$, $login_i$)
* **k** (kick) – посылается при удалении участника из чата
  Тело сообщения: произвольная строка (причина удаления)

4)
- Логин не может содержать символов меньших, чем ' ' (пробел). Логин должен быть непустой. Длина логина и пароля не может быть меньше двух и больше 31 символов. Длина regular-сообщения не может быть больше 65535 символов.
- При присоединении, удалении и выходе участника из чата сервер отправляет всем клиентам соответствующее m-сообщение.
- Порт задаётся в параметрах сервера, дефолтный - 1337.
- Порядок авторизации: при создании сессии от клиента ожидается пароль, который сервер запоминает до выключения. Потом, при авторизации с другого устройства под тем же логином или при перезапуске клиента, будет ожидаться тот же пароль. User_id у одного и того же пользователя, залогинившегося с разных устройств, должен быть одинаковый.
- В системе будет один суперпользователь (root), который может посылать сообщения kick. При осуществлении kick сервер обрывает соединение с клиентом. Если root пытается кикнуть несуществующего или незалогиненного пользователя, сервер отвечает ему статусным сообщением 2 (незалогиненный пользователь).
- При запуске сервера спрашивается пароль для рута.
- Если клиент присылает битое сообщение (неправильная длина сообщения, невалидные символы), то сервер отвечает статусным сообщением 6 (invalid message).

Lab 3: Shell

Дедлайн: no deadline

Требуется написать программу на языке C, осуществляющую частичную эмуляцию командной оболочки (shell).

Командная оболочка представляет собой интерактивную программу, последовательно выполняющую вводимые пользователем команды. Поток команд берется со стандартного потока ввода. Считывать команды нужно в цикле до тех пор, пока на очередной итерации не будет получен конец файла (детектируется при помощи константы EOF или проверкой возвращаемых значений функции чтения из файлов/потоков), либо не будет введена команда **exit**. Перед считыванием нужно вывести на экран приглашение к набору команд, чтобы после набора команд строка выглядела, например, так:

```
username$ ./path/to/binary with "$SOME" 'arguments' | ./piped to > file
```

Длина считываемой строки с командами ограничена только размером виртуальной памяти, доступной процессу.

Программа должна корректно обрабатывать все ошибки, в том числе ошибки синтаксиса, выдавая осмысленные информационные сообщения о них в стандартный поток ошибок.

Описание языка команд shell

Программа на языке shell - это последовательность групп команд, отделённых друг от друга символами-разделителями. Группа команд состоит из одиночных команд, объединённых в конвейер при помощи символа `'|'`. Одиночная команда состоит из запуска некоторой программы (или специальной встроенной команды) с произвольным набором параметров, при этом ввод-вывод этой программы (или встроенной команды) может быть перенаправлен в файл. Все значащие конструкции языка (строки) разделяются любым ненулевым количеством пробельных символов. Последовательность пробельных символов интерпретируется как один символ пробела.

Грамматика языка shell записывается следующим образом:

`program` = [command SEPARATOR]*

`command_group` = single_command [PIPE single_command]* [AMPERSAND]

`single_command` = command argument_list [IO_REDIR filename]*

`command` = {path | special_command}

`special_command` ∈ ['cd', 'history', 'exit']

`IO_REDIR` ∈ ['>', '>>', '<']

`PIPE` = '|'

`AMPERSAND` = '&'

`SEPARATOR` ∈ ['\n', ';', EOF]

`path` - это путь до исполняемого файла в операционной системе.

`argument_list` - это набор строк, разделённых пробельными символами.

Если команда слишком длинная, можно переносить часть команды на новую строку, завершив предыдущую строку символом `'\'`. Это правило не описывается приведённой выше грамматикой, и описано текстом, чтобы не усложнять грамматику.

В командах могут встречаться следующие конструкции:

- *Подстановка значений переменных*

  Перед тем, как выполнять команду, необходимо подставить значения переменных во все места их использования. Переменные shell разделяются на служебные и пользовательские. Данное задание предполагает подстановку только служебных переменных.

  Значение переменной – это некоторый текст, который сопоставлен имени переменной. В случае обнаружения в строке конструкции вида `${VARIABLE_NAME}`, вместо этой конструкции в строку будет подставлена строка, являющаяся значением переменной.

- *Текст в кавычках*

  Кавычки бывают двух видов: двойные (`"`) и одинарные (`'`). В тексте, содержащем подстроку в двойных кавычках, производится подстановка значений переменных; в тексте в одинарных кавычках подстановка не производится.

  Например, для пользователя с именем vasya будет напечатано следующее:

```
vasya$ echo "I am ${USER}"
I am vasya
vasya$ echo 'I am ${USER}'
I am ${USER}
```

- *Экранирование символа*

  Экранирование осуществляется при помощи символа `'\'`. Символ, идущий после `'\'`, не будет иметь «служебного» смысла. Например, обратный слэш позволяет поставить символ кавычки внутри текста, обёрнутого в кавычки. Последовательность `'\\'` позволяет ввести обычный одинарный символ обратного слэша.

- *Комментарии*

  Если во входной строке встречается символ `'#'`, который не находится внутри одинарных или двойных кавычек и не экранирован обратным слэшом, то все дальнейшие символы в этой строке (и сам символ `'#'`) считаются комментариями и игнорируются.

Команды и запуск команд

Одиночные команды можно разделить на две группы – встроенные команды shell и внешние программы, которые запускаются так, что каждая программа оказывается в своём отдельном процессе.

При запуске команды нужно передавать ей список её аргументов. Например, одиночная команда

```
ls -l -a ../.. #конец аргументов
```

должна передать программе ls после её запуска такой набор аргументов:

```
argv = ['ls', '-l', '-a', '../..'].
```

После списка аргументов прогаммы допускается указывать символы '<' и '>', которые позволяют считать стандартный ввод из файла или вывести стандартный вывод в файл, а также последовательность символов '>>', которая позволяет дописать вывод программы в конец указанного файла. Например:

```
# cat принимает на вход данные
# из файла file.in и дописывает
# их в файл file.out
username$ cat < file.in >> file.out
```

Допускается перенаправлять вывод в несколько файлов, в таком случае вся информация должна выводиться в последний из них. То же правило выполняется и для ввода из нескольких файлов – данные вводятся только из последнего указанного файла.

В конце группы команд, после перенаправлений ввода/вывода, допускается указать символ '&', который означает запуск команды в фоновом режиме. Подробнее про фоновый режим можно прочитать в следующем разделе.

При помощи символа '|' организуется конвейер. Если две команды разделены символом конвейера, то команда справа получает на стандартный поток ввода то, что команда слева выводит на свой стандартный поток вывода. В конвейер можно объединять произвольное число команд. В случае, если имело место перенаправление ввода/вывода, приоритет отдаётся перенаправлению, а не конвейеру. В этом случае процесс, подключённый к конвейеру справа, получит на вход символ конца файла (EOF), либо процесс слева будет некому «слушать» из конвейера, и он, вероятно, получит себе сигнал *SIGPIPE*.

Специальные команды

- **history** – выводит историю запущенных групп команд, пронумерованных по возрастанию от самой ранней (номер 1) до самой недавней, в следующем виде:

```
username$ history | grep gcc
 1022  gcc -E  test_abc.c
 1023  gcc -E  test_abc.c > /tmp/filo.c
 1025  gcc -S  test_abc.c
 1027  gcc -c  test_abc.c
 1029  gcc -o test_abc test_abc.c
 1032  gcc -o test_abc test_abc.c -lm
 1035  gcc -o test_abc test_abc.c -lm
 1206  gcc -g redactor.c
 1229  gcc -g ilya.c
 1476  gcc prog32.c
 1852  gcc terminal_manipulation.c
 1855  gcc term_mode_change.c
 1859  gcc term_mode_change.c
 1862  gcc term_mode_change.c
 1964  gcc -Wall -ansi -pedantic -g main9.c
 1966  gcc -Wall -ansi -pedantic -g main9.c
 1968  gcc -Wall -ansi -pedantic -g main9.c
 1994  if gcc foo ; then echo "OK"; fi
 1995  if gcc foo ; then echo "OK"; else echo "fooo";  fi
 2005  history | grep gcc
```

- **cd** – осуществляет переход в каталог, указанный первым в аргументах команды. Если вместо каталога в аргументах указан символ '-', то необходимо перейти в предыдущий посещённый каталог, если он определён, или выдать сообщение об ошибке.

- **exit** – завершает работу shell, завершает все запущенные дочерние процессы, освобождает занятую память.
- **jobs** – выводит список активных в текущий момент групп комманд. Каждая группа команд должна иметь уникальный идентификатор, по которому можно вывести эту группу в режим переднего плана.
- **fg** – выводит группу команд в режим переднего плана. Принимает аргумент – идентификатор группы команд. Если аргумент не задан, выводит на передний план последнюю группу команд или печатает сообщение об ошибке, если список jobs пустой или не удаётся вывести указанную группу в режим переднего плана.
- **bg** – уводит группу команд в фоновый режим. Принимает аргумент – идентификатор группы команд. Если аргумент не задан, уводит в фоновый режим последнюю группу команд или печатает сообщение об ошибке, если список jobs пустой или не удаётся вывести указанную группу в фоновый режим.

Режим переднего плана и фоновый режим

В режиме переднего плана (foreground) группы команд исполняются последовательно, одна за другой. Каждая группа команд на время своего выполнения получает терминал в свой монопольный доступ. Нажатие *Ctrl+c* должно приводить к посылке сигнала SIGINT текущей выполняющейся группе команд, но не самому процессу, реализующему shell. По нажатию *Ctrl+z* группа команд должна приостанавливаться (сигнал SIGSTP), после чего должно выводиться приглашение shell, которое позволяет запустить новую порцию команд или продолжить выполнение приостановленной ранее группы команд путём указания её номера во встроенной команде **fg** или **bg**. Список запущенных групп команд можно посмотреть при помощи команды **jobs**.

Группа команд, запущенная в фоновом режиме (background), не должна обращаться к терминалу. В случае попытки обращения к терминалу процесс из фоновой группы команд должен быть приостановлен (не завершён). Если группа команд запускается в фоновом режиме, то процессы в ней не должны получать сигнал *SIGINT* в случае нажатия *Ctrl+c* на клавиатуре. В случае запуска работы в фоновом режиме shell не ждёт её завершения, а сразу выдаёт приглашение к вводу следующего набора команд, либо выполняет следующую группу команд (например, если несколько групп команд в одной строке были разделены символом `';'`). В случае завершения фоновой работы shell информирует об этом пользователя, выдавая соответствующее сообщение в стандартный поток ошибок основным процессом shell.

Дополнительные требования

Требуется реализовать подстановку любых переменных, которые перед запуском выставлены в переменных окружения, в том числе служебных:

- `$цифра` – подстановка соответствующего аргумента командной строки самого shell
- `$#` – число параметров, переданное shell
- `$?` – значение статуса последнего завершившегося процесса в последней выполнившейся группе команд переднего плана
- `${USER}` – login пользователя
- `${HOME}` – домашний каталог пользователя
- `${SHELL}` – имя shell (путь до того места в файловой системе, где находится исполняемый файл с ним)
- `${UID}` – идентификатор пользователя
- `${PWD}` – текущий каталог
- `${PID}` – pid shell

О работе с переменными окружения можно прочитать в man-странице про getenv.

Рекомендации по написанию кода задания

На начальном этапе при выполнении задания необходимо научиться считывать команды shell и сохранять их в оперативной памяти (не выполняя сами команды). Рекомендуется хранить данные в памяти как массив структур следующего вида:

```
struct command {
    char* name;
    int arg_number;
    char** arguments;
    char* input_file;
    char* output_file;
    int output_type; /* 1 - rewrite, 2 - append */
};

struct command_group {
    int background;
    struct program* programs;
    int programs_number;
};
```

После того, как вы убедились, что разбор команд происходит правильно, можно приступать к реализации запуска действий, связанных с командами. Для реализации команды **history** целесообразно использовать очередь с ограничением на её длину.

Для корректного отслеживания завершающихся процессов полезно определить callback на сигнал SIGCHLD.

Не стоит надеяться, что служебные переменные, такие, как `${HOME}`, будут для вас выставлены; нужно определить их значения изнутри shell.