

### Задание практикума Командный интерпретатор (Shell).

Необходимо реализовать под управлением ОС Unix интерактивный командный интерпретатор (некоторый аналог shell), осуществляющий в цикле считывание командной строки со стандартного ввода, ее анализ и исполнение соответствующих действий.

В командной строке могут присутствовать следующие операции:

указаны в порядке убывания приоритетов (на одной строке приоритет одинаков)

, >, >>, <
&&,
;, &

Допустимы также **круглые скобки**, которые позволяют изменить порядок выполнения операций.

В командной строке допустимо также произвольное количество пробелов между составляющими ее словами.

Разбор командной строки осуществляется Shell-ом по следующим правилам:

**<Команда Shella >** →

< Команда с условным выполнением > { [ ; | & ] < Команда Shella > } { ; | & }

**<Команда с условным выполнением >** →

<Команда> { [ && | || ] <Команда с условным выполнением> }

**<Команда>** → { <перенаправление ввода/вывода> } <Конвейер> |

<Конвейер> { <перенаправление ввода/вывода> } | ( <Команда Shella> )

**<перенаправление ввода/вывода>** →

{ <перенаправление ввода > } <перенаправление вывода> |

{ <перенаправление вывода> } <перенаправление ввода >

**<перенаправление ввода >** → '<' файл

**<перенаправление вывода>** → '>' файл | '>>' файл

**<Конвейер>** → <Простая команда> { '|' <Конвейер> }

**<Простая команда>** → <имя команды> <список аргументов>

**{X}** – означает, что X может отсутствовать;

**[x|y]** – значит, что должен присутствовать один из вариантов : x либо y

| - в описании правил то же, что «ИЛИ»

**pr1 | ... | prN** – конвейер: стандартный вывод всех команд, кроме последней, направляется на стандартный ввод следующей команды конвейера. Каждая команда выполняется как самостоятельный процесс, все **pr*i*** выполняются **параллельно**. Shell ожидает

завершения последней команды для проверки ее статуса в команде с условным выполнением.

Код завершения конвейера = код завершения последней команды конвейера. Остальные команды конвейера также не должны стать «зомби». Остальные команды конвейера также не должны стать «зомби».

Простую команду можно рассматривать как частный случай конвейера (конвейер из одной команды).

**com1 ; com2** – означает, что команды будут выполняться последовательно.

**com &** - запуск команды в фоновом режиме (т.е. Shell готов к вводу следующей команды, не ожидая завершения данной команды com, а com не реагирует на сигналы завершения, посылаемые с клавиатуры, например, на нажатие Ctrl-C ). После завершения выполнения фоновой команды не должно остаться процесса – зомби. Посмотреть список работающих процессов можно с помощью команды **ps**.

**com1 && com2** - выполнить **com1**, если она завершилась успешно, выполнить **com2**;

**com1 || com2** - выполнить **com1**, если она завершилась неуспешно, выполнить **com2**.

Должен быть проверен и системный успех и значение, возвращенное **exit** ( 0 – успех).

### Перенаправление ввода-вывода :

< **файл** - файл используется в качестве стандартного ввода;

> **файл** - стандартный вывод направляется в файл (если файла не было - он создается, если файл уже существовал, то его старое содержимое отбрасывается, т.е. происходит вывод с перезаписью);

>> **файл** – стандартный вывод направляется в файл ( если файла не было - он создается, если файл уже существовал, то его старое содержимое сохраняется, а запись производится в конец файла).

Замечание. В приведенных правилах указаны все возможные способы размещения команд перенаправления ввода/вывода в командной строке, допустимые стандартом POSIX.

Shell, как правило, поддерживает лишь какую-то часть из них. Для реализации можно выбрать любой (достаточно один) вариант размещения.

**Круглые скобки ()** - группируют команды для запуска в отдельном экземпляре Shella.

**com1 && (com2; com3)**

Здесь команды com2 и com3 будут выполнены только при успешном завершении com1.

**(com1; com2) &**

В фоновом режиме будет выполняться последовательность команд com1 и com2.

Круглые скобки (), кроме выполнения функции группировки, выполняют и функцию вызова нового экземпляра интерпретатора Shell. В последовательности команд

**cd ..; ls; ls** две команды **ls** выдадут 2 экземпляра содержимого родительского каталога , а последовательность

**(cd ..; ls) ; ls** выдаст сначала содержимое родительского каталога, а затем содержимое текущего каталога, т.к. при входе в скобки вызывается новый экземпляр Shell, в рамках которого и осуществляется переход. При выходе из круглых скобок происходит возврат в старый Shell и в старый каталог.

Пример, на котором можно проверить работу со скобками, требует того, чтобы в программе была написана встроенная команда **cd** для смены каталога. Встроенная, значит запускается как функция, а не как исполняемый файл. При желании можно создать исполняемый файл для смены каталога.

**ps; ls; (cd namedir; ls; ps) && ls && ps**

Первый и последний **ls** отработают в текущем каталоге, **ls** в скобках будет выполнен в каталоге **namedir**, **ps** в скобках должен показать наличие дополнительного процесса Shell.

Для выполнения команды в скобках создаётся отдельный экземпляр Shella. Поскольку команда в скобках является тут частью команды с условным выполнением, необходимо ожидание и проверка успешности его ( процесса Shell) выполнения.

Еще пример : **date; (ls -l|cat -n)>f & pwd**

Обязательный минимум (достаточный для получения оценки 3) – это реализация конвейера, фонового режима и перенаправлений ввода-вывода.

### **Про моделирование фонового режима.**

Основные требования, которым должен удовлетворять фоновый процесс в вашей программе:

Он должен работать параллельно с основной программой.

После запуска фонового процесса Шелл может запускать на выполнение следующую команду, не дожидаясь, пока фоновый процесс закончит работу.

Он не должен реагировать на сигналы, приходящие с клавиатуры.

Вообще таких сигналов несколько, но в вашей программе достаточно не реагировать на SIGINT (сигнал, который вызывается нажатием Ctrl-C). Сигналы с клавиатуры получают только процессы основной (не фоновой) группы. Они завершаются, а фоновые процессы продолжают работать.

Фоновый процесс не имеет доступа к терминалу, т.е. не должен читать со стандартного ввода (это достигается перенаправлением стандартного ввода на файл устройства /dev/null, чтение из которого сразу дает EOF). Вывод на экран можно разрешить для отладки, а можно и запретить, перенаправив стандартный вывод на тот же /dev/null (вывод будет просто пропадать).

После завершения фонового процесса не должно остаться процесса «зомби». А его не остается либо, когда родительский процесс завершается раньше, чем «сын», либо, когда в родительском процессе вызывается функция wait или waitpid.

Первый вариант моделирования фонового режима, применявшийся в шеллах до того как появились системы управления заданиями, использует сигналы.

Схема такая:

Процесс, созданный для запуска фоновой команды, перенаправляет стандартный ввод на файл **“/dev/null”** – теперь при попытке чтения со стандартного ввода сразу будет получен конец файла, так что не будет конфликта чтения между основным процессом и фоновым; вывод тоже можно перенаправить на **“/dev/null”**, тогда он будет просто пропадать, но можно и оставить для отладки;

устанавливает игнорирование сигнала SIGINT ( signal(SIGINT,SIG\_IGN));

запускает на выполнение собственно фоновый процесс.

Другой, простой способ сделать процесс фоновым (разумеется, простой для нашего случая моделирования, поскольку реально усилий требуется больше) – это выделить его в отдельную группу, фоновую.

При создании новый процесс автоматически помещается в ту же группу, что и его родительский процесс.

Поместить процесс с номером pid в группу с номером pgid можно с помощью функции **int setpgid (pid\_t pid, pid\_t pgid)** , возвращает 0 при успехе, -1 при возникновении ошибки.

Вызов функции `setpgid(0, 0)` (в некоторых системах вызов должен быть без параметров, а функция может называться **setpgrp**) помещает текущий процесс в новую группу, номер которой становится равным номеру текущего процесса.

Чтобы не оставалось процесса-«зомби», запускать фоновую команду можно, например, следующим образом:

Основной процесс- шелл создает «сына», дожидается его окончания и считывает следующую команду.

«Сын» запускает «внука» и умирает. При этом «отцом» «внука» становится `init` (процесс с номером 1), что избавляет от возникновения «зомби» после окончания «внука».

Во «внуке» запускается уже собственно фоновая команда.

Впрочем, решать проблему «зомби» можно и другим способом, обеспечивая вызов функции `waitpid` без блокирования нужное количество раз, например, перед вводом очередной команды или при получении сигнала `SIGCHLD`.