

Утверждаю

Ректор университета

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_О.Н.Федонин

«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_2020 г.

**UNIX-СИСТЕМЫ**

**ПРОЦЕССЫ В UNIX**

Методические указания

к выполнению лабораторной работы №6

для студентов очной формы обучения

по направлениям подготовки

09.03.01 «Информатика и вычислительная техника»,

09.03.04 «Программная инженерия»,

02.03.03 «Математическое обеспечение и администрирование информационных систем»

**Брянск 2020**

**УКД 004.43**

UNIX-системы. Процессы в UNIX [Электронный ресурс]: методические указания к выполнению лабораторной работы №6 для студентов очной формы обучения по направлениям подготовки 09.03.01 «Информатика и вычислительная техника», 09.03.04 «Программная инженерия», 02.03.03 «Математическое обеспечение и администрирование информационных систем». – Брянск: БГТУ, 2020. – 25 с.

Разработал

Е.О.Трубаков,

канд. техн. наук, доц.

Рекомендовано кафедрой «Информатика и программное обеспечение» БГТУ (протокол №5 от 26.01.2020)

**Методические указания публикуются в авторской редакции**

# Цель работы

Цель работы – получить представление о принципах организации исполнения программ в ОС *UNIX*. Ознакомиться с некоторыми средствами управления процессами. Приобрести навыки программирования в ОС *Unix* с использованием функций межпроцессного взаимодействия. А именно: изучить системные вызовы для работы с процессами: *fork*(), *exec*(), *wait*(), *exit*(), *getpid*(), *getppid*(); изучить системные вызовы для организации взаимодействия между процессами: *kill*(), *signal*(), *pipe*(); изучить системные вызовы для работы с разделяемой памятью: *shmget*(), *shmat*(), *shmdt*(); ознакомится с примерами программирования на базе исходных текстов утилит *Unix*; получить практические навыки разработки программ для ОС *Unix* с использованием функций межпроцессного взаимодействия.

Продолжительность работы – 2 часа.

# Теоретическая часть

Процессом называется программа в стадии ее выполнения. Процесс включает:

* программный код – последовательность команд, исполняемых процессором;
* данные, обрабатываемые программой;
* стек – динамически выделяемая память для хранения оперативной информации;
* системную информацию о выполняемой задаче (размещаемая память, открытые файлы, статус процесса, системные переменные окружения).

ОС *UNIX* является многозадачной. Это означает, что одновременно могут выполняться несколько процессов, причем часть процессов могут являться образцами одной программы. Выполнение процесса заключается в исполнении набора инструкций, который никогда непосредственно не передает управление набору инструкций другого процесса. Процесс считывает и записывает информацию в область данных и в стек, но ему недоступны данные из и стеки других процессов. Современные реализации *UNIX*, как правило, выделяют каждому процессу независимые адресные пространства. В то же время процессы имеют возможность обмениваться друг с другом данными с помощью предоставляемых *UNIX* средств межпроцессного взаимодействия (сигналы, каналы, разделяемая память, семафоры, сообщения, файлы).

Процесс обладает средствами порождения новых процессов – системными вызовами *exec* (прекращение своей работы с замещением самого себя новым процессом) и *fork* (порождение полной копии процесса). Типичной также является конструкция «*fork*-и-*exec*» - процесс порождает свою копию, которая сразу же замещает себя на новый процесс.

К основным атрибутам процесса в *UNIX* относятся:

* *PID* – уникальный целочисленный идентификатор процесса;
* *PPID* – идентификатор родительского процесса (процесса, породившего данный);
* Приоритет процесса – число в пределах от –20 до 20, определяющее относительную долю процессорного времени, отводимого данному процессу. Значение –20 соответствует максимальному приоритету, 0 – стандартному приоритету;
* *TTY* – терминальная линия, т.е. устройство, с которым по умолчанию ассоциирован стандартный ввод и вывод процесса (обычно – рабочее место, с которого запущен процесс).
* *RID* и *RGID* – соответственно идентификаторы пользователя, запустившего процесс, и его группы.
* *EID* и *EGID* – то же, но определяют пользователя, от имени которого действует процесс, что задает права доступа к файлам и системным операциям. Обычно (но не всегда) совпадают с *RID* и *RGID*.

Существуют следующие типы процессов:

1. Системные процессы. Являются частью ядра системы и всегда расположены в оперативной памяти. Обладают наивысшим приоритетом и полным доступом ко всем ресурсам системы. Выполняют системные функции: начальный запуск системы, управление памятью, диспетчеризация прикладных процессов, управление физическими устройствами, а также ряд важнейших сервисных функций – организация файловой системы, базовые сетевые операции (основные протоколы, *firewall* и т.п.) и т.д. Обычно ядро целиком загружается из единого двоичного файла (/*unix*, /*vmunix*, /*bzImage* или др.) при старте системы (технология монолитного ядра). В некоторых системах часть функций вынесены в отдельные, динамически подключаемые и удаляемые файлы – модули ядра*.* Системы, в которых само ядро реализует лишь минимальный набор функций, а модули ядра несут основную функциональную нагрузку, называют системами, построенными по технологии микроядра.
2. Процессы-«демоны» (англ*. daemon* – от аббревиатуры *Disk And Execution Monitor* – следящий за диском и исполнением) – не интерактивные процессы, которые запускаются обычным образом – путем загрузки в память соответствующих программных (исполняемых) файлов и выполняются в фоновом режиме. Обычно демоны запускаются при инициализации системы, но после инициализации ядра, и обеспечивают работу различных подсистем *UNIX*: системы доступа с терминалов, печати, сетевых услуг и т.д., а также серверных приложений (сервера *Internet*, баз данных и др.). Демоны не связаны ни с одним пользовательским сеансом работы и обычно не могут непосредственно управляться пользователем. Большую часть времени демоны ожидают, пока тот или иной процесс запросит определенную услугу, например, печать документа или доступ к базе данных.
3. Прикладные (пользовательские) процессы. К прикладным относятся все остальные процессы, выполняющиеся в системе. Как правило, это процессы, порожденные в рамках пользовательского сеанса работы. Например, запуск любой команды (*ls*, *mv*…) породит соответствующий процесс этого типа. Важнейшим пользовательским процессом является основной командный интерпретатор (оболочка, *shell*), который обеспечивает работу пользователя в UNIX. Он запускается сразу же после успешного входа, обеспечивает ввод и выполнение команд; завершение работы основной оболочки приводит к отключению от системы.

Пользовательские процессы могут выполняться как в интерактивном (диалоговом), так и в фоновом режиме. При выходе пользователя из системы все его процессы уничтожаются.

В *UNIX* различают два класса исполняемых файлов (программ) – двоичные (бинарные) и интерпретируемые (сценарии, скрипты).

Двоичныепрограммные файлы получаются путем компиляции исходных текстов программ (как правило, реализуемых на языке *С* или *С++*) и содержат набор машинных команд (инструкций процессора), непосредственно исполняемых процессором, а также статические исходные данные. Существуют собственно исполняемые двоичные файлы, а также библиотеки. Библиотекипредставляют собой фрагменты программного кода, вынесенные в отдельный файл и подключаемые к основной программе либо на этапе компиляции (статическое связывание), либо на этапе исполнения (динамическое связывание, называемое также использованием разделяемых библиотек). Среди стандартных двоичных форматов в современных *UNIX*-системах наиболее распространены *COFF* и *ELF*, а также «старый» формат – *a.out*. Двоичные программные файлы обеспечивают максимальное быстродействие.

Интерпретируемые программы представляют собой текстовые файлы, составленные из команд некоторого командного языка (обычно это один из стандартных командных интерпретаторов (*shell*) либо интерпретирующий язык программирования, напр. *Perl*). При запуске программы-сценария система загружает в память соответствующий интерпретатор, обычно выполненный в виде бинарного файла, который считывает команды и исполняет их. Производительность программ-сценариев обычно ниже за счет времени, необходимого на анализ текста команд и преобразования из в соответствующий набор процессорных инструкций.

Особое (промежуточное) место занимают исполняемые файлы, реализованные средствами языка *Java*. Такие программы представляют собой скомпилированный двоичный код, независимый от типа ОС и архитектуры ЭВМ. Выполнение бинарных *Java*-файлов происходит в среде специального программного средства, называемого виртуальной машиной *Java* и по сути является интерпретацией.

С каждым процессом ассоциируются потоки ввода-вывода *–* интерфейсы, позволяющие процессу обмениваться информацией с внешней средой. Каждый процесс имеет потоки (рисунок 1):

* поток стандартного ввода *stdin*(по умолчанию - клавиатура);
* поток стандартного вывода *stdout*(по умолчанию – экран терминала);
* поток сообщений *stderr*(служит для вывода сообщений об ошибках и предупреждений; по умолчанию – также экран терминала).

При непосредственном запуске процесса (программы) средствами командного интерпретатора *shell* (ввод команды в командной строке, фрагмент *shell*-сценария) установкам по умолчанию соответствует простая команда, состоящая из имени исполняемого файла, и, возможно, параметров.

Пусть *proc1* – некоторый исполняемый файл, содержащий программу, которая выполняет чтение со стандартного ввода и вывод информации в стандартный выходной поток. Тогда запуск этого исполняемого файла командой *proc1* будет означать связывание входящего потока с клавиатурой, а обоих исходящих – с экраном терминала.

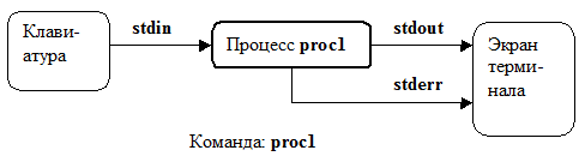


Рис. 1. Входные и выходные потоки (установки по умолчанию)

Командный интерпретатор позволяет перенаправить потоки стандартного ввода и вывода путем использования специальных параметров командной строки.

Запись *proc1 <f1.dat* приведет к связыванию потока стандартного ввода с файлом *f1.dat* (рисунок 2). При этом те данные, которые предполагалось вводить с клавиатуры, будут считаны из файла.

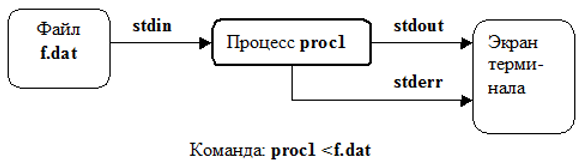


Рис. 2. Перенаправление входного потока (чтение из файла)

Запись *proc1 >f1.dat* приведет к связыванию потока стандартного вывода с файлом *f1.dat* (рисунок 2). При этом информация, которая по умолчанию выводится на экран, будет записана в файл *f1.dat*; предыдущее содержимое файла будет уничтожено.

Для дозаписи в конец файла (без уничтожения содержимого) следует использовать конструкцию *proc1 >>f1.dat*

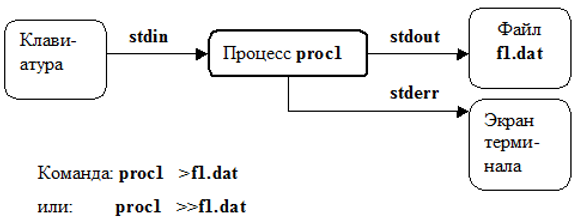


Рис. 3. Перенаправление выходного потока (запись результатов в файл)

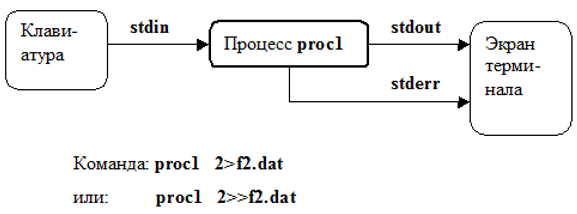


Рис. 4. Перенаправление потока сообщений об ошибках (запись протокола ошибках в файл)

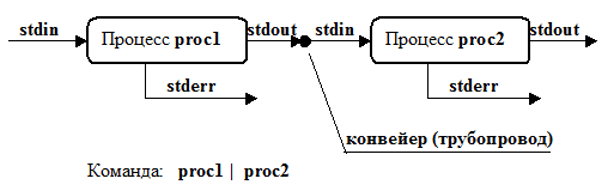


Рис. 5. Организация конвейера (трубопровода) – результаты работы первого процесса становятся исходными данными для второго

Интерпретатор *shell* выполняет и некоторые другие операции, помимо связывания через программный канал. Рассмотрим кратко вопрос одновременного выполнения нескольких программ, о чем уже упоминалось при обсуждении программных каналов. Например, можно запустить две команды с помощью одной командной строки, разделив их точкой с запятой; интерпретатор *shell* распознает этот символ и разобьет строку на две команды:

$ date; who

Tue Sep 27 01:03:17 EDT 1983

ken tty0 Sep 27 00:43

dmr tty1 Sep 26 23:45

rob tty2 Sep 26 23:59

bwk tty3 Sep 27 00:06

jj tty4 Sep 26 23:31

you tty5 Sep 26 23:04

her tty7 Sep 26 23:34

Обе команды будут выполнены (подряд) прежде, чем интерпретатор вновь вернется с приглашением.

Можно также при желании запустить несколько команд одновременно. Предположим, что вы собираетесь заняться длительными вычислениями, например, подсчитать число слов в вашей книге, но не хотите ждать окончания команды *wc* для перехода к другой работе. Тогда можно задать:

$ wc ch\* > wc.out &

6944 Shell дает номер процесса

$

Амперсанд (*&*) в конце командной строки указывает интерпретатору, что нужно запустить данную команду, а затем сразу перейти к получению последующих команд с терминала, т. е. не ждать ее завершения. Итак, команда будет выполняться, а вы можете отвлечься на что-нибудь другое. Переключение выходного потока на файл *wc.out* предотвращает возможность его смешивания с той информацией, которая появится на терминале в процессе дальнейшей работы.

Каждый экземпляр запущенной программы называется процессом. Число, выдаваемое *shell* в ответ на команду, введенную с *&*, является номером процесса. Его можно использовать в других командах в качестве ссылки на данный экземпляр выполняемой программы.

Важно понимать различие между программами и процессами. Скажем, *wc* – это программа, но каждый запуск программы *wc* создает новый процесс. Если одновременно выполняется несколько экземпляров одной программы, то любой из них считается отдельным процессом с отличным от других номером.

Если конвейер завершается операцией *&*

$ pr ch \* | lpr &

6951 Номер процесса

$

то все процессы этого конвейера начинают выполняться сразу, и *&* относится ко всем программам, участвующим в конвейере. Однако выдается только номер процесса, относящийся к последней программе в конвейере. Команда *$ wait* ожидает, пока не завершатся все процессы, запущенные с помощью *&*. Если она не возвращается сразу, значит, у вас есть незавершенные команды. Прервать выполнение команд можно, нажав клавишу *DELETE*.

Можно использовать номер процесса, сообщаемый интерпретатором, для остановки процесса, инициированного операцией *&*: *$ kill 6944.*

Если вы забыли номер процесса, команда *ps* выведет сообщение обо всех ваших процессах. В том случае, когда вам некогда, команда *kill 0* уничтожит все ваши процессы, за исключением начального процесса-интерпретатора. Если же вам интересно, что делают другие пользователи, команда *ps -ag* сообщит обо всех выполняемых процессах. Пример вывода:

$ ps -ag

PID TTY TIME CMD

36 со 6:29 /etc/cron

6423 5 0:02 -sh

6704 1 0:04 -sh

6722 1 0:12 vi paper

4430 2 0:03 -sh

6612 7 0:03 -sh

6628 7 1:13 rogue

6843 2 0:02 write dmr

6949 4 0:01 login bimmler

6952 5 0:08 pr ch1.1 ch1.2 ch1.3 ch1.4

6951 5 0:03 lpr

6959 5 0:02 ps -ag

6844 1 0:02 write rob $

Здесь *PID* — номер процесса; *TTY* — терминал, связанный с процессом (как в команде who); *TIME* — затраченное время процессора в минутах и секундах, а в конце строки - выполняемая команда. Команда *ps* — одна из тех команд, которые выполняются по-разному в различных версиях системы, так что вывод в вашей системе может иметь другой формат. Даже аргументы могут отличаться – см. в своем справочном руководстве страницу *ps*.

Процессы, подобно файлам, имеют иерархическую структуру: у каждого процесса есть родитель и могут быть потомки. Ваша копия интерпретатора *shell* была создана процессом, обслуживающим связь через терминал с системой. Когда вы запускаете команды, их процессы становятся прямыми потомками вашей копии *shell*. Если вы запускаете программу "внутри" одной из этих команд, например команду ! для выхода из редактора *ed*, то создается новый процесс-потомок, который является, таким образом, уже внуком для *shell*.

Иногда процесс выполняется столь долго, что вы уже жалеете, что запустили его. Выключите терминал и идите домой, не дожидаясь его окончания. Но если вы выключите терминал или отсоедините его от сети, то процесс будет уничтожен, даже если применен *&*. Специально для такого случая предусмотрена команда *nohup* (*no hangup* — без отбоя).

Введите *$ nohup* команда *&* и команда будет продолжать выполняться, даже если выйти из системы. Любой результат выполнения команды будет сохранен в файле, называемом *nohup.out*. После запуска программы никакая команда *nohup* уже не поможет.

Если ваш процесс требует много процессорного времени, вы можете облегчить участь тех, кто работает вместе с вами, запустив его с приоритетом ниже обычного. Это можно сделать с помощью программы *nice*: *$ nice* большая-команда *&*

Команда *nohup* автоматически вызывает *nice*, поскольку раз уж вы собираетесь выйти из системы, то можете позволить, чтобы ваша команда выполнялась дольше.

Наконец, вы можете дать указание системе запустить ваш процесс в необычное время, скажем, утром, когда все нормальные люди спят, а не работают на машине. Команда называется *at*:

$ at время

любые команды

какие угодно...

ctl-d

$

Это пример типичного использования команды *at*, но, конечно, команды можно брать и из файла:

$ at 3am << файл

$

Время можно задавать исходя из 24-часового цикла как 2130 или 12-часового как 930*pm*.

Создание среды. Одним из достоинств системы *UNIX* является то, что вы можете легко адаптировать ее по своему вкусу либо в соответствии с местными традициями программистского мира. Например, как отмечалось выше, существуют разные стандарты для символов стирания и удаления; по умолчанию используются *#* и *@*. Вы можете изменить их в любой момент с помощью команды *$ stty erase e kill k*, где *e* обозначает нужный вам символ стирания, а *k* — символ удаления. Однако задавать эти символы при каждом входе в систему — довольно нудное занятие.

На помощь здесь приходит *shell*. Если в вашем начальном каталоге есть файл *.profile*, интерпретатор будет запускать команды из него при каждом входе в систему перед выдачей первого приглашения. Поэтому можно поместить команды в *.profile* для установки требуемой среды, и они будут выполняться всякий раз при входе в систему.

Большинство пользователей первым делом помещают в свой файл *.profile* команду *$ stty erase <-*. Тут используется *<-* , чтобы сделать символ стирания видимым, но вы должны поместить в *.profile* настоящий символ "шаг назад". Команда *stty* воспринимает также обозначение *^х* в качестве *ctl-x*, поэтому тот же результат можно получить, вводя: *$ stty erase '^h'* поскольку *ctl-h* и есть шаг назад. Символ *^* ранее применялся для операции программного канала *|*, поэтому его следует экранировать с помощью кавычек. Если на вашем терминале нет возможности задать интервалы табуляции, можно добавить к строке с *stty* аргумент *-tabs* : *stty erase '^h' –tabs.*

Если у вас есть желание посмотреть, как загружена система, при входе в нее добавьте для подсчета пользователей конвейер

*who | wc -l*

Если имеется служба новостей, можно добавить команду *news*. Те, кому нравится игра *fortune*, могут добавить */usr/games/fortune*.

Спустя некоторое время вы обнаружите, что период входа в систему существенно возрос, и выберете для себя оптимальный вариант.

Некоторыми возможностями системы можно управлять с помощью так называемых *shell*-переменных, значения которых пользователь может и посмотреть, и установить. Например, строка-приглашение, обозначаемая ранее как *$*, на самом деле хранится в *shell*-переменной, называемой *PS1*, и можно присвоить ей любое значение: *PS1 = 'Yes, dear ?' Да, дорогой?*

Кавычки необходимы, поскольку в строке-приглашении есть пробелы, а в этой операции пробелы перед и после "=" не допускаются.

Интерпретатор также выделяет переменные *HOME* и *MAIL*. *HOME* представляет собой имя вашего начального каталога; переменная обычно имеет правильное значение даже без установки ее в *.profile*. Переменная *MAIL* содержит имя стандартного файла, в котором хранится ваша почта. Если вы переопределите ее для интерпретатора, то будете в случае появления новой почты получать извещение после ввода каждой команды. Эта возможность плохо реализована в *shell*. Просмотр файла после ввода каждой команды сказывается на времени работы системы. Кроме того, если вы достаточно долго работаете в редакторе, то не узнаете о новой почте, поскольку не задаете новых команд в начальном интерпретаторе *shell*. Лучшим решением была бы проверка через каждые несколько минут, а не после очередной команды. Третьим решением, хотя и не для всех доступным, может быть извещение, посылаемое самой программой *mail*, — она точно "знает", когда появится почта.

*MAIL= /usr/spool/mail/you*

В вашей системе файл для почты может быть другим; распространенным является и имя */usr/mail/you*.

Наиболее полезной переменной интерпретатора *shell*, вероятно, считается та, которая определяет, где проводится поиск команд. Вспомните, что, когда вы вводите имя команды, интерпретатор обычно вначале ищет его в текущем каталоге, затем в */bin* и далее в */usr/bin*. Эта последовательность каталогов называется путем поиска и хранится в переменной интерпретатора с именем *PATH*. Если определенный по умолчанию путь поиска вас не устраивает, то его можно изменить (опять в файле *.profile*). Например, строкой ниже к стандартному пути поиска добавляется */usr/games*:

*PATH=.:/bin:/usr/bin:/usr/games/ Один способ...*

Синтаксис может показаться несколько странным: последовательность имен каталогов разделяется двоеточием. Напоминаем, что '.' обозначает текущий каталог. Можно опустить имя '.', пустой компонент в *PATH* обозначает текущий каталог.

Другой способ установить значение *PATH* — просто добавить к предыдущему значению

*PATH=$PATH:/usr/games ... Другой способ*

Можно получить значение любой переменной интерпретатора, предварив ее имя символом $. В приведенном примере выражение *$РАТН* выбирает текущее значение, к которому добавляется новый компонент, и результат вновь присваивается *PATH*. Можно проверить это с помощью команды *echo*:

$ echo PATH is $PATH

PATH is :/bin:/usr/bin:/usr/games

$ echo $HOME Ваш начальный каталог

/usr/you

$

Если у вас есть свои собственные команды, то, возможно, вы захотите собрать их в свой каталог и добавить его к пути поиска. В таком случае переменная *PATH* может принять подобное значение:

*PATH=:$HOME/bin:/bin:/usr/bin:/usr/games*

Существует еще одна переменная, часто используемая текстовыми редакторами, – *TERM*, которая указывает тип используемого терминала. Эта информация позволяет программам более эффективно работать с экраном. Поэтому можно в *.profile* добавить, например, следующее: *TERMнm3*

Можно применять переменные и для сокращения записи. Если вам часто приходится ссылаться на каталог с длинным именем, имеет смысл добавить строку типа *d = /horribly/long/directory/name* к файлу *.profile*, чтобы использовать: *$ cd $d*

Ваши собственные переменные, например *d*, по традиции обозначаются строчными буквами, что позволяет отличить их от тех, которые, как *PATH*, использует сам интерпретатор.

Наконец, вы должны сообщить интерпретатору, что будете использовать некоторые переменные в других программах; для этого служит команда *export*: *export MAIL PATH TERM*

Подводя итоги, посмотрим, как может выглядеть типичный файл *.profile*:

$ cat .profile

stty erase '^h' -tabs

MAIL=/usr/spool/mail/you

PATH=:$HOME:/bin:/usr/bin:/usr/games

TERMнm3

b=$HOME/book

export MAIL PATH ТЕRМ b

date

who | wc -l

Неименованный канал.Пример организации неименованного канала:

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

main()

{

int fd[2];

pid\_t childpid;

pipe(fd);

if((childpid = fork()) == -1)

{

perror("fork");

exit(1);

}

if(childpid == 0)

{

/\* Порожденный процесс закрывает дескриптор входа канала \*/

close(fd[0]);

}

else

{

/\* Родительский процесс закрывает дексриптор выхода канала \*/

close(fd[1]);

}

}

В данном примере первый процесс создает массив *fd* для хранения дескрипторов входа и выхода программного канала, который создается далее функцией *pipe*. Затем системным вызовом *fork* порождается дочерний процесс, который закрывает вход канала. Родительский процесс закрывает выход канала. Теперь процессы могут использовать созданный канал для односторонней передачи данных (от родительского процесса дочернему).

# ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

1. Просмотреть список активных процессов. Объяснить результаты.
2. Выполнить команды с использованием конструкций:

* переназначение выходного канала (на примере *cat, ls, ps*…);
* переназначение канала сообщений (на примере команды с заведомо ошибочным аргументом);
* переназначение входного канала;
* конвеер (на примере *ls, ps* и *more, less, grep*).

1. Войти в систему со второй виртуальной консоли под тем же именем.
2. Запустить в одной из консолей интерактивный процесс (*mc, vi, less*), переключиться в другую виртуальную консоль и уничтожить этот процесс командой *kill*. Объяснить результаты.
3. В этом разделе приведены задания, в каждом из которых требуется написать программу или несколько программ на языке C. Студенту предлагается выполнить одно из предложенных заданий и продемонстрировать разработанную программу и ее исходные коды преподавателю.
4. Прототип командного интерпретатора с функциями:

* запуск процессов в синхронном режиме;
* завершение процессов по сигналу с клавиатуры (*Ctrl+C*).

В следующем примере продемонстрирован запуск прототипа командного интерпретатора, выполнение из его командной строки команды *cat*, затем прекращение ее работы комбинацией клавиш *Ctrl+C*.

$ ./proto-sh

> cat

^C

> exit

$

1. Упрощенная модель сетевого сервера, работающего по следующей схеме:
   * вначале запускается управляющий процесс *P*, ожидающий подключения сетевых клиентов на определенный порт;
   * при подключении клиента вначале в асинхронном режиме запускается процесс *P1*, который порождает копию – дочерний процесс *С1*;
   * процессы обмениваются информацией с помощью сигналов, работают безостановочно (в цикле), все свои действия записывают в журнал;
   * при завершении процесса *P1* (по сигналу), процесс *C1* порождает новую копию *C2*, и таким образом становится родительским процессом (*P2*).
   * при завершении дочернего процесса (*С1, С2, ..., Сn*), завершается также и родительский процесс. а процесс P создает обслуживающий процесс *С* (для каждого клиента), взаимодействующий с клиентом любым образом (например, просто передает клиенту свой *PID*);
   * процесс *C* завершает свою работу при отключении соответствующего клиента;
   * при завершении процесса *P* (по сигналу) завершаются все процессы *C*.
2. Задача «акробаты»:
   * вначале в асинхронном режиме запускается процесс *P1*, который порождает копию — дочерний процесс *С1*;
   * процессы обмениваются информацией с помощью сигналов, работают безостановочно (в цикле), все свои действия записывают в журнал;
   * при завершении процесса *P1* (по сигналу), процесс *C1* порождает новую копию *C2*, и таким образом становится родительским процессом (*P2*).
   * при завершении дочернего процесса (*С1, С2, ..., Сn*), завершается также и родительский процесс.
3. Программа-конвейер:
   * в качестве параметров передаются имена двух исполняемых файлов *f1* и *f2*;
   * основной процесс *P* создает неименованный канал и порождает процесс *C*;
   * стандартный ввод *P* связывается с выходом канала, а стандартный вывод *C* связывается со входом канала;
   * процесс *P* загружает в себя *f2*, а процесс *C* загружает *f1*;
4. Аналог утилиты *nohup* с выводом перехваченных сигналов. Пример запуска команды *sleep* с последующей проверкой игнорирования процессом сигнала «*HUP*»:

$ ./nohup sleep 3600

[21305]

$ kill -HUP 21305

$ ps -p 21305 | grep sleep

21305 pts/0 S 0:00 sleep 3600

1. Программа «будильник». Позволяет запустить программу с заданным значением времени (передается как значение ключа «-t», по умолчанию составляет 5 минут), при наступлении которого программа выполнить должна одно из действий, заданных аргументами:
   * ключ «*-c*» со значением *cmd* — запустить команду *cmd*;
   * ключ «*-p*» со значением *file* — проиграть звуковой файл *file*;
   * ключ «*-k*» со значением *pid* — отправить сигнал *USR1* процессу с идентификатором *pid*.

Сама программа при получении сигнала должна выдать текущее время с момента запуска. Пример запуска двух процессов программы, один их которых проиграет звуковой файл спустя минуту после запуска, а второй через 5 секунд после запуска отправляет сигнал *USR1* первому процессу:

$ ./alarm -t 00:01:00 -p "alarm.mp3" & [21305]

$ ./alarm -t 00:00:05 -k 21305

Время с момента запуска: 00:00:33

1. Программа «ящерица». При получении сигнала *USR1* процесс порождает дочерний процесс-копию и сам завершается. При получении сигнала *USR2* процесс выводит свой номер (по порядку от первого запущенного процесса). Пример:

$ ./lizard &

[21305]

$ kill -USR1 21305

$ ps | grep lizard

21307 pts/1 00:00:00 lizard

$ kill -USR2 21307 8

Порядковый номер процесса: 2

[1]+ Готово ./lizard

1. Программа «контейнер». Позволяет запустить заданную команду с ограничением по времени выполнения. После запуска программа ожидает завершения процесса команды и, если по истечении заданного времени команда не завершилась, отправляет сигнал завершения процессу («*TERM*»). Время задается в секундах с помощью ключа «*-t*». Пример запуска команды «sleep 100» с ограничением времени выполнения 5 секунд:

$ ./limit -t 5 sleep 100

1. Распределенный подсчет размера каталога. При запуске программе передается путь к каталогу в качестве параметра. Запущенный процесс получает список файлов в этом каталоге и для каждого подкаталога запускает свою копию с соответствующим параметром. Не дожидаясь завершения порожденных процессов программа суммирует размеры файлов из списка, затем к этой сумме прибавляет числа, которые выведут дочерние процессы по завершении. Полученный суммарный размер каталога процесс выводит на стандартный вывод и завершается. Пример запуска программы для каталога */usr* с правами суперпользователя:

$ sudo ./size /usr

1035468

Пример запуска программы для каталога /usr с правами суперпользователя в асинхронном режиме с промежуточным контролем запускаемых процессов:

$ sudo ./size /usr &

$ ps x | grep

21305 pts/1 D 0:03 ./size /usr

21306 pts/1 D 0:02 ./size /usr/home

21307 pts/1 D 0:01 ./size /usr/lib

1. Программа ввода с поддержкой множественного доступа. При запуске программа проверяет, существует ли сегмент общей (разделяемой) памяти с предопределенным идентификатором. Если не существует, то создает, иначе — получает доступ к существующему сегменту и выводит содержимое общей памяти на стандартный вывод. Любой из запущенных процессов считывает стандартный ввод (пока не получит символ окончания файла) и записывает полученный ввод в общую память. Каждые 200 миллисекунд запущенные процессы проверяют состояние общий памяти; в случае увеличения размера содержимого процессы выводят добавленную часть на стандартный вывод. Последний процесс при выходе уничтожает освобождает общую память. Таким образом, все копии процессов всегда должны отображать один и тот же текст, независимо от того в каком процессе введен этот текст.

Пример запуска двух процессов в одном терминале с передачей на стандартный ввод первого процесса тестовой строки:

$ echo Test | ./shared-input

[21305]

$ ./shared-input

Test

# Контрольные вопросы

1. Что такое процесс?
2. Что в себя включает процесс?
3. Какие основные атрибуты есть у процесса?
4. Какие типы процессов существуют?
5. Какие процессы называют системными?
6. Какие процессы называют «демонами»?
7. Какие процессы называют прикладными?
8. Статическое и динамическое связывание.
9. Потоки ввода-вывода.
10. На что указывает амперсанд (&) в конце командной строки?
11. Для чего используется stty?
12. Переменные интерпретатора HOME, MAIL, PATH и TERM.
13. Какова последовательность действий системного вызова *fork*?
14. Что наследует порожденный процесс от родительского?
15. Какова последовательность действий системного вызова *exec*?
16. Какое действие осуществляет системный вызов *kill*?
17. Какой системный вызов используются для установки собственного обработчика сигнала процессом?
18. Как организуется неименованный канал между процессами?
19. В чем преимущества и недостатки обмена данными через разделяемую память в сравнении с использованием неименованных каналов?
20. Что такое разделяемая память?

# Список рекомендуемой литературы

## Основная литература

1. Курячий Г.В. Операционная система UNIX: учебное пособие [Электронный ресурс] / Курячий Г.В.— М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. – 258c. – Режим доступа: http://www.iprbookshop.ru/52199.
2. Администрирование ОС Unix [Электронный ресурс] / – 2-е изд. –М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. – 303 c. – Режим доступа: http://www.iprbookshop.ru/73659.html.

## Дополнительная литература

1. Командная строка UNIX: практикум [Электронный ресурс] — М.: Московский государственный строительный университет, ЭБС АСВ, 2013. – 44c. – Режим доступа: http://www.iprbookshop.ru/23729.
2. Мошков, М.Е. Введение в системное администрирование Unix [Электронный ресурс] / М.Е. Мошков. – М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. – 208 c. – Режим доступа: http://www.iprbookshop.ru/73672.html.

# ПРИЛОЖЕНИE

Таблица

Некоторые полезные команды

|  |  |
| --- | --- |
| **Команда** | **Пояснение** |
| chmod +x | Установка разрешения на выполнение |
| date +"%Y-%m-%d %H:%M:%S" | Вывод даты в формате ГГГГ-ММ-ДД ЧЧ:ММ:СС |
| expr 2 + 2 | Вычисление выражения 2 + 2 |
| expr 123 % 2 | Получение остатка от деления 123 на 2 |
| grep | Поиск строк по шаблону |
| grep -n | Вывод номеров строк с совпадениями |
| grep -l | Вывод списка файлов с совпадениями |
| grep -5 source.txt | Вывод первых пяти строк |
| read var | Ввод значения переменной пользователем |
| cut -c 1,3-5 | Вывод 1-го, 3-го, 4-го и 5-го символов из стандартного ввода |
| paste f1 f2 | Объединение файлов по колонкам |
| sed 's/txt/TXT/g' | Поиск и замена подстрок |
| sed 's/txt/d' | Поиск и удаление подстрок |
| sort | Сортировка строк |
| sort +1 | Пропуск первого поля при сортировке |
| sort -n | Сортировка чисел |
| sort -r | Сортировка в обратном порядке |
| tail -5 source.txt | Вывод последних пяти строк |
| tail +5 source.txt | Вывод всех строк после пятой |
| tr '[A-Z]' '[a-z]' | Преобразование в нижний регистр букв |
| tr '[a-z]' '[A-Z]' | Преобразование в верхний регистр букв |
| tr -d '\_' | Удаление символов подчеркивания |
| uniq | Поиск уникальных строк |
| wc -w | Подсчет количества слов |
| wc -l | Подсчет количества строк |

UNIX-системы. Процессы в UNIX: методические указания к выполнению лабораторной работы № 6 для студентов очной формы обучения по направлениям подготовки 09.03.01 «Информатика и вычислительная техника», 09.03.04 «Программная инженерия», 02.03.03 «Математическое обеспечение и администрирование информационных систем»

ТРУБАКОВ ЕВГЕНИЙ ОЛЕГОВИЧ

Научный редактор Д. А. Коростелев

Компьютерный набор Е.О. Трубаков

Иллюстрации Е.О. Трубаков

Подписано в печать \_\_.\_\_.\_\_. Усл.печ.л. 1,52 Уч.-изд.л. 1,52

Брянский государственный технический университет

241035, Брянск, бульвар 50 лет Октября, 7 БГТУ

Кафедра «Информатика и программное обеспечение», тел. 56-09-84