Раздел 1

Среда исполнения

1. Вывод различных атрибутов процесса в соответствии с указанными опциями

═══════ Напишите программу, которая будет обрабатывать опции, приведенные ниже. Опции должны быть обработаны в соответствии с порядком своего появления справа налево. Одной и той же опции разрешено появляться несколько раз. Используйте **`getopt(3C)`** для определения имеющихся опций. *Сначала пусть ваша программа обрабатывает только некоторые опции*. Затем добавьте еще, до тех пор, пока все требуемые опции не будут обрабатываться. Вы можете скопировать воспользоваться программой **`getopt_ex.c`** и изменить ее.

- **`-i`** Печатает реальные и эффективные идентификаторы пользователя и группы.
- **`-s`** Процесс становится лидером группы. *Подсказка:* смотриии **`setpgid(2)`**.
- **`-p`** Печатает иидентификаторы процесса, процесса-родителя и группы процессов.
- **`-u`** Печатает значение **`ulimit`**
- **`-Unew_ulimit`** Изменяет значение **`ulimit`**. *Подсказка:* смотри **`atol(3C)`** на странице руководства **`strtol(3C)`**
- **`-c`** Печатает размер в байтах core-файла, который может быть создан.
- **`-Csize`** Изменяет размер core-файла
- **`-d`** Печатает текущую рабочую директорию
- **`-v`** Распечатывает переменные среды и их значения
- **`-Vname=value`** Вносит новую переменную в среду или изменяет значение существующей переменной.

Проверьте вашу программу на различных списках аргументов, в том числе:

- Нет аргументов
- Недопустимую опцию.
- Опции, разделенные знаком минус.
- Неудачное значение для **`U`**.

2. Время в Калифорнии

═══════ Измените программу **`ex_time.c`**, чтобы она выводила дату и время в Калифорнии (Pacific Standard Time, PST). *Подсказка:* Если время UTC 20 часов, то в Калифорнии 12 часов.

3. Установка идентификатора пользователя для доступа к

файлу

═══ Создайте файл данных, который может писать и читать только владелец (это можно сделать командой shell **`chmod 600 file`**) и напишите программу, которая

1. Печатает реальный и эффективный идентификаторы пользователя.
2. Открывает файл с помощью **`fopen(3)`**. Если **`fopen()`** завершился успешно, файл должен быть закрыт с помощью **`fclose(3)`**. Напечатайте сообщение об ошибке, используя **`perror(3C)`**, если файл не удалось открыть.
3. Сделайте, чтобы реальный и эффективный идентификаторы пользователя совпадали. *Подсказка:* **`setuid(2)`**
4. Повторите перввые два шага.

Проверьте работу вашей программы.

1. Исполните программу и посмотрите вывод
2. Сделайте программу доступной для запуска членам вашей группы и пусть ваши одногруппники исполнят программу.
3. Командой **`chmod u+s prog`** установите бит *установки идентификатора пользователя* и пусть ваши одногруппники опять исполнят эту программу.

Управление памятью

4. Список строк

═══ Напишите программу, которая вставляет строки, введенные с клавиатуры, в список. Память под узлы списка выделяйте динамически с использованием **`malloc(3)`**. Ввод завершается, когда в начале строки вводится точка (.). Затем все строки из списка выводятся на экран.

═══ Подсказка: Объявите массив символов размера, достаточного чтобы вместить самую длинную введенную строку. Используйте **`gets(3)`**, чтобы прочитать строку, и **`strlen(3)`**, чтобы определить ее длину. Помните, что **`strlen(3)`** не считает нулевой символ, завершающий строку. После определения длины строки, выделите блок памяти нужного размера и внесите новый указатель в список.

Системные вызовы ввода/вывода

5. Таблица поиска строк в текстовом файле.

═══ Написать программу, которая анализирует текстовый файл, созданный текстовым редактором, таким как **`ed(1)`** или **`vi(1)`**. После запроса, который предлагает ввести номер строки, с использованием **`printf(3)`** программа печатает соответствующую строку текста. Ввод нулевого номера

завершает работу программы. Используйте **`open(2)`**, **`read(2)`**, **`lseek(2)`** и **`close(2)`** для ввода/вывода. Постройте таблицу отступов в файле и длин строк для каждой строки файла. Как только эта таблица построена, позиционируйтесь на начало заданной строки и прочтите точную длину строки. Предполагайте, что файл не длиннее сотни строк.
═══ Подсказка: Выберите или создайте текстовый файл с короткими строками. Помните, что первая строка начинается с нулевого отступа в файле. Найдите каждый символ перевода строки, запишите его позицию; в программе следует использовать вызов lseek(fd, 0L, 1). Для отладки распечатайте эту таблицу и сравните с таблицей, полученной вручную. Как только таблицы начнут совпадать, можно приступать к запросу номера строки.

6. Таблица поиска строк в текстовом файле.

═══ Измените программу так, чтобы пользователю отводилось 5 секунд на ввод номера строки. Если пользователь не успевает, программа должна распечатать все содержимое файла и завершиться. Если же пользователь успел в течение пяти секунд ввести номер строки, то программа должна работать как в предыдущей задаче.

7. Таблица поиска строк в текстовом файле 2.

═══ Измените предыдущую программу так, чтобы использовалось отображение файла в память взамен использования **`read(2)`**, **`lseek(2)`** и **`write(2)`**.

Захват файлов и записей

8. Защищенный текстовый редактор

═══ Напишите программу, которая захватывает весь файл перед вызовом редактора. Это защитит файл с правами доступа группы на изменение. После того как программа захватит файл, вызовите свой любимый редактор с помощью библиотечной функции **`system(3)`**. Попытайтесь в вашей программе использовать допустимое захватывание. Когда ваша программа захватит файл, попросите одногруппника исполнить вашу программу. Объясните результат. Потом попытайтесь использовать обязательное захватывание. Объясните результат.

Создание процессов и исполнение программ. Управление процессами

9. Создание двух процессов

═══ Напишите программу, которая создает подпроцесс. Этот

подпроцесс должен исполнить **`cat(1)`** длинного файла. Родитель должен вызвать **`printf(3)`** и распечатать какой-либо текст. После выполнения первой части задания модифицируйте программу так, чтобы последняя строка, распечатанная родителем, выводилась после завершения порожденного процесса. Используйте **`wait(2)`**, **`waitid(2)`** или **`waitpid(3)`**.

10. Код завершения команды

═══════ Напишите программу, которая запускает команду, заданную в качестве первого аргумента, в виде порожденного процесса. Все остальные аргументы программы передаются этой команде. Затем программа должна дождаться завершения порожденного процесса и распечатать его код завершения.

11. Функция execvpe()

═══════ Напишите функцию execvpe(), которая работает как **`execvp(2)`**, но позволяет изменять среду исполнения, как **`execve(2)`**.
═══════ Совет: используйте внешнюю переменную **`environ`**.

12. Командный интерпретатор shell (задание ≟1 по shell)

═══════ В этом упражнении вы должны разработать упрощенную версию командного интерпретатора системы UNIX - shell. В дальнейшем этот shell будет улучшен добавлением программных каналов и обработки сигналов. Для вас заранее написаны некоторые функции этой программы. В файле **shell.c** содержится скелет функции main() с комментариями, говорящими, где следует вставить ваш код. Функция main() вызывает функцию prompline() из файла **promptline.c**, которая читает строку из стандартного ввода. Затем main() вызывает parseline() из файла **parseline.c**, которая проверяет ввод на отсутствие синтаксических ошибок. Для каждой команды в строке, parseline() формирует дескриптор команды типа struct command, определенного в shell.h Существует глобальный массив таких структур, называемый cmds[] и объявленный в shell.c. Дескриптор команды состоит из двух полей. Первое поле - массив указателей, называемый cmdargs[]. Функция parseline() инициализирует этот массив указателями на строки - аргументы. Каждый аргумент завершается нулевым байтом. Во входной строке аргументы разделяются пробелами. Кроме того, parseline() помещает после последнего аргумента в списке cmdargs[] нулевой указатель. Команды могут разделяться переводом строки, точкой с запятой, амперсендом (&) или символами <, >, >> или |. Другие глобальные переменные, объявленные в shell.c определяют, должна ли команда быть запущена в основном (foreground) или фоновом (background) режиме, а также для перенаправления ввода/вывода из/в файл. Эти переменные (bkgrnd, infile, outfile и appfile) устанавливаются в функции parseline(). Если bkgrnd ненулевой, команда должна быть запущена в фоновом режиме, то есть ваш shell не должен

ожидать завершения соответствующего процесса. Если infile ненулевой, он указывает на имя файла, из которого должен быть получен стандартный ввод первой команды в строке, т.е. символ <. Аналогично, если outfile ненулевой, он указывает на имя файла, куда следует перенаправить вывод последней команды в строке, т.е. символ >. Наконец, если appfile ненулевой, это имя файла, к содержимому которого следует добавить вывод последней команды в строке, т.е. символ >>. В следующих упражнениях вы будете модифицировать shell.c. Каждое последующее упражнение является улучшением предыдущего. Вы должны использовать следующую версию вызова **`exec(2)`**:
execvp(cmds[i].cmdargs[0], cmds[i].cmdargs);

13. Исполнение команд (задание ⊥2 по shell)

═══════ Модифицируйте shell.c для исполнения каждой команды в массиве cmds[] как подпроцесса. Во время исполнения этого подпроцесса, ваша программа должна ждать его завершения. Если программа, заданная во входной строке, не может быть найдена, shell должен распечатать сообщение об ошибке и продолжить исполнение. Предполагается, что пользователь набирает только простые команды, т.е. только одна команда на строке, без метасимволов, перенаправления ввода/вывода, конвейеров и символа &.

14. Исполнение в фоновом режиме (задание ⊥3 по shell)

═══════ Измените вашу программу так, чтобы позволить исполнение команд в фоновом режиме, обозначаемое символом & в конце командной строки. Программа должна выводить идентификатор фонового процесса на стандартный вывод.

15. Перенаправление ввода/вывода (задание ⊥4 по shell)

═══════ Измените вашу программу так, чтобы позволить перенаправление ввода/вывода. Заметьте, что перенаправление ввода ("< filename") используется только с первой командой в строке, а вывода ("> filename" или ">> filename") - только с последней.
═══════ Совет: используйте **`dup(2)`**.

Управление терминальным вводом/выводом

16. Ответ без ввода новой строки

═══════ Напишите программу, которая печатает вопрос и требует односимвольного ответа. Измените атрибуты вашего терминала так, чтобы пользователю не нужно было вводить новую строку после ответа.

17. Строчный редактор

Многие программы, принимающие ввод с терминала, позволяют редактировать строку перед использованием. Напишите программу, которая выключает эхо и каноническую обработку, таким образом выключив и обработку символа забоя. Ваша программа должна получать ввод с клавиатуры и показывать его на терминале в соответствии со следующими правилами:

a. Каждый введенный символ должен немедленно появляться на дисплее.
b. Когда вводится символ ERASE, стирается последний символ в текущей строке.
c. Когда вводится символ KILL, стираются все символы в текущей строке.
d. Когда вводится CTRL-W, стирается последнее слово в текущей строке, вместе со всеми следующими за ним пробелами.
e. Программа завершается, когда введен CTRL-D и курсор находится в начале строки.
f. Все непечатаемые символы, кроме перечисленных выше, должны издавать звуковой сигнал, выводя на терминал символ CTRL-G.
g. Длина строки ограничена 40 символами. Если какое-то слово пересекает 40-й столбец, это слово должно быть помещено в начало следующей строки.

Управление файлами

18. Листинг каталога

Напишите программу - аналог команды **`ls -ld`**. Для каждого своего аргумента эта команда должна распечатывать:

- Биты состояния файла в воспринимаемой человеком форме:
  - **`d`** если файл является каталогом
  - **`-`** если файл является обычным файлом
  - **`?`** во всех остальных случаях
- Три группы символов, соответствующие правам доступа для хозяина, группы и всех остальных:
  - **`r`** если файл доступен для чтения, иначе √
  - **`w`** если файл доступен для записи, иначе √
  - **`x`** если файл доступен для исполнения, иначе **-**
- Количество связей файла
  - Имена собственника и группы файла (совет - используйте **`getpwuid`** и **`getgrgid`**).
  - Если файл является обычным файлом, его размер. Иначе оставьте это поле пустым.
  - Дату модификации файла (используйте **`ctime`**).
  - Имя файла (если было задано имя с путем, нужно распечатать только имя).

Желательно, чтобы поля имели постоянную ширину, т.е. чтобы листинг имел

вид таблицы.
═══ Совет - используйте **`printf`**.

Управление каталогами

19. Шаблоны имен файлов

═══ Напишите программу, которая приглашает пользователя ввести шаблон имени файла, аналогичный тому, который используется в shell. Синтаксис шаблона таков:

- **`*`** соответствует последовательности любых символов кроме **`/`**, имеющей любую длину; возможно - пустой последовательности.
- **`?`** соответствует любому одному символу.
- **`/`** не может встречаться.
- любой другой символ соответствует самому себе.

Символы **`*`** и **`?`** в шаблоне могут встречаться в любом количестве и в любом порядке.

═══ Затем программа должна найти и распечатать имена всех файлов в текущем каталоге, соответствующих шаблону. Если таких файлов нет, программа должна распечатать сам шаблон.
═══ Совет: используйте **`readdir`**, чтобы считать все имена файлов в текущем каталоге, и выберите из них соответствующие шаблону.

20. Шаблоны имен файлов (2)

═══ Измените предыдущую программу так, чтобы в шаблоне могли встречаться символы **`/`**. При этом программа должна распечатывать все файлы, путевые имена которых соответствуют шаблону. Так, шаблону **`*/*`** соответствуют все файлы во всех подкаталогах текущего каталога.

Раздел 2

Сигналы

21. Пищалка

═══ Напишите программу, которая входит в бесконечный цикл и издает звуковой сигнал на вашем терминале каждый раз, когда вы нажимаете *DELETE*. При получении SIGQUIT, она должна вывести сообщение, говорящее, сколько раз прозвучал сигнал, и завершиться.

22. Мультиплексирование ввода

═══ Напишите программу, которая читает из нескольких файлов по очереди, т.е. после чтения строки из одного файла, читается строка из следующего и т.д. Если в течение TIME_OUT секунд ничего не было прочитано, берется следующий файл.

═══ Программа получает в качестве аргументов имена одного или нескольких файлов, из которых она будет читать. Обычно это терминальные файлы (т.е. /dev/ttynn), но могут быть файлы и других типов. (`read(2)` с нетерминального устройства может прочитать несколько строк, в зависимости от количества требуемых байтов и длины этих строк.) Если в одном из файлов достигнут конец файла, из него больше не читают. Когда конец файла достигнут во всех файлах, программа завершается. Проверьте вашу программу так:

═══ `$ multiplex /dev/tty 'tty'`

23. Защита от сигналов, посылаемых с терминала (задание ⊥5 по shell)

═══ Измените исходный текст программы shell так, чтобы она не завершалась, когда вы генерируете сигнал SIGINT. Вместо этого должен завершаться процесс первого плана, а ваш shell должен немедленно выдавать приглашение. Кроме того, предохраните команды, исполняемые в фоновом режиме, от прерывания сигналами SIGINT и SIGQUIT. (Если вы не имеете собственной версии этой программы, вы можете посмотреть исходные тексты заготовки в файлах **shell.c**, **parseline.c**, **promptline.c** и **shell.h**. Откомпилировав их, вы получите простой командный интерпретатор, способный выполнять программы в виде порожденных процессов, запускать фоновые процессы и перенаправлять ввод/вывод.)

24. Простое управление заданиями (задание ⊥6 по shell)

═══ Если вы реализовали защиту от сигналов, модифицируйте ваш интерпретатор shell, так чтобы обеспечить следующие возможности для управления заданиями: SIGTSTP, посланный с клавиатуры (*CTRL-Z* по умолчанию), должен заставить основную программу перейти в фоновый режим и возобновить исполнение. PID переведенного на фон процесса будет выведен на stderr, и командный интерпретатор выдаст приглашение для следующей команды.

═══ Совет: сделайте каждый порожденный процесс лидером группы процессов и явным образом переводите его на первый план. Когда порожденный процесс первого плана завершается или останавливается, переводите на первый план интерпретатор.

Программные каналы

25. Связь через программный канал

Напишите программу, которая создает два подпроцесса, взаимодействующих через программный канал. Первый процесс выдает в канал текст, состоящий из символов верхнего и нижнего регистров. Второй процесс переводит все символы в верхний регистр, и выводит полученный текст на терминал. Подсказка: см. **`toupper(3)`**.

26. Связь с использованием функций стандартной библиотеки

Используйте стандартные библиотечные функции **`popen(3)`** и **`pclose(3)`** для выполнения тех же операций, что и в предыдущем упражнении.

27. Подсчет пустых строк в файле

Напишите программу, которая подсчитывает пустые строки в файле, используя команду **`wc(1)`**.

28. Генератор случайных чисел

Напишите программу, которая генерирует сортированный список из ста случайных чисел в диапазоне от 0 до 99. Распечатайте числа по десять в строке. Используйте **`p2open(3)`**, чтобы запустить **`sort(1)`** и **`rand(3)`** и **`srand(3)`** для генерации случайных чисел.

29. Конвейеры (задание ⊥7 по shell)

Измените ваш командный интерпретатор так, чтобы он позволял создавать конвейеры. Если вы добавили управление заданиями в упражнениях Раздела 1, вы можете модифицировать программу так, чтобы все процессы в конвейере принадлежали к одной группе. Тогда, например, SIGINT мог бы прервать все процессы в конвейере первого плана.

Очереди сообщений

30. Очереди сообщений

Напишите две программы, одна из которых шлет сообщения другой. Получатель распечатывает содержимое сообщения, тип и значение, возвращенное **`msgrcv(2)`**. Напишите программы так, чтобы они не исполнялись родственными процессами. Сделайте программы как можно проще. Для этого упражнения вам не нужна никакая проверка ошибок. Используйте численную форму вашего идентификатора пользователя в качестве ключа. Если позволит время, вы можете установить права доступа для очереди так, чтобы ваши

одногруппники, зная ключ очереди, могли читать из нее или писать в нее. К обоим программам должен быть добавлен цикл, чтобы иметь дело с многими сообщениями. Если вы будете разделять идентификатор очереди с другими пользователями, вы должны сказать друг другу ваши идентификаторы пользователя и использовать их как типы сообщений.
═══════ Замечание: `ipcs -q` распечатывает количество сообщений в очереди и их общий размер.

31. Рассылка одного сообщения нескольким процессам

═══════ Напишите две программы: отправитель и получатель. Отправитель принимает текст с клавиатуры и рассылает сообщения нескольким копиям программы-получателя. Получатель сообщения распечатывает свое имя программы (argv[0]) и содержимое сообщения. Отправитель продолжает работу, пока не встретит конец файла. Тогда он рассылает всем получателям сообщение-ограничитель. Как только получатель заканчивает свою очередь сообщений, он отправляет сообщение отправителю. После получения такого подтверждения от всех получателей, отправитель удаляет очередь.

32. Несколько процессов, сообщающих состояние процессу-мастеру

═══════ Напишите две программы, общающиеся через очередь. Одна из программ - мастер - будет получать сообщения, генерируемые другими программами - отправителями. Эти сообщения могут изображать состояние или степень готовности отправителя. При получении, мастер распечатывает сообщение и какую-либо идентификацию его отправителя. Перед завершением, каждый отправитель посылает сообщение-ограничитель. После получения таких ограничителей от всех отправителей, мастер удаляет очередь и завершается.

Семафоры

33. Моделирование производственной линии

═══════ Вы должны смоделировать производственную линию, производящую Виджеты. Каждый Виджет состоит из Детали С и Модуля 1. Модуль 1 состоит из Детали А и Детали В. Изготовление Детали А требует 2 секунды, Детали В - 3 секунды и Детали С - 4 секунды.
═══════ Подсказка: элементы производственной линии должны быть представлены процессами. Используйте набор семафоров, по одному для каждой детали и модуля. Как только деталь или модуль произведены, добавляйте единицу к соответствующему семафору. Когда объект используется в на следующем этапе, вычитайте из того же семафора 1. Используйте

`sleep(3)` для моделирования частоты, с которой производятся детали.

Разделяемая память

34. Производитель и Потребитель

═══════ Напишите две программы, Производитель и Потребитель, такие что Производитель заполняет буфер в разделяемой памяти, а Потребитель читает его. Производитель должен помещать новые данные в буфер только после того, как Потребитель прочитает его.
Совет: Для синхронизации можно использовать два семафора (эффективнее всего - набор из двух семафоров). Один из семафоров нужно ассоциировать с записью новых данных в буфер. Другой должен быть ассоциирован с чтением данных Потребителем.

35. Кольцевая очередь в разделяемой памяти

═══════ Реализуйте кольцевую очередь фиксированной длины в разделяемой памяти. Программа - производитель читает с клавиатуры и помещает прочитанный текст в виде записи в конец очереди. Потребитель берет записи из начала очереди. При обнаружении некоторого условия окончания, производитель должен перед выходом поместить в очередь запись-ограничитель. Когда потребитель получит такую запись, он также завершится.
Советы:

A. Очередь должна представлять собой массив записей. Таким образом, размер каждой записи не может превосходить фиксированную максимальную величину.
B. добавление записи в хвост очереди (псевдокод)
   - tail <- tail +1
   - if tail = QUE_SIZE, tail <- 0
   - if tail = head, queue full condition
   - else queue[tail] <- item
C. получение записи из начала очереди (псевдокод)
   - if tail = head, queue empty condition
   - head <- head +1
   - if head = QUE_SIZE, head <- 0
   - else item <- queue[head]

36. Один производитель и несколько потребителей буфера

═══════ Напишите программу-производитель, которая помещает текст в буфер, размещенный в сегменте разделяемой памяти. Напишите

программу-потребитель, которая будет читать из этого буфера. Может существовать несколько копий потребителя. Производитель может обновлять буфер только после того, как все потребители считали его содержимое.
═══════ Совет: Эта программа похожа на Задание 34. Нужны два семафора, но их значения будут меняться в диапазоне от 0 до количества потребителей.

37. Несколько читающих процессов и один эксклюзивный процесс записи в разделяемую память

═══════ Предположим, что исполняется n процессов. Часть из них читает, а остальные пишут в разделяемый сегмент памяти. Несколько читающих процессов могут работать с буфером одновременно. Если один из пишущих процессов выполняет запись, все остальные процессы должны ждать. Кроме того, если пишущий процесс хочет обновить данные, он должен ждать, пока все читающие процессы закончатся. Для обеспечения такого взаимного исключения надо использовать семафоры.
═══════ Замечание: В решении приведена одна программа, которая иногда читает, а иногда пишет.
═══════ Совет: Один из способов решения состоит в использовании набора из трех семафоров со следующими значениями:

- индекс 0 счетчик процессов, выполняющих чтение
- индекс 1 двоичный семафор, гарантирующий, что только один процесс может писать в буфер
- индекс 2 используется для блокировки чтения на время записи или ожидания записи