# Лабораторная работа 9. Разработка программ для управления процессами в Linux

**Аннотация:** Цель работы: получить представление об особенностях компиляции программ на языке программирования C в Linux, изучить основные системные вызовы и функции, используемые для управления процессами в Linux.

**Задание 1**. Изучите особенности компиляции программ на языке C/C++ в Linux.

Ход выполнения:

1. Прочитайте вводную информацию о компиляторе gcc.

Компилятор превращает код программы в объектный код понятный компьютеру. Компиляторов под Linux существует много, практически для каждого распространенного языка. Большинство самых востребованных компиляторов входит в набор GNU Compiler Collection, известных под названием GCC ([http://gcc.gnu.org](http://gcc.gnu.org/)).

Изначально аббревиатура GCC имела смысл GNU C Compiler, но в апреле 1999 года сообщество GNU решило взять на себя более сложную миссию и начать создание компиляторов для новых языков с новыми методами оптимизации, поддержкой новых платформ, улучшенных runtime-библиотек и других изменений (<http://gcc.gnu.org/gccmission.html>). Поэтому сегодня коллекция содержит в себе компиляторы для языков C, C++, Objective C, Chill, Fortran, Ada и Java, как библиотеки для этих языков (libstdc++, libgcj, ...).

Компиляция программ производится командой:

gcc <имя\_файла>

После этого, если процесс компиляции пройдет успешно, то вы получите загружаемый файл a.out, запустить который можно командой:

./a.out

2. В Linux создайте пустой документ с именем 12\_1.c. В созданном документе напишите следующую программу:

#include <stdio.h>

int main()

{

printf("Первая программа для Linux.\n");

return 0;

}

3. Откомпилируйте программу:

gcc 12\_1.c

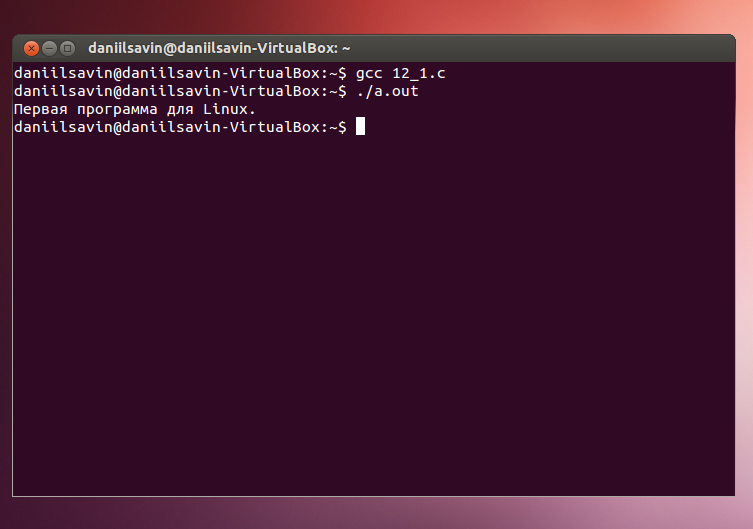
Если программа была написана без ошибок, то компилятор создаст исполняемый файл с именем **a.out**. Изменить имя создаваемого исполняемого файла можно, задав его с помощью опции *-o*:

gcc <имя\_исходного\_файла> -o <имя\_исполняемого\_файла>

3. Запустите программу:

./a.out

4. Проверьте правильность работы программы.



**Задание 2**. Изучите особенности компиляции нескольких файлов.

Ход выполнения:

1. Прочитайте информацию о необходимости компиляции нескольких файлов.

Обычно простые программы состоят из одного исходного файла. При работе с большой программой может возникнуть несколько достаточно серьезных проблем:

* Файл, становясь большим, увеличивает время компиляции, и малейшие изменения в исходном тексте автоматически вынуждают тратить время программиста на перекомпиляцию программы.
* Если над программой работает много человек, то практически невозможно отследить сделанные изменения.
* Процесс правки и само ориентирование при большом исходном тексте становится сложным и поиск небольшой ошибки может повлечь за собой вынужденное "изучение" кода заново.

При разработке программ рекомендуется их разбивать на куски, которые функционально ограничены и закончены.

Для того, чтобы вынести функцию или переменную в отдельный файл надо перед ней поставить зарезервированное слово *extern*.

2. Создайте программу из нескольких файлов. Сначала создайте главную программу, в которой будут две внешние процедуры. Назовем этот файл main.c:

#include <stdio.h>

// описываем функцию f1() как внешнюю

extern int f1();

// описываем функцию f2() как внешнюю

extern int f2();

int main()

{

int n1, n2;

n1 = f1();

n2 = f2();

printf("f1() = %d\n",n1);

printf("f2() = %d\n",n2);

return 0;

}

3. Создайте два файла, каждый из которых будет содержать полное определение внешней функции из главной программы. Файлы назовем f1.c и f2.c:

// файл f1.c

int f1()

{

return 2;

}

// файл f2.c

int f2()

{

return 10;

}

4. Создайте два файла, каждый из которых будет содержать полное определение внешней функции из главной программы. Файлы назовите f1.c и f2.c:

// файл f1.c

int f1()

{

return 2;

}

// файл f2.c

int f2()

{

return 10;

}

5. Откомпилируйте созданные файлы.

Компилировать можно все файлы одновременно одной командой, перечисляя составные файлы через пробел после ключа *-c*:

gcc -c main.c f1.c f2.c

Или каждый файл в отдельности:

gcc -c f1.c

gcc -c f2.c

gcc -c main.c

В результате работы компилятора получите три отдельных объектных файла:

main.o

f1.o

f2.o

6. Чтобы их собрать в один файл с помощью gcc надо использовать ключ -o:

gcc main.o f1.o f2.o -o result

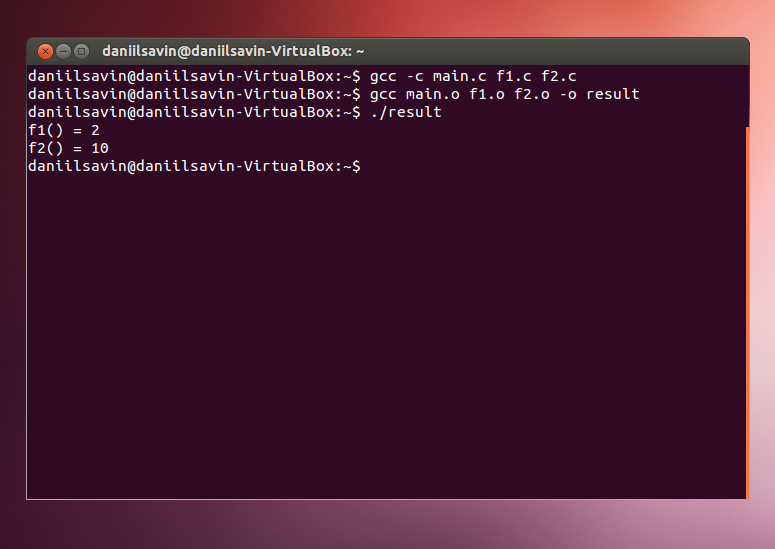
7. В результате вызова полученной программы rezult командой:

./result

На экране появится результат работы:

f1() = 2

f2() = 10



8. Теперь, если изменить какую-то из процедур, например f1():

int f1()

{

return 25;

}

то компилировать заново все файлы не придется, а понадобится лишь скомпилировать измененный файл и собрать результирующий файл из кусков:

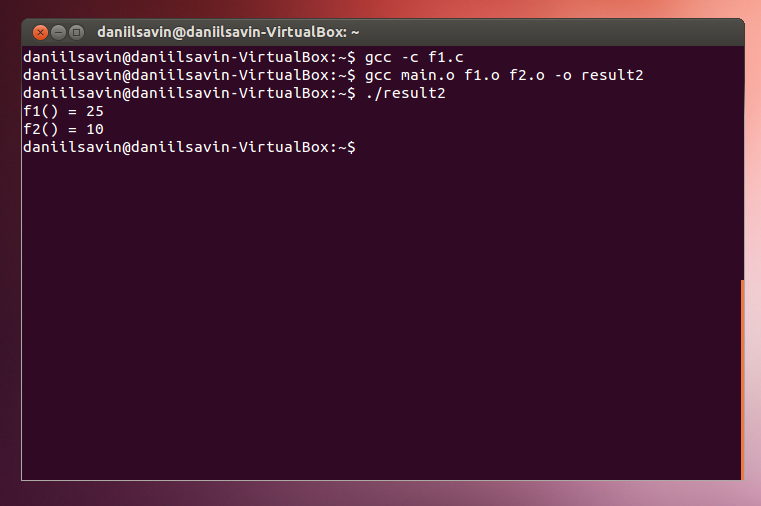
gcc -c f1.c

gcc main.o f1.o f2.o -o result2

./result2

f1() = 25

f2() = 10



**Задание 3**. Напишите программу, печатающую следующие значения:

* идентификаторов процесса (PID) и его родителя (PPID) для текущего процесса
* идентификатор пользователя, запустившего программу, и идентификатор его группы.

Ход выполнения:

1. Прочитайте информацию о типе данных *pid\_t*.

С точки зрения программирования идентификатор процесса обозначается типом *pid\_t*, величина которого определяется в заголовочном файле *<sys/types.h>*. Тип данных *pid\_t* является синонимом для одного из целочисленных типов языка C.

2. Прочитайте информацию о системных вызовах *getpid()* и *getppid().*

Системный вызов *getpid()* возвращает идентификатор вызывающего процесса:

#include <sys/types.h>

#include <unistd.h>

pid\_t getpid (void);

Системный вызов *getppid()* возвращает идентификатор родителя вызывающего процесса:

#include <sys/types.h>

#include <unistd.h>

pid\_t getppid (void);

Использование данных вы­зовов несложно, например:

printf ("My pid=%d\n",(int)getpid());

3. Напишите программу, печатающую значения PID и PPID для текущего процесса. Запустите ее несколько раз подряд. Посмотрите, как меняется идентификатор текущего процесса. Объясните наблюдаемые изменения.

4. Прочитайте информацию о системных вызовах *getuid()* и *getgid().*

Узнать идентификатор пользователя, запустившего программу на исполнение – UID и идентификатор группы, к которой он относится, – GID можно с помощью системных вызовов *getuid()* и *getgid()*, применив их внутри программы.

Системный вызов *getuid* возвращает идентификатор пользователя для текущего процесса.

Системный вызов *getgid* возвращает идентификатор группы пользователя для текущего процесса.

Типы данных uid\_t и gid\_t являются синонимами для одного из целочисленных типов языка C.

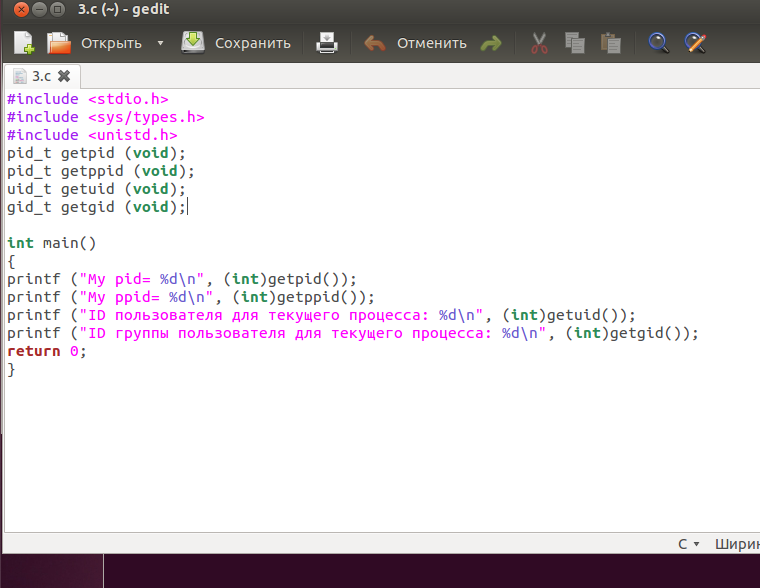
#include <sys/types.h>

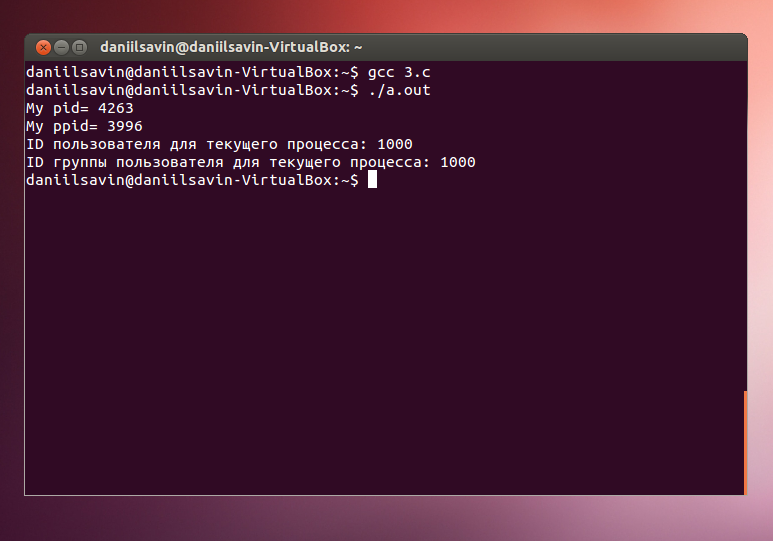
#include <unistd.h>

uid\_t getuid(void);

gid\_t getgid(void);

5. Дополните созданную программу так, чтобы она печатала идентификатор пользователя, запустившего программу, и идентификатор его группы.





**Задание 4**. Изучите особенности использования системного вызова *fork()* и функции *exit()*:

Ход выполнения:

1. Прочитайте информацию о системном вызове *fork()*.

Новый процесс, запускающий тот же системный образ, что и текущий, может быть создан с помощью системного вызова fork():

#include <sys/types.h>

#include <unistd.h>

pid\_t fork(void);

В случае успешного обращения к *fork()* создается новый процесс. Оба процесса выполняются от точки обраще­ния к *fork()*, как будто ничего не происходило.

Новый процесс является дочерним по отношению к вызывающему, который, в свою очередь, называется родительским. В дочернем процессе успешный запуск *fork()* возвращает *0*. В родительском *fork()* возвращает *pid* дочернего. Родительский и дочерний процессы практически идентичны, за исключением некоторых особен­ностей:

* pid дочернего процесса, назначается заново и отличается от родитель­ского;
* родительский pid дочернего процесса установлен равным pid родительского процесса;
* ресурсная статистика дочернего процесса обнуляется;
* любые ожидающие сигналы прерываются и не наследуются дочерним процессом;
* никакие вовлеченные блокировки файлов не наследуются дочерним процессом.

В случае ошибки дочерний процесс не создается, *fork()* возвращает *–1*, устанав­ливая соответствующее значение *errno*. Вот два возможных значения *errno* и их смысл:

* EAGAIN – ядро не способно выделить определенные ресурсы, например новый pid, или достигнуто ограничение по ресурсам RLIMIT\_NPROC;
* ENOMEM – недостаточно ресурсов памяти ядра, чтобы завершить запрос.

Использование очень простое:

pid\_t pid;

pid = fork ();

if (pid > 0)

printf ("Я родительский процесс сpid=%d!\n", pid);

else if (!pid)

printf ("А я дочерний!\n");

else if (pid == –1)

perror ("fork");

2. Наберите программу, откомпилируйте ее и запустите на исполнение.

#include <sys/types.h>

#include <unistd.h>

#include <stdio.h>

int main()

{

pid\_t pid, ppid;

int a = 0;

(void)fork();

a = a+1;

pid = getpid();

ppid = getppid();

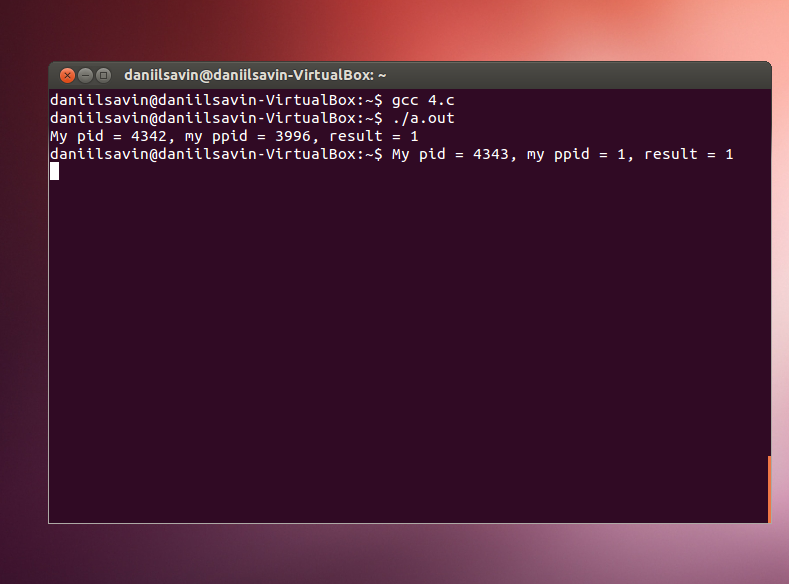
printf("My pid = %d, my ppid = %d,

result = %d\n", (int)pid, (int)ppid, a);

return 0;

}

3. Проанализируйте полученный результат.

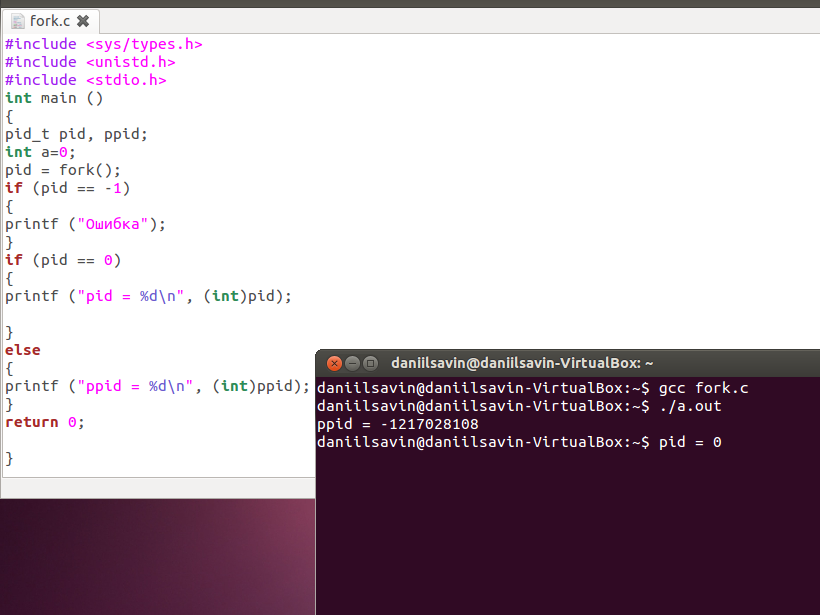


4. Какие фрагменты кода выполняются одним процессом? Какие двумя процессами?

Код после вызова fork() выполняется дважды. Значение a стало равным 1 и процесс-родитель получил это значение.

5. Как меняется значение переменной a?

6. Измените предыдущую программу с *fork()* так, чтобы родитель и ребенок совершали разные действия (какие – не важно).



7. Прочитайте информацию о функции *exit()*.

Существует два способа корректного завершения процесса в программах, написанных на языке C:

* Процесс корректно завершается по достижении конца функции *main()* или при выполнении оператора *return* в функции *main()*.
* При необходимости завершить процесс в каком-либо другом месте программы используется функция exit() из стандартной библиотеки функций для языка C.

#include <stdlib.h>

void exit(int status);

При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков, после чего инициируется системный вызов прекращения работы процесса и перевода его в состояние ***закончил исполнение***.

Если процесс завершает свою работу раньше, чем его родитель, и родитель явно не указал, что он не хочет получать информацию о статусе завершения порожденного процесса, то завершившийся процесс не исчезает из системы окончательно, а остается в состоянии ***закончил исполнение*** либо до завершения процесса-родителя, либо до того момента, когда родитель получит эту информацию. Такие процессы в операционной системе UNIX принято называть ***процессами-зомби*** (zombie, defunct).

Возврата из функции в текущий процесс не происходит, и функция ничего не возвращает.

Значение параметра *status* – кода завершения процесса – передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс. При этом используются только младшие 8 бит параметра, так что для кода завершения допустимы значения от 0 до 255. По соглашению, код завершения 0 означает безошибочное завершение процесса.

**Задание 5**. Напишите программу, распечатывающую значения аргументов командной строки и параметров окружающей среды для текущего процесса:

Ход выполнения:

1. Прочитайте информацию о параметрах функции *main()* в языке C.

У функции *main()* в языке программирования C существует три параметра, которые могут быть переданы ей операционной системой.

int main(int argc, char \*argv[], char \*envp[]);

Первые два параметра при запуске программы на исполнение командной строкой позволяют узнать полное содержание командной строки. Вся командная строка рассматривается как набор слов, разделенных пробелами. Через параметр *argc* передается количество слов в командной строке, которой была запущена программа. Параметр *argv* является массивом указателей на отдельные слова. Так, например, если программа была запущена командой

a.out 12 abcd

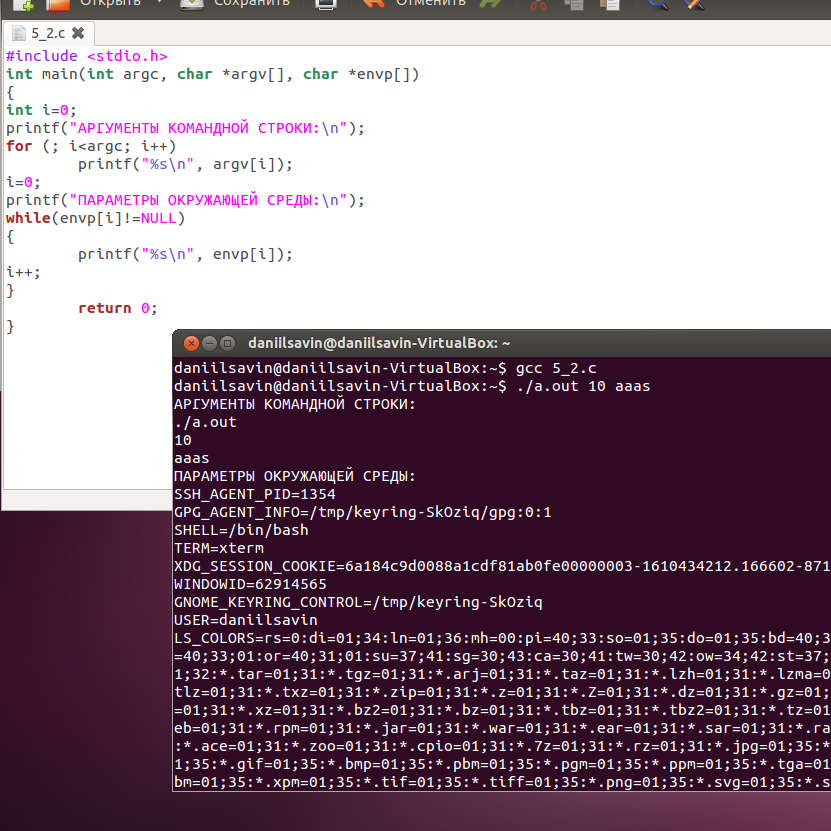
то значение параметра *argc* будет равно 3, *argv[0]* будет указывать на *имя программы* – первое слово *–* *"a.out"*, *argv[1]* – на *слово* "12", *argv[2]* — на *слово* *"abcd".* Так как имя программы всегда присутствует на первом месте в командной строке, то *argc* всегда больше 0, а *argv[0]* всегда указывает на имя запущенной программы.

Третий параметр – *envp* – является массивом указателей на параметры окружающей среды процесса. Начальные параметры окружающей среды процесса задаются в специальных конфигурационных файлах для каждого пользователя и устанавливаются при входе пользователя в систему. В дальнейшем они могут быть изменены с помощью специальных команд операционной системы *UNIX*.

Каждый параметр имеет вид: *переменная=строка*. Такие переменные используются для изменения долгосрочного поведения процессов, в отличие от аргументов командной строки. Например, задание параметра *TERM=vt100* может говорить процессам, осуществляющим вывод на экран дисплея, что работать им придется с терминалом *vt100*. Меняя значение переменной среды*TERM*, например на *TERM=console*, мы сообщаем таким процессам, что они должны изменить свое поведение и осуществлять *вывод* для *системной консоли*.

Размер массива аргументов командной строки в функции *main()* мы получали в качестве ее параметра. Так как для массива ссылок на параметры окружающей среды такого параметра нет, то его размер определяется другим способом. Последний элемент этого массива содержит указатель *NULL*.

2. Напишите программу, распечатывающую значения аргументов командной строки и параметров окружающей среды для текущего процесса.



**Задание 6**. Изучите особенности использования системного вызова *exec()*:

Ход выполнения:

1. Прочитайте информацию о системном вызове *exec()*.

Вызов *exec()* заменяет пользовательский контекст текущего процесса на содержимое некоторого исполняемого файла и устанавливает начальные значения регистров процессора (в том числе устанавливает программный счетчик на начало загружаемой программы). Этот вызов требует для своей работы задания имени исполняемого файла, аргументов командной строки и параметров окружающей среды.

Для осуществления вызова программист может воспользоваться одной из шести функций: *execlp() ,**execvp() ,**execl() и,**execv() ,**execle() ,**execve() ,* отличающихся друг от друга представлением параметров, необходимых для работы системного вызова *exec()*.

#include <unistd.h>

int execlp(const char \*file,

const char \*arg0,

... const char \*argN,(char \*)NULL)

int execvp(const char \*file, char \*argv[])

int execl(const char \*path,

const char \*arg0,

... const char \*argN,(char \*)NULL)

int execv(const char \*path, char \*argv[])

int execle(const char \*path,

const char \*arg0,

... const char \*argN,(char \*)NULL,

char \* envp[])

int execve(const char \*path, char \*argv[],

char \*envp[])

Аргумент *file* является указателем на имя файла, который должен быть загружен. Аргумент *path* – это указатель на полный путь к файлу, который должен быть загружен.

Аргументы *arg0, ..., argN* представляют собой указатели на аргументы командной строки. Заметим, что аргумент *arg0* должен указывать на имя загружаемого файла. Аргумент *argv* представляет собой массив из указателей на аргументы командной строки. Начальный элемент массива должен указывать на имя загружаемой программы, а заканчиваться массив должен элементом, содержащим указатель *NULL*.

Аргумент *envp* является массивом указателей на параметры окружающей среды, заданные в виде строк *"переменная=строка"*. Последний элемент этого массива должен содержать указатель *NULL*.

Поскольку вызов функции не изменяет системный контекст текущего процесса, загруженная программа унаследует от загрузившего ее процесса следующие атрибуты:

* идентификатор процесса;
* *идентификатор родительского процесса*;
* групповой идентификатор процесса;
* идентификатор сеанса;
* время, оставшееся до возникновения сигнала SIGALRM ;
* текущую рабочую директорию;
* маску создания файлов;
* идентификатор пользователя;
* групповой идентификатор пользователя;
* явное игнорирование сигналов;
* таблицу открытых файлов (если для файлового дескриптора не устанавливался признак "закрыть файл при выполнении *exec()* ").

В случае успешного выполнения возврата из функций в программу, осуществившую вызов, не происходит, а управление передается загруженной программе. В случае неудачного выполнения в программу, инициировавшую вызов, возвращается отрицательное значение.

Символы *l* и *v* указывают, передаются ли аргументы списком или массивом. Символ *p* указывает, что система будет искать указанный файл по полному пользовательскому пути. В командах, где используются варианты с *p*, можно указать только имя файла, если он находится в пределах пользовательского пути. Наконец, *е* обозначает, что для нового процесса создается новое окружение.

2. Откомпилируйте программу, сохраните ее под именем 12\_6.c и запустите на исполнение.

#include <sys/types.h>

#include <unistd.h>

#include <stdio.h>

int main(int argc, char \*argv[],

char \*envp[]){

(void) execle("/bin/cat", "/bin/cat", "12\_6.c ", 0, envp);

printf("Error on program start\n");

exit(-1);

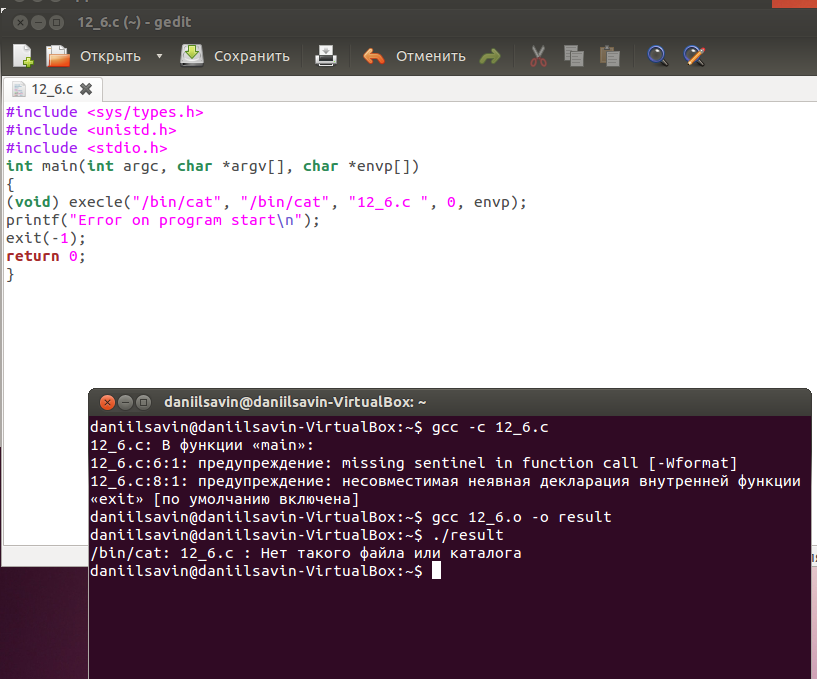
return 0;

}

3. Проанализируйте результат.

Как программа изменяет пользовательский контекст процесса?

Что произойдет при возникновении ошибки в ходе работы программы?



**Задание 7**. Изучите особенности работы с файлами через системные вызовы и стандартную библиотеку ввода-вывода для языка C.

Ход выполнения:

1. Прочитайте информацию о механизмах, обеспечивающих потоковую передачу данных в операционной системе Linux.

Среди всех категорий средств коммуникации наиболее употребительными являются каналы связи, обеспечивающие достаточно безопасное и достаточно информативное взаимодействие процессов.

Существует две модели передачи данных по каналам связи – поток ввода-вывода и сообщения. Из них более простой является потоковая модель, в которой операции передачи/приема информации вообще не интересуются содержимым того, что передается или принимается. Вся информация в канале связи рассматривается как непрерывный поток байт, не обладающий никакой внутренней структурой.

Потоковая передача информации может осуществляться не только между процессами, но и между процессом и устройством ввода-вывода, например, между процессом и диском, на котором данные представляются в виде файла. Поскольку системные вызовы, использующиеся для потоковой работы с файлом, во многом соответствуют системным вызовам, применяемым для потокового общения процессов, мы начнем наше рассмотрение именно с механизма потокового обмена между процессом и файлом.

В UNIX-подобных ОС можно упрощенно полагать, что информация о файлах, с которыми процесс осуществляет операции потокового обмена, наряду с информацией о потоковых линиях связи, соединяющих процесс с другими процессами и устройствами ввода-вывода, хранится в некотором массиве, получившем название ***таблицы открытых файлов*** или ***таблицы******файловых дескрипторов***.

Индекс элемента этого массива, соответствующий определенному потоку ввода-вывода, получил название файлового дескриптора для этого потока. Таким образом, файловый дескриптор представляет собой небольшое целое неотрицательное число, которое для текущего процесса в данный момент времени однозначно определяет некоторый действующий канал ввода-вывода.

Некоторые файловые дескрипторы на этапе старта любой программы ассоциируются со стандартными потоками ввода-вывода. Так, например, файловый дескриптор ***0*** соответствует стандартному потоку ввода, файловый дескриптор ***1*** – стандартному потоку вывода, файловый дескриптор ***2*** – стандартному потоку для вывода ошибок. В нормальном интерактивном режиме работы стандартный поток ввода связывает процесс с клавиатурой, а стандартные потоки вывода и вывода ошибок – с текущим терминалом (дисплеем).

Наиболее простой способ доступа к файлу связан с использованием системных вызовов *read()* и *write()*. Однако прежде, чем к файлу можно будет получить доступ, его требуется открыть с помощью системного вызова *open()* или *creat()*. Когда работа с файлом закончена, его нужно закрыть посредством системного вызова *close()*.

2. Прочитайте информацию о системном вызове *open()*.

#include <fcntl.h>

int open(char \*path, int flags);

int open(char \*path, int flags, int mode);

Системный вызов *open()* предназначен для выполнения операции открытия файла и, в случае ее удачного осуществления, возвращает файловый дескриптор открытого файла (небольшое неотрицательное целое число, которое используется в дальнейшем для других операций с этим файлом).

Параметр *path* является указателем на строку, содержащую полное или относительное имя файла.

Параметр *flags* может принимать одно из следующих трех значений:

* O\_RDONLY – если над файлом в дальнейшем будут совершаться только операции чтения;
* O\_WRONLY – если над файлом в дальнейшем будут осуществляться только операции записи;
* O\_RDWR – если над файлом будут осуществляться и операции чтения, и операции записи.

Каждое из этих значений может быть скомбинировано посредством операции "побитовое или ( | )" с одним или несколькими флагами:

* O\_CREAT – если файла с указанным именем не существует, он должен быть создан;
* O\_EXCL – применяется совместно с флагом O\_CREAT. При совместном их использовании и существовании файла с указанным именем, открытие файла не производится и констатируется ошибочная ситуация;
* O\_NDELAY – запрещает перевод процесса в состояние ожидание при выполнении операции открытия и любых последующих операциях над этим файлом;
* O\_APPEND – при открытии файла и перед выполнением каждой операции записи (если она, конечно, разрешена) указатель текущей позиции в файле устанавливается на конец файла;
* O\_TRUNC – если файл существует, уменьшить его размер до 0, с сохранением существующих атрибутов файла, кроме, быть может, времен последнего доступа к файлу и его последней модификации.

Кроме того, в некоторых версиях операционной системы UNIX могут применяться дополнительные значения флагов:

* O\_SYNC – любая операция записи в файл будет блокироваться (т. е. процесс будет переведен в состояние ожидание) до тех пор, пока записанная информация не будет физически помещена на соответсвующий нижележащий уровень hardware;
* O\_NOCTTY – если имя файла относится к терминальному устройству, оно не становится управляющим терминалом процесса, даже если до этого процесс не имел управляющего терминала.

Параметр *mode* устанавливает атрибуты прав доступа различных категорий пользователей к новому файлу при его создании. Он обязателен, если среди заданных флагов присутствует флаг O\_CREAT, и может быть опущен в противном случае. Этот параметр задается как сумма следующих восьмеричных значений:

* 0400 – разрешено чтение для пользователя, создавшего файл;
* 0200 – разрешена запись для пользователя, создавшего файл;
* 0100 – разрешено исполнение для пользователя, создавшего файл;
* 0040 – разрешено чтение для группы пользователя, создавшего файл;
* 0020 – разрешена запись для группы пользователя, создавшего файл;
* 0010 – разрешено исполнение для группы пользователя, создавшего файл;
* 0004 – разрешено чтение для всех остальных пользователей;
* 0002 – разрешена запись для всех остальных пользователей;
* 0001 – разрешено исполнение для всех остальных пользователей.

При создании файла реально устанавливаемые права доступа получаются из стандартной комбинации параметра mode и маски создания файлов текущего процесса umask, а именно – они равны mode & ~umask.

Системный вызов *open()* использует набор флагов для того, чтобы специфицировать операции, которые предполагается применять к файлу в дальнейшем или которые должны быть выполнены непосредственно в момент открытия файла. Из всего возможного набора флагов на текущем уровне знаний нас будут интересовать только флаги O\_RDONLY, O\_WRONLY, O\_RDWR, O\_CREAT и O\_EXCL. Первые три флага являются взаимоисключающими: хотя бы один из них должен быть применен и наличие одного из них не допускает наличия двух других. Эти флаги описывают набор операций, которые, при успешном открытии файла, будут разрешены над файлом в дальнейшем: только чтение, только запись, чтение и запись.

В случае, когда мы **допускаем**, что файл на диске может отсутствовать, и хотим, чтобы он был создан, флаг для набора операций должен использоваться в комбинации с флагом O\_CREAT. Если файл существует, то все происходит по рассмотренному выше сценарию. В случае, когда мы **требуем**, чтобы файл на диске отсутствовал и был создан в момент открытия, флаг для набора операций должен использоваться в комбинации с флагами O\_CREAT и O\_EXCL.

3. Прочитайте информацию о системных вызовах *read(), write(), close()*.

#include <sys/types.h>

#include <unistd.h>

size\_t read(int fd, void \*addr,

size\_t nbytes);

size\_t write(int fd, void \*addr,

size\_t nbytes);

Системные вызовы *read()* и *write()* предназначены для осуществления потоковых операций ввода (чтения) и вывода (записи) информации над каналами связи, описываемыми файловыми дескрипторами, т.е. для файлов, pipe, FIFO и socket.

Параметр *fd* является файловым дескриптором созданного ранее потокового канала связи, через который будет отсылаться или получаться информация, т. е. значением, которое вернул один из системных вызовов *open()* , *pipe()* или *socket()*.

Параметр *addr* представляет собой адрес области памяти, начиная с которого будет браться информация для передачи или размещаться принятая информация.

Параметр *nbytes* для системного вызова *write* определяет количество байт, которое должно быть передано, начиная с адреса памяти *addr*. Параметр *nbytes* для системного вызова *read* определяет количество байт, которое мы хотим получить из канала связи и разместить в памяти, начиная с адреса *addr*.

В случае успешного завершения системный вызов возвращает количество реально посланных или принятых байт. Это значение может не совпадать с заданным значением параметра *nbytes*, а быть меньше, чем оно, в силу отсутствия места на диске или в линии связи при передаче данных или отсутствия информации при ее приеме. При возникновении какой-либо ошибки возвращается отрицательное значение.

При работе с файлами информация записывается в файл или читается из файла, начиная с места, определяемого указателем текущей позиции в файле. Значение указателя увеличивается на количество реально прочитанных или записанных байт. При чтении информации из файла она не пропадает из него. Если системный вызов *read()* возвращает значение *0*, то это означает, что файл прочитан до конца.

#include <unistd.h>

int close(int fd);

Системный вызов *close* предназначен для корректного завершения работы с файлами и другими объектами ввода-вывода, которые описываются в операционной системе через файловые дескрипторы: pipe, FIFO, socket.

Параметр *fd* является дескриптором соответствующего объекта, т. е. значением, которое вернул один из системных вызовов *open()*, *pipe()* или *socket()*.

Системный вызов возвращает значение *0* при нормальном завершении и значение *-1* при возникновении ошибки.

3. Наберите, откомпилируйте программу и запустите ее на исполнение. Обратите внимание на использование системного вызова *umask()* с параметром *0* для того, чтобы права доступа к созданному файлу точно соответствовали указанным в системном вызове *open()*.

#include <sys/types.h>

#include <fcntl.h>

#include <stdio.h>

int main(){

int fd;

size\_t size;

char string[] = "Hello, world!";

/\* Обнуляем маску создания файлов текущего процесса для того,

чтобы права доступа у создаваемого файла точно соответствовали параметру вызова open() \*/

(void)umask(0);

/\* Попытаемся открыть файл с именем myfile в текущей директории только для операций вывода. Если файла не существует, попробуем его создать с правами доступа 0666, т. е. read-write для всех категорий пользователей \*/

if((fd = open("myfile", O\_WRONLY | O\_CREAT,

0666)) < 0){

/\* Если файл открыть не удалось, печатаем об этом сообщение и прекращаем работу \*/

printf("Can\'t open file\n");

exit(-1);

}

/\* Пробуем записать в файл 14 байт из нашего массива, т.е. всю строку "Hello, world!" вместе с признаком конца строки \*/

size = write(fd, string, 14);

if(size != 14){

/\* Если записалось меньшее количество байт, сообщаем об

ошибке \*/

printf("Can\'t write the whole string\n");

exit(-1);

}

/\* Закрываем файл \*/

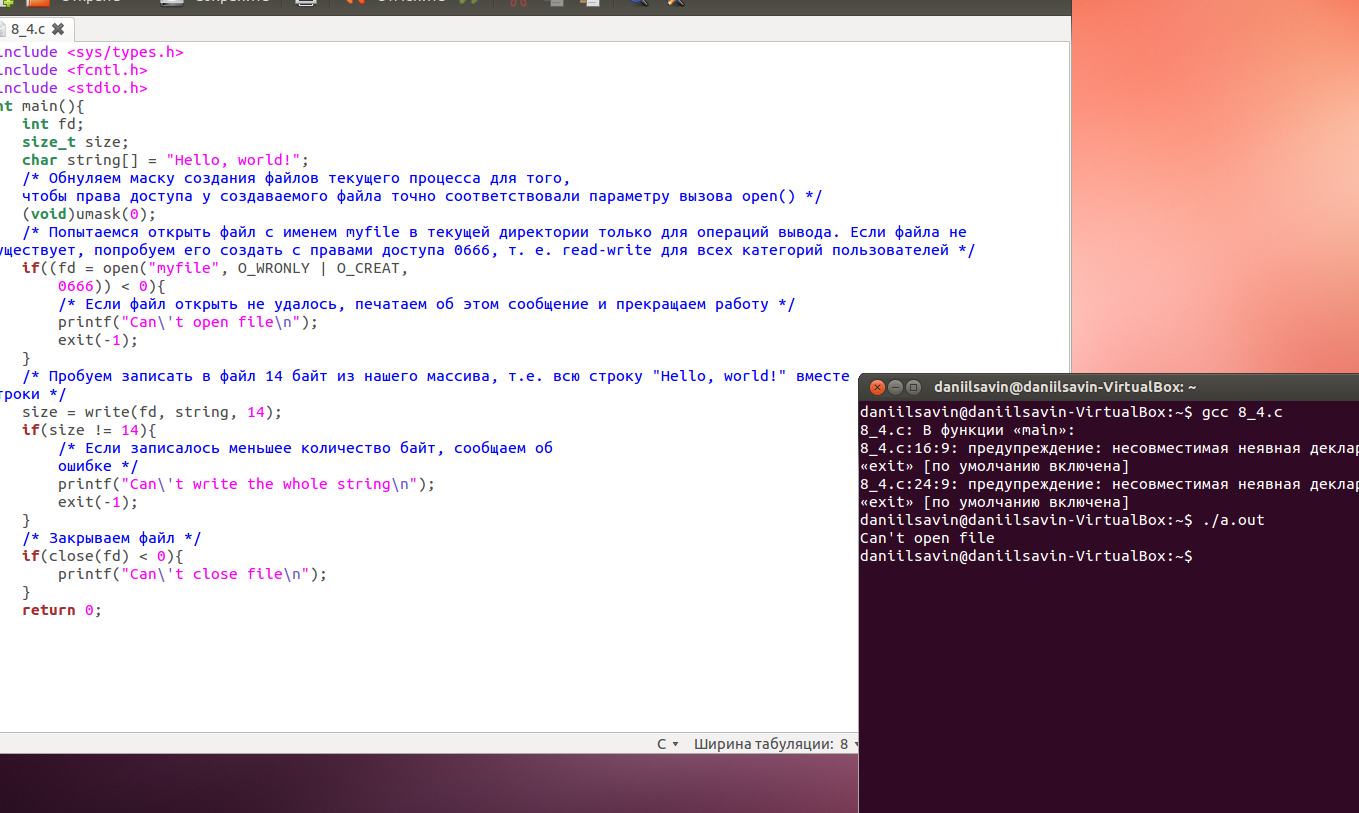
if(close(fd) < 0){

printf("Can\'t close file\n");

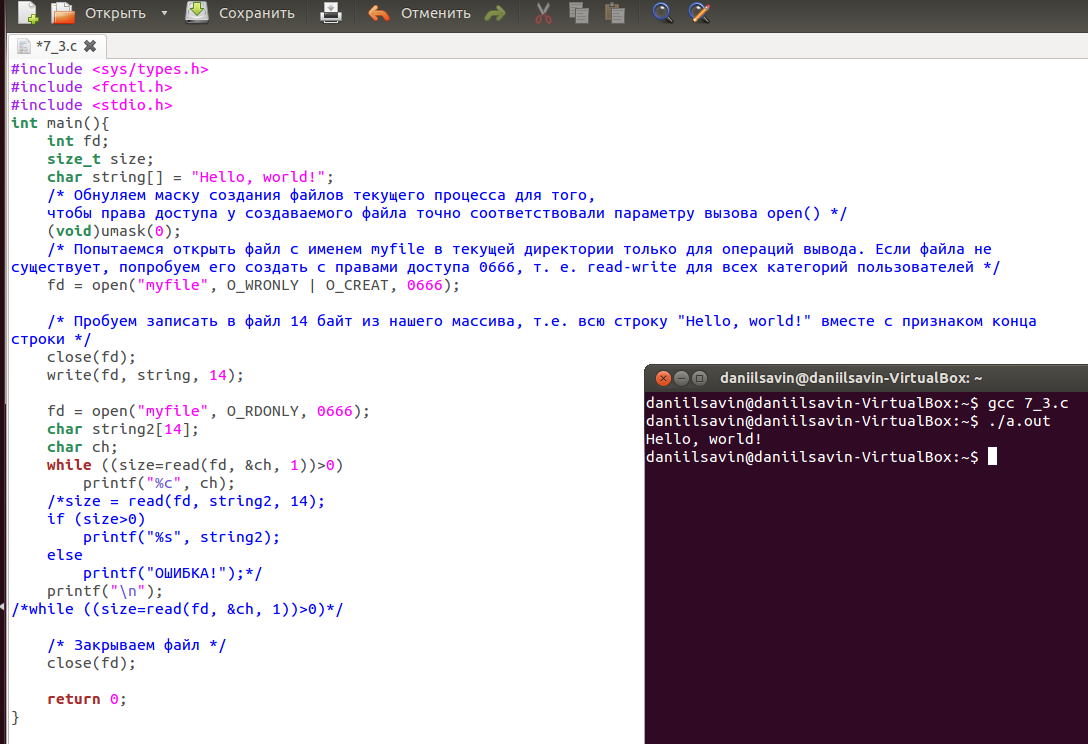
}

return 0;

}



4. Измените программу из предыдущего раздела так, чтобы она читала записанную ранее в файл информацию и печатала ее на экране. Все лишние операторы желательно удалить.



**Задание 8**. Изучите особенности организации взаимодействия процессов через конвейеры.

Ход выполнения:

1. Прочитайте информацию о системном вызове *pipe()*.

Наиболее простым способом для передачи информации с помощью потоковой модели между различными процессами или даже внутри одного процесса в операционной системе UNIX является конвейер (pipe).

Важное отличие конвейера от файла заключается в том, что прочитанная информация немедленно удаляется из него и не может быть прочитана повторно.

Конвейер представляет собой область памяти, недоступную пользовательским процессам напрямую, зачастую организованную в виде кольцевого буфера (хотя существуют и другие виды организации). По буферу при операциях чтения и записи перемещаются два указателя, соответствующие входному и выходному потокам. При этом выходной указатель никогда не может перегнать входной и наоборот. Для создания нового экземпляра такого кольцевого буфера внутри операционной системы используется системный вызов *pipe()*.

#include <unistd.h>

int pipe(int \*fd);

Параметр *fd*является указателем на массив из двух целых переменных. При нормальном завершении вызова в первый элемент массива – *fd[0]* – будет занесен файловый дескриптор, соответствующий выходному потоку данных конвейера и позволяющий выполнять только операцию чтения, а во второй элемент массива – *fd[1]* – будет занесен файловый дескриптор, соответствующий входному потоку данных и позволяющий выполнять только операцию записи.

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибок.

2. Наберите программу, организующую работу с конвейером в рамках одного процесса откомпилируйте ее и запустите на исполнение.

#include <sys/types.h>

#include <unistd.h>

#include <stdio.h>

int main(){

int fd[2];

size\_t size;

char string[] = "Hello, world!";

char resstring[14];

/\* Попытаемся создать pipe \*/

if(pipe(fd) < 0){

/\* Если создать pipe не удалось, печатаем об этом сообщение

и прекращаем работу \*/

printf("Can\'t create pipe\n");

exit(-1);

}

/\* Пробуем записать в pipe 14 байт из нашего массива, т.е. всю строку "Hello, world!" вместе с признаком конца строки \*/

size = write(fd[1], string, 14);

if(size != 14){

/\* Если записалось меньшее количество байт, сообщаем об

ошибке \*/

printf("Can\'t write all string\n");

exit(-1);

}

/\* Пробуем прочитать из pip'а 14 байт в другой массив, т.е. всю записанную строку \*/

size = read(fd[0], resstring, 14);

if(size < 0){

/\* Если прочитать не смогли, сообщаем об ошибке \*/

printf("Can\'t read string\n");

exit(-1);

}

/\* Печатаем прочитанную строку \*/

printf("%s\n",resstring);

/\* Закрываем входной поток\*/

if(close(fd[0]) < 0){

printf("Can\'t close input stream\n");

}

/\* Закрываем выходной поток\*/

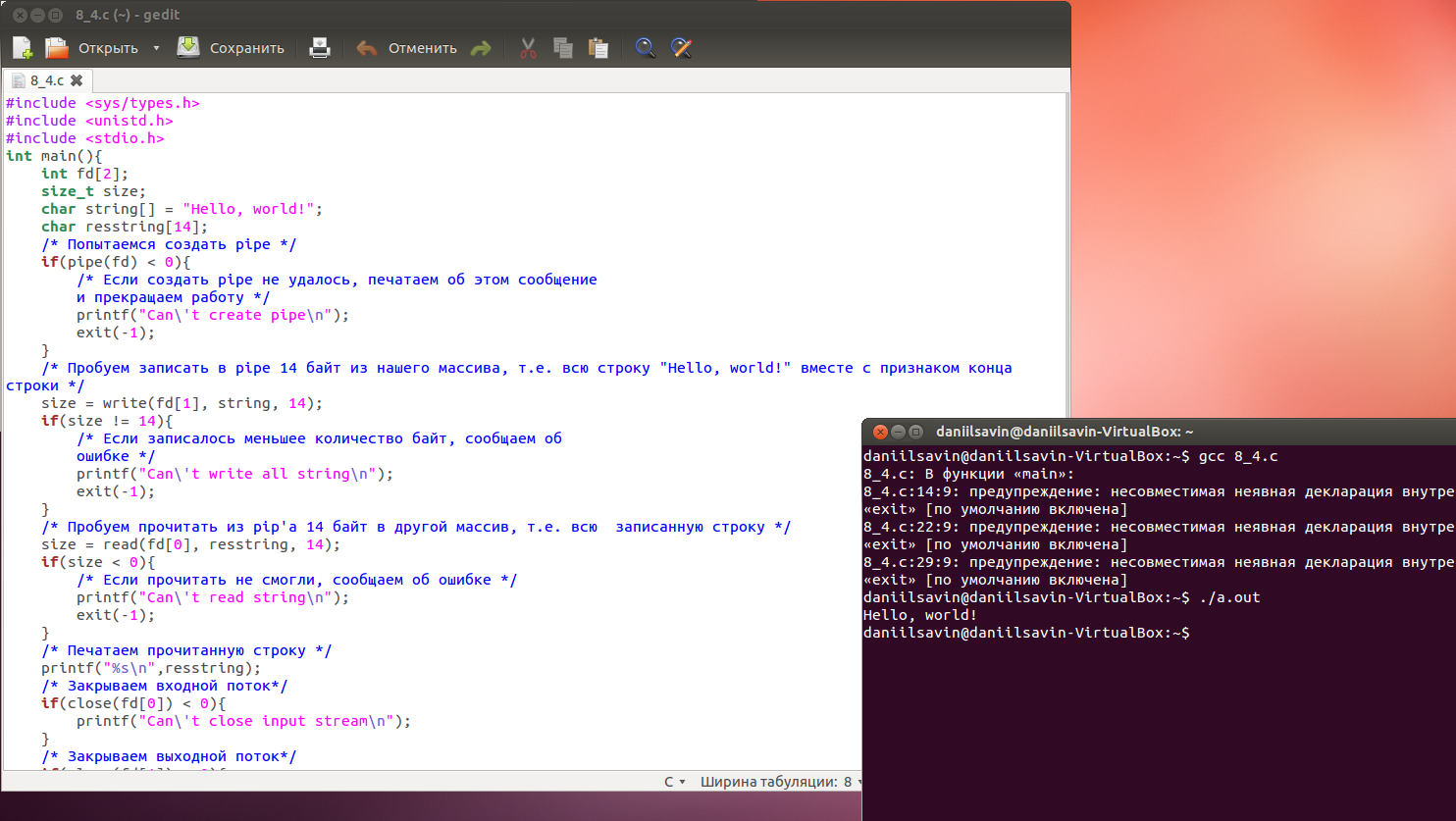
if(close(fd[1]) < 0){

printf("Can\'t close output stream\n");

}

return 0;

}



3. Наберите программу, осуществляющую однонаправленную связь между процессом-родителем и процессом-ребенком, откомпилируйте ее и запустите на исполнение.

#include <sys/types.h>

#include <unistd.h>

#include <stdio.h>

int main(){

int fd[2], result;

size\_t size;

char resstring[14];

/\* Попытаемся создать конвейер \*/

if(pipe(fd) < 0){

/\* Если создать конвейер не удалось, печатаем об этом сообщение и прекращаем работу \*/

printf("Can\'t create pipe\n");

exit(-1);

}

/\* Порождаем новый процесс \*/

result = fork();

if(result){

/\* Если создать процесс не удалось, сообщаем об этом и

завершаем работу \*/

printf("Can\'t fork child\n");

exit(-1);

} else if (result > 0) {

/\* Мы находимся в родительском процессе, который будет

передавать информацию процессу-ребенку. В этом процессе

выходной поток данных нам не понадобится, поэтому

закрываем его.\*/

close(fd[0]);

/\* Пробуем записать в pipe 14 байт, т.е. всю строку

"Hello, world!" вместе с признаком конца строки \*/

size = write(fd[1], "Hello, world!", 14);

if(size != 14){

/\* Если записалось меньшее количество байт, сообщаем

об ошибке и завершаем работу \*/

printf("Can\'t write all string\n");

exit(-1);

}

/\* Закрываем входной поток данных, на этом

родитель прекращает работу \*/

close(fd[1]);

printf("Parent exit\n");

} else {

/\* Мы находимся в порожденном процессе, который будет

получать информацию от процесса-родителя. Он унаследовал

от родителя таблицу открытых файлов.

В этом процессе входной поток данных нам не

понадобится, поэтому закрываем его.\*/

close(fd[1]);

/\* Пробуем прочитать из конвейера 14 байт в массив, т.е. всю записанную строку \*/

size = read(fd[0], resstring, 14);

if(size < 0){

/\* Если прочитать не смогли, сообщаем об ошибке и

завершаем работу \*/

printf("Can\'t read string\n");

exit(-1);

}

/\* Печатаем прочитанную строку \*/

printf("%s\n",resstring);

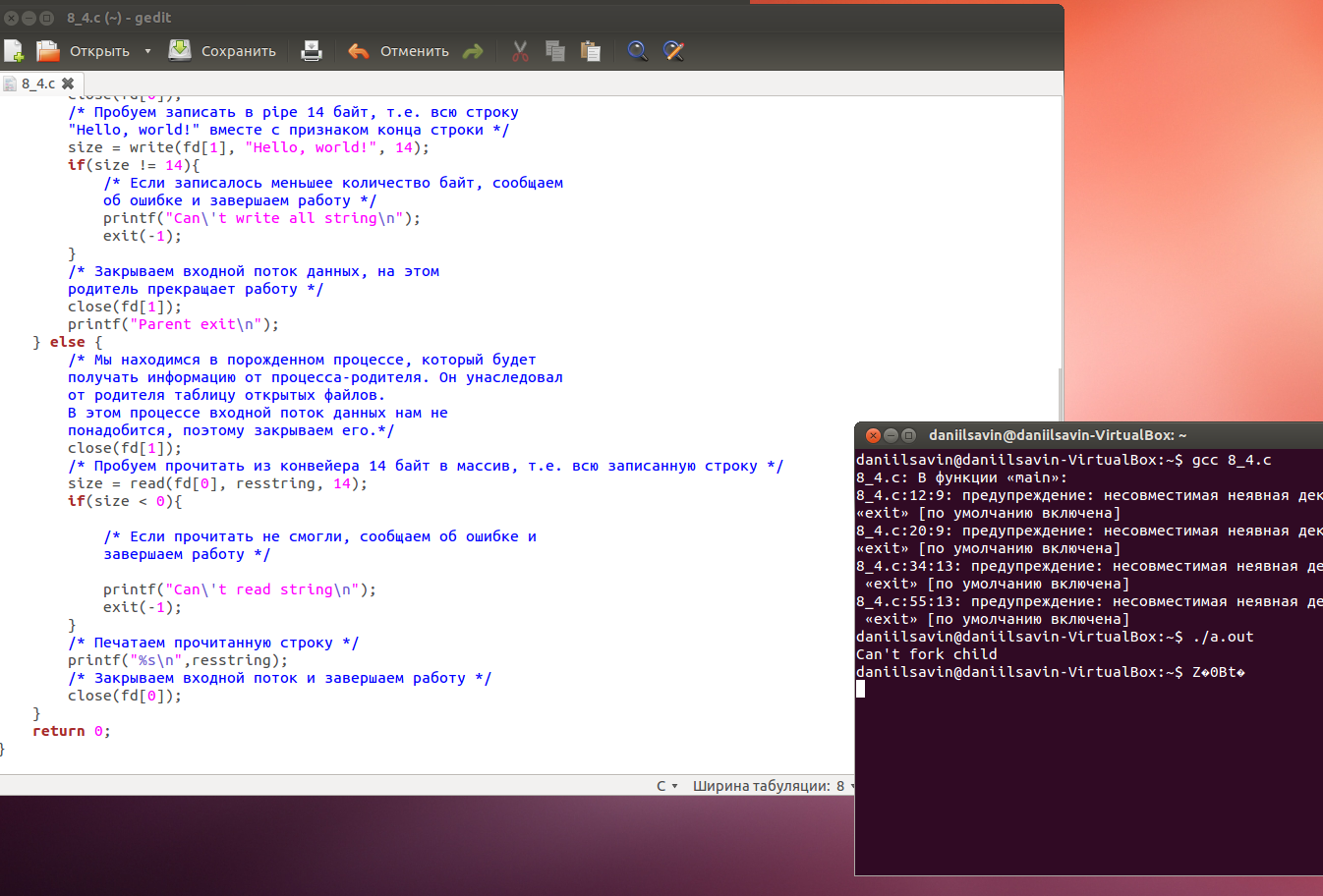
/\* Закрываем входной поток и завершаем работу \*/

close(fd[0]);

}

return 0;

}



4. Измените приведенную программу для двусторонней связи между двумя родственными процессами, исполняющими разные программы.

