2024

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Раздел #2

АВТОСБОРКА ПРИЛОЖЕНИЙ

МНОГОЗАДАЧНОСТЬ И МНОГОФАЙЛОВЫЕ ПРОЕКТЫ

Лабораторная работа #22

Многозадачность процессов и этапы компиляции

РАЗРАБОТАЛ

СЕРГЕЙ СТАНКЕВИЧ (SERHEY STANKEWICH)

Оглавление

ЛАБОРАТОРНАЯ РАБОТА #22

Многозадачность процессов и этапы компиляции

Цель работы

Изучить встроенный инструментарий для разработки приложений под семейство ОС Linux и фундаментальные основы системного программирования с использованием компиляторов ***gcc/g++***, отладчика ***gdb*** и других для проектирования, компиляции, отладки и запуска приложений на языке программирования C/C++.

Изучить базовую концепцию операционной системы, «Процесс». Получить представление о многозадачности процессов.

Краткие теоретические сведения.

РАЗДЕЛ #1 – КОМПИЛЯЦИЯ И СБОРКА

Процесс компиляции

Компиляция исходных текстов на Си в исполняемый файл происходит в несколько этапов, которые представлены на рисунке 1. В результате получается исполняемый, бинарный код. Файл, содержащий исполняемый код, обычно называют *исполняемым файлом* или *бинарником* (binary).

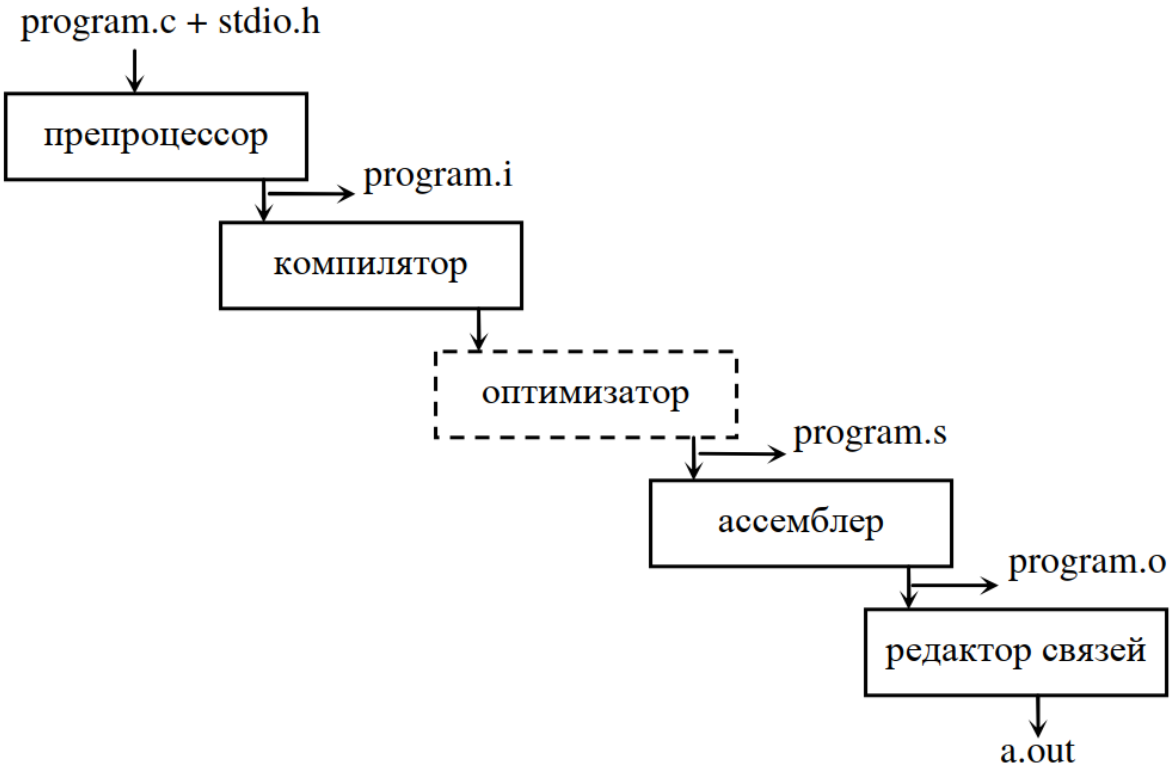


Рисунок 1 – Процесс генерации исполняемого файла по исходному тексту

В системе UNIX Си-компилятор, как правило, состоит из трех программ: *препроцессора*, *синтаксического анализатора* и *генератора кода*.

Результатом его работы является программа на языке ассемблера, которая затем транслируется в объектный файл, компонуемый с другими модулями загрузчиком. В итоге образуется выполняемая программа.

Процесс компиляции состоит из следующих этапов:

1. **Лексический анализ**. Последовательность символов исходного файла преобразуется в последовательность лексем.
2. **Синтаксический анализ**. Последовательность лексем преобразуется в дерево разбора.
3. **Семантический анализ**. Дерево разбора обрабатывается с целью установления его семантики (смысла) – например, привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т.д.
4. **Оптимизация**. Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла.
5. **Ассемблирование**. Промежуточное представление кода на языке ассемблера.
6. **Генерация кода**. Из промежуточного представления порождается объектный код. Результатом генерации является *объектный код*.
7. **Связывание**. Это последний этап, который либо ведет к получению исполняемого файла.

Препроцессор

Первым этапом работы компилятора является обработка исходного кода препроцессором языка C. Препроцессор выполняет 3 операции: *текстовые замены, вырезание комментариев и включение файлов*.

Текстовые замены и включения файлов запрашиваются программистом с помощью директив препроцессора. Директивы препроцессора – это строки, начинающиеся с символа "#".

Директивы препроцессора **#include** это постоянно используемая директива. Она создает копию указанного файла, которая включается в программу вместо директивы. Существует две формы использования директивы **#include**:

```
#include <filename>
#include “filename”
```

Разница между ними заключается в том, где компилятор будет искать файлы, которые необходимо включить. Если имя заключено в кавычки (“ и ”), система ищет его в том же каталоге, что и компилируемый файл. Такую запись обычно используют для включения определенных пользователем заголовочных файлов. Если же имя файла заключено в угловые скобки (< и >), это используется для файлов стандартной библиотеки, то поиск будет вестись в зависимости от конкретной реализации компилятора, обычно в предопределенных операционной системой каталогах.

Рассмотрим пример:

```
#include <stdio.h>

// Это комментарий.

#define STRING "This is a test"
#define COUNT (5)

int main () {
        int i;

// Это еще какой-то комментарий.
        for (i=0; i<COUNT; i++) {
                puts(STRING);
        }

        return 1;
}
```

Первая директива в нашем примере подключает стандартный заголовочный файл **stdio.h**, его содержимое подставляется в наш исходный файл. Вторая и третья директивы заменяют строки в нашем коде.

Запустив **gcc** с ключом "-E", можно остановить его работу после первого этапа и увидеть результаты работы препроцессора над нашим кодом.

`gcc -E test.c -o test.i` так `gcc -E test.c > test.txt` или так `gcc -i test.c`

флаги -i, -ii зависят от языка программирования, C или C++ соответственно.

Файл **stdio.h** довольно велик, поэтому опустим некоторые ненужные для нас строки.

```
# 1 "test.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 28 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 330 "/usr/include/features.h" 3 4
# 1 "/usr/include/sys/cdefs.h" 1 3 4
# 348 "/usr/include/sys/cdefs.h" 3 4
# 1 "/usr/include/bits/wordsize.h" 1 3 4
# 349 "/usr/include/sys/cdefs.h" 2 3 4
# 331 "/usr/include/features.h" 2 3 4
# 354 "/usr/include/features.h" 3 4
# 1 "/usr/include/gnu/stubs.h" 1 3 4

# 653 "/usr/include/stdio.h" 3 4
extern int puts (__const char *__s);
…
…
int main ()
{
    int i;
```

```
    for (i=0; i<5; i++)
    {
        puts("This is a test");
    }

    return 1;
}
```

Препроцессор дописал к нашей простой программе много новых строк. А сколько строк в файле, полученном в результате препроцессинга вашей программы?

Мы запросили у препроцессора включение заголовочного файла stdio.h в нашу программу. В свою очередь, stdio.h запросил включение других заголовочных файлов, и так далее. Препроцессор сделал отметки о том, включение какого файла и на какой строке было запрошено. Эта информация будет использована на следующих этапах компиляции.

Так, строки

```
# 28 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
```

означают, что файл features.h был запрошен на строке 28 файла stdio.h. Препроцессор создает эту отметку перед соответствующим "интересным" местом, так что, если встретится ошибка, компилятор сможет нам сообщить, где именно произошла ошибка.

Теперь посмотрим на эти строки:

```
# 653 "/usr/include/stdio.h" 3 4
extern int puts (__const char *__s);
```

Здесь **puts()** объявлена как внешняя функция (extern), возвращающая целочисленной значение и принимающая массив постоянных символов в качестве параметра. Если бы случилась несостыковка, касающаяся этой функции, тогда компилятор смог бы сообщить нам, что данная функция была объявлена в файле stdio.h на строке 653. Интересно, что на данном этапе функция puts() не определена, а лишь объявлена. Здесь пока нет реального кода, который будет работать при вызове функции puts(). Определение функций будет происходить позже.

Также обратите внимание на то, что все комментарии были удалены препроцессором, и произведены все запрошенные текстовые замены. Можно сделать вывод, что препроцессор нужен для исследования кода программ и его зависимостей от других библиотек.

В данный момент программа готова к следующему этапу, трансляции на язык ассемблера.

Ассемблирование

Ассемблирование – это процесс преобразования программы на язык ассемблера. Результат трансляции можно увидеть с помощью ключа -S:

gcc -S test.c или так gcc -S test.i -o test.s

Будет создан файл с именем **test.s**, содержащий реализацию нашей программы на языке ассемблера.

```
        .file   "test.c"
        .section        .rodata
.LC0:
        .string "This is a test"
        .text
.globl main
        .type   main, @function
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $20, %esp
        movl    $0, -8(%ebp)
        jmp     .L2
.L3:
        movl    $.LC0, (%esp)
        call    puts
        addl    $1, -8(%ebp)
.L2:
        cmpl    $4, -8(%ebp)
        jle     .L3
        movl    $1, %eax
        addl    $20, %esp
        popl    %ecx
        popl    %ebp
        leal    -4(%ecx), %esp
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 4.2.4 (Gentoo 4.2.4 p1.0)"
        .section        .note.GNU-stack,"",@progbits
```

Я не силен в языке ассемблера, однако некоторые моменты можно выделить сразу. Строка сообщения была перемещена в другую область памяти и стала называться .LC0. Основную часть кода занимают операции, от начала выполнения программы и до ее завершения. Очевидна реализация цикла for на метке .L2: это просто проверка (cmpl) и инструкция "переход, если меньше" ("Jump if Less Than", jle). Инициализация цикла осуществляется оператором movl перед меткой .L3. Между метками .L3 и .L2 очевиден вызов функции puts(). Ассемблер знает, что вызов функции puts() по имени здесь корректен, и

что это не метка памяти, как например .L2. Обсудим этот механизм далее, когда будем говорить о заключительном этапе компиляции, связывании. Наконец, наша программа завершается операцией возвращения (ret).

Генерация объектного кода

Объектный код – это программа на языке машинных кодов с частичным сохранением символьной информации, необходимой в процессе сборки.

Если указать компилятору, чтобы он пропустил этап связывания, с помощью ключа -c и -o:

```
gcc -c test.c -o test.o
```

тогда мы получим объектный файл размеров всего 888 байт. При отладочной сборке возможно сохранение большого количества символьной информации (идентификаторов переменных, функций, а также типов). Если скомпилировать нашу программу с этапом связывания, мы получим исполняемый файл размером 6885 байт. Разницу составляет как раз код для запуска и завершения программы, а также вызов функции puts() из библиотеки libc.so.

Связывание (линковка)

Связывание ведет к получению исполняемого файла, либо объектного файла, который можно объединить с другим объектным файлом, и таким образом получить исполняемый файл. Проблема с вызовом функции puts() разрешается именно на этапе связывания. Помните, в stdio.h функция puts() была объявлена как внешняя функция? Это и означает, что функция будет определена (или реализована) в другом месте. Если бы у нас было несколько исходных файлов нашей программы, мы могли бы объявить некоторые функции как внешние и реализовать их в различных файлах; такие функции можно использовать в любом месте нашего кода, ведь они объявлены как внешние. До тех пор, пока компилятор не знает, откуда берется реализация такой функции, в получаемом коде лишь остается ее "пустой" вызов. Линковщик разрешит все эти зависимости и в процессе работы подставит в это "пустое" место реальный адрес функции.

Линковщик выполняет также и другую работу. Он соединяет нашу программу со стандартными процедурами, которые будут запускать нашу программу. К примеру, есть стандартная последовательность команд, которая настраивает рабочее окружение, например, принимает аргументы командной строки и переменные системного окружения. В завершении программы должны также присутствовать определенные операции, чтобы помимо всего прочего

программа могла возвратить код ошибки. Очевидно, эти стандартные операции порождают немалое количество кода. На рисунке 2 наглядно представлены этапы создания, компиляции и запуска приложения.

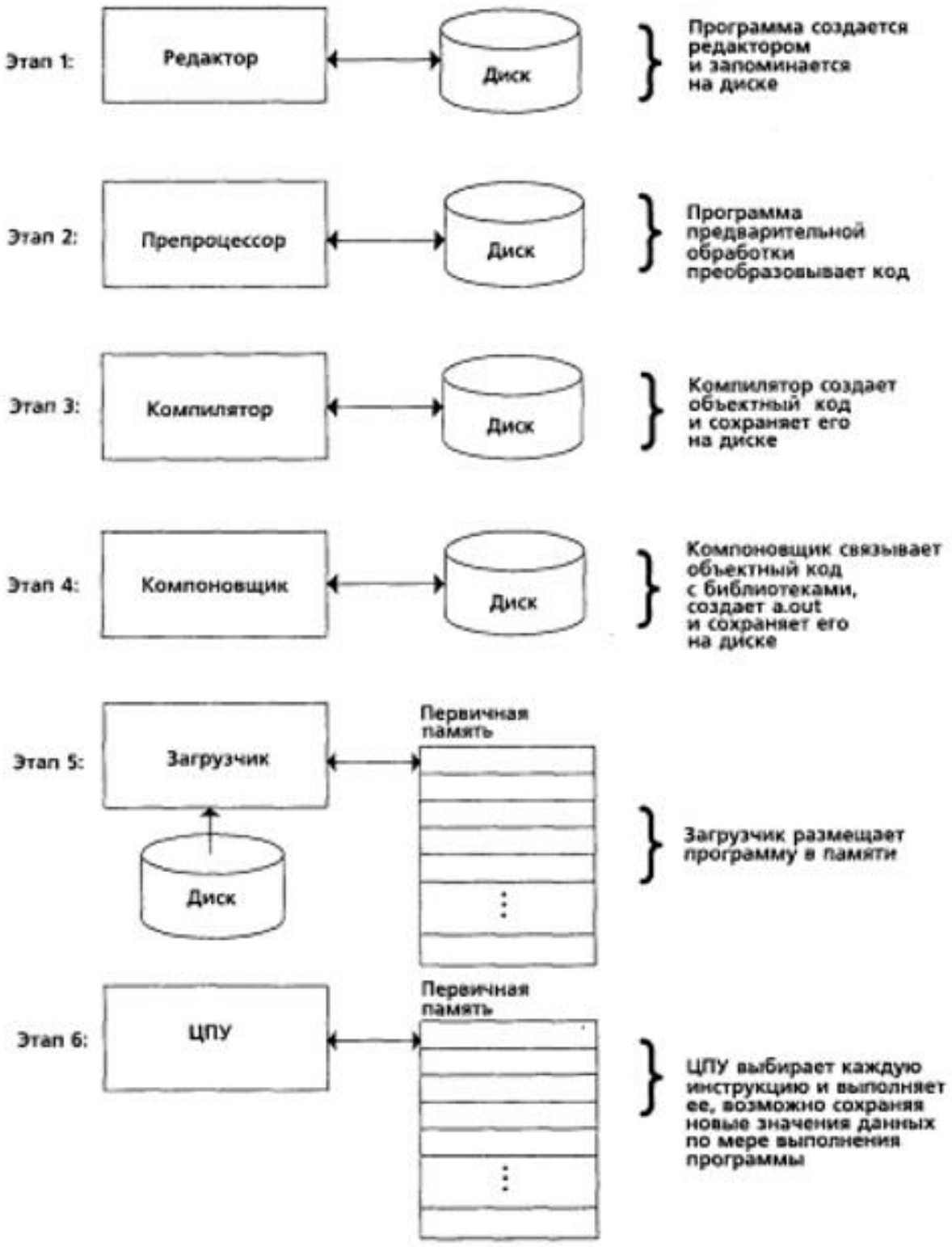


Рисунок 2 – Этапы создания и запуска программы

Простейшая автосборка оболочкой Shell

Сборкой называется процесс подготовки программы к непосредственному использованию. Простейший пример сборки – компиляция и компоновка. Более сложные проекты могут также включать в себя дополнительные промежуточные этапы (операции над файлами, конфигурирование и т. п.). Отличием скриптовых языков программирования, в том, что они позволяют запускать программы сразу после подготовки исходного кода, минуя стадию сборки. Примером такого языка является популярный Python.

Собирать программы вручную неудобно, поэтому программисты, как правило, прибегают к различным приемам, позволяющим автоматизировать этот процесс. Самый простой способ – написать сценарий оболочки (shell-скрипт), который будет автоматически выполнять все то, что вы обычно вводите вручную.

Пример, как можно собрать многофайловый проект при помощи скрипта **makeprintup_script**:

```
#!/bin/sh
gcc -c print_up.c
gcc -c main.c
gcc -o printup print_up.o main.o
```

Осталось только вызвать скрипт makeprintup_script, и проект будет создан:

```
$ ./ makeprintup_script
```

Перечислим некоторые проблемы использования скриптов для сборки проектов:

- Скрипты оболочки статичны. Их работа не зависит от состояния текущей задачи. Даже если нужно заново откомпилировать только один файл, скрипт будет собирать проект "с нуля".
- В скриптах плохо просматриваются связи между различными элементами проекта.
- Скрипты не обладают возможностью самодиагностики.

К счастью, в наличии имеется достаточно большой арсенал специализированных средств для автоматической сборки программных проектов. Такие средства называют автосборщиками или утилитами автоматической сборки. Благодаря специализированным автосборщикам программист может сосредоточиться, собственно, на программировании, а не на процессе сборки.

Простейшая автосборка утилитой make

Утилита make (GNU make) – наиболее популярное и проверенное временем средство автоматической сборки программ в Linux. Даже "гигант" автосборки, пакет GNU Autotools, является лишь *надстройкой* над make.

Автоматическая сборка программы на языке C обычно осуществляется по следующему алгоритму.

1. Подготавливаются исходные и заголовочные файлы.

2. Подготавливаются make-файлы, содержащие сведения о проекте. Порой даже крупные проекты обходятся одним make-файлом. Вообще говоря, make-файл может называться, как угодно, однако обычно выбирают одно из трех стандартных имен (Makefile, makefile или GNUmakefile), которые распознаются автосборщиком автоматически.

3. Вызывается утилита make, которая собирает проект на основании данных, полученных из make-файла. Если в проекте используется нестандартное имя make-файла, то его нужно указать после опции -f при вызове автосборщика.

Разработчики GNU make рекомендуют использовать имя Makefile. В этом случае у вас больше шансов, что make-файл будет стоять обособленно в отсортированном списке содержимого репозитория.

Базовый синтаксис Makefile

Чтобы работать с make, необходимо создать файл с именем Makefile. В make-файлах могут присутствовать следующие конструкции:

- **Комментарии**. В make-файлах допустимы однострочные комментарии, которые начинаются символом # (решетка) и действуют до конца строки.
- **Объявления констант**. Константы в make-файлах служат для подстановки. Они во многом схожи с константами препроцессора языка C.
- **Целевые связки**. Эти элементы несут основную нагрузку в make-файле. При помощи целевых связок задаются зависимости между различными частями программы, а также определяются действия, которые будут выполняться при сборке программы. В любом make-файле должна быть хотя бы одна целевая связка.

В Makefile обязательны только целевые связки.

Каждая целевая связка состоит из следующих компонентов:

- Имя цели. Если целью является файл, то указывается его имя. После имени цели следует двоеточие.
- Список зависимостей. Здесь просто перечисляются через пробел имена файлов или имена промежуточных целей. Если цель ни от чего не зависит, то этот список будет пустым.
- Инструкции. Это команды, которые должны выполняться для достижения цели.

Рассмотрим пример. Сначала создаем Makefile программы printup:

```
# Makefile for printup
printup: print_up.o main.o
	gcc -o printup print_up.o main.o

print_up.o: print_up.c print_up.h
	gcc -c print_up.c
main.o: main.c
	gcc -c main.c

clean:
	rm -f *.o
	rm -f printup
```

Строки после комментария, это целевые связки.

Первая связка отвечает за создание исполняемого файла printup и формируется следующим образом:

1. Сначала записывается имя цели (printup).
2. После двоеточия перечисляются зависимости (print_up.o и main.o).
3. На следующей строке после **знака табуляции** пишется правило для получения бинарника printup.

Аналогичным образом оформляются остальные целевые связки.

Вызовем утилиту make с указанием цели, которую нужно достичь. В нашем случае это будет выглядеть так:

```
$ make printup
gcc -c print_up.c
gcc -c main.c
gcc -o printup print_up.o main.o
```

Итак, проект собран.

Вообще говоря, при запуске make имя цели можно не указывать. Тогда основной целью будет считаться первая цель в Makefile. Следовательно, в нашем случае, чтобы собрать проект, достаточно вызвать make без аргументов:

```
$ make
gcc -c print_up.c
gcc -c main.c
gcc -o printup print_up.o main.o
```

Целевая связка (**clean**) требует особого рассмотрения:

1. Сначала указывается имя цели (clean).
2. После двоеточия следует пустой список зависимостей. Это значит, что данная связка не требует наличия каких-либо файлов и не предполагает предварительного выполнения промежуточных целей.
3. На следующих двух строках прописаны инструкции, удаляющие объектные файлы и бинарник.

Эта цель очищает проект от всех файлов, автоматически созданных при сборке. Итак, чтобы очистить проект, достаточно набрать следующую команду:

```
$ make clean
rm -f *.o
rm -f printup
```

Очистка проекта обычно выполняется в следующих случаях:

- при подготовке исходного кода к отправке конечному пользователю или другому программисту, когда нужно избавить проект от лишних файлов;
- при изменении или добавлении в проект заголовочных файлов;
- при изменении make-файла.

Иногда требуется вписать в make-файл нечто длинное, например инструкцию, не умещающуюся в одной строке. В таком случае строки условно соединяются символом \ (обратная косая черта):

```
gcc -Wall -pedantic -g -o my_very_long_output_file one.o two.o \
three.o four.o five.o
```

Автосборщик при обработке make-файла будет интерпретировать такую конструкцию как единую строку.

В следующей лабораторной работе #23 мы подробно рассмотрим *расширенные* возможности наиболее популярного и проверенного временем средства автоматической сборки программ в Linux, утилиту make.

РАЗДЕЛ #2 – МНОГОЗАДАЧНОСТЬ ПРОЦЕССОВ

Абстракция процесса

В основе многозадачности лежит понятие «процесс». *Процесс* – это нечто, совершающее в системе какие-либо действия. В основе программного процесса лежит понятие «*программа*». Программу можно рассматривать с нескольких сторон. Первое состояние программы, это «исходный текст», второе состояние, это «исполняемый файл», и третье состояние, это «процесс». Компьютерная программа сама по себе – лишь пассивная последовательность инструкций. В то время как процесс – непосредственное выполнение этих инструкций.

Исполняемый файл программы долгое время может храниться на жестком диске компьютера без дела. Но как только пользователь запускает программу она загружается в *оперативную память* с помощью ядра операционной системы. В оперативной памяти для программы выделяется *виртуальное адресное пространство*, и в определенный момент инструкции программы отправляются на *процессор* компьютера и там происходит их выполнение. Так программа становится процессом. Итак, процессом называют выполняющуюся программу и все её элементы: адресное пространство, глобальные переменные, регистры, стек, открытые файлы и так далее.

Три абстракции (состояния) программы:

1. **Исходные файлы** программы, тексты.
2. **Исполняемые файлы** программы. Скомпилированная программа.
3. **Процесс**. Программа запущенная на исполнение.

Многозадачность

В основе многозадачности лежит понятие "процесс". Процесс – это запущенная работающая программа. Один процесс может инициировать (породить) другой. В таком случае породивший процесс называют родителем или родительским процессом, а порожденный – потомком или дочерним процессом. Все процессы в системе – элементы одной иерархии, называемой деревом процессов. На вершине этой иерархии находится процесс init. Это единственный процесс, не имеющий родителя.

Каждый раз, когда вы запускаете в командной строке какую-нибудь программу, например ls, в системе рождается новый процесс, для которого процесс «командная оболочка» является родителем.

Иерархия процессов

Во многих операционных системах вся информация каждого процесса, дополняющая содержимое его собственного адресного пространства, хранится в таблице операционной системы. Эта таблица называется таблицей процессов и представляет собой массив (или связанный список) структур, по одной на каждый существующий в данный момент процесс.

Процесс может инициировать (породить) несколько других процессов (называющихся дочерними), а эти процессы, в свою очередь, тоже могут создать дочерние процессы. В таком случае породивший процесс называют родителем или родительским процессом, а порожденный – потомком или дочерним процессом. Все процессы в операционной системе – элементы одной иерархии, называемой деревом процессов (рисунок 3). Процессы могут быть связаны между собой. *Связанные процессы* – это процессы, которые объединены для решения некоторой задачи, и им нужно часто передавать данные от одного к другому и синхронизировать свою деятельность. Такое взаимодействие называется *межпроцессным*.

На вершине этой иерархии находится процесс **init**. Это единственный процесс, не имеющий родителя. Каждый раз, когда вы запускаете в командной строке какую-нибудь программу, например **ls**, в системе рождается новый процесс, для которого командная оболочка является родителем.

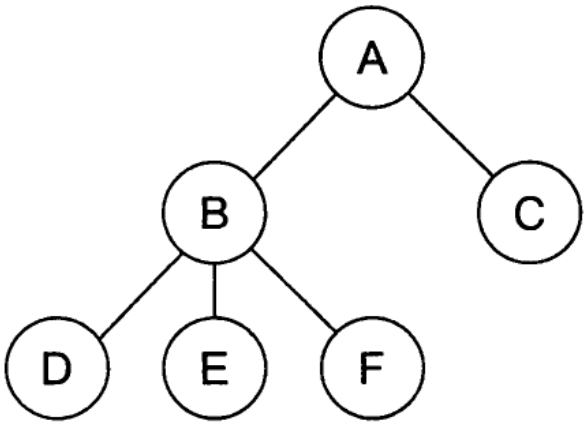


Рисунок 3- Дерево процессов.
Процесс A создал два дочерних процесса B и C.
Процесс B создал три дочерних процесса D, E и F

Окружение и его переменные

Окружение (environment) – это определяемое пользователем значения, которые могут влиять на поведение запущенных процессов на компьютере.

Переменная окружения (*среды*), это набор специфичных для конкретного пользователя пар ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ, с помощью которых можно передавать в программу какие-нибудь данные общего назначения.

Переменные окружения являются частью среды, в которой выполняется процесс. Например, запущенный процесс может запросить значение переменной среды TEMP, чтобы найти подходящее место для хранения временных файлов, или переменную HOME или USERPROFILE, чтобы найти структуру каталогов, принадлежащую пользователю, запускающему процесс.

Каждый процесс располагает и свободно распоряжается своей копией окружения, которую он получает от родителя. Таким образом, если процесс сначала модифицирует свое окружение, а потом породит новый процесс, то потомок получит копию уже измененного окружения. Чтобы посмотреть, что представляет собой окружение в действительности, введите следующую команду: `$ env`

На экран будет выведено окружение программы env. Можете не сомневаться, это точная копия окружения вашей командной оболочки.

В современных Linux-дистрибутивах окружение представлено десятками переменных, но широко используются только некоторые из них:

USER — имя пользователя;
HOME — домашний каталог;

PATH — список каталогов, в которых осуществляется поиск исполняемых файлов программ;
SHELL — используемая командная оболочка;
PWD — текущий каталог.

Следующий шаблон позволяет посмотреть значение конкретной переменной:

```
$ echo $VARIABLE
```

Здесь VARIABLE – это имя переменной. Оболочка вместо записи $VARIABLE всегда подставляет значение соответствующей переменной окружения.

Рассмотрим пример: `$ touch $USER`

Вместо $USER здесь подставляется значение переменной. Такая команда создаст пустой файл, присвоив ему имя текущего пользователя.

Пользователь может *модифицировать* окружение командной оболочки. В разных оболочках это делается по-разному. Например, оболочка **bash**, помимо свойственного любому процессу окружения, поддерживает также свой собственный набор переменных, которые называют *локальными переменными оболочки*.

Рассмотрим пример:

```
$ MYVAR=Hello
$ echo $MYVAR
$ env | grep MYVAR
$
```

Первой командой создается и инициализируется локальная переменная оболочки MYVAR. Следующая команда выводит значение этой переменной. Третья команда ищет переменную MYVAR среди окружения программы env (точная копия окружения оболочки). Но поскольку MYVAR не является переменной окружения, то последняя команда ничего не выводит.

Как видно из этого примера, локальные переменные "работают" только на уровне оболочки. Не являясь частью окружения, они не могут наследоваться дочерними процессами. Поэтому программа **env** не нашла переменную MYVAR. Но иногда требуется включить (экспортировать) локальную переменную в окружение. Для этого в оболочке **bash** есть внутренняя (не являющаяся отдельной программой) команда export:

```
$ export MYVAR
$ env | grep MYVAR
MYVAR=Hello
```

Команду export можно также использовать для инициализации и экспорта переменной в один прием:

```
$ export MYVAR_NEW=Goodbye
$ env | grep MYVAR_NEW
MYVAR_NEW=Goodbye
```

Важно понимать, что полученные переменные MYVAR и MYVAR_NEW существуют только в текущем окружении процесса оболочки и ее потомков. Очевидно, что после перезагрузки эти переменные "исчезнут", если их, например, не объявить в файле инициализации bash. Даже такая жизненно важная переменная, как PATH, не появляется в окружении оболочки сама собой: строка для ее инициализации обычно находится в файле **/etc/profile**.

Исключить переменную из окружения оболочки позволяет команда export -n. При этом остается локальная переменная. Когда же надо удалить переменную отовсюду, вызывают команду unset:

```
$ env | grep -i myvar
MYVAR_NEW=Goodbye
MYVAR=Hello
$ export -n MYVAR
$ env | grep MYVAR
MYVAR_NEW=Goodbye
$ echo $MYVAR
Hello
$ unset MYVAR_NEW
$ env | grep MYVAR
$ echo -n $MYVAR_NEW
$
```

> Важно понимать, что export, export -n и unset – это не отдельные программы, а внутренние команды bash. Оболочки ksh (KornShell) и zsh (Z shell), например, не поддерживают опцию -n команды export. Оболочки семейства C-Shell (csh, tcsh) работают с окружением совсем другими командами setenv и unsetenv.

Процессы в Linux

В Linux каждому работающему процессу соответствует уникальное положительное целое число, которое называется *идентификатором процесса* и часто обозначается аббревиатурой PID (Process IDentifier).

К процессу также привязано еще одно положительное целое число – *идентификатор родительского процесса*, которое часто обозначается аббревиатурой PPID (Parent Process IDentifier).

Мы уже говорили о том, что на вершине дерева процессов находится процесс init. Идентификатор этого процесса всегда равен 1. Родителем init условно считается процесс с идентификатором 0, но фактически init не имеет родителя.

Для получения информации о процессах предназначена программа **ps**, поддерживающая большое количество опций. Некоторые из этих опций являются стандартными для Unix-подобных систем, другие зависят от конкретной реализации ps. В Linux обычно установлена программа ps из пакета procps.

Если вызвать ps без аргументов, то на экране появится список процессов, запущенных под текущим терминалом:

```
$ ps
  PID TTY          TIME CMD
25164 pts/0    00:00:00 bash
26310 pts/0    00:00:00 ps
```

Появилась таблица, состоящая из четырех столбцов. Первый столбец (PID) – это идентификатор процесса. В четвертом столбце (CMD) записывается имя исполняемого файла программы, запущенной внутри процесса. В нашем случае под текущим терминалом работают два процесса: командная оболочка и, собственно, программа ps.

Если выполнить приведенную команду еще раз, то можно увидеть некоторые изменения:

```
$ ps
  PID TTY          TIME CMD
25164 pts/0    00:00:00 bash
26444 pts/0    00:00:00 ps
```

В данном случае изменился PID процесса для программы ps. Всякий раз, когда в системе рождается новый процесс, ядро Linux автоматически выделяет для него уникальный идентификатор.

Идентификаторы процессов могут через какое-то время повторяться, но два одновременно работающих процесса не могут иметь одинаковый идентификатор. Если идентификаторы процессов достигли максимального числа в системе, то следующему процессу присваивается минимальное число доступных пользовательских процессов в системе. Присваивание идентификаторов идет по кругу.

Если теперь запустить под оболочкой какую-нибудь программу, то она будет связана с текущим терминалом и появится в выводе программы ps. Чтобы продемонстрировать это, запустим программу yes в фоновом режиме, перенаправив ее стандартный вывод на устройство /dev/null:

```
$ yes > /dev/null &
[1] 26600
$ ps
  PID TTY          TIME CMD
25164 pts/0    00:00:00 bash
26600 pts/0    00:00:05 yes
26618 pts/0    00:00:00 ps
```

Если теперь ничего не трогать, то даже через год процесс yes будет усердно пересылать поток символов "y" в "никуда". Чтобы предотвратить эту бессмыслицу, выведем процесс из фонового режима и закончим его комбинацией клавиш <Ctrl>+<C>:

```
$ fg yes
yes > /dev/null
^C
$ ps
  PID TTY          TIME CMD
25164 pts/0    00:00:00 bash
26729 pts/0    00:00:00 ps
```

Если вызвать программу ps с опцией -f, то вывод пополнится несколькими новыми столбцами:

```
$ ps -f
UID          PID    PPID  C    STIME    TTY          TIME     CMD
nnivanov   25164   25161  0    13:43    pts/0    00:00:00 bash
nnivanov   26790   25164  0    13:59    pts/0    00:00:00 ps -f
```

Столбец UID (User IDentifier) в расширенном выводе программы ps содержит имя пользователя, от лица которого запущен процесс, в столбце PPID выводится идентификатор родительского процесса.

Если запустить программу ps с флагом -e, то будет выведен полный список работающих в системе процессов. Многие флаги программы ps можно комбинировать. Например, следующие две команды эквивалентны и выводят полный список процессов в расширенном виде:

`$ ps -e -f` или `$ ps -ef`

Чтобы получить сводку опций программы ps, просто введите команду:

```
$ ps –help
```

Дерево процессов

Если вызывать программу ps с флагами -e и -H, то можно увидеть дерево процессов в наглядной форме:

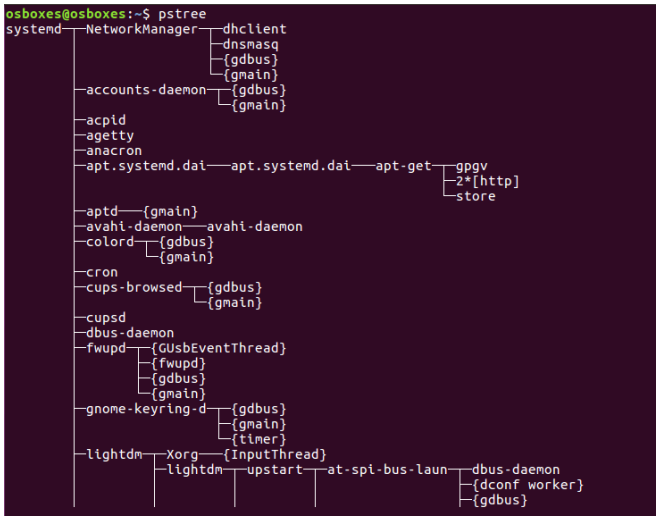
```
root         987  0.0  0.4  31956  4332 ?        Ss   08:52   0:00 /usr/lib/blueto
root        1046  0.0  0.1  17276  1812 tty1     Ss+  08:52   0:00 /sbin/agetty --
root        1050  0.0  0.8 143176  8612 ?        Ss   08:52   0:00 /usr/sbin/mdm -
root        1059  0.0  0.9 193244  9420 ?        S    08:52   0:00  \_ /usr/sbin/m
root        1065  0.6  8.7 411968 88656 tty7     Ss+  08:52   0:03      \_ /usr/li
student     1097  0.0  4.1 520592 42488 ?        Ssl  08:52   0:00      \_ cinnamo
student     1213  0.0  0.0  11140   316 ?        Ss   08:52   0:00          \_ /us
student     1259  0.1  5.0 1171748 51564 ?       Sl   08:52   0:00          \_ /us
student     1358  0.0  2.9 188440 29708 ?        S    08:52   0:00          \_ /us
student     1375  4.7 21.2 1354556 215544 ?      Sl   08:52   0:25          |   \_
student     1382  0.1  5.3 819240 54180 ?        Sl   08:52   0:00          \_ nem
student     1383  0.2  3.1 522284 32472 ?        Sl   08:52   0:01          \_ nm-
student     1384  0.0  2.8 413336 29332 ?        Sl   08:52   0:00          \_ /us
student     1385  0.0  1.9 327076 19700 ?        Sl   08:52   0:00          \_ /us
student     1387  0.0  0.5  25880  6000 ?        S    08:52   0:00          \_ /us
student     1396  0.0  0.0   4508   704 ?        S    08:52   0:00          |   \_
student     1397  0.0  4.3 629264 43824 ?        Sl   08:52   0:00          |
student     1418  0.0  0.0   4364   728 ?        S    08:52   0:00          |
student     1509  0.0  1.9 407736 20088 ?        Sl   08:52   0:00          \_ cin
```

Если ваша терминальная программа использует пропорциональный шрифт, то следующая команда выведет дерево процессов с соединительными ветвями:

```
$ ps -e --forest
```

> Пропорциональным называется шрифт, каждый символ которого занимает на экране прямоугольник фиксированного размера. Чтобы выяснить, работает ли ваша терминальная программа с пропорциональным шрифтом, просто запустите Midnight Commander; если меню и панели этой программы отображаются нормально (в виде ровных прямоугольников), то терминальная программа задействует пропорциональный шрифт.

Если в вашей системе установлен пакет программ psmisc, то дерево процессов можно увидеть при помощи программы pstree. Существуют также графические утилиты для просмотра дерева процессов: ksysguard, gnome-system-monitor и т.д.



Возникает вопрос: что произойдет с работающим потомком при завершении родительского процесса?

Родительским процессом для «осиротевшего» потомка становится процесс init и PPID «сироты» становится равным 1 (это PID процесса init).

Многозадачность и операции ввода-вывода

Вспомним ненадолго окружение. Мы знаем, что оно наследуется (копируется) от родительского процесса к дочернему. Примерно то же самое происходит и с файлами: таблица дескрипторов при запуске процесса заполняется открытыми файлами родителя. Но, в отличие от окружения, которое в каждом процессе представлено независимой копией, наследование дескрипторов обладает двумя особенностями: файл не будет закрыт до тех пор, пока его не закроют все процессы, в которых он открыт. Иначе говоря, если процесс A с открытым файлом F породил процесс B, то для закрытия файла F необходимо, чтобы оба процесса (A и B) закрыли этот файл. Обратите внимание, что это условие не является достаточным, поскольку файл F мог также достаться процессу A от его родителя; если процессы разделяют общий файловый дескриптор, то они разделяют также и текущую позицию ввода-вывода.

Получение информации о процессе

Каждому процессу привязаны следующие важные данные:

- идентификатор процесса (PID);
- идентификатор родительского процесса (PPID);
- идентификатор пользователя (UID).

Есть и другие данные, которыми располагает процесс, но об этом позже.

Получить PID, PPID и UID текущего процесса можно с помощью следующих системных вызовов, объявленных в заголовочном файле unistd.h:

- pid_t getpid (void);
- pid_t getppid (void);
- uid_t getuid (void);

Типы данных pid_t и uid_t являются целыми числами, размерность которых зависит от реализации. Чтобы использовать эти типы, нужно включить в программу заголовочный файл sys/types.h.

Возникает вопрос, почему последний из этих вызовов возвращает целое число? Снова заглянем в файл /etc/passwd, где хранится информация обо всех пользователях системы. Каждая строка файла /etc/passwd – это запись, соответствующая конкретному пользователю системы. В свою очередь каждая запись разделяется двоеточиями на семь полей. Третье поле, это идентификатор пользователя, то есть то самое уникальное для каждого пользователя системы число, которое возвращает системный вызов getuid().

> В Unix-подобных системах каждому имени пользователя соответствует один (и только один) числовой идентификатор. В широком смысле под идентификатором пользователя (UID) может пониматься не только число, но и имя, однозначно соответствующее этому числу. Так, например, программа ps в столбце UID выводит имя пользователя вместо числа. Получение числового идентификатора из имени пользователя называется разрешением имени пользователя (username resolution).

Чтобы узнать числовой UID текущего пользователя, можно выполнить следующую команду: `$ id -u`

Можно также узнать UID любого пользователя системы: `$ id -u USERNAME`

UID суперпользователя (с именем root в подавляющем большинстве случаев) всегда равен 0: `$ id -u root`

Для получения имени пользователя из числового UID применяется библиотечная функция getpwuid(), объявленная в заголовочном файле pwd.h:

```
struct passwd * getpwuid (uid_t UID);
```

Базовая многозадачность

Для реализации базовой многозадачности реализованы две концепции: это развилки – **fork()** и передача управления – **execve()**.

Для порождения нового процесса предназначен *системный вызов* **fork()**, объявленный в заголовочном файле unistd.h:

```
pid_t fork (void);
```

Системный вызов fork() порождает процесс методом "*клонирования*". Это значит, что новый процесс является *точной копией* своего родителя и выполняет ту *же самую программу*.

Различаются — только идентификатором процесса PID.

Процессы в Linux работают на самом деле не одновременно. Ядро периодически дает возможность каждому процессу передать свой исполняемый код процессору (или процессорам, если их несколько). Эти переключения обычно происходят настолько быстро, что у нас создается иллюзия одновременной работы нескольких программ.

За переключение процессов отвечают специальные алгоритмы, находящиеся в ядре Linux. Считается, что пользователи и

программисты должны условно считать алгоритмы переключения процессов *непредсказуемыми*.

Иными словами, процессы в Linux работают *независимо друг от друга*. Подобная договоренность позволяет ядру Linux "*интеллектуально*" распределять системные ресурсы между процессами, повышая производительность системы в целом.

Чтобы два «родственных» процесса не выполняли одно и тоже действие, в системном программировании есть *механизм передачи управления процессу*.

Системный вызов execve() загружает в процесс другую программу и передает ей безвозвратное управление. Этот системный вызов объявлен в заголовочном файле unistd.h:

```
int execve (const char * PATH, const char ** ARGV, const char ** ENVP);
```

Этот системный вызов принимает три аргумента:

- PATH – это путь к исполняемому файлу программы, которая будет запускаться внутри процесса. Здесь следует учитывать, что одноименная переменная окружения в системном вызове execve() не используется. В результате ответственность за поиск программы возложена только на программиста.
- ARGV — это уже знакомый нам массив аргументов программы. Здесь важно помнить, что первый аргумент (ARGV[0]) этого массива является именем программы или чем-то другим (на ваше усмотрение), но не фактическим аргументом. Последним элементом ARGV должен быть NULL.
- ENVP — это тоже уже знакомый нам массив, содержащий окружение запускаемой программы. Этот массив также должен заканчиваться элементом NULL. Позже будут описаны надстройки над execve(), позволяющие использовать переменную окружения PATH для запуска программы.

Выполняющийся внутри процесса код называется *образом процесса* (process image). Важно понимать, что системный вызов execve() заменяет текущий образ процесса на новый. Следовательно, возврата в исходную программу не происходит. В случае ошибки execve() возвращает -1, но если новая программа начала выполняться, то execve() уже ничего не вернет, поскольку работа исходной программы в текущем процессе на этом заканчивается.

Учитывая, что execve() не может возвращать ничего кроме -1, следующая проверка будет явно избыточной:

```
if (execve (binary_file, argv, environ) == -1) { /* обработка ошибки */ }
```

Целесообразнее более простая форма обнаружения ошибки:

```
execve (binary_file, argv, environ);
/* обработка ошибки */
```

Совместное использование системных вызовов fork() и execve() позволяет запускать программы в отдельных процессах, что чаще всего и требуется от программиста.

Семейство exec()

В стандартной библиотеке языка C есть пять дополнительных функций, которые реализованы с использованием execve() и вместе называются семейством exec(). Все функции семейства exec() объявлены в заголовочном файле unistd.h:

```
int execl (const char * PATH, const char * ARG, ...);
int execle (const char * PATH, const char * ARG, ..., const char ** ENVP);
int execlp (const char * FILE, const char * ARG, ...);
int execv (const char * PATH, const char ** ARGV);
int execvp (const char * FILE, const char ** ARGV);
```

Системный вызов execve() является "лидером" и полноправным членом семейства exec(). Напомним еще раз его прототип:

```
int execve (const char * PATH, const char ** ARGV, const char ** ENVP);
```

Универсальный шаблон всех функций семейства exec():

```
execX[Y] (...);
```

Имя функции формируется добавлением к префиксу exec обязательного X и произвольного Y. В качестве X могут выступать символы l и v (без кавычек), а Y может принимать значения e и p. В таблице 22.1 представлено назначение символов в имени exec()

> Для вас не должно иметь значения, является ли конкретный представитель семейства **exec()** библиотечной функцией или системным вызовом. Выбор конкретной формы **exec()** должен зависеть не от особенностей реализации, а от поставленной задачи.

Таблица 22.1. Назначение символов в имени exec()

Символ	Наличие	Комментарий
l (эль)	Есть	Аргументы программы передаются не в виде массива ARGV, а в виде отдельных аргументов функции exec(), список которых заканчивается аргументом NULL
	Нет	Набор аргументов запускаемой программы передается в виде массива строк ARGV
v	Есть	Аргументы программы передаются в виде единого массива строк ARGV, последним элементом которого является NULL
	Нет	Вместо "v" присутствует символ "l", который задает список программы в виде отдельных аргументов-строк, список которых заканчивается аргументом NULL
e	Есть	В качестве последнего аргумента используется массив строк ENVP (окружение будущей программы)
	Нет	Начальным окружением запускаемой программы будет окружение текущего процесса
p	Есть	В качестве первого аргумента используется имя исполняемого файла программы, поиск которого будет производиться в каталогах, перечисленных в переменной окружения PATH
	Нет	В качестве первого аргумента выступает полный путь к исполняемому файлу программы (относительно текущего и корневого каталога)

Best of LUCK with it, and remember to HAVE FUN while you're learning :)
Sergey Stankewich

УПРАЖНЕНИЯ

Упражнение 1

Системный вызов fork () и поэтапная компиляция

Системное программирование предоставляет возможность запускать программы из других программ. Запустим редактор, наберем следующий код и скомпилируем программу:

procexample.c

```
#include <stdlib.h>
char *procstr="sh -c ./timeexample";

  void main()
{
   system(procstr);
   exit(0);
}
```

Здесь мы используем *команду* system для запуска внешней программы. Вместо procstr нужно указать имя запускаемого процесса. Запустим календарь:

```
#include <stdlib.h>
char procstr[]="cal";

    void main()
   {
     system(procstr);
     exit(0);

   }
```

Также попробуйте запустить *браузер*, или *текстовый редактор*, или *проводник*, или *калькулятор* если они есть в вашей системе.

Операционная система позволяет породить параллельную ветвь программы с помощью команды fork().

Рассмотрим сначала небольшой пример (листинг 22.1):

Листинг 22.1 (10.1). Программа fork1.c

```
#include <unistd.h>
#include <stdio.h>

int main (void) {
        fork ();
        printf ("I'm lake Linux\n");
        sleep (15);
        return 0;
}
```

Компилируем: `$ gcc -o fork1 fork1.c`

Передаем содержимое в файл

`$ ./fork1 > output &`

Выполните команды:

`$ ps`

…

`$ cat output`

объясните полученный результат.

Программа **fork1** породила новый процесс, о чем свидетельствует вывод программы ps. Сразу после вызова fork() каждый процесс продолжил самостоятельно выполнять одну и ту же программу fork1. Этим и объясняется наличие двух приветствий " I'm lake Linux" в файле output.

Рассмотрим пример (листинг 22.2), в котором родительский и дочерний процессы выполняют разные действия. Чтобы в контексте программы отделить один процесс от другого, достаточно знать, что системный вызов fork() возвращает в текущем процессе PID порожденного потомка (или -1 в случае ошибки). А в новый процесс fork() возвращает 0.

Листинг 22.2 (10.2). Программа fork2.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main (void) {

pid_t result = fork();
if (result == -1) { fprintf (stderr, "Error\n");
return 1;
}
if (result == 0)
printf ("I'm child with PID=%d\n", getpid ());
else
printf ("I'm parent with PID=%d\n", getpid ());

return 0;
}
```

Компилируем и выполняем программу:

```
$ ./fork2
```

Запустите программу несколько раз. Обратите внимание как изменяется PID процессов и в какой последовательности они выводятся.

В приведенном примере сообщение родителя может появиться первым либо вторым. Как уже говорилось ранее, за переключение процессов отвечают специальные алгоритмы, находящиеся в ядре Linux. Для пользователей и программистов алгоритмы переключения процессов непредсказуемыми, процессы в Linux работают **независимо друг от друга**.

Рассмотрим пример (листинг 22.3), демонстрирующий "непредсказуемость" переключения процессов в Linux.

Листинг 23.4 (10.3). Программа race1.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h>
#define WORKTIME      3

int main (void) {
        unsigned long parents = 0;
        unsigned long children = 0;
        pid_t result;
    time_t nowtime = time (NULL);
    result = fork ();
    if (!result) {
        while (time (NULL) < nowtime+WORKTIME) chil-
        dren++; printf ("children: %ld\n", children);
    }
    else { while (time (NULL) < nowtime+WORKTIME) par-
    ents++; printf ("parents: %ld\n", parents);
    }
    return 0;
}
```

Компилируем и выполняем программу:

```
$ gcc -o race1 race1.c
$ ./race1
```

Запустите программу несколько раз. Обратите внимание как изменяется PID процессов и в какой последовательности они выводятся.

Если произойдет случай, при котором "родитель" закончиться чуть раньше и оболочка "выкинет" приглашение командной строки перед отчетом процесса-потомка, то вы получили доказательство того, что *процессы работают независимо*.

Рассмотрим пример программы в которой один процесс (дочерний или родительский) выполняется, пока не закончится (листинг 22.4.а). Запустите программу и посмотрите какой получился результат и объясните его. Однако, чтобы дать возможность смены процессов, нужно как-то приостановить выполняющийся процесс. Этого можно добиться с помощью команды sleep(число_секунд). Вот, как мы изменили программу (листинг 22.4.б). Запустите обе программы программу и посмотрите какой получился результат и объясните его.

Листинг 22.4.а.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
    void main()
   { int sum=0;
     pid_t child_pid;
     child_pid= fork();
      while(sum<5)
      {if(child_pid !=0)
           {printf ("main process working! Sum is %d\n",sum);
            sum++;
           }
      else
           {
        printf ("child process working! Sum is %d\n",sum);
        sum++ ;
           }
     }
    }
```

Листинг 22.4.б.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
    void main()
   { int sum=0;
     pid_t child_pid;
     child_pid= fork();
      while(sum<5)
      {if(child_pid !=0)
           {printf ("main process working! Sum is %d\n",sum);
            sum++;
            sleep(1);
           }
      else
           {
        printf ("child process working! Sum is %d\n",sum);
        sum++ ;
        sleep(1);
           }
     }
    }
```

Попробуйте написать похожий скрипт для компиляции вашей программы:

```
#!/bin/sh
gcc -E test.c -o test.i
gcc -S test.i -o test.s
gcc -c test.s
gcc -o testprogram test.o
```

С помощью утилиты **ls -l** или *программы-проводника* посмотрите какие фалы появились в текущей директории. Далее с помощью команды **file** определите типы каждого файла, результаты подкрепите скриншотами в отчете.

Упражнение 2

Передача управления: системный вызов execve()

Рассмотрим программу (листинг 22.5), демонстрирующую работу системного вызова execve().

Листинг 22.5 (10.4). Программа execve1.c

```
#include <stdio.h>
#include <unistd.h>
extern char ** environ;

int main (void) {
    char * uname_args[] = {
    "uname", "-a", NULL
    };
    execve ("/bin/uname", uname_args, environ);
    fprintf (stderr, "Error\n");
    return 0;
}
```

Скомпилируйте программу программой make с использованием примерно вот такого мейк-файла **Makefile**:

```
execve1: execve1.c
        gcc -o $@ execve1.c
clean:
        rm -f execve1
```

```
$ make
...
$ ./execve1
```

Здесь мы заменили образ текущего процесса программой /bin/uname с опцией -a. Если программа была успешно вызвана, то сообщение "Error" не выводится. В качестве окружения мы воспользовались массивом environ.

Рассмотрим пример, который демонстрирует *важные особенности*: три аспекта работы системного вызова execve():

- обе программы выполнялись в одном и том же процессе;
- при помощи системного вызова execve() программе можно "подсунуть" любое окружение;
- в элементе argv[0] действительно может быть все, что угодно.

Необходимо создать в одном каталоге две программы. Первая (листинг 22.6) будет при помощи execve() запускать вторую программу (листинг 22.7).

Листинг 22.6 (10.5). Программа execve2.c

```
#include <stdio.h>
#include <unistd.h>

int main (void) {
char * newprog_args[] = { "Tee-hee!", "new-
prog_arg1", "newprog_arg2", NULL };

char * newprog_envp[] = { "USER=abrakadabra",
"HOME=/home/abrakadabra", NULL };

printf ("Old PID: %d\n", getpid ());
```

Листинг 22.7 (10.6). Программа newprog.c

```
#include <stdio.h>
#include <unistd.h>

extern char ** environ;

int main (int argc, char ** argv) {
int i;
printf ("ENVIRONMENT:\n");
for (i = 0; environ[i] != NULL; i++) printf ("envi-
ron[%d]=%s\n", i, environ[i]);
```

```
//продолжение Листинг 22.6

execve ("./newprog", newprog_args, new-
prog_envp);
fprintf (stderr, "Error\n");

return 0;
}
```

```
// продолжение Листинг 22.7

printf ("ARGUMENTS:\n");
for (i = 0; i < argc; i++) printf ("argv[%d]=%s\n", i,
argv[i]);

printf ("New PID: %d\n", getpid ());
return 0;
}
```

Здесь же в текущей директории создадим небольшой make-файл **Makefile**.

```
execve2: execve2.c newprog
	gcc -o $@ execve2.c

newprog: newprog.c
	gcc -o $@ $^

clean:
	rm -f newprog execve2
```

Соберите проект:

```
$ make
...
$ ./execve2
```

Объясните полученный результат.

Запускаемой программе можно передавать пустое окружение. Для этого достаточно указать NULL в третьем аргументе системного вызова execve(), (Листинг 22.8.).

Листинг 22.8 (10.8). Программа execve3.c

```
#include <stdio.h>
#include <unistd.h>

int main (void) {
	char * env_args[] = { "env", NULL };

	execve ("/usr/bin/env", env_args, NULL);
	fprintf (stderr, "Error\n");

	return 0;
}
```

Соберите программу с помощью утилиты **make** и запустите. Если ваша программа не нашла утилиту env, то решите проблему самостоятельно. Объясните полученный результат.

Совместное использование системных вызовов fork() и execve() позволяет запускать программы в отдельных процессах, что чаще всего и требуется от программиста.

Рассмотрим пример, демонстрирующий *многозадачность* "во всей красе!" (листинг 22.9).

Листинг 22.9 (10.9). Пример forkexec1.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

extern char ** environ;

int main (void) {
pid_t result;
char * sleep_args[] = { "sleep", "5", NULL };

result = fork ();
if (result == -1) { fprintf (stderr, "fork error\n");
return 1;
}
if (result == 0) {
execve ("/bin/sleep", sleep_args, environ);
fprintf (stderr, "execve error\n");
return 1; }
else {
fprintf (stderr, "I'm parent with PID=%d\n",
getpid());
}

return 0;
}
```

Соберите программу с помощью утилиты **make** и запустите. Объясните полученный результат.

Если внимательно разобраться в программе, приведенной в листинге 22.8, то станет очевидным, что конструкция else является избыточной, поскольку дочерний процесс может только безвозвратно запустить другую программу или завершиться ошибкой. Все инструкции, находящиеся далее вызова execve(), могут выполняться только родительским процессом.

Закомментируйте избыточный код, перекомпилируйте программу, запустите, объясните полученный результат.

Упражнение 3
Семейство библиотечных функций exec()

Системный вызов execve() является "главой семейства". Выбор конкретной формы exec() должен зависеть не от особенностей реализации, а от поставленной задачи.

Рассмотрим примеры работы функций семейства exec() "на все случаи жизни". Поочередно для каждой функции семейства exec() реализуем фактически одну и ту же программу, которая будет запускать программу **ls** для подробного просмотра корневого каталога.

Напишем программу (листинг 22.9), которая при помощи execve() запускает ls с указанными ранее аргументами в отдельном процессе. Для уменьшения текста программы уберем различные проверки на ошибки.

Листинг 22.9 (10.10). Программа execvels.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

extern char ** environ;

int main (void) {
pid_t result;
result = fork ();
/* Child */
char *ls_args[] = { "ls", "-l", "/", NULL };
execve ("/bin/ls", ls_args, environ);

/* Parent */
return 0;
}
```

Соберите программу с помощью утилиты **make** и запустите. Объясните полученный результат.

Теперь реализуем ту же программу на основе execl() (листинг 22.10).

Листинг 22.10 (10.11). Программа execlls.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main (void) {
pid_t result;

result = fork ();
/* Child */
execl ("/bin/ls", "ls", "-l", "/", NULL);

/* Parent */
return 0;
}
```

Соберите программу с помощью утилиты **make** и запустите. Объясните полученный результат.

Использование execl() в нашем случае значительно упростило программу.

Но иногда требуется тот же execl(), но с возможностью передачи окружения в вызываемую программу. Для этого существует функция execle(), работа которой продемонстрирована в следующем примере (листинг 22.11).

Листинг 22.11 (10.12). Программа execlels.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

extern char ** environ;

int main (void) {
pid_t result;
result = fork ();
/* Child */
execle ("/bin/ls", "ls", "-l", "/", NULL, environ);

/* Parent */
return 0;
}
```

Соберите программу с помощью утилиты **make** и запустите. Объясните полученный результат.

Функции execve(), execl() и execle() вызывают программу, указывая абсолютный путь к ее исполняемому файлу. Это не всегда бывает хорошо. Предпочтителен вариант использования *переменной окружения* **PATH** для поиска указанного

исполняемого файла. Такую возможность предоставляет функция execlp(), что иллюстрирует листинг 22.12.

Листинг 22.12 (10.13). Программа execlpls.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main (void) {
pid_t result;

result = fork ();

/* Child */
execlp ("ls", "ls", "-l", "/", NULL);

/* Parent */
return 0;
}
```

Соберите программу с помощью утилиты **make** и запустите. Объясните полученный результат.

Функция execv(), рассмотренная далее (листинг 22.13), работает так же, как и execve(), только без возможности передачи "особенного" окружения.

Листинг 22.13 (10.14). Программа execvls.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main (void) {
pid_t result;
char *ls_args[] = { "ls", "-l", "/", NULL };

result = fork ();

/* Child */
execv ("/bin/ls", ls_args);

/* Parent */
return 0;
}
```

Соберите программу с помощью утилиты **make** и запустите. Объясните полученный результат.

Функция execvp() делает то же, что и execv(), но с возможностью поиска исполняемого файла программы в переменной окружения PATH (листинг 22.14).

Листинг 22.14 (10.15). Программа execvpls.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main (void) {
pid_t result;
char *ls_args[] = { "ls", "-l", "/", NULL };

result = fork ();

/* Child */
execvp ("ls", ls_args);

/* Parent */
return 0;
}
```

Соберите программу с помощью утилиты **make** и запустите. Объясните полученный результат.

We hope you enjoy working with Linux!

ЗАДАНИЯ

Задание 1

Используя материалы (исходные файлы) **упражнения №1** разработайте программу, в которой дочерний процесс запускается системным вызовом fork в цикле, ограниченном счетчиком проходов по циклу. Через пять секунд родительский процесс должен закончить работу, а дочерний процесс выдать команду **ps**. **Внимание**. Список выполняющихся процессов можно получить с помощью команды **$ps**.

Теперь нужно осознать, как осуществляется взаимодействие процессов. Стандартным образом для этой цели используется канал (pipe). Но сейчас мы осуществим взаимодействие через текстовый файл. *Создайте программу*, в которой один процесс будет писать в файл какую-то строку, а второй будет считывать.

Проведите компиляцию однофайлового проекта помощью скрипты **bash** с прохождением **всех стадий компиляции**.

Сборка проекта должна содержать файлы с результатами **препроцессинга**. Исследуйте фалы препроцессора, найдите в них код своей программы.

Определите **размеры** *исходных*, *препроцессорных, ассемблерных, объектных* и *исполняемых* файлов. С помощью соответствующей консольной команды или с помощь проводника определите **форматы этих файлов**. Результаты подтвердите скриншотами. Сделайте краткий вывод по выполненной работе.

Задание 2

Используя материалы (исходные файлы) **упражнения №2** проведите компиляцию проекта **листингов 22.5 и 22.6** с помощью утилиты **make** с прохождением **всех стадий компиляции** обеих программ.

Сборка проекта должна содержать файлы с результатами **препроцессинга**. Исследуйте фалы препроцессора, найдите в них код своей программы.

Определите **размеры** *исходных*, *препроцессорных, ассемблерных, объектных* и *исполняемых* файлов. С помощью соответствующей консольной команды или с помощь проводника определите **форматы этих файлов**. Результаты подтвердите скриншотами. Сделайте краткий вывод по выполненной работе.

Программа **листинга 22.8** должна запускать **браузер**, установленный в вашей системе, или текстовый **редактор**, или **проводник**, или другую, написанную

вами программу, а также можно чтобы ваша программа запускала и то и другое и третье.

Проявите возможности многозадачности «во всей красе»!

Задание 3

Используя материалы, исходные файлы **упражнения №3** (листинги 22.9 – 22.14) создайте программу запускающую «микросервисы»: программу-календарь, браузер, проводник, текстовый редактор, установленные в вашей системе.

С помощью очень полезной утилиты **strace**, запустите переданную ей вами разработанную программу и выведите в стандартный поток ошибок отчет об использованных системных вызовах. Полученный результат представьте в отчете и дайте объяснения.

«Easy things should be easy and hard things should be possible»
«Простые вещи должны быть простыми, а сложные вещи должны быть возможными»

Контрольные вопросы:

1) Перечислите основные инструментарии для разработки программ?
2) Что такое исходные файлы программы?
3) Что такое исполняемые файлы программы?
4) Какой командой можно определить тип файла? Определите типы файлов вашей сборки проекта?
5) Назовите основные стадии компиляции программы?
6) В чем различие компиляции и интерпритации программы?
7) Приведите известные вам примеры компилируемых и интерпритируемых языков программирования?
8) Что такое препроцессор и зачем он нужен?
9) В чем разница между понятиями «*аргумент функции*» и «*параметр функции*» ?
10) В чем разница использования двух форм директивы: **#include <filename>** или **#include “filename”**?
11) Что такое автосборка? Какие приемущества она дает?
12) Какой командой можно определить определить формат файла?
13) Что такое утилита **make**?
14) В чем разница автосборки скриптами **bash** и утилитой **make**?
15)

Дополнительная информация

Подробоно песочница придставлена к книге: Шоттс У. «Командная строка Linux. Полное руководство.» — СПб.: Питер, 2017. — 480 с.: ил. — (Серия «Для профессионалов»). на страницах 225-226.

Иванов Н.Н. И20 Программирование в Linux. Самоучитель. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2012. — 400 с.: ил. **Смотрите страницы 15 – 36, 58 – 65.**

Мэтью, Н. М97 Основы программирования в Linux: Пер. с англ. / Н. Мэтью, Р. Стоуне. — 4-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2009. — 896 е.: ил.

Робачевский А. М. Операционная система UNIX®. - СПб.: 2002. - 528 ил.

Харви Дейтел, Пол Дейтел «Как программировать на С», с.994. **Смотрите страницы 587 – 595**.

Интернет источники

https://librebay.blogspot.com/2016/12/how-to-compile-c-cpp-program-in-Ubuntu.html

http://cs.mipt.ru/cpp/labs/lab1.html#id28

http://rus-linux.net/lib.php?name=/MyLDP/algol/compilation/compilation-1.html

http://linux.yaroslavl.ru/docs/prog/gcc/gcc1-2.html (можно использовать как справочник)