**Лабораторная работа 3**

**Гусаров Андрей Пин-24**

**Аргументы командной строки**

**Сборка с помощью gcc (clang)**

Полная декларация main выглядит следующим образом:

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

Целое число, argc - количество аргументов. Это количество аргументов, переданных в программу из командной строки, включая имя программы.

Массив указателей на символы - это список всех аргументов. argv [0] - это имя программы или пустая строка, если имя недоступно. После этого каждый номер элемента меньше argc является аргументом командной строки. Вы можете использовать каждый элемент argv как строку или использовать argv как двумерный массив. argv [argc] является нулевым указателем.

Компиляция программы "C" из одного источника

Самый простой случай компиляции - это когда весь ваш исходный код установлен в одном файле. Это устраняет любые ненужные шаги синхронизации нескольких файлов или слишком много думает. Предположим, что есть файл с именем 'single\_main.c', который мы хотим скомпилировать. Мы сделаем это с помощью командной строки, подобной этой:

cc single\_main.c

Обратите внимание, что мы предполагаем, что компилятор называется «cc». Если вы используете компилятор GNU, вместо этого вы напишите 'gcc'.

Предположим, вы хотите, чтобы результирующая программа называлась "single\_main". В этом случае вы можете использовать следующую строку для его компиляции:

cc single\_main.c -o single\_main

Запуск результирующей программы

После того, как мы создали программу, мы хотим запустить ее. Обычно это делается простым вводом его имени, например:

single\_main

Однако для этого необходимо, чтобы текущий каталог находился в нашем PATH (это переменная, указывающая нашей оболочке Unix, где искать программы, которые мы пытаемся запустить). Во многих случаях этот каталог не помещается в наш путь. Тогда давайте покажем этому компьютеру, кто умнее, и таким образом мы попробуем:

./single\_main

**Аргументы командной строки**

**Системный вызов fork**

**Системный вызов pipe**

**Работа с файлами в Си**

**fork()** — [системный вызов](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BD%D1%8B%D0%B9_%D0%B2%D1%8B%D0%B7%D0%BE%D0%B2) в [Unix-подобных операционных системах](https://ru.wikipedia.org/wiki/Unix-%D0%BF%D0%BE%D0%B4%D0%BE%D0%B1%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0" \o "Unix-подобная операционная система), создающий новый [процесс](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) (потомок), который является практически полной копией процесса-родителя, выполняющего этот вызов.

Между процессом-потомком, порождаемым вызовом fork(), и процессом-родителем существуют различия:

[PID](https://ru.wikipedia.org/wiki/%D0%98%D0%B4%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%82%D0%BE%D1%80_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%B0) процесса-потомка отличен от PID процесса-родителя;

значению PPID процесса-потомка присваивается значение PID процесса-родителя;

процесс-потомок получает собственную таблицу файловых дескрипторов, являющуюся копией таблицы процесса-родителя на момент вызова fork(); это означает, что открытые файлы наследуются, но если процесс-потомок, например, закроет какой-либо файл, то это не повлияет на таблицу дескрипторов процесса-родителя;

для процесса-потомка очищаются все ожидающие доставки [сигналы](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%B3%D0%BD%D0%B0%D0%BB%D1%8B_(UNIX));

временная статистика выполнения процесса-потомка в таблицах ОС обнуляется;

блокировки памяти и записи, установленные в процессе-родителе, не наследуются.

fork - создание дочернего процесса

ОБЪЯВЛЕНИЕ

       #include <sys / types.h>

       #include <unistd.h>

       pid\_t fork (void);

ОПИСАНИЕ

       fork () создает новый процесс, дублируя вызывающий процесс. Новый процесс называется дочерним процессом. Вызывающий процесс вызывается как родительский процесс.

В [программировании](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) **именованный канал** или **именованный конвейер** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *named pipe*) — один из методов [межпроцессного взаимодействия](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BD%D0%BE%D0%B5_%D0%B2%D0%B7%D0%B0%D0%B8%D0%BC%D0%BE%D0%B4%D0%B5%D0%B9%D1%81%D1%82%D0%B2%D0%B8%D0%B5" \o "Межпроцессное взаимодействие), расширение понятия [конвейера](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%B2%D0%B5%D0%B9%D0%B5%D1%80_(UNIX)) в [Unix](https://ru.wikipedia.org/wiki/Unix" \o "Unix) и подобных [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0).

ОБЪЯВЛЕНИЕ

       pipe, pipe2 - создать конвеер

ОПИСАНИЕ top

       #include <unistd.h>

       / \* На Alpha, IA-64, MIPS, SuperH и SPARC / SPARC64; см. примечания \* /

       struct fd\_pair {

           long fd [2];

       };

       struct fd\_pair pipe ();

       / \* На всех других архитектурах \* /

       int pipe (int pipefd [2]);

       #define \_GNU\_SOURCE / \* См. feature\_test\_macros (7) \* /

       #include <fcntl.h> / \* Получить O\_ \* определения констант \* /

       #include <unistd.h>

       int pipe2 (int pipefd [2], int flags);

ОПИСАНИЕ

       pipe () создает канал, однонаправленный канал данных, который можно использовать для межпроцессного взаимодействия. Массив pipefd используется для возврата двух файловых дескриптора, относящиеся к концам канала. pipefd [0] относится к концу чтения канала. pipefd [1] относится к записи в конец канала. Данные, записанные в конец записи канала буферизируется ядром, пока не будет прочитано из конца чтения конвеера.

       Если flags равен 0, то pipe2 () совпадает с pipe ().

FILE \*fopen(name, type);

name – имя открываемого файла (включая путь),  
type — указатель на строку символов, определяющих способ доступа к файлу:

"r" — открыть файл для чтения (файл должен существовать);

"w" — открыть пустой файл для записи; если файл существует, то его содержимое теряется;

"a" — открыть файл для записи в конец (для добавления); файл создается, если он не существует;

"r+" — открыть файл для чтения и записи (файл должен существовать);

"w+" — открыть пустой файл для чтения и записи; если файл существует, то его содержимое теряется;

"a+" — открыть файл для чтения и дополнения, если файл не существует, то он создаётся.

**Чтение символа из файла**:

**char fgetc(поток);**

Аргументом функции является указатель на поток типа FILE. Функция возвращает код считанного символа. Если достигнут конец файла или возникла ошибка, возвращается константа EOF.  
  
**Запись символа в файл**:

**fputc(символ,поток);**

Аргументами функции являются символ и указатель на поток типа FILE. Функция возвращает код считанного символа.  
  
Функции fscanf() и fprintf() аналогичны функциям scanf() и printf(), но работают с файлами данных, и имеют первый аргумент — указатель на файл.

**fscanf(поток, "ФорматВвода", аргументы);**

**fprintf(поток, "ФорматВывода", аргументы);**

Функции fgets() и fputs() предназначены для ввода-вывода строк, они являются аналогами функций gets() и puts() для работы с файлами.

**fgets(УказательНаСтроку, КоличествоСимволов, поток);**

Символы читаются из потока до тех пор, пока не будет прочитан символ новой строки ‘\n’, который включается в строку, или пока не наступит конец потока EOF или не будет прочитано максимальное количество символов. Результат помещается в указатель на строку и заканчивается нуль- символом ‘\0’. Функция возвращает адрес строки.

**fputs(УказательНаСтроку,поток);**

Копирует строку в поток с текущей позиции. Завершающий нуль- символ не копируется.

size\_t fread(void \*ptr, size\_t size\_of\_elements, size\_t number\_of\_elements, FILE \*a\_file);

size\_t fwrite(const void \*ptr, size\_t size\_of\_elements, size\_t number\_of\_elements, FILE \*a\_file);

|  |  |  |
| --- | --- | --- |
| ​int printf( const char \*format, ... );​ | (1) |  |
| int fprintf( [FILE](http://ru.cppreference.com/w/c/io) \*stream, const char \*format, ... ); | (2) |  |
| int sprintf( char \*buffer, const char \*format, ... ); | (3) |  |
| int snprintf( char \*buffer, int buf\_size, const char \*format, ... ); | (4) | (начиная с C99) |
|  |  |  |

Загружает данные из данного места, преобразует их в строку символов эквиваленты и записывает результаты в различных поглотителей.

1)

Записывает результаты в [stdout](https://ru.cppreference.com/w/c/io" \o "c/io).

2)

Записывает результаты в файл потока stream.

3)

Записывает результаты в buffer буквенная.

4)

Записывает результаты в buffer строку символов. В большинстве buf\_size - 1 символов написаны. В результате символьная строка будет завершен нулевой символ, если buf\_size равна нулю.

|  |  |  |
| --- | --- | --- |
| ​int scanf( const char \*format, ... );​ | (1) |  |
| int fscanf( [FILE](http://ru.cppreference.com/w/c/io) \*stream, const char \*format, ... ); | (2) |  |
| int sscanf( const char \*buffer, const char \*format, ... ); | (3) |  |
|  |  |  |

Читает данные из различных источников, интерпретирует его в соответствии с format и сохраняет результаты в данной мест.

1)

считывает данные из [stdin](https://ru.cppreference.com/w/c/io" \o "c/io)

2)

считывает данные из файла stream поток

3)

считывает данные с нулевым buffer строку символов

**Make- основные сведения**

[make](http://ru.wikipedia.org/wiki/Make) — утилита предназначенная для автоматизации преобразования файлов из одной формы в другую. Правила преобразования задаются в скрипте с именем Makefile, который должен находиться в корне рабочей директории проекта. Сам скрипт состоит из набора правил, которые в свою очередь описываются:  
  
1) целями (то, что данное правило делает);  
2) реквизитами (то, что необходимо для выполнения правила и получения целей);  
3) командами (выполняющими данные преобразования).  
  
В общем виде синтаксис makefile можно представить так:

*# Индентация осуществляется исключительно при помощи символов табуляции,*

*# каждой команде должен предшествовать отступ*

<цели>: <реквизиты>

<команда *#1>*

...

<команда *#n>*

То есть, правило make это ответы на три вопроса:

{Из чего делаем? (реквизиты)} ---> [Как делаем? (команды)] ---> {Что делаем? (цели)}

Несложно заметить что процессы трансляции и компиляции очень красиво ложатся на эту схему:

{исходные файлы} ---> [трансляция] ---> {объектные файлы}

{объектные файлы} ---> [линковка] ---> {исполнимые файлы}

**Простейший Makefile**

Предположим, у нас имеется программа, состоящая всего из одного файла:

*/\**

*\* main.c*

*\*/*

#include <stdio.h>

int main()

{

printf("Hello World!\n");

return 0;

}

Для его компиляции достаточно очень простого мэйкфайла:

hello: main.c

gcc -o hello main.c

Данный Makefile состоит из одного правила, которое в свою очередь состоит из цели — «hello», реквизита — «main.c», и команды — «gcc -o hello main.c». Теперь, для компиляции достаточно дать команду make в рабочем каталоге. По умолчанию make станет выполнять самое первое правило, если цель выполнения не была явно указана при вызове:

$ make <цель>

Представим, что наша программа состоит из десятка- другого исходных файлов. Мы вносим изменения в один из них, и хотим ее пересобрать. Использование подхода описанного в предыдущем примере приведет к тому, что все без исключения исходные файлы будут снова скомпилированы, что негативно скажется на времени перекомпиляции. Решение — разделить компиляцию на два этапа: этап трансляции и этап линковки.  
  
Теперь, после изменения одного из исходных файлов, достаточно произвести его трансляцию и линковку всех объектных файлов. При этом мы пропускаем этап трансляции не затронутых изменениями реквизитов, что сокращает время компиляции в целом. Такой подход называется инкрементной компиляцией. Для ее поддержки make сопоставляет время изменения целей и их реквизитов (используя данные файловой системы), благодаря чему самостоятельно решает какие правила следует выполнить, а какие можно просто проигнорировать:

main.o: main.c

gcc -c -o main.o main.c

hello.o: hello.c

gcc -c -o hello.o hello.c

hello: main.o hello.o

gcc -o hello main.o hello.o

All GNU programs should have the following targets in their Makefiles:

**‘all’**

Compile the entire program. This should be the default target. This target need not rebuild any documentation files; Info files should normally be included in the distribution, and DVI (and other documentation format) files should be made only when explicitly asked for.

By default, the Make rules should compile and link with ‘-g’, so that executable programs have debugging symbols. Otherwise, you are essentially helpless in the face of a crash, and it is often far from easy to reproduce with a fresh build.

**‘install’**

Compile the program and copy the executables, libraries, and so on to the file names where they should reside for actual use. If there is a simple test to verify that a program is properly installed, this target should run that test.

Do not strip executables when installing them. This helps eventual debugging that may be needed later, and nowadays disk space is cheap and dynamic loaders typically ensure debug sections are not loaded during normal execution. Users that need stripped binaries may invoke the install-strip target to do that.

If possible, write the install target rule so that it does not modify anything in the directory where the program was built, provided ‘make all’ has just been done. This is convenient for building the program under one user name and installing it under another.

The commands should create all the directories in which files are to be installed, if they don’t already exist. This includes the directories specified as the values of the variables prefix and exec\_prefix, as well as all subdirectories that are needed. One way to do this is by means of an installdirs target as described below.

Use ‘-’ before any command for installing a man page, so that make will ignore any errors. This is in case there are systems that don’t have the Unix man page documentation system installed.

The way to install Info files is to copy them into $(infodir) with $(INSTALL\_DATA) (see [Command Variables](https://www.gnu.org/software/make/manual/html_node/Command-Variables.html#Command-Variables)), and then run the install-info program if it is present. install-info is a program that edits the Info dir file to add or update the menu entry for the given Info file; it is part of the Texinfo package.

Here is a sample rule to install an Info file that also tries to handle some additional situations, such as install-info not being present.

When writing the install target, you must classify all the commands into three categories: normal ones, *pre-installation* commands and *post-installation* commands. See [Install Command Categories](https://www.gnu.org/software/make/manual/html_node/Install-Command-Categories.html#Install-Command-Categories).

**‘install-html’**

**‘install-dvi’**

**‘install-pdf’**

**‘install-ps’**

These targets install documentation in formats other than Info; they’re intended to be called explicitly by the person installing the package, if that format is desired. GNU prefers Info files, so these must be installed by the install target.

When you have many documentation files to install, we recommend that you avoid collisions and clutter by arranging for these targets to install in subdirectories of the appropriate installation directory, such as htmldir. As one example, if your package has multiple manuals, and you wish to install HTML documentation with many files (such as the “split” mode output by makeinfo --html), you’ll certainly want to use subdirectories, or two nodes with the same name in different manuals will overwrite each other.

Please make these install-*format* targets invoke the commands for the *format* target, for example, by making *format* a dependency.

**‘uninstall’**

Delete all the installed files—the copies that the ‘install’ and ‘install-\*’ targets create.

This rule should not modify the directories where compilation is done, only the directories where files are installed.

The uninstallation commands are divided into three categories, just like the installation commands. See [Install Command Categories](https://www.gnu.org/software/make/manual/html_node/Install-Command-Categories.html#Install-Command-Categories).

**‘install-strip’**

Like install, but strip the executable files while installing them. In simple cases, this target can use the install target in a simple way:

install-strip:

$(MAKE) INSTALL\_PROGRAM='$(INSTALL\_PROGRAM) -s' \

install

But if the package installs scripts as well as real executables, the install-strip target can’t just refer to the install target; it has to strip the executables but not the scripts.

install-strip should not strip the executables in the build directory which are being copied for installation. It should only strip the copies that are installed.

Normally we do not recommend stripping an executable unless you are sure the program has no bugs. However, it can be reasonable to install a stripped executable for actual execution while saving the unstripped executable elsewhere in case there is a bug.

**‘clean’**

Delete all files in the current directory that are normally created by building the program. Also delete files in other directories if they are created by this makefile. However, don’t delete the files that record the configuration. Also preserve files that could be made by building, but normally aren’t because the distribution comes with them. There is no need to delete parent directories that were created with ‘mkdir -p’, since they could have existed anyway.

Delete .dvi files here if they are not part of the distribution.

**‘distclean’**

Delete all files in the current directory (or created by this makefile) that are created by configuring or building the program. If you have unpacked the source and built the program without creating any other files, ‘make distclean’ should leave only the files that were in the distribution. However, there is no need to delete parent directories that were created with ‘mkdir -p’, since they could have existed anyway.

**‘mostlyclean’**

Like ‘clean’, but may refrain from deleting a few files that people normally don’t want to recompile. For example, the ‘mostlyclean’ target for GCC does not delete libgcc.a, because recompiling it is rarely necessary and takes a lot of time.

**‘maintainer-clean’**

Delete almost everything that can be reconstructed with this Makefile. This typically includes everything deleted by distclean, plus more: C source files produced by Bison, tags tables, Info files, and so on.

The reason we say “almost everything” is that running the command ‘make maintainer-clean’ should not delete configure even if configure can be remade using a rule in the Makefile. More generally, ‘make maintainer-clean’ should not delete anything that needs to exist in order to run configure and then begin to build the program. Also, there is no need to delete parent directories that were created with ‘mkdir -p’, since they could have existed anyway. These are the only exceptions; maintainer-clean should delete everything else that can be rebuilt.

The ‘maintainer-clean’ target is intended to be used by a maintainer of the package, not by ordinary users. You may need special tools to reconstruct some of the files that ‘make maintainer-clean’ deletes. Since these files are normally included in the distribution, we don’t take care to make them easy to reconstruct. If you find you need to unpack the full distribution again, don’t blame us.

To help make users aware of this, the commands for the special maintainer-clean target should start with these two:

@echo 'This command is intended for maintainers to use; it'

@echo 'deletes files that may need special tools to rebuild.'

**‘TAGS’**

Update a tags table for this program.

**‘info’**

Generate any Info files needed. The best way to write the rules is as follows:

info: foo.info

foo.info: foo.texi chap1.texi chap2.texi

$(MAKEINFO) $(srcdir)/foo.texi

You must define the variable MAKEINFO in the Makefile. It should run the makeinfo program, which is part of the Texinfo distribution.

Normally a GNU distribution comes with Info files, and that means the Info files are present in the source directory. Therefore, the Make rule for an info file should update it in the source directory. When users build the package, ordinarily Make will not update the Info files because they will already be up to date.

**‘dvi’**

**‘html’**

**‘pdf’**

**‘ps’**

Generate documentation files in the given format. These targets should always exist, but any or all can be a no-op if the given output format cannot be generated. These targets should not be dependencies of the all target; the user must manually invoke them.

Here’s an example rule for generating DVI files from Texinfo:

dvi: foo.dvi

foo.dvi: foo.texi chap1.texi chap2.texi

$(TEXI2DVI) $(srcdir)/foo.texi

You must define the variable TEXI2DVI in the Makefile. It should run the program texi2dvi, which is part of the Texinfo distribution. (texi2dvi uses TeX to do the real work of formatting. TeX is not distributed with Texinfo.) Alternatively, write only the dependencies, and allow GNU make to provide the command.

Here’s another example, this one for generating HTML from Texinfo:

html: foo.html

foo.html: foo.texi chap1.texi chap2.texi

$(TEXI2HTML) $(srcdir)/foo.texi

Again, you would define the variable TEXI2HTML in the Makefile; for example, it might run makeinfo --no-split --html (makeinfo is part of the Texinfo distribution).

**‘dist’**

Create a distribution tar file for this program. The tar file should be set up so that the file names in the tar file start with a subdirectory name which is the name of the package it is a distribution for. This name can include the version number.

For example, the distribution tar file of GCC version 1.40 unpacks into a subdirectory named gcc-1.40.

The easiest way to do this is to create a subdirectory appropriately named, use ln or cp to install the proper files in it, and then tar that subdirectory.

Compress the tar file with gzip. For example, the actual distribution file for GCC version 1.40 is called gcc-1.40.tar.gz. It is ok to support other free compression formats as well.

The dist target should explicitly depend on all non-source files that are in the distribution, to make sure they are up to date in the distribution. See [Making Releases](https://www.gnu.org/prep/standards/html_node/Releases.html#Releases) in *GNU Coding Standards*.

**‘check’**

Perform self-tests (if any). The user must build the program before running the tests, but need not install the program; you should write the self-tests so that they work when the program is built but not installed.

В [вычислениях](https://ru.qwe.wiki/wiki/Computing) , **Exec** является функциональностью в [операционной системе](https://ru.qwe.wiki/wiki/Operating_system) , которая запускает [исполняемый файл](https://ru.qwe.wiki/wiki/Executable_file) в контексте уже существующий [процесс](https://ru.qwe.wiki/wiki/Process_(computing)) , заменяя предыдущий исполняемый файл. Этот акт также называют **накладкой** . Это особенно важно в [Unix-подобных](https://ru.qwe.wiki/wiki/Unix-like" \o "Unix-подобные) систем, хотя другие операционные системы реализуют его. Поскольку новый процесс не создается, исходный [идентификатор процесса](https://ru.qwe.wiki/wiki/Process_identifier) (PID) не изменяется, но [машинный код](https://ru.qwe.wiki/wiki/Machine_code) , [данные](https://ru.qwe.wiki/wiki/Data_(computing)) , [куча](https://ru.qwe.wiki/wiki/Heap_(programming)) , и [стек](https://ru.qwe.wiki/wiki/Run-time_stack) процесса заменяются теми из новой программы.

языка C прототипы

[POSIX](https://ru.qwe.wiki/wiki/POSIX) стандарт объявляет *EXEC* функции в [unistd.h](https://ru.qwe.wiki/wiki/Unistd.h" \o "unistd.h) файл заголовка, в [языке C](https://ru.qwe.wiki/wiki/C_(programming_language)) . Те же функции объявлены в [process.h](https://ru.qwe.wiki/wiki/Process.h" \o "Process.h) для DOS (см [ниже](https://ru.qwe.wiki/wiki/Exec_(system_call)#DOS_operating_systems) ), OS / 2 и Microsoft Windows.

int execl(char **const** \*path, char **const** \*arg0, ...);

int execle(char **const** \*path, char **const** \*arg0, ..., char **const** \*envp[]);

int execlp(char **const** \*file, char **const** \*arg0, ...);

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

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

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

Некоторые реализации предоставляют эти функции, названные с начальным символом подчеркивания (например, \_execl).

База каждый **Exec** (выполнить), за которым следует один или более букв:

**е** - массив указателей на [переменные окружения](https://ru.qwe.wiki/wiki/Environment_variable) явно передаются в новый образ процесса.

**л** - [Аргументы командной строки](https://ru.qwe.wiki/wiki/Command-line_interface#Arguments_section) передаются по отдельности (в **л** IST) к функции.

**р** - Использует [переменную PATH среды](https://ru.qwe.wiki/wiki/PATH_(variable)) , чтобы найти файл с именем в *файл* аргумент выполняются.

**v** - аргументы командной строки передаются функции в качестве массива ( **об** Экторе) указателей.

В execl , функции имеют , execv и execvp вызовов, новый образ процесса наследует текущие переменные окружения.

аргументы

**дорожка**

*Путь* аргумент задает путь к файлу для выполнения в качестве нового образа процесса.

**arg0, ...**

Аргументы , начинающиеся на *arg0* являются [указатели](https://ru.qwe.wiki/wiki/Pointer_(computer_programming)) на аргументы , которые будут переданы в новый образ процесса.

Первый аргумент *arg0* должно быть имя исполняемого файла. Как правило , это то же самое значение как *путь* аргумент. Некоторые программы могут неправильно полагаться на этот аргумент , обеспечивающий расположение исполняемого файла, но нет никакой гарантии , это а также не стандартизированы на разных платформах.

**ARGV**

*ARGV* значение представляет собой массив указателей на аргументы.

**envp**

Аргумент *envp* представляет собой массив указателей на настройки окружения.

В *Exec* вызовы по имени , оканчивающиеся на *е* изменить окружение нового образа процесса, передавая список параметров среды через *envp* аргумент. Этот аргумент является массивом символьных указателей; каждый элемент (за исключением конечного элемента) указывает на [строку с нулевым символом](https://ru.qwe.wiki/wiki/Null-terminated_string) , определяющей [переменную среды](https://ru.qwe.wiki/wiki/Environment_variable) .

Каждый завершающим нулем строка имеет вид: name=value

где *имя* это имя переменной среды, а также *значение* является значением этой переменной. Последний элемент *envp* массива должен быть [нулевым](https://ru.qwe.wiki/wiki/Null_pointer) .

[Небольшой пример, как комбинировать fork и exec](https://ece.uwaterloo.ca/~dwharder/icsrts/Tutorials/fork_exec/)

#include <stdio.h>

/\* This program forks and and the prints whether the process is

\* - the child (the return value of fork() is 0), or

\* - the parent (the return value of fork() is not zero)

\*

\* When this was run 100 times on the computer the author is

\* on, only twice did the parent process execute before the

\* child process executed.

\*

\* Note, if you juxtapose two strings, the compiler automatically

\* concatenates the two, e.g., "Hello " "world!"

\*/

int main( void ) {

char \*argv[3] = {"Command-line", ".", NULL};

int pid = fork();

if ( pid == 0 ) {

execvp( "find", argv );

}

/\* Put the parent to sleep for 2 seconds--let the child finished executing \*/

wait( 2 );

printf( "Finished executing the parent process\n"

" - the child won't get here--you will only see this once\n" );

return 0;

}