Лабораторная работа № 3. Знакомство с компилятором GCC. Make-файлы. Процессы.

Средствами, традиционно используемыми для создания программ для открытых операционных систем, являются инструменты разработчика GNU. Проект GNU был основан в 1984 году Ричардом Столлманом. Его необходимость была вызвана тем, что в то время сотрудничество между программистами было затруднено, так как владельцы коммерческого программного обеспечения чинили многочисленные препятствия такому сотрудничеству. Целью проекта GNU было создание комплекта программного обеспечения под единой лицензией, которая не допускала бы возможности присваивания кем-то эксклюзивных прав на это ПО. Частью этого комплекта и является набор инструментов для разработчика, которым мы будем пользоваться, и который должен входить во все дистрибутивы Linux.

Одним из этих инструментов является компилятор GCC. Первоначально эта аббревиатура расшифровывалась, как GNU C Compiler. Сейчас она означает - GNU Compiler Collection.

Файлы с исходными кодами программ, которые мы будем создавать, это обычные текстовые файлы, и создавать их можно с помощью любого текстового редактора (например GEdit, KWrite, Kate, а также более традиционные для пользователей Linux - vi, emacs).

Помимо текстовых редакторов, существуют специализированные среды разработки со своими встроенными редакторами. Одним из таких средств является KDevelop. Интересно, что в нём есть встроенный редактор и встроенная консоль, расположенная прямо под редактором. Так что можно прямо в одной программе, не переключаясь между окнами, и редактировать код и давать консольные команды.

Создайте отдельный каталог hello. Это будет каталог нашего первого проекта. В нём создайте текстовый файл hello.c со следующим текстом:

```
#include <stdio.h>
int main(void){
      printf(”Hello world!\n”);
      return(0);
}
```

Затем в консоли зайдите в каталог проекта. Наберите команду:

```
gcc hello.c
```

Теперь посмотрите внимательно, что произошло. В каталоге появился новый файл a.out. Это и есть исполняемый файл. Запустим его. Наберите в консоли:

```
./a.out
```

Программа должна запуститься, то есть должен появиться текст:

```
Hello world!
```

Компилятор `gcc` по умолчанию присваивает всем созданным исполняемым файлам имя `a.out`. Если хотите назвать его по-другому, нужно к команде на компиляцию добавить флаг -o и имя, которым вы хотите его назвать. Давайте наберём такую команду:

```
gcc hello.c -o hello
```

Мы видим, что в каталоге появился исполняемый файл с названием hello. Запустим его.

```
./hello
```

Как видите, получился точно такой же исполняемый файл, только с удобным для нас названием. Флаг `-o` является лишь одним из многочисленных флагов компилятора `gcc`. Некоторые другие флаги мы рассмотрим позднее. Чтобы просмотреть все возможные флаги, можно воспользоваться справочной системой man. Наберите в командной строке:

```
man gcc
```

Перед вами предстанет справочная система по этой программе. Просмотрите, что означает каждый флаг. С некоторыми из них мы скоро встретимся. Выход из справочной системы осуществляется с помощью клавиши q.

Вы, конечно, обратили внимание, что, когда мы запускаем программу из нашего каталога разработки, мы перед названием файла набираем точку и слэш. Зачем же мы это делаем?

Дело в том, что, если мы наберём только название исполняемого файла, операционная система будет искать его в каталогах `/usr/bin` и `/usr/local/bin`, и, естественно, не найдёт.

Каталоги `/usr/bin` и `/usr/local/bin` - системные каталоги размещения исполняемых программ. Первый из них предназначен для размещения стабильных версий программ, как правило,

входящих в дистрибутив Linux. Второй - для программ, устанавливаемых самим пользователем (за стабильность которых никто не ручается). Такая система нужна, чтобы отделить их друг от друга. По умолчанию при сборке программы устанавливаются в каталог `/usr/local/bin`. Крайне нежелательно помещать что-либо лишнее в `/usr/bin` или удалять что-то оттуда вручную, потому что это может привести к краху системы. Там должны размещаться программы, за стабильность которых отвечают разработчики дистрибутива.

Чтобы запустить программу, находящуюся в другом месте, надо прописать полный путь к ней, например так:

```
/home/user/projects/hello/hello
```

Или другой вариант: прописать путь относительно текущего каталога, в котором вы в данной момент находитесь в консоли. При этом одна точка означает текущий каталог, две точки - родительский. Например, команда `./hello` запускает программу hello, находящуюся в текущем каталоге, команда `../hello` - программу hello, находящуюся в родительском каталоге, команда `./projects/hello/hello` - программу во вложенных каталогах, находящихся внутри текущего.

Есть возможность добавлять в список системных путей к программам дополнительные каталоги. Для этого надо добавить новый путь в системную переменную `PATH`. Но давайте пока не будем отвлекаться от главной темы. Переменные окружения - это отдельный разговор.

Теперь рассмотрим, что же делает программа `gcc`. Её работа включает три этапа:

- обработка препроцессором;
- компиляция и компоновка (или линковка).

Препроцессор включает в основной файл содержимое всех заголовочных файлов, указанных в директивах `#include`. В заголовочных файлах обычно находятся объявления функций, используемых в программе, но не определённых в тексте программы.

Их определения находятся где-то в другом месте: или в других файлах с исходным кодом или в бинарных библиотеках *(ключ -E)*.

Вторая стадия - **компиляция**. Она заключается в превращении текста программы на языке C/C++ в набор машинных команд. Результат сохраняется в объектном файле. Разумеется, на машинах с разной архитектурой процессора двоичные файлы получаются в разных форматах, и на одной машине невозможно запустить бинарник, собранный на другой машине (разве только, если у них одинаковая архитектура процессора и одинаковые операционные системы). Вот почему программы для UNIX-подобных систем распространяются в виде исходных кодов: они должны быть доступны всем пользователям, независимо от того, у кого какой процессор и какая операционная система *(ключ -c)*.

Последняя стадия - **компоновка**. Она заключается в связывании всех объектных файлов проекта в один, связывании вызовов функций с их определениями, и присоединением библиотечных файлов, содержащих функции, которые вызываются, но не определены в проекте. В результате формируется запускаемый файл - наша конечная цель. Если какая-то функция в программе используется, но компоновщик не найдёт место, где эта функция определена, он выдаст сообщение об ошибке, и откажется создавать исполняемый файл *(ключ -o)*.

Если мы создаём объектный файл из исходника, уже обработанного препроцессором (например, такого, какой мы получили выше), то мы должны обязательно указать явно, что компилируемый файл является файлом исходного кода, обработанный препроцессором, и имеющий теги препроцессора. В противном случае он будет обрабатываться, как обычный файл C++, без учёта тегов препроцессора, а значит связь с объявленными функциями не будет устанавливаться. Для явного указания на язык и формат обрабатываемого файла служит опция `-x`. Файл C++, обработанный препроцессором обозначается cpp-output.

```
gcc -x cpp-output -c proga.cpp
```

Наконец, последний этап - компоновка. Получаем из объектного файла исполняемый.

```
gcc proga.o -o proga
```

Можно его запускать.

```
./proga
```

Вы спросите: «Не лучше ли просто один раз скомандовать gcc proga.c -o proga?»

Дело в том, что настоящие программы очень редко состоят из одного файла. Как правило исходных файлов несколько, и они объединены в проект. И в некоторых исключительных случаях

программу приходится компоновать из нескольких частей, написанных на разных языка. В этом случае приходится запускать компиляторы разных языков, чтобы каждый получил объектный файл из своего исходника, а затем уже эти полученные объектные файлы компоновать в исполняемую программу.

Пример проекта из нескольких файлов

Напишем теперь программу, состоящую из двух исходных файлов и одного заголовочного.. Создадим каталог проекта kalkul. В нём создадим три файла: calculate.h, calculate.c, main.c.

Файл calculate.h:

```
////////////////////////////////////
// calculate.h
#ifndef CALCULATE_H_
#define CALCULATE_H_
float Calculate(float Numeral, char Operation[4]);
#endif /*CALCULATE_H_*/
```

Файл calculate.c:

```
// calculate.c

#include <stdio.h>
#include <math.h>
#include <string.h>
#include "calculate.h"

float Calculate(float Numeral, char Operation[4])
{
	float SecondNumeral;
	if(strncmp(Operation, "+", 1) == 0){
		printf("Второе слагаемое: ");
		scanf("%f",&SecondNumeral);
		return(Numeral + SecondNumeral);
	}
	else if(strncmp(Operation, "-", 1) == 0){
		printf("Вычитаемое: ");
		scanf("%f",&SecondNumeral);
		return(Numeral - SecondNumeral);
	}

	else if(strncmp(Operation, "*", 1) == 0) {
		printf("Множитель: ");
		scanf("%f",&SecondNumeral);
		return(Numeral * SecondNumeral);
	}

	else if(strncmp(Operation, "/", 1) == 0) {
		printf("Делитель: ");
		scanf("%f",&SecondNumeral);
			if(SecondNumeral == 0) {
				printf("Ошибка: деление на ноль! ");
				return(HUGE_VAL);
			}
			else return(Numeral / SecondNumeral);
	}

	else if(strncmp(Operation, "pow", 3) == 0) {
		printf("Степень: ");
```

```
            scanf("%f",&SecondNumeral);
            return(pow(Numeral, SecondNumeral));
      }

      else if(strncmp(Operation, "sqrt", 4) == 0) return(sqrt(Numeral));
      else if(strncmp(Operation, "sin", 3) == 0) return(sin(Numeral));
      else if(strncmp(Operation, "cos", 3) == 0) return(cos(Numeral));
      else if(strncmp(Operation, "tan", 3) == 0) return(tan(Numeral));


      else {
            printf("Неправильно введено действие ");
            return(HUGE_VAL);
      }
}
```

Файл main.c:

```
// main.c
#include <stdio.h>
#include "calculate.h"
int main(void) {
      float Numeral;
      char Operation[4];
      float Result;
      printf("Число: ");
      scanf("%f",&Numeral);
      printf("Арифметическое   действие   (+,-,*,/,pow,sqrt,sin,cos,tan):
");   scanf("%s",&Operation);
      Result = Calculate(Numeral, Operation);
      printf("%6.2f\n",Result);
      return 0;
}
```

У нас есть два файла исходного кода (c-файлы) и один заголовочный (h-файл). Заголовочный включается в оба c-файла.

Скомпилируем calculate.c.

```
gcc -c calculate.c
```

Получили calculate.o. Затем main.c.

```
gcc -c main.c
```

И вот он main.o перед нами! Теперь, как вам уже, наверное, подсказывает интуиция, надо из этих двух объектных файлов сделать запускаемый.

```
gcc calculate.o main.o -o kalkul
```

Не получилось... Вместо столь желаемого запускаемого файла, в консоли появилась какая-то ругань:

```
calculate.o(.text+0x1b5): In function 'Calculate’:
calculate.c: undefined reference to 'pow’
calculate.o(.text+0x21e):calculate.c: undefined  reference  to
'sqrt’
calculate.o(.text+0x274):calculate.c: undefined  reference  to   'sin’
calculate.o(.text+0x2c4):calculate.c: undefined  reference  to   'cos’
calculate.o(.text+0x311):calculate.c: undefined  reference  to   'tan’
collect2: ld returned 1 exit status
```

Undefined reference означает ссылку на функцию, которая не определена. В данном случае gcc не нашёл определения функций pow, sqrt, sin, cos, tan. Где же их найти?

Как уже говорилось раньше, определения функций могут находиться в библиотеках. Это скомпилированные двоичные файлы, содержащие коллекции однотипных операций, которые часто вызываются из многих программ, а потому нет смысла многократно писать их код в программах.

Стандартное расположение файлов библиотек - каталоги `/usr/lib` и `/usr/local/lib` (при желании можно добавить путь). Если библиотечный файл имеет расширение **.a**, то это статическая библиотека, то есть при компоновке весь её двоичный код включается в исполняемый файл. Если расширение **.so**, то это динамическая библиотека. Это значит в исполняемый файл программы помещается только ссылка на библиотечный файл, а уже из него и запускается функция.

Когда мы писали программу **hello**, мы использовали функцию **printf** для вывода текстовой строки. Однако, как вы помните, мы нигде не писали определения этой функции. Откуда же она тогда вызывается?

Просто при компоновке любой программы компилятор gcc по умолчанию включает в запускаемый файл библиотеку **libc**. Это стандартная библиотека языка С. Она содержит рутинные функции, необходимые абсолютно во всех программах, написанных на С, в том числе и функцию **printf**. Поскольку библиотека **libc** нужна во всех программах, она включается по умолчанию, без необходимости давать отдельное указание на её включение.

Остальные библиотеки надо требовать включать явно. Ведь нельзя же во все программы помещать абсолютно все библиотеки. Тогда исполняемый файл раздуется до немыслимо крупных размеров. Одним программам нужны одни функции, другим - другие. Зачем же засорять их ненужным кодом! Пусть остаётся только то, что реально необходимо.

Нам в данном случае нужна библиотека libm. Именно она содержит все основные математические функции. Она требует включения в текст программы заголовочного файла <math.h>.

Помимо этого дистрибутивы Linux содержат и другие библиотеки, например:
libGL Вывод трёхмерной графики в стандарте OpenGL. Требуется заголовочный файл <GL/gl.h>.
libcrypt Криптографические функции. Требуется заголовочный файл <crypt.h>.
libcurses Псевдографика в символьном режиме. Требуется заголовочный файл <curses.h>.
libform Создание экранных форм в текстовом режиме. Требуется заголовочный файл <form.h>.
libgthread Поддержка многопоточного режима. Требуется заголовочный файл <glib.h>.
libgtk Графическая библиотека в режиме X Window. Требуется заголовочный файл <gtk/gtk.h>.
libhistory Работы с журналами. Требуется заголовочный файл <readline/readline.h>.
libjpeg Работа с изображениям в формате JPEG. Требуется заголовочный файл <jpeglib.h>.
libncurses Работа с псевдографикой в символьном режиме. Требуется заголовочный файл <ncurses.h>.
libpng Работа с графикой в формате PNG. Требуется заголовочный файл <png.h>.
libpthread Многопоточная библиотека POSIX. Стандартная многопоточная библиотека для Linux. Требуется заголовочный файл <pthread.h>.
libreadline Работа с командной строкой. Требуется заголовочный файл <readline/readline.h>.
libtiff Работа с графикой в формате TIFF. Требуется заголовочный файл <tiffio.h>.
libvga Низкоуровневая работа с VGA и SVGA. Требуется заголовочный файл <vga.h>.
А также многие-многие другие.

Обратите внимание, что названия всех этих библиотек начинаются с буквосочетания lib-. Для их явного включения в исполняемый файл, нужно добавить к команде gcc опцию -l, к которой слитно прибавить название библиотеки без lib-. Например, чтобы включить библиотеку libvga надо указать опцию -lvga.

Нам нужны математические функции pow, sqrt, sin, cos, tan. Они, как уже было сказано, находятся в математической библиотеке libm. Следовательно, чтобы подключить эту библиотеку, мы должны указать опцию -lm.

```
gcc calculate.o main.o -o kalkul -lm
```

Наконец-то наш запускаемый файл создан!

Make-файлы

У вас, вероятно, появился вопрос: можно ли не компилировать эти файлы по отдельности, а собрать сразу всю программу одной командой? Можно.

```
gcc calculate.c main.c -o kalkul -lm
```

Вы скажете, что это удобно? Удобно для нашей программы, потому что она состоит всего из двух с-файлов. Однако профессиональная программа может состоять из нескольких десятков таких файлов. Каждый раз набирать названия их всех в одной строке было бы делом чрезмерно утомительным. Но есть возможность решить эту проблему. Названия всех исходных файлов и все команды для сборки программы можно поместить в отдельный текстовый файл. А потом считывать их оттуда одной короткой командой.

Давайте создадим такой текстовый файл и воспользуемся им. В каталоге проекта **kalkul** удалите все файлы, кроме **calculate.h, calculate.c, main.c**. Затем создайте в этом же каталоге новый файл, назовите его Makefile (без расширений). Поместите туда следующий текст.

```
kalkul:          calculate.o main.o
                 gcc calculate.o main.o -o kalkul -lm

calculate.o:     calculate.c calculate.h
                 gcc -c calculate.c

main.o:         main.c calculate.h
                gcc -c main.c
clean:
                rm -f kalkul calculate.o main.o
install:
                cp kalkul /usr/local/bin/kalkul
uninstall:
                rm -f /usr/local/bin/kalkul
```

Обратите внимание на строки, введённые с отступом от левого края. Этот отступ получен с помощью клавиши Tab. Только так его и надо делать! Если будете использовать клавишу «Пробел», команды не будут исполняться.

Затем дадим команду, состоящую всего из одного слова:

```
make
```

И сразу же в нашем проекте появляются и объектные файлы и запускаемый. Программа make как раз и предназначена для интерпретации команд, находящихся в файле со стандартным названием Makefile. Рассмотрим его структуру.

Makefile является списком правил. Каждое правило начинается с указателя, называемого «Цель». После него стоит двоеточие, а далее через пробел указываются зависимости. В нашем случае ясно, что конечный файл **kalkul** зависит от объектных файлов calculate.o и main.o. Поэтому они должны быть собраны прежде сборки **kalkul**. После зависимостей пишутся команды. Каждая команда должна находиться на отдельной строке, и отделяться от начала строки клавишей Tab. Структура правила Makefile может быть очень сложной. Там могут присутствовать переменные, конструкции ветвления, цикла. Этот вопрос требует отдельного подробного изучения.

Если мы посмотрим на три первых правила, то они нам хорошо понятны. Там те же самые команды, которыми мы уже пользовались. А что же означают правила clean, install и uninstall?

В правиле clean стоит команда rm, удаляющая исполняемый и объектные файлы. Флаг -f означает, что, если удаляемый файл отсутствует, программа должна это проигнорировать, не выдавая никаких сообщений. Итак, правило clean предназначено для «очистки» проекта, приведения его к такому состоянию, в каком он был до команды make.

Запустите

```
make
```

Появились объектные файлы и исполняемый. Теперь

```
make clean
```

Объектные и исполняемый файлы исчезли. Остались только С-файлы, h-файл и сам Makefile.

То есть, проект «очистился» от результатов команды make.

Правило install помещает исполняемый файл в каталог /usr/local/bin - стандартный каталог размещения пользовательских программ. Это значит, что её можно будет вызывать из любого места простым набором её имени. Но помещать что-либо в этот каталог можно только, зайдя в систему под «суперпользователем». Для этого надо дать команду su и набрать пароль «суперпользователя». В противном случае система укажет, что вам отказано в доступе. Выход из «суперпользователя» осуществляется командой exit. Итак,

```
make
su
make install
exit
```

Теперь вы можете запустить это программу просто, введя имя программы, без прописывания пути.

```
kalkul
```

Можете открыть каталог /usr/local/bin. Там должен появиться файл с названием kalkul. Давайте теперь «уберём за собой», не будем засорять систему.

```
su
make uninstall
exit
```

Посмотрите каталог /usr/local/bin. Файл **kalkul** исчез. Итак, правило uninstall удаляет программу из системного каталога.

Процессы.

Контекст процесса

Контекст процесса складывается из пользовательского контекста и контекста ядра.

Под пользовательским контекстом процесса понимают код и данные, расположенные в адресном пространстве процесса. Все данные подразделяются на:

- инициализируемые неизменяемые данные (например, константы);
- инициализируемые изменяемые данные (все переменные, начальные значения которых присваиваются на этапе компиляции);
- неинициализируемые изменяемые данные (все статические переменные, которым не присвоены начальные значения на этапе компиляции);
- стек пользователя;
- данные, расположенные в динамически выделяемой памяти (например, с помощью стандартных библиотечных C функций malloc(), calloc(), realloc()).

Исполняемый код и инициализируемые данные составляют содержимое файла программы, который исполняется в контексте процесса. Пользовательский стек применяется при работе процесса в пользовательском режиме (user-mode).

Под понятием "контекст ядра" объединяются системный контекст и регистровый контекст, рассмотренные на лекции. Мы будем выделять в контексте ядра стек ядра, который используется при работе процесса в режиме ядра (kernel mode), и данные ядра, хранящиеся в структурах, являющихся аналогом блока управления процессом — PCB. Состав данных ядра будет уточняться на последующих семинарах. На этом занятии нам достаточно знать, что в данные ядра входят: идентификатор пользователя — UID, групповой идентификатор пользователя — GID, идентификатор процесса — PID, идентификатор родительского процесса — PPID.

Идентификация процесса

Каждый процесс в операционной системе получает уникальный идентификационный номер - PID (process identificator). При создании нового процесса операционная система пытается присвоить ему свободный номер больший, чем у процесса, созданного перед ним. Если таких свободных номеров не оказывается (например, мы достигли максимально возможного номера для процесса), то операционная система выбирает минимальный номер из всех свободных номеров. В операционной системе Linux присвоение идентификационных номеров процессов начинается с номера 0, который получает процесс kernel при старте операционной системы. Этот номер впоследствии не можетбыть присвоен никакому другому процессу. Максимально возможное значение для номера процесса в Linux на базе 32разрядных процессоров Intel составляет 2^{31}-1.

Иерархия процессов

В операционной системе UNIX все процессы, кроме одного, создающегося при старте операционной системы, могут быть порождены только какими-либо другими процессами. В качестве прародителя всех остальных процессов в подобных UNIX системах могут выступать процессы с номерами 1 или 0. В операционной системе Linux таким родоначальником, существующим только при загрузке системы, является процесс kernel с идентификатором 0.

Таким образом, все процессы в UNIX связаны отношениями процесс-родитель - процесс-ребенок и образуют генеалогическое дерево процессов. Для сохранения целостности генеалогического дерева в ситуациях, когда процесс-родитель завершает свою работу до завершения выполнения процесса-ребенка, идентификатор родительского процесса в данных ядра процесса-ребенка (PPID - parent process identificator) изменяет свое значение на значение 1, соответствующее идентификатору процесса init, время жизни которого определяет время функционирования операционной системы. Тем самым процесс init как бы усыновляет осиротевшие процессы. Наверное, логичнее было бы заменять PPID не на значение 1, а на значение идентификатора ближайшего существующего процесса- прародителя умершего процесса-родителя, но в UNIX почему-то такая схема реализована не была.

Системные вызовы getppid() и getpid()

Данные ядра, находящиеся в контексте ядра процесса, не могут быть прочитаны процессом

непосредственно. Для получения информации о них процесс должен совершить соответствующий системный вызов. Значение идентификатора текущего процесса может быть получено с помощью системного вызова getpid(), а значение идентификатора родительского процесса для текущего процесса - с помощью системного вызова getppid(). Прототипы этих системных вызовов и соответствующие типы данных описаны в системных файлах <sys/types.h> и <unistd.h>. Системные вызовы не имеют параметров и возвращают идентификатор текущего процесса и идентификатор родительского процесса соответственно.

Прототипы системных вызовов:

```
#include <sys/types.h>
#include <unistd.h> pid_t getpid(void);
pid_t getppid(void);
```

Описание системных вызовов

Системный вызов getpid возвращает идентификатор текущего процесса. Системный вызов getppid возвращает идентификатор процесса-родителя для текущего процесса. Тип данных pid_t является синонимом для одного из целочисленных типов языка С.

Создание процесса в UNIX. Системный вызов fork()

В операционной системе UNIX новый процесс может быть порожден единственным способом - с помощью системного вызова fork(). При этом вновь созданный процесс будет являться практически полной копией родительского процесса. У порожденного процесса по сравнению с родительским процессом (на уровне уже полученных знаний) изменяются значения следующих параметров: идентификатор процесса - PID; идентификатор родительского процесса - PPID.

Системный вызов для порождения нового процесса:

Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h> pid_t fork(void);
```

Описание системного вызова

Системный вызов fork служит для создания нового процесса в операционной системе UNIX. Процесс, который инициировал системный вызов fork, принято называть родительским процессом (parent process). Вновь порожденный процесс принято называть процессом-ребенком (child process). Процесс-ребенок является почти полной копией родительского процесса. У порожденного процесса по сравнению с родительским изменяются значения следующих параметров:

- идентификатор процесса;
- идентификатор родительского процесса;
- время, оставшееся до получения сигнала SIGALRM;
- сигналы, ожидавшие доставки родительскому процессу, не будут доставляться порожденному процессу.

При однократном системном вызове возврат из него может произойти дважды: один раз в родительском процессе, а второй раз в порожденном процессе. Если создание нового процесса произошло успешно, то в порожденном процессе системный вызов вернет значение 0, а в родительском процессе - положительное значение, равное идентификатору процесса-ребенка. Если создать новый процесс не удалось, то системный вызов вернет в инициировавший его процесс отрицательное значение.

Системный вызов fork является единственным способом породить новый процесс после инициализации операционной системы UNIX.

В процессе выполнения системного вызова fork() порождается копия родительского процесса и возвращение из системного вызова будет происходить уже как в родительском, так и в порожденном процессах. Этот системный вызов является единственным, который вызывается один раз, а при успешной работе возвращается два раза (один раз в процессе- родителе и один раз в процессе-ребенке)! После выхода из системного вызова оба процесса продолжают выполнение регулярного пользовательского кода, следующего за системным вызовом.

Для того чтобы после возвращения из системного вызова fork() процессы могли определить, кто из них является ребенком, а кто родителем, и, соответственно, по-разному организовать свое поведение, системный вызов возвращает в них разные значения. При успешном создании нового

процесса в процесс-родитель возвращается положительное
значение, равное идентификатору процесса-ребенка. В процесс-ребенок же возвращается значение 0. Если по какой-либо причине создать новый процесс не удалось, то системный вызов вернет в инициировавший его процесс значение -1. Таким образом, общая схема организации различной работы процесса-ребенка и процесса-родителя выглядит так:

```
pid = fork();
if(pid == -1){
      ...
      /* ошибка */

} else if (pid == 0){
      ...
      /* ребенок */
      ...

} else {
      ...
      /* родитель */
      ...
}
```

Завершение процесса. Функция exit()

Существует два способа корректного завершения процесса в программах, написанных на языке С. Первый способ мы использовали до сих пор: процесс корректно завершался по достижении конца функции main() или при выполнении оператора return в функции main(), второй способ применяется при необходимости завершить процесс в каком-либо другом месте программы. Для этого используется функция exit() из стандартной библиотеки функций для языка С. 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**.

Возврата из функции в текущий процесс не происходит и функция ничего не возвращает. Значение параметра функции exit() - кода завершения процесса - передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс. На самом деле при достижении конца функции main() также неявно вызывается эта функция со значением параметра 0.

Функция для нормального завершения процесса:

Прототип функции

```
#include <stdlib.h> void exit(int status);
```

Описание функции

Функция exit служит для нормального завершения процесса. При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков (файлов, pipe, FIFO, сокетов), после чего инициируется системный вызов прекращения работы процесса и перевода его в состояние **закончил исполнение**

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.

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. Процессы, находящиеся в состоянии **закончил исполнение**, в операционной системе UNIX принято называть процессами-зомби (zombie, defunct).

Системный вызов exec()

Изменение пользовательского контекста процесса. Семейство функций для системного вызова exec()

Для изменения пользовательского контекста процесса применяется системный вызов exec(), который пользователь не может вызвать непосредственно. 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().

Функции изменения пользовательского контекста процесса

Прототипы функций

```
#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[])
```

Описание функций

Для загрузки новой программы в системный контекст текущего процесса используется семейство взаимосвязанных функций, отличающихся друг от друга формой представления параметров.

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.

Дерево процессов. Идентификаторы процессов.

В загрузочном образе ОС GNU/Linux присутствует специальный процесс - init. В начале своей работы он считывает файл, сообщающий о количестве терминалов. Затем он разветвляется, порождая по процессу на терминал. Эти процессы ждут, пока кто-нибудь не зарегистрируется в системе. Процесс регистрации порождает оболочку для приема команд, которые могут породить другие процессы и т.д. Поэтому все процессы принадлежат единому дереву, в корне которого находится init. Получить дерево процессов можно при помощи команды pstree, рис. 1.

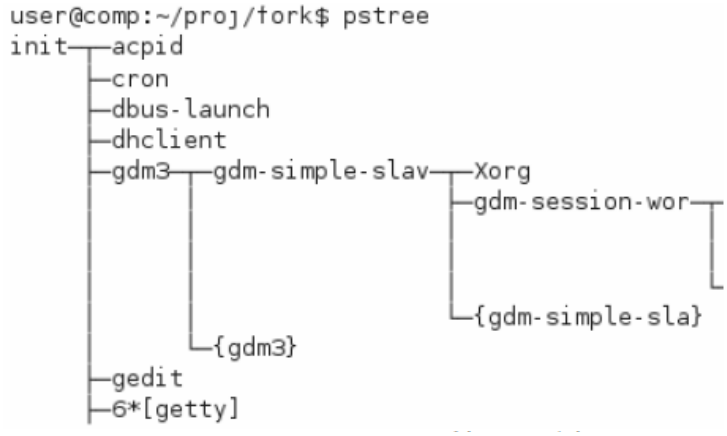

Рис 1. Дерево процессов в ОС Debian GNU/Linux.

На рисунке видно, что главный процесс init запустил дочерние процессы: acpid, cron и другие. Один из дочерних процессов, gdm3 запустил, в свою очередь, два других дочерних процесса: gdm-simple-slav и {gdm3}. Процесс gdm-simple-slav также запустил несколько дочерних процессов и т.д. Таким образом при запуске команды pstree можно наблюдать дерево процессов, образованное от первичного процесса init. Каждый процесс имеет уникальный идентификатор (PID).

Функции int getpid(); int getppid() возвращают идентификаторы текущего и родительского процессов соответственно.

Пример 1. Программа выводит на дисплей PID собственный и родительского процесса.

```
/* Получение идентификатора процесса */
#include<stdio. h>
#include<unistd.h>
int main(){
      printf ("Process ID: %d\n", (int) getpid() );
      printf ("Parent process ID: %d\n", (int) getppid());
      getchar();
      return 0;
}
```

Листинг 1. Результат работы программы.

```
user@comp:~/proj/fork$ ./getpid

Process ID: 2302
```

```
Parent process ID: 2261
```

После вывода идентификаторов процессов программа ожидает нажатия клавиши. В это время следует запустить отдельный терминал bash и ввести команду `ps -axf`, выводящую список процессов.

Листинг 1а. Список процессов

```
user@comp:~$ ps -axf
  PID TTY      STAT   TIME COMMAND
    2 ?        S      0:00 [kthreadd]
    3 ?        S      0:00 \_ [migration/0]
.....
 2259 ?        Sl     0:02 gnome-terminal
 2260 ?        S      0:00 \_ gnome-pty-helper
 2261 pts/0    Ss     0:00 \_ bash
 2302 pts/0    S+     0:00 |  \_ ./getpid
 2305 pts/1    Ss     0:00 \_ bash
 2318 pts/1    R+     0:00     \_ ps -axf
```

В списке по полученным идентификаторам можно отыскать:

- родительский процесс - терминал bash (PID=2261);
- являющийся дочерним по отношению к bash процесс getpid (PID=2302);
- отдельно запущенный для вызова команды ps -axf терминал bash (PID=2305);
- процесс, отображающий список процессов ps (PID=2318).

Следует обратить внимание, что идентификаторы процессов увеличиваются, т.е. последний запущенный процесс будет иметь наибольший идентификатор.

Создание процессов.

Создание новых процессов в ОС GNU/Linux производится при помощи системного вызова, описанного в заголовочном файле <`unistd.h`> `pid_t fork()`.

В результате этого вызова создается новый процесс, являющийся почти точной копией исходного. У дочернего процесса свое адресное пространство, поэтому если в нем изменить значение переменной, то это никак не повлияет на значение переменной в родительскому процессе.

Следующая после вызова fork команда будет выполнена уже обоими процессами.

Для того, чтобы процессы знали, кто из них родительский, а кто - дочерний существует два способа. Первый способ основан на получении идентификатора при помощи вызова `getpid`. У родительского процесса он будет равен идентификатору до выполнения вызова fork. У дочернего он будет иметь большее значение, т. к. процесс запущен позже родительского.

Второй способ основан на анализе результата вызова fork. Возвращаемое значение типа pid_t (аналог типа int):

- = PID дочернего для родительского процесса;
- = 0 для дочернего процесса.

Функция `wait` приостанавливает родительский процесс до тех пор, пока не завершится один из его дочерних процессов:

```
pid_t wait(int *stat_loc);
```

Для ожидания завершения конкретного дочернего процесса используется функция:

```
pid_t waitpid(pid_t pid, int *stat_loc, int options),
```

где `pid` - идентификатор дочернего процесса

Пример 2. Программа создает дочерний процесс. Для определения «кто есть кто» используется значение, возвращаемое вызовом fork. Перед вызовом fork переменной A присваивается значение 0. В дочернем процессе переменная увеличивается на 1. Родительский процесс ожидает окончания дочернего, после чего выводит значение переменной A.

```
/* вызов fork */
```

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char* argv[]) {
     int A = 0,child_pid;
     printf ("Before fork():\nA = %d\n",A);
     child_pid = fork(); // создание дочернего процесса

     if (child_pid == 0) {  // это дочерний процесс
          printf("It's the child process. Child_pid = %d; getpid = %d;
                    A = %d\n",child_pid,getpid(),A);
          A++;
          printf("It's the child process. Change A = %d\n Press
                    Enter..\n",A);
          getchar();
     }

     else if (child_pid > 0)  {  // это родительский процесс
          wait(child_pid);
          printf ("It's the main process. Child_pid = %d; getpid = %d;
                    getppid = %d; A = %d\nPress enter \n",.
                    child_pid,getpid(),getppid(),A);
          getchar();
     }
     else printf("fork error");
     return 0;
}
```

Листинг 2. Результат работы программы.

```
user@comp:~/proj/fork$ ./fork Before fork():
A = 0
It's the child process. Child_pid = 0; getpid = 1594; A = 0 It's the
child process. Change A = 1 Press enter..

It's the main process. Child_pid = 1594; getpid = 1593; getppid = 1515;
A =0
Press enter..
```

В отдельном терминале до завершения дочернего процесса следует выполнить команду ps -axf.

Листинг 2а. Список процессов.

```
1514 ?  S  0:00 \_ gnome-pty-helper
1515    Ss 0:00 \_ bash
1593    S+ 0:00 | \ ./fork
1594    S+ 0:00 | \ ./fork
1538    Ss 0:00 \_ bash
1595    R+ 0:00 \ ps -axf
```

Для замещения образа памяти процесса используется одна из разновидностей вызова `exec`
`int execvp(const char *file, char *const argv[]).`

Если применить его к обычному процессу, то выполнится программа, указанная в переменной file.

Пример 3. Программа запускает команду ls с аргументом /home/user. После успешного завершения ls команды, указанные после execvp не выполняются.

```
/* Замещение выполняемого процесса другой программой */
#include <stdio.h>
#include <unistd.h>
int main(int argc, char* argv[], char* envp[]) {
```

```
      char program [] = "ls";
      char* arg_list [10] = {"ls","/home/user"};
      execvp (program,arg_list);
      printf("Error on program start");
      return 123;
}
```

Листинг 4. Результат работы программы.

```
user@comp:~/proj/fork$ ./exec

1.txt Desktop distr proj spisok tmp tmp1 tmp2
```

Совместная работа вызовов fork и exec.

При последовательном использовании вызовов fork и exec выполняемый процесс вначале разветвляется на два, а затем вместо второго запускается другая программа.

Пример 5. Программа создает дочерний процесс, в котором выполняется вызов exec.

```
/* Запуск программы при помощи fork и exec */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]){
      int child_pid = fork();
      if (child_pid == 0) {
            printf("It's the children process:\n");
            char* arg_list [] = {"ls","/home/user"};
            execvp ("is", arg_list); exit(1);
            }
      else {
            wait(child_pid);
            printf("It's the main process. Press Enter.. \n");
            getchar();
            }
      return 0;
}
```

Листинг 5. Результат работы программы.

```
user@comp:~/proj/fork$ ./fork_exec
It's the children process:
1.txt Desktop distr er proj spisok tmp tmp1 tmp2
It's the main process. Press Enter..
```

Для завершения процессов используется функция `int kill(pid_t pid, int sig)`,

- где pid - идентификатор процесса;
- sig -сигнал, посылаемый процессу.

Пример 6. Программа создает дочерний процесс и после паузы отправляет ему сигнал завершения.

```
/* Завершение дочернего процесса */
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
int main(int argc, char* argv[]){
      int child_pid = fork();
```

```
      if (child_pid == 0) {
            printf("It's the children process:\n"); sleep(1);
            char* arg_list[] = {""};
            execvp ("top", arg_list);
            exit(1);
            }
      else {
            int k;
            sleep(7);
            k = kill (child_pid,SIGTERM);
            wait(child_pid);
            printf("Children process terminated..\n");
            printf("It's the main process. Press Enter.. \n");
            getchar();
            }
      return 0;
}
```

Листинг 6. Результат работы программы.

```
user@comp:~/proj/fork$ ./kill

It's the children process:
- 19:38:15 up 21 min, 2 users, load average: 0.00, 0.05, 0.04 Tasks: 92
total, 1 running, 91 sleeping, 0 stopped, 0 zombie Cpu(s): 0.7%us,
1.0%sy, 0.0%ni, 98.0%id, 0.3%wa, 0.0%hi, 0.0%si,
0.0%st
Mem: 124784k    total,     120140k    used, 4644k free, 2228k buffers
Swap: 223224k total, 4724k used, 218500k free, 50796k cached
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+
COMMAND
1585 user 20 0 2440 1144 900 R 1.0 0.9 0:00.09 top 143 root 20 0  0    0
     0    S    0.3  0.0  0:00.32    ata/0
965 root 20 0 30396 16m 5884 S 0.3 13.8 0:12.24 Xorg
1  root  20  0  2036  644  612  S  0.0  0.5  0:03.42  mit  Children  process
terminated..
It's the main process. Press Enter..
```

Задание:

1. Написать, скомпилировать и отладить примеры программы из теоретической части.
2. Выполнить задание по варианту (см. далее).
3. Составить отчет о выполненной работе

Варианты заданий.

1. В дочернем процессе получить список процессов, записать его в файл и завершить процесс. В родительском процессе дождаться окончания дочернего, получить список процессов и дописать его в созданный дочерним процессом файл. Сравнить списки процессов.
2. Главный процесс в течение 20 секунд через каждые 5 секунд порождает по одному дочернему процессу. Дочерние процессы дописывают в файл свои идентификаторы (PID). Главный процесс ждет окончания дочерних и дописывает в файл идентификаторы дочерних процессов.
3. Определить в главном процессе целочисленную переменную. Дочерний процесс увеличивает эту переменную в 2 раза и проверяет, не превысило ли значение переменной числа, заданного в качестве аргумента командной строки. Если превысило, то процесс завершается, иначе вновь создается аналогичный дочерний процесс. Главный процесс определяет количество созданных дочерних процессов.
4. Открыть файл и записать в него первое сообщение («Hello »). После разветвления процесса в дочернем записать в файл второе сообщение («World»). Выполнить программу, просмотреть созданный файл и объяснить результат. Что будет, если попытаться закрыть файл в

дочернем процессе, а в родительском после ожидания wait(child_pid) записать в файл третье сообщение?

5. Программа принимает в качестве аргумента имя файла. Родительский процесс создает дочерний, через временный файл передает ему имя файла, принимает в качестве ответа количество слов в файле и выводит результат на дисплей.
6. Программа принимает в качестве аргумента строку. Родительский процесс создает дочерний, через временный файл передает ему строку, принимает в качестве ответа количество символов в строке и выводит результат на дисплей.
7. Программа принимает в качестве аргумента полный путь к каталогу. Родительский процесс создает дочерний, через временный файл передает ему путь, принимает в качестве ответа количество файлов в каталоге и выводит результат на дисплей.
8. Родительский процесс последовательно запускает в дочерних все исполняемые файлы, которые он сможет найти в указанном каталоге. После запуска последнего происходит задержка в 10 секунд, после чего все дочерние процессы закрываются и выводится на дисплей их количество.