Министерство образования Республики Беларусь

Учреждение образования

“Гомельский государственный университет имени Ф. Скорины”

Отчет по лабораторной работе

Лабораторная работа №8.

**Утилита Make**

Выполнил:

Студент группы КФ-17

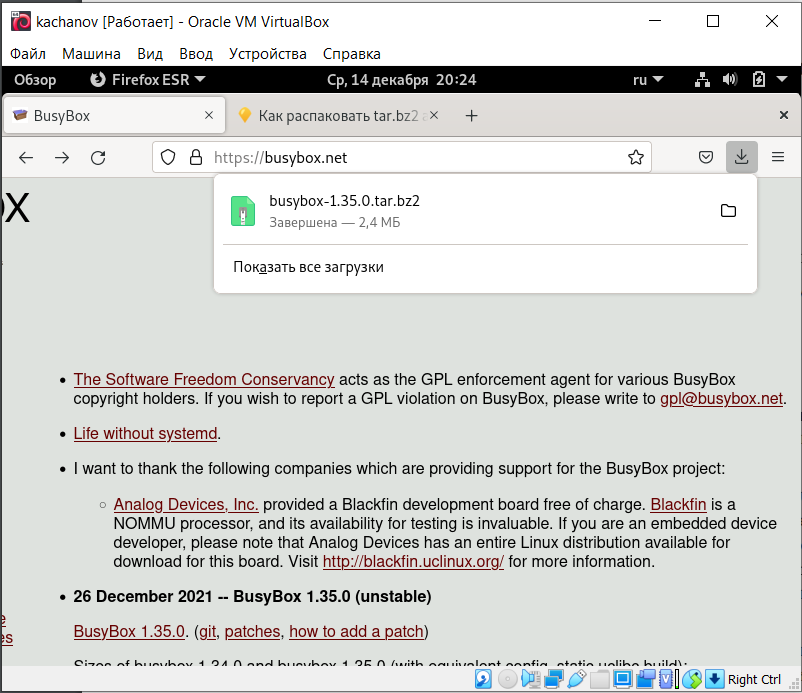
Гуревич П.A.

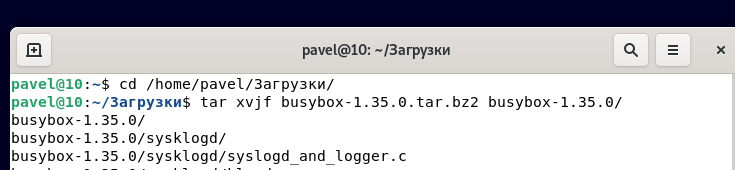
Гомель 2022

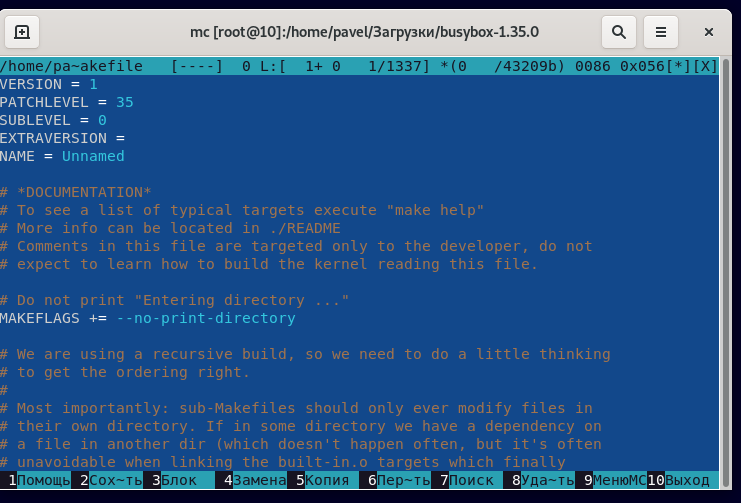
**Лабораторная работа №8.**

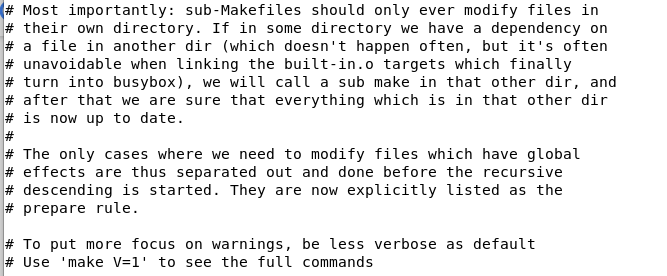
**Утилита Make**

1. Изучите главу 9 теоретического курса «Утилита Make».
2. При необходимости установите программный пакет make.
3. Скачайте с сайта www.busybox.net исходный код программы Busybox и распакуйте его в подкаталог домашнего каталога. Изучите файл Makefile из основного каталога проекта. Определите главную цель в файле и зависимости главной цели. Обратите внимание на переменные CROSS\_COMPILE и ARCH в данном файле. Они предназначены для кросскомпиляции проекта. Исследуйте, каким образом они используются в файле

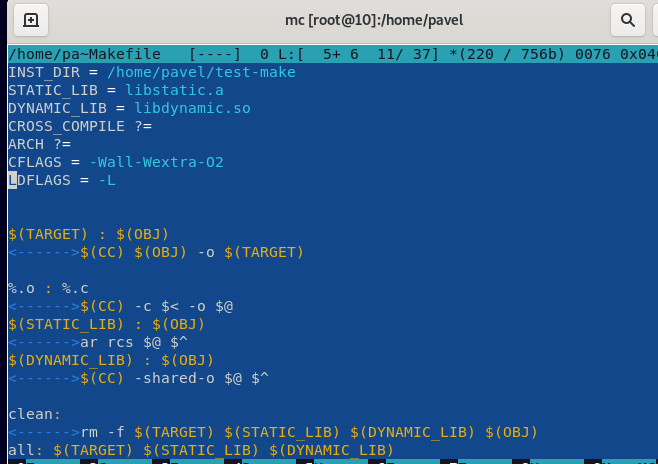
****



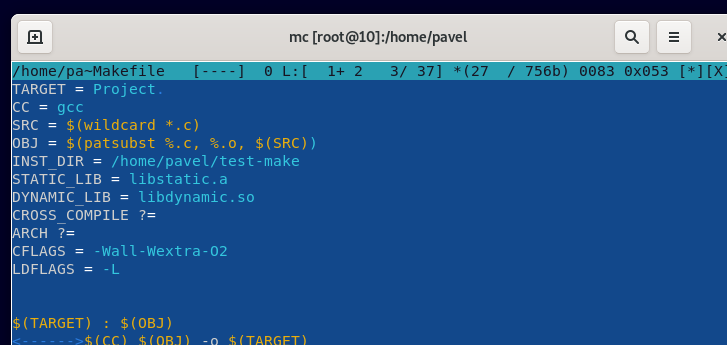


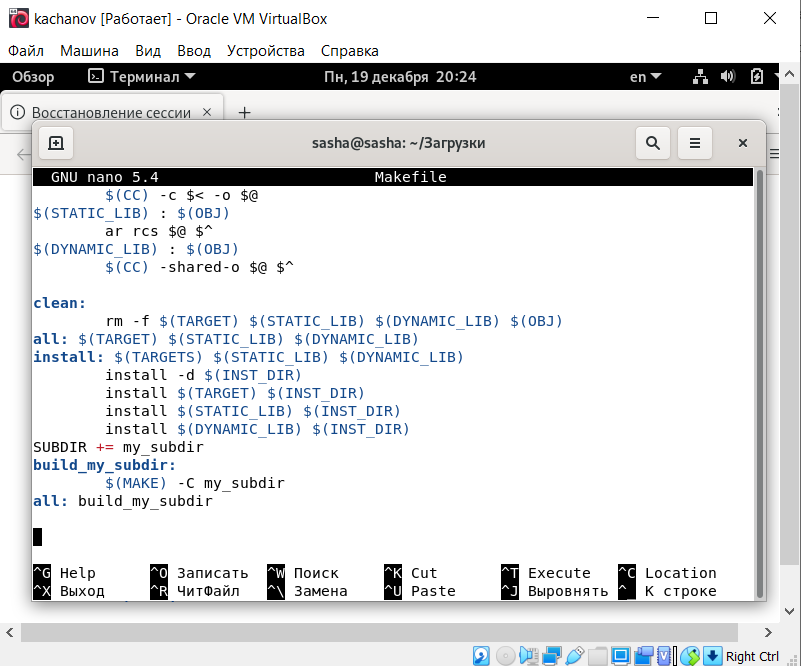
****

1. Разработайте свой собственный Makefile для задачи компиляции, выполняемой в предыдущей лабораторной работе. В файле должны присутствовать цели all, clean, install. Установка при выполнении цели install должна производиться в каталог, который определен в переменной INST\_DIR, имеющей значение по умолчанию «/home/user/test\_for\_make/». Также в файле должны быть цели для сборки исполняемого файла, статической библиотеки и динамической библиотеки.



1. Доработайте созданный Makefile так, чтобы при компиляции можно было использовать кросс-компилятор (используя переменные CROSS\_COMPILE и ARCH).



****

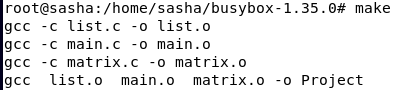
1. Запишите полученный Makefile и файлы, необходимые для компиляции, в подкаталог исходных кодов программы Busybox. Внесите необходимые изменения для того, чтобы разработанный вами файл вызывался при сборке программы Busybox (для этого нужно добавить в общий файл сборки дополнительную цель, которая запустит команду make в другом каталоге). Выполните сборку программы Busybox следующими командами:

$ make defconfig

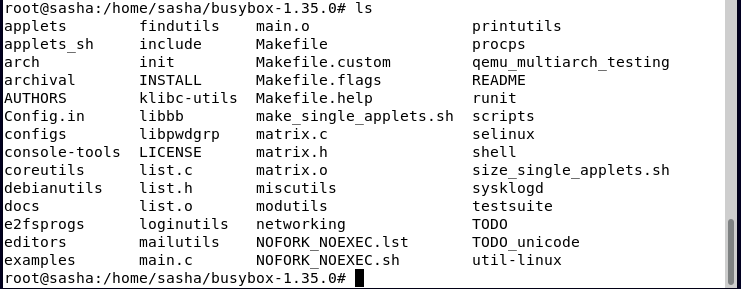
$ make

Убедитесь, что сборка вашей программы тоже выполнена.

1)make

****

**2)ls**

****

**КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. В предыдущей главе мы с вами рассмотрели, как можно скомпилировать сравнительно простые программы, используя компилятор GCC. При этом во многих случаях для компиляции программы нам требовалось выполнить не одну, а несколько команд. В сложных программах для их полной сборки может потребоваться выполнить десятки и даже сотни команд. Конечно, можно было бы записать эти команды в скрипт и выполнять его. Сложность такого метода состоит в том, что все команды должны идти в определенной последовательности, т.к. результаты выполнения некоторых из них нужны для выполнения других команд.Такой файл будет непросто написать, а еще сложнее — его сопровождать.

К счастью, есть метод лучше — использовать специализированные системы сборки. Системы сборки сами определяют порядок выполнения команд. Они так же способны определить, какие файлы исходных текстов программы были изменены с момента последней компиляции, и перекомпилировать только их.

Кроме собственно сборки программы из исходных текстов, системы сборки позволяют автоматизировать любые операции, которые требуется выполнять в порядке, определяемом зависимостями по данным между ними. Обычно системы сборки используются для:

* конфигурации программы — определения, какие именно функции сложной программы нужно включать в создаваемый исполняемый файл;
* компиляции программы из исходных текстов;
* установки программы — записи созданных программных файлов в целевые каталоги;
* удаления временных файлов, созданных при компиляции и т.п.

Утилита Make обладает двумя важными характеристиками:

* она автоматически определяет порядок, в котором нужно выполнить команды для сборки программы, исходя из условия, что файлы, которые требуются для компиляции на каждом этапе, должны быть получены на предыдущих этапах;
* она предотвращает повторную компиляцию тех файлов, которые не изменялись с момента предыдущей компиляции.

Для использования утилиты **Make** нужно в рабочем каталоге программы создать файл с именем Makefile, содержащий инструкции по сборке программы в специальном формате. В этом случае сборка будет производиться простым вызовом команды **make**, возможно с некоторыми параметрами.

Цель может быть *именем файла*, который необходимо получить, а может быть просто обозначением некоторой операции, не совпадающим с именем файла (в этом случае говорят об *абстрактной цели*).

Далее после двоеточия через пробел перечисляются *зависимости* — определяющие действия, которые должны быть выполнены заранее для получения цели. Обычно в качестве зависимости указывается имя файла или имя другой цели.

После этого запускается на выполнение **рекурсивный алгоритм обработки правил**. Обработка одного правила включает в себя два этапа:

* обработка зависимостей;
* выполнение команд.

При этом выполнение команд производится не всегда, а только при выполнении условий, о которых будет сказано ниже.

*Первый*этап при обработке правил — *обработка зависимостей*. Здесь для каждой из зависимостей проверяется, является ли данная зависимость целью какого-либо другого правила. Если ответ на этот вопрос утвердительный, то найденное правило также обрабатывается в соответствии с алгоритмом обработки правил.

*Второй*этап при обработке правил — *выполнение команд*. Команды, содержащиеся в правиле, выполняются, если оказывается истинным одно из следующих условий:

* цель данного правила является абстрактной;
* цель данного правила является именем файла, и такого файла не существует;
* какая-либо из зависимостей данного правила является абстрактной целью;
* цель данного правила является именем файла, и какая-либо из зависимостей, являющихся именем файла, имеет более позднее время модификации, чем цель.

|  |  |
| --- | --- |
| Цель | Описание |
| all | Сборка основных исполняемых файлов и библиотек. |
| clean | Удаление всех временных файлов, созданных на этапе сборки |
| configure | Запуск процедуры конфигурации программы – определения компонентов, которые необходимо включить в исполняемый файл |
| distclean | Удаление временных файлов, созданных на этапах сборки и конфигурации |
| install | Установка созданных на этапе сборки файлов в соответствующие каталоги в системе [1] |
| uninstall | Удаление файлов из служебных каталогов, которые были размещены там командой make install |

В утилите **Make** существует механизм, аналогичный макросам в языке C. И хотя в Make чаще используется термин «переменная», по сути это все те же макросы. Мы будем использовать термин «переменная». Рассмотрим, для чего же они нужны.

При сборке может потребоваться передавать компилятору те или иные параметры, например, нам потребуется скомпилировать программу с отладочной информацией, задав для **gcc** ключ **–g**. Ясно, что записывать и удалять вручную ключ **–g** во всех вызовах компилятора — неудачная идея. Для решения подобных задач в Make предназначены переменные. Они позволяют определить переменную с этим ключом только в одном месте Make-файла, а во всех местах, где требуется данный ключ, вставить значение этой переменной. Таким образом, переменные в Make, как и макросы в C — это способ борьбы с избыточностью.

В именах переменных в Make часто используют только заглавные буквы, как и в макросах C, хотя это и не является обязательным.

Кроме этого, пользователи могут создавать свои *шаблонные правила*. Шаблонное правило должно содержать знак «%» в имени цели, и притом только один. Цели при этом рассматривается как шаблон имени файла. Знак «%» соответствует любой непустой подстроке, остальные символы должны совпадать. Например, шаблонное правило для компиляции файлов программ на языке C может иметь вид:

$ cat Makefile\_pattern   
%.o: %.c   
   gcc -c $< -o $@   
$ make -f Makefile\_pattern file1.o   
gcc -c file1.c -o file1.o

|  |  |
| --- | --- |
| Переменная | Описание |
| $@ | Имя файла цели правила |
| $< | Имя первой зависимости |
| $? | Имена всех зависимостей, которые являются более новыми, чем цель |
| $^ | Имена всех зависимостей |

1. Утилита Make используется достаточно широко. Исторически она является первой системой сборки, и, по этой причине, стала стандартом де-факто. Тем не менее, она не лишена недостатков.

Первая группа недостатков связана с низкой производительностью Make.

Вторая группа недостатков связана с некоторой запутанностью языка Make. Здесь можно упомянуть и необходимость использования табуляции перед командами в правилах, и отсутствие строгого синтаксиса языка, и некоторую запутанность механизма работы с переменными, и просто отсутствие в языке множества полезных конструкций, например, циклов.

Для исправления недостатков Make было разработано достаточно большое количество альтернативных систем сборки проектов. В качестве примеров можно привести системы:

CMake, использующая синтаксис языка C;

Apache Ant и NAnt, использующие язык XML;

SCons, использующая синтаксис языка Python;

Rake, использующая синтаксис языка Ruby.