Lab Work 1 - Tools and Platform

The practical side of this course naturally requires a lot of programming work. We are going to write codes using C programming language - the one that is most suitable to develop an operating system. We will be working on Linux platform.

Revisiting C Programming

This is mainly just to look back at basic C code structure - some of these are never taught in the basic programming course. The codes in this topic emphasize on C language standard - you should be able to compile and run the codes on any platform that has a C compiler. You just have to remember that some libraries only exists on specific platforms - so, make sure you know your target and development platforms. There are also some info on project management using makefiles.

Development Environment

If you really understood what you did in your introductory programming course, development on Linux platform is no different compared to using Windows platform. You obviously need a text editor to write your code, a C compiler to compile your code and a linker to create the executable. What you most probably have used is an Integrated Development Environment (IDE) which combines all of them into a single environment which is mainly the editor - the compiler and linker are just menu items or buttons somewhere within.

It is advisable to manage your source file in proper folder/path hierarchy. One thing about naming files and paths is NEVER use a whitespace character (' ') in them, no matter what Windows says about them not being a problem.

Code Editing

The source file is just a text file - so, you can actually use any text editor. On Windows platform, you can in fact use `notepad` or even the console-based `edit` (relic from DOS era) to write your code. The downside of using these editors are they lack syntax-highlighting which is very useful to start learning programming by helping you spot keyword/structure-related syntax errors early on. There are a lot of free code editors (text editor specifically for programming purpose that provides syntax highlighting) - a recommended editor is Geany.

On Linux, IDEs are also available but many veteran programmers simply use Linux text editors which naturally supports syntax highlighting. Among the popular editors are `Emacs` and `Vim`. On popular distributions like Ubuntu, the supplied GUI-based editor (GNOME editor a.k.a. `gedit`, which is available on all distributions that use GNOME desktop) should be sufficient. On KDE desktop, `kwrite` or `kate` does the job pretty well.

Compiling (and Linking)

The instructions in this section require you to use the terminal. If you are using an IDE, just look for a menu item or button that says compile or build or make or something similar (they actually do the same thing as what is mentioned here).

If you have a single source file `this.c` this process can be done in a single step:

```
gcc this.c
```

By default, the `gcc` (GNU Compiler Collection instead of GNU C Compiler) compiler will create an executable called `a.out`. If you want to specify a name (e.g. `this`) you can use the -o parameter:

```
gcc -o this this.c
```

It does not need be similar to the name of your source file:

```
gcc -o what_the this.c
```

Note that both compiling and linking can be done in this single command. They are actually separate processes and most of the time, it is very useful to execute them in separate stages especially when managing a software with multiple source files.

Managing a Project

If you are using a full-fledged IDE, you will find everything but the kitchen sink available (sometimes a sink can also be found ). One of the things that `make` an IDE seems to work better is having a project management flow, i.e. keeping tracks of source files required to build the project, which compiler to use, etc. For an old school developer, using a `makefile` (requires `make` tool) makes more sense. This is what I am going to introduce here.

Like other configuration files in Unix/Linux systems, writing a `makefile` (which is also a simple text file) is relatively easy. Any strings following the '#' character are treated as comments. A simple example of a `makefile` that can be used to compile a C program from a single 'single.c' source file is shown below.

makefile

```
# makefile to compile a c program
single: single.c
    gcc -o single single.c
```

I will explain the syntax after the next example. Take note that there should be a single hard tab before the gcc command. The given example however is not very convenient because I have to change a lot of things (three words at least in the example) if I decided to change file or project name.

The example below gives us a more generic `makefile`.

makefile

```
# makefile to compile a c program
PROJECT = single
SOURCE = $(PROJECT).c
CC = gcc
$(PROJECT): $(SOURCE)
    $(CC) -o $(PROJECT) $(SOURCE)
```

The two common things that we do in a makefile is to define build rules and to define variables. The lines that contains a '=' is usually a variable definition. Any environment variable is imported into our `make` environment. A rule is defined by specifying a target that is done by specifying a label (or target object name) followed by a ':' character. In our example, the target is $(PROJECT) - which is a variable and expands to `single`. Anything that comes after ':' will be considered as a prerequisite. If a target is older than any of its prerequisites, it will be rebuilt by `make` tool. The build process is defined by the lines following the target (there can be more than 1 line). Notice that the MUST BE a hard tab character at the beginning of a process line.

Imagine now that you have a hundred source files. Using the above rule, changing one source file will cause all the other source files to be recompiled again. This is not very efficient. Thus, if you remember you introductory lecture for computer programming, an executable is built by combining object files and linking them to the respective libraries. Object files (*.o or *.obj) can be built independently. So, to have that, we alter our `makefile` to accommodate this.

makefile

```
# makefile to compile a c program
PROJECT = single
OBJECTS = $(PROJECT).o
CC = gcc
CFLAGS =
LDFLAGS =
$(PROJECT): $(OBJECTS)
    $(CC) $(CFLAGS) -o $@ $+ $(LDFLAGS)
%.o: %.c
    $(CC) $(CFLAGS) -c $<
```

There are a few new things here. Special variables like $@, $+ and $< are now used. To know what they mean, you may refer to the note at the end of this section. The prerequisites are now object files that needs to be built as well. In order to build these, `make` will refer to the second rule `%.o: %.c` - which says that any object files (*.o) needs to be built depends on a source C file (*.c) that will be built using gcc with a `-c` parameter (compile only).

One thing you should know is `make` will use the first matching rule that it finds. So, if you have some object files that depends on both source (*.c) and header (*.h) files, you need to create the rule (`%.o: %.c %.h`) *before* the rule given above.

The given information so far about makefiles should be sufficient for basic projects. You will learn, in time, of more advanced ways to specify rules that will show you how the `make` tool can be a very powerful build management tool.

Some notes on special `make` variables:

```
# $@ ==> target file
# $? ==> changed dependents
# $* ==> shared target/dependent prefix
# $< ==> first dependent
# $^ ==> all dependents, duplicates omitted
# $+ ==> all dependents, order retained
```

Writing Programs in C

All programs written in C must have a `main()` function, which is the starting point of a program. A complete and valid C code can be:

nothing.c

```
void main(){}
```

Notice that, unlike assembly language, a C program is not processed line by line - so, you can actually have a source code file without any newline character. However, this will cause a terrible headache when you need to debug the code.

The code in `nothing.c` does not do anything - absolutely nothing! This, in fact, just shows the syntax for a C function. A slightly bigger code can make the program do something:

zero.c

```
int main() { return 0; }
```

The code now returns (to the operating system) an integer value of 0. You may think this still does nothing but in a practical operating systems, this can be very helpful for system administrators. In Unix systems there are binaries to do just that - return 0 (/usr/bin/true) and 1 (/usr/bin/false). Notice also that a zero is usually a success (true) so that we can use the non-zero failure (false) values as error codes.

It is sometimes very useful to be able to provide information (@input) to the program at command line. This can be made possible by declaring 2 parameters:

param.c

```
#include<stdio.h>
int main(int argc, char *argv[])
```

```
{
    int loop;
    printf("Argument count: %d\n",argc);
    for(loop=0;loop<argc;loop++)
    {
        printf("Argument #%02d: '%s'\n",loop,argv[loop]);
    }
    return 0;
}
```

Notice that this is the first time `#include<stdio.h>` is used - only because we now want to use the `printf` function which is declared (prototyped) in that particular header file. Take note that we should only include the relevant header file(s) when we need to use a declaration provided by that include file. In the above code, the first parameter is an integer value that provides the number of arguments that is about to be passed, while the second parameter is an array of pointers to the argument strings. Unlike most programming languages, C does not have that (string) as a primitive data type and this is simply because there is no such thing at hardware level. The processor will always process a string one character at a time.

Temporary Dumping Ground

Some stuffs to share...

Primitive Variable Size

Variable in C - a simple C code to check C variable size.

chksize.c

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

int main(int argc, char** argv)
{
    int loop;
    if (argc<2)
    {
        printf("Sizeof 'unsigned int' is %d\n",(int)sizeof(unsigned
int));
        printf("Sizeof 'unsigned char' is %d\n",(int)sizeof(unsigned
char));
        printf("Sizeof 'unsigned short' is %d\n",(int)sizeof(unsigned
short));
        printf("Sizeof 'unsigned long' is %d\n",(int)sizeof(unsigned
long));
```

```
        printf("Sizeof 'unsinged long long' is
%d\n",(int)sizeof(unsigned long long));
        printf("Sizeof 'float' is %d\n",(int)sizeof(float));
        printf("Sizeof 'double' is %d\n",(int)sizeof(double));
        printf("Sizeof 'long double' is %d\n",(int)sizeof(long
double));
    }
    for (loop=1;loop<argc;loop++)
    {
        if (!strcasecmp(argv[loop],"int"))
        {
            printf("Sizeof 'unsigned int' is %d\n",(int)sizeof(unsigned
int));
        }
        else if (!strcasecmp(argv[loop],"char"))
        {
            printf("Sizeof 'unsigned char' is
%d\n",(int)sizeof(unsigned char));
        }
        else if (!strcasecmp(argv[loop],"short"))
        {
            printf("Sizeof 'unsigned short' is
%d\n",(int)sizeof(unsigned short));
        }
        else if (!strcasecmp(argv[loop],"long"))
        {
            printf("Sizeof 'unsigned long' is
%d\n",(int)sizeof(unsigned long));
        }
        else if (!strcasecmp(argv[loop],"float"))
        {
            printf("Sizeof 'float' is %d\n",(int)sizeof(float));
        }
        else if (!strcasecmp(argv[loop],"double"))
        {
            printf("Sizeof 'double' is %d\n",(int)sizeof(double));
        }
        else if (!strcasecmp(argv[loop],"llong"))
        {
            printf("Sizeof 'long long' is %d\n",(int)sizeof(unsigned
long long));
        }
        else if (!strcasecmp(argv[loop],"ldouble"))
        {
            printf("Sizeof 'long double' is %d\n",(int)sizeof(long
double));
        }
    }
    return 0;
}
```

Up-to-date version should always be available here.

Practical makefile

This is what I use to quickly compile a C source code (usually to test some things).

makefile

```
# makefile for single source file app (from my1codeapp)

ALLAPP = $(subst .c,,$(wildcard *.c))
ALLAPP += $(subst .cpp,,$(wildcard *.cpp))

TARGET ?=
TGTLBL ?= $(TARGET)
ifeq ($(TGTLBL),)
    TGTLBL = app
endif
OBJLST ?=
DOFLAG ?=
DOLINK ?=

CC = gcc
CP = g++

DELETE = rm -rf

CFLAGS += -Wall
# i prefer to build static bin
CFLAGS += -static
LFLAGS += $(LDFLAGS)
OFLAGS +=
# when working with large files
XFLAGS += -D_LARGEFILE_SOURCE=1 -D_FILE_OFFSET_BITS=64
ifneq ($(DOFLAG),)
    CFLAGS += $(DOFLAG)
endif
ifneq ($(DOLINK),)
    LFLAGS += $(DOLINK)
endif

# i can still squeeze in some external code(s) => OBJLST!
EXTPATH = /home/share/store/repomine/my1codelib/src
ifneq ($(wildcard $(EXTPATH)),)
    CFLAGS += -I$(EXTPATH)
endif

.PHONY: dummy $(TARGET)
```

```
# TARGET can be temporary code (reside at top level)
$(TARGET): $(OBJLST) $(TARGET).o
    $(CC) $(CFLAGS) -o $(TGTLBL) $^ $(LFLAGS) $(OFLAGS)

dummy:
    @echo "Run 'make <app>' or 'make TARGET=<app>'"
    @echo "  <app> = { $(ALLAPP) }"
    @echo
    @echo "To set compiler flag (e.g. static), do 'make <app> DOFLAG=-
static'"
    @echo "To link a library (e.g. math), do 'make <app> DOLINK=-lm'"

%: %.c
    $(CC) $(CFLAGS) -o $@ $< $(LFLAGS) $(OFLAGS)

%: %.cpp
    $(CP) $(CFLAGS) -o $@ $< $(LFLAGS) $(OFLAGS)

# to compile those external code(s) => OBJLST!
%.o: $(EXTPATH)/%.c $(EXTPATH)/%.h
    $(CC) -c $(CFLAGS) -o $@ $<

clean:
    -$(DELETE) $(ALLAPP) $(TGTLBL) *.o
```