The Abdus Salam
International Centre for Theoretical Physics

United Nations
Educational, Scientific
and Cultural Organization

International Atomic
Energy Agency

310/1779-5

Fourth Workshop on Distributed Laboratory Instrumentation Systems
(30 October - 24 November 2006)

Linux Refresher (2)

Razaq Babalola IJADUOLA

Strada Costiera 11, 34014 Trieste, Italy - Tel. +39 040 2240 111; Fax +39 040 224 163 - sci_info@ictp.it, www.ictp.it

make

- 'make' is a command generator.
- From a description file and general templates it creates commands for the shell.
- 'make' sorts out dependencies among files.

make continue..

A program often consists of several source files, header files, libraries.

When one of these is modified, you must rebuild the program, by re-compiling some of the files, but not necessarily all, and then re-linking the object files.

'make' decides what must be done, basing itself on the dependencies and the last modification date/time of each file.

If file A depends on file B and B was modified after A had been built, then A must be re-built:

compiled, linked, edited, substituted in a library or what have you.

make *continue..*

Generally you invoke 'make' by typing:

make myprogram

'myprogram' is the target, it is built from one or more files, the 'prerequisites' or 'dependencies'.

The dependencies are specified in a description file (Makefile or makefile), together with the commands to be executed to build the target.

make continue..

Example of a Makefile:

```
program : main.o iodat.o dorun.o lo.o /usr/lib/crtn.a
	cc -o program main.o iodat.o dorun.o lo.o /usr/lib/crtn.a

main.o : main.c
	cc -c main.c

iodat.o : iodat.c
	cc -c iodat.c

dorun.o : dorun.c
	cc -c dorun.c

lo.o : lo.s
	as -o lo.o lo.s
```

Note that there are dependency lines or rule lines containing a ':' and command lines, starting with a 'tab' character.

make *continue..*

The example is rather clumsy, repetitive. Things become simpler by using macros and by exploiting the suffix rules.

A macro is defined as:

name = "a text string" ('quoted' if necessary).

You refer to a macro with:

$(name) or ${name}

make continue..

Example:

```
LIB = -lX11
objs = drawable.o plot_points.o root_data.o
CC = /usr/bin/gcc
23 = "This is the 23rd run"
OPT = # empty now, use later
DB = -g
BINDIR = /usr/local/bin
plot : ${objs}
        ${CC} -o plot ${DB} ${OPT} ${objs} ${LIB}
        mv plot ${BINDIR}
```

make *continue..*

When you now type: make plot the shell will receive the following commands:

/usr/bin/gcc -o plot -g drawable.o plot_points.o root_data.o -lX11

mv plot /usr/local/bin

make continue..

Macros can be nested. The order of definition is immaterial. 'make' has also macros defined internally.

Shell environment variables can be used as macros in a Makefile. So, if you have a shell environment variable:

DIR = /usr/proj
and you have 'exported' it:
export DIR (for bash) or:
setenv DIR /usr/proj (for tcsh)

make continue..

then you may use ${DIR} in your Makefile:

```
SRC = ${DIR}/src

myprog : ....
        cd ${SRC}
```

make *continue..*

These are the very essentials only of 'make'. For the details (many!), see the man pages or the book by A.Oram and S.Talbott, Managing projects with make. The examples were taken from this book.

'make' is absolutely essential for developing software of some complexity, especially if done by a team of programmers.

gcc

gcc stands for "GNU Compiler Collection."
It contains compilers for C, C++, Objective C, Fortran, Java and a few more.

gcc continue..

Usually the compiler chain is invoked with:

gcc [-options] -o Prog file1.c file2.c...filek.s

(or something similar), where Prog is the name of the executable file that is to be produced as the end product.

gcc continue..

C compiler consists of a chain of programs:

- preprocessor (cpp) transforms the '.c' program into a '.i' file, using the '.h' files.
- compiler (cc1) does the real job:

translates C code into assembly code ('.s' file)

- assembler (as) translates the '.s' file into object code ('.o' file)
- linker (ld) 'glues' all '.o' files together

and includes functions from libraries.

A cross-compiler has an overall wrapper program, usually called xgcc.

gcc continue..

The most important options are:

- -g produce information for the debugger.
- -I incdir include '.h' files from 'incdir'.
- -L libdir search 'libdir' for libraries.
- -l libname search the library 'libname' for functions to 'link' in.
- -o prog name the executable: 'prog'.
- -S produce only '.s' file(s). Don't assemble.
- -c produce object ('.o') files. Don't call the linker to produce an executable.
- -Wall give all possible warnings.
- -O2 optimize the code produced.
- -v verbose mode (I find it useful).

gcc continue..

Normally, you will wisely specify the options to use in your Makefile.

gdb

gdb is a symbolic debugger, which means that you can use line numbers and names of variables and functions as defined in the C source code.

The executable to be debugged must have been compiled using the '-g' option of gcc.

What Next?

Finally, after this workshop those who are new to linux and windows users can continue to work in a linux environment such as:

- cygwin can be use to run unix commands and applications in windows. Can also be use to connect to linux remotely. http://www.cygwin.com

- There are projects like Ubuntu, knoppix etc for those who are new to linux.They are easy to install and work with.

http://www.ubuntulinux.org/
http://www.knoppix.org/

What Next? continue...

- Small linux - Damn Small Linux(DSL) and Puppy Linux. These are fully functional linux that can fit on a small CD or jumpdrive between 50-70Meg.

http://www.damnsmalllinux.org/
http://www.puppyos.com/

For the various distributions of linux, try:
http://distrowatch.com

More information about linux from:
http://www.linux.org