Lecture 6: C programming in Linux

Lecturer: Prof. Zichen Xu

Outline

- Tarball
- GCC
- Make and Makefile
- GDB

Tarballs

- Most non-package software is distributed in source code format. The most common format being C project “libraries in compressed TAR files known as “tarballs.
- These files usually contain the string “tar” indicating the file it a tar (tape archive) format file.
- The file may also be compressed indicated by a filename suffix ending in .Z (compress) or .gz (gnuzip).
- These file are then “untarr’red” into the product library as setup by the project programmer

Tar commands

- tar –cvf <tarfilename.tar> <target directories> - creates tar file.
- tar –tvf <tarfilename.tar> - list tar file contents
- tar –xvf <tarfilename.tar> - extracts tar file
- Can add –z flag for newer LINUX distros with gnuzip for automatic compress/decompress (.gz suffux).
- Otherwise try compress command (.Z suffix)
- USAGE:
  - Always create tarfile in target directory (relative file/directory names)
  - Always list tarfile before extracting (insure relative file names)
  - Always extact tarfile in target directory (relative file/directory names)
- Example:
  - tar –xzvf ~/bb-1_3a_tar.gz

What is gcc?

- gcc
  - stands for GNU C/C++ Compiler
  - a popular console-based compiler for *NIX platforms and others; can cross-compile code for various architectures
  - gcc to compile C programs; g++ for C++
  - can actually work with also ADA, Java, and a couple other languages
  - gcc performs all of these:
    - preprocessing,
    - compilation,
    - assembly, and
    - linking
  - we are to use it for our last C assignment
- As always: there is man gcc

The “C” compiler

- “Standard” C is the original language as developed by Ken Thompson et. al.

  This is the format taught in most basic “C” language classes.

- ANSI C first developed in 1987, approved in 1989 (C89). ISO in 1990.

  Introduced OOP programming structures – inheritance etc.

  Source code designated as .c. GNU compiler is gcc.

- C99 came out as 1999 ANSI/ISO standard.

  Enforced stricter syntax rules introduced new arithmetic, data types and structures,

- Current 2011 “standard” version is ISO C11.

  Which includes numerous memory management and multithreading structures.

- C++ (C with classes) developed in 1979 by Bjarne Stroustrup of Bell Labs as a full-blown OOP language. Standardized buy ANSI in 1998 as C++99. Not strictly a superset of ANSI “C” but is backwards compatible with some exceptions.

  Source code designated as .cpp.

  GNU complier is g++.

  Used primarily for hardware level progrmming.

Compile Process

- **Compiler Stage**: All C language code in the .c file is converted into a lower-level language called Assembly language; making .s files.

- **Assembler Stage**: The assembly language code made by the previous stage is then converted into object code which are fragments of code which the computer understands directly. An object code file ends with .o.

- **Linker Stage:** The final stage in compiling a program involves linking the object code to code libraries which contain certain "built-in" functions, such as printf. This stage produces an executable program, which is named a.out by default.

“C” Compile Example

- main.c

```
#include <stdio.h>
#include <iostream>
#include "functions.h"
using namespace std;
int main(){
   print_hello();
   std::cout << endl;
   std::cout << "The factorial of 5 is " << factorial(5) << endl;
   return 0;
}
```

Note the deprecated “cout” instruction requiring special handling under c++.

“C” Compile Example

- hello.c

```
#include <stdio.h>
#include <iostream>
#include "functions.h"
using namespace std;
void print_hello(){
  std::cout << "Hello World!";
}
```

Note the deprecated “cout” instruction requiring special handling under c++.

“C” Compile Example

- factorial.c

```
#include "functions.h"
int factorial(int n){
   if(n!=1){
      return(n * factorial(n-1));    # recursive call
   }
   else return 1;
}
```

- functions.h

```
void print_hello();
int factorial(int n);
```

“C” Compile Example

- Compile the “project”

  g++ main.c hello.c factorial.c –o hello

  This above command will generate hello binary. In our example we have only four files and we know the sequence of the function calls so it may be feasible to write the above command by hand and prepare a final binary. But for the large project where we will have thousands of source code files, it becomes difficult to maintain the binary builds.

- Run it..

  ./hello

  =>Hello World!

  =>The factorial of 5 is 120

Options

- There are zillions of them, but there are some the most often used ones:
  - To compile: -c
  - Specify output filename: -o <filename>
  - Include debugging symbols: -g
  - GDB friendly output: -ggdb
  - Show all (most) warnings: -Wall
  - Be stubborn about standards: -ansi and -pedantic
  - Optimizations: -O, -O*

Options: -c

- gcc performs compilation and assembly of the source file without linking.
- The output are usually object code files, .o; they can later be linked and form the desired executables.
- Generates one object file per source file keeping the same prefix (before .) of the filename.

Options: -o <filename>

- Places resulting file into the filename specified instead of the default one.
- Can be used with any generated files (object, executables, assembly, etc.)
- If you have the file called source.c; the defaults are:
  - source.o if -c was specified
  - a.out if executable
- These can be overridden with the -o option.

Options: -g

- Includes debugging info in the generated object code. This info can later be used in gdb.
- gcc allows to use -g with the optimization turned on (-O) in case there is a need to debug or trace the optimized code.

Options: -ggdb

- In addition to -g produces the most GDB-friendly output if enabled.

Options: -Wall

- Shows most of the warnings related to possibly incorrect code.
- -Wall is a combination of a large common set of the -W options together. These typically include:
  - unused variables
  - possibly uninitialized variables when in use for the first time
  - defaulting return types
  - missing braces and parentheses in certain context that make it ambiguous
  - etc.
- Always a recommended option to save your bacon from some “hidden” bugs.
- Try always using it and avoid having those warnings.

Options: -ansi and -pedantic

- For those who are picky about standard compliance.
- -ansi ensures the code compiled complies with the ANSI C standard; -pedantic makes it even more strict.
- These options can be quite annoying for those who don’t know C well since gcc will refuse to compile unkosher C code, which otherwise it has no problems with.

Options: -O, -O1, -O2, -O3, -O0, -Os

- Various levels of optimization of the code
- -O1 to -O3 are various degrees of optimization targeted for speed
- If -O is added, then the code size is considered
- -O0 means “no optimization”
- -Os targets generated code size (forces not to use optimizations resulting in bigger code).

Options: -I

- Tells gcc where to look for include files (.h).
- Can be any number of these.
- Usually needed when including headers from various-depth directories in non-standard places without necessity specifying these directories with the .c files themselves, e.g.:
  #include “myheader.h” vs.
  #include “../foo/bar/myheader.h”

For Your Assignments

- For your assignments, I'd strongly suggest to always include -Wall and -g.
- Optionally, you can try to use -ansi and –pedantic, which is a bonus thing towards your grade.
- Do not use any optimization options.
- You won't need probably the rest as well.

Shared Object Libraries

- "shared components" or "archive libraries", groups together multiple compiled object code files into a single file known as a library.

- There are two Linux C/C++ library types:

  - Static linked libraries (.a): Library of object code which is linked with, and becomes part of the application.

  - Dynamically linked shared object libraries (.so):

    There is only one form of this library but it can be used in two ways.

    1) Dynamically linked at run time but statically aware. The libraries must be available during compile/link phase. The shared objects are not included into the executable component but are tied to the execution.
    2) Dynamically loaded/unloaded and linked during execution (i.e. browser plug-in) using the dynamic linking loader system functions.

- Libraries are typically names with the prefix "lib". This is true for all the C standard libraries. When linking, the command line reference to the library will not contain the library prefix or suffix. Thus the following link command: gcc src-file.c -lm -lpthread The libraries referenced in this example for inclusion during linking are the math library and the thread library. They are found in/usr/lib/lib**m**.a and /usr/lib/lib**pthread**.a.
- See also **ar**, **ld**, **ldd** and **strings** commands.

Shared Object Libraries

- Creating a shared library:
  - gcc -Wall -fPIC -c *.c
  - gcc -shared -Wl,-soname,libctest.so.1 -o libctest.so.1.0 *.o
  - mv libctest.so.1.0 /opt/lib

The following creates the library libctest.so.1.0 and symbolic links to it.

- ln -sf /opt/lib/libctest.so.1.0 /opt/lib/libctest.so.1
- ln -sf /opt/lib/libctest.so.1.0 /opt/lib/libctest.so ..
- ln -sf /opt/lib/libctest.so.1.0 /opt/lib/libctest.so.1
- ln -sf /opt/lib/libctest.so.1 /opt/lib/libctest.s

Shared Object Libraries

- Creating a static library:
  - Compile: cc -Wall -c ctest1.c ctest2.c
  - Create library "libctest.a": ar -cvq libctest.a ctest1.o ctest2.o
  - List files in library: ar -t libctest.a
  - Linking with the library:
    - cc -o *executable-name* prog.c libctest.a
    - cc -o *executable-name* prog.c -L/path/to/library-directory –lctest
- Linking the library to the program:
  - gcc -Wall -I*/path/to/include-files* -L*/path/to/libraries* prog.c -lctest -o prog
  - gcc -Wall -L/opt/lib prog.c -lctest -o prog
- Using:
  - Set path: export LD_LIBRARY_PATH=/opt/lib:$LD_LIBRARY_PATH
  - Run: prog

Example

- For example, if you have the following source files in some project of yours:
  - ccountln.h
  - ccountln.c
  - fileops.h
  - fileops.c
  - process.h
  - process.c
  - parser.h
  - parser.c
- You could compile every C file and then link the objet files generated, or use a single command for the entire thing.
  - This becomes unfriendly when the number of files increases; hence, use Makefiles!
- NOTE: you don’t NEED to compile .h files explicitly.

Example (2)

- One by one:
  - gcc -g -Wall -ansi -pedantic -c ccountln.c
  - gcc -g -Wall -ansi -pedantic -c parser.c
  - gcc -g -Wall -ansi -pedantic -c fileops.c
  - gcc -g -Wall -ansi -pedantic -c process.c
- This will give you four object files that you need to link and produce an executable:
  - gcc ccountln.o parser.o fileops.o process.o -o ccountln

Example (3)

- You can do this as well:
  - gcc -g -Wall -ansi -pedantic ccountln.c parser.c fileops.c process.c -o ccountln
- Instead of typing this all on a command line, again: use a Makefile.

Example (4)

```
# Simple Makefile with use of gcc could look like this
CC=gcc
CFLAGS=-g -Wall -ansi -pedantic
OBJ:=ccountln.o parser.o process.o fileops.o
EXE=ccountln

all: $(EXE)

$(EXE): $(OBJ)
	$(CC) $(OBJ) -o $(EXE)

ccountln.o: ccountln.h ccountln.c
	$(CC) $(CFLAGS) -c ccountln.c
...
```

Motivation

- Small programs ⟶ all in single cpp file
- “Not so small” programs :
  - Many lines of code
  - Multiple components
  - More than one programmer
  - Require coordination of files to build

Motivation – continued

- Problems:
  - Long files are harder to manage
    (for both programmers and machines)
  - Every change requires compilation of all constituent files?
  - Many programmers can not modify the
    same file simultaneously
    - Large projects are not implemented in a single module/file

Motivation – continued

- Solution : Separation of Concerns
- Targets:
  - Proper and optimal division of components
  - Minimum compilation when something is changed
  - Easy maintenance of project structure, dependencies and creation

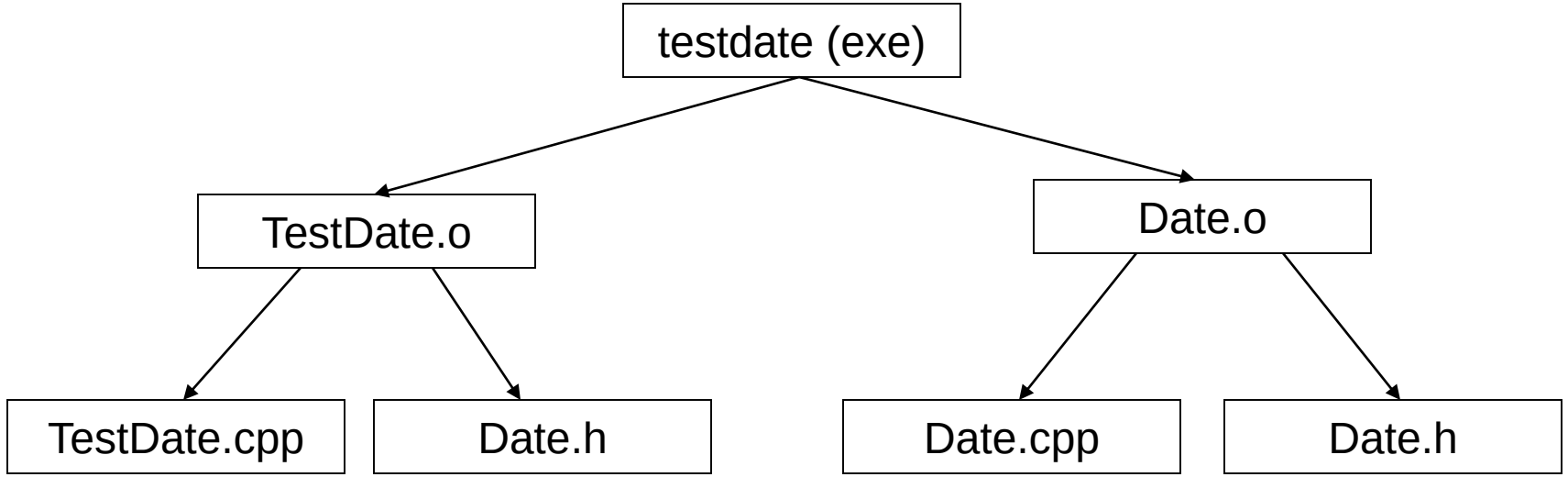
Project maintenance

- Performed in Linux by the Make utility
- A makefile is a file (script) containing:
  - Project structure (files, dependencies)
  - Instructions for creation of object code, executables, other tasks
- Note:A makefile script is not limited to C++ programs

Project structure

- Project structure and dependencies can be represented as a DAG (= Directed Acyclic Graph)
- Example :
  - Date class:
    - contains 3 files
  - Date.h, Date.cpp, TestDate.cpp
  - Date.h included in both .cpp files
  - Executable should be named testdate

Makefile Tree



Makefile Rules:

Targets, Dependencies, and Associated Commands

```
testdate: Date.o TestDate.o
	g++ -o testdate TestDate.o Date.o

TestDate.o: TestDate.cpp  Date.h
	g++ -c TestDate.cpp

Date.o: Date.cpp Date.h
	g++ -c Date.cpp
```

Rule syntax

Macros

- Macros are used to simplify and automate
  - Often used like constants in programs
- We can compress identical dependencies and use built-in macros to get another (shorter) equivalent makefile :

```
testdate: TestDate.o Date.o
  g++ –o $@ TestDate.o Date.o

TestDate.o Date.o: Date.h
  g++ –c $*.cpp
```

Macro $@ represents rule target:

- $@ = testdate

Macro $* represents prefixes shared by target and dependant files:

$* = TestDate, if target is TestDate.o

Date, if target is Date.o

Make Operation

- Project's dependency tree is constructed
- Target of first rule is to be created
- Go down the dependency tree to see if there is a target that should be recreated:
  - target file is older than one of its dependencies
  - recreate the target file according to the action specified, on our way up the tree. Consequently, more files may need to be recreated
- If something is changed, linking is usually necessary
  - Top target in dependency tree

make operation - continued

- make operation ensures minimum compilation, when the project structure is written properly

- Do not write something like:

```
testdate: TestDate.cpp Date.cpp Date.h
        g++ –o testdate TestDate.cpp Date.cpp
```

  which requires compilation of all files when something is changed
  - Example: If Date.cpp changes, why compile TestDate.cpp?

Passing parameters to makefile

- Parameters can be passed to a makefile by specifying them along with their values in the command line.
- For example:

  make PAR1=1 PAR2=soft1

  will call the makefile with 2 parameters: PAR1 is assigned the value “1” and PAR2 is assigned the value “soft1”. The same names should be used within the makefile to access these variables (using the usual “$(VAR_NAME)” syntax)

Passing parameters - continued

- Note that assigning a value to a variable within the makefile overrides any value passed from the command line.

- For example:

  command line : make PAR=1

  in the makefile:

  PAR = 2

- PAR value within the makefile will be 2, overriding the value sent from the command line

Conditional statements

- Simple conditional statements can be included in a makefile.
- Usual syntax is:

```
ifeq (value1, value2)
        body of if
else
        body of else
endif
```

Conditional statements - example

```
sum: main.o sum.o
    gcc –o sum main.o sum.o

main.o: main.c sum.h
    gcc –c main.c

#deciding which file to compile to create sum.o
ifeq ($(USE_SUM), 1)

sum.o: sum1.c sum.h
    gcc –c sum1.c –o $@

else
sum.o: sum2.c sum.h
    gcc –c sum2.c –o $@

endif
```

Replacement Makefile

- Makefile

```
SHELL = /bin/sh
MAKE = make
CC = g++
LIBS=
CFLAGS=-DSIGSETJMP -O

hello: main.o hello.o factorial.o functions.h
	${CC} ${CFLAGS} -o $@  main.o hello.o factorial.o ${LIBS}

clean:
	rm -f *.o
```

- Run as make; make clean

Standard “targets”

- People have come to expect certain targets in Makefiles. You should always browse first, but it's reasonable to expect that the targets all (or just make), install, and clean will be found.

- **make all** - should compile everything so that you can do local testing before installing things.

- **make install** - should install things in the right places. But watch out that things are installed in the right place for your system.

- **make clean** - should clean things up. Get rid of the executables, any temporary files, object files, etc.

Reference

- GNU Project's Tutorial for makefiles

  http://www.gnu.org/manual/make-3.80/make.html

General tarball/make example

tar -vxzf <gzipped-tar-file>

cd <dist-dir>

./configure

make

make install

make clean

Debugging – gdb (gnu debugger)

- Use the -g command line parameter to cc, gcc, or CC: cc -g -c foo.c
- gdb [<program> | [corefile>|<pid>]]
- gcc –g –c –o hello hello.c
- gdb ./hello

- See also strace (Linux), truss (Solaris)

gdb commands

- General Commands:

file [<file>]	selects *<file>* as the program to debug
run [<args>]	runs selected program with arguments <*args>*
attach <pid>	attach gdb to a running process <*pid>*
kill	kills the process being debugged
quit	quits the gdb program
help [<topic>]	accesses the internal help documentation

- Stepping and Continuing:

c[ontinue]	continue execution (after a stop)
s[tep]	step one line, entering called functions
n[ext]	step one line, without entering functions
finish	finish the function and print the return value

- Useful breakpoint commands:

b[reak] [<where>]	sets breakpoints. *<where>*
[r]watch <expr>	sets a watchpoint, which will break when *<expr>* is written to [or read]
info break[points]	prints out a listing of all breakpoints
clear [<where>]	clears a breakpoint at *<where>*
d[elete] [<nums>]	deletes breakpoints by number

- Commands for looking around:

list [<where>]	prints out source code at *<where>*
search <regexp>	searches source code for *<regexp>*
backtrace [<n>]	prints a backtrace *<n>* levels deep
info [<what>]	prints out info on *<what>* (like local variables or function args)
p[rint] [<expr>]	prints out the evaluation of *<expr>*

- Commands for altering data and control path:

set <name> <expr>	sets variables or arguments
return [<expr>]	returns *<expr>* from current function
jump <where>	jumps execution to *<where>*

Conclusion

- We have started on C programming in Linux
- We have revisited the process of compiling C.
- We will start on Linux Programming on File I/O next week
- Reading Assignment: Chapter 3&4 in your textbook