Makefile and profiling/debugging tool

LAB 1 MODULE, UCI, SPRING 2020

SHU-MEI TSENG

What is Makefile?

```
/* driver.cc */
#include “sort.hh”
int main()
{
    seqSort();
    mySort();

    return 0;
}
```

```
/* sort.hh */
#if !defined (INC_SORT_HH)
#define INC_SORT_HH

void seqSort (int N, int* A);
void mySort (int N, int* A);
int* NewCopy (int N, const int* A)

#endif
```

```
/* sort.cc */
#include “sort.hh”

void seqSort (int N, int* A)
{

}
int* NewCopy (int N, int* A)
{

}
```

```
/* mysort.cc */
#include “sort.hh”

void mySort(int N, int* A)
{

}
```

How to compile?

```
$g++ -c driver.cc sort.cc mysort.cc
$g++ -o sorting driver.o sort.o mysort.o
```

Tedious and not easy to manage and modify

Makefile

- A way of automation on software building
- Define a set of rules and dependencies to build the targeted program
- If the target is older than one of the dependencies, re-build the target by specified command

Dependency Rule

target: dependencies
<Tab> commands to make target

target: name of file or action

dependencies: files that are used to create the target

<Tab> cannot be replaced by spaces

command: interpreted by a shell to be executed

To build the executable
- $make <target>

Phony target:
- No dependencies, only a list of commands
- $make clean

```
# Makefile for previous example

sorting: driver.o sort.o mysort.o
     g++ -o sorting driver.o sort.o mysort.o
```

```
$make sorting
```

```
# Makefile for previous example

sorting: driver.o sort.o mysort.o
     g++ -o sorting driver.o sort.o mysort.o
clean:
     rm -f core *.o *~ sorting
```

Macros

NAME = text string

Similar to variables in common programming languages

Purpose: avoid repeating texts and make it easy to modify Makefile

Example

- CC=g++

Macros are referred to by putting them in parentheses and precede them with $ sign

- $(CC)

Special Macros

- Macro @ evaluates to the current target => Usage: $@
- $< : first prerequisite
- $^ : all prerequisites

```
# simple example

CC = g++
objs = hello.o main.o
print_hello: $(objs)
	$(CC) -o $@ $(objs)
	#equivalent to $(CC) -o print_hello $(objs)
```

Macros (Cont'd)

Pattern rule

- Looks like normal rule except that the target contains "%"
- Can be used for matching file names
- % can match **any** non-empty strings; other characters match **only themselves**
  - Specify how to make any *.o file with *.c file (E.g., hello.o with hello.c, foo.o with foo.c)

```
# simple example

%.o: %.cc
  $(CC) $(CFLAGS) $(COPTFLAGS) -o $@ -c $<
```

specify how to make any *.o file with *.c file (e.g., hello.o with hello.c, foo.o with foo.c)

Conventional Macros

- Check the list by using `$make -p`

Macro	Description
CC	Program to compile C programs
CXX	Program to compile C++ programs
CFLAGS CXXFLAHS	Extra flags to give to the C/C++ compiler
LDFLAGS	Extra flags to the C compiler if they need to invoke the linker

GDB – GNU Debugger

DEBUGGING TOOL FOR C

Introduction

A debugging tool that allows you to view your program's state as it is executing

First, build a debug version of your program

- Compile a program with debugging symbol: -g, which enables built-in debugging support
- Add compiler flags –g (add debug symbols) and –O0 (disable optimization)
- `$g++ -g –O0 main.cc hello.cc –o print_hello`

Start debugging session by the following approaches

Program has no input arguments

```
$gdb <exec>
$(gdb) r
```

Program has input arguments

```
$gdb --args <exec> arg1 arg2 … argN
$(gdb) run
```

```
$gdb <exec>
$(gdb) run arg1 arg2 … argN
```

If a core dump file is generated 

```
$gdb <exec> core
```

Common useful commands

Command	Desciption
run or r	Execute the program from start to end
break or b <line number> <function name>	Set breakpoint on a particular line of code
next or n	Execute the next line of code, but don't dive into functions
step	go to next instruction, diving into the function
list or l <line number> or <function name>	display the code (Note that it usually shows a few lines back to the point you specify)
print or p <variable name>	Display the stored value
quit or q	Exit gdb
continue	Continue normal execution
backtrace or bt	Generate a stack trace of function calls that lead to the error
frame or f <frame number>	Select a stack frame or select currently selected stack frame

Memory Debugging Tool

Memory checker

Some memory bugs do not crash a program, GDB cannot tell you where the bug is

Valgrind

- A tools suite that provides both debugging and profiling tools. One of the most famous tools is Memcheck, which can detect memory-related errors

Memory leak: incorrect management of memory allocation, memory which is no longer needed is not released

- Issue: reduce the performance of the memory since it reduces the amount of available memory

```
$valgrind --leak-check=yes <exec> arg1 arg2 …
```

Reference: https://valgrind.org/docs/manual/manual.html

Profiling using perf

Perf

A performance analyzing tool in Linux.

It uses a sampling approach to gather data about important hardware and kernel events, such as cache misses, branch misses, page faults, and context switches.

Using stat subcommand `$perf stat <exec>`

```
 Performance counter stats for './hello':

          2.602670      task-clock (msec)         #    0.002 CPUs utilized
                10      context-switches          #    0.004 M/sec
                 5      cpu-migrations            #    0.002 M/sec
               105      page-faults               #    0.040 M/sec
         7,743,667      cycles                    #    2.975 GHz
         4,803,316      instructions              #    0.62  insn per cycle
           842,276      branches                  #  323.620 M/sec
            30,380      branch-misses             #    3.61% of all branches

       1.397611004 seconds time elapsed
```