Recitation 1: Basic Tools

This recitation introduces basic tools for building, debugging and profiling software.

1 Software engineering practices

You probably learned this from previous courses; nonetheless, we will remind you again.

- Maintainability: comment your code, use meaningful variable names, insert white-spaces.
- Code organization: modularize large routines, reuse code, do not copy and paste.
- Version control: write commit messages, commit often, do not break the master repository.

2 Pair programming

Pair programming is a technique in which two programmers work on the same machine. “One of the programmers, the driver, has control of the keyboard/mouse and actively implements the program. The other programmer, the observer, continuously observes the work of the driver to identify tactical (syntactic, spelling, etc.) defects, and also thinks strategically about the direction of the work.” The programmers work equally to develop a software as they periodically switch roles.

3 Getting started

We recommend that you work on course machines. You can access it using ssh

```bash
% ssh username@cloudN.csail.mit.edu
```

To get a local copy of the repository for your work, you need to use git to clone it.

```bash
% git clone /afs/csail.mit.edu/proj/courses/6.172/
student-repos/recitation1/username recitation1
```

4 Building your code

You can use a compiler to compile your code to a binary. We use Intel compiler in this course.

```bash
% icc -Wall main.c -o rollingsum
```

This will compile `main.c` to `rollingsum`. `-Wall` tells the compiler that you want to see all warnings. You can run it by typing

```bash
% ./rollingsum
```

The current version of `main.c` just generate an array and print it out. Next, we will modify it to measure the running time of the program. Use your favorite editor to edit `main.c`. If you don’t have one, you can use `emacs`.
Add the timing code in function main.

```
clockmark_t time1 = ktiming_getmark();
gen_array(array, ARRAY_SIZE);
print_array(array, ARRAY_SIZE);
clockmark_t time2 = ktiming_getmark();

float elapsedf = ktiming_diff_sec(&time1, &time2);
printf("Elapsed execution time: %f sec\n", elapsedf);
```

Since timing functions are in ktiming.c, we have to include ktiming.h at the top of main.c.

```
#include "ktiming.h"
```

Then, compile the code again

```
% icc -Wall main.c -o rollingsum
```

You will get “undefined reference” errors because the compiler cannot find the code for some functions. You need to add ktiming.c in the argument to the compiler

```
% icc -Wall main.c ktiming.c -o rollingsum
```

You still get an error. This time icc cannot find clock_gettime, which is called in ktiming.c. This function is in realtime library, so you also need to tell the compiler to include that.

```
% icc -Wall main.c ktiming.c -lrt -o rollingsum
```

The problem with this command is that if you modify some files, you need to compile everything again. The better way is to compile each file separately, and link them together later.

```
% icc -Wall -c main.c
icc -Wall -c ktiming.c
icc main.o ktiming.o -lrt -o rollingsum
```

-c tells the compiler to generate an object file (main.o). Using the latter commands, you don’t need to compile ktiming.c again when you modify main.c.

### 5 Using Make

Typing these commands every time is tedious. Plus, you might forget to recompile some files. make can check the modification time of the dependency files to determine whether the target needs to be recompiled. To use make, you need to specify build rules in Makefile. Each rule is in the following format.

```
target: dependencies
    command
```
Your Makefile should look like this.

```
CC := icc
CFLAGS := -Wall
LDFLAGS := -lrt

all: rollingsum

main.o: main.c
  $(CC) $(CFLAGS) -c main.c

ktiming.o: ktiming.c
  $(CC) $(CFLAGS) -c ktiming.c

rollingsum: main.o ktiming.o
  $(CC) -o rollingsum main.o ktiming.o $(LDFLAGS)

clean:
  rm -f rollingsum *.o

Note: the command lines must be indented with TAB, not spaces. Now, you can compile the program by typing

% make
```

### 6 Using `printf`

Next, modify `main` to call `sum_array`

```c
sum_array(array, ARRAY_SIZE, 0, ARRAY_SIZE);
```

*Don’t forget to import `sum.h` and modify Makefile to include the new dependency.*

When you make, you will get an error about a format string conversion. The problem is `printf` requires a type of each variable. In this case, `sum` is an int, so you need to use `%d` instead of `%f`. The compiler detects this error and warns you. Nonetheless, it still generate a binary, which prints a wrong number. You can read more about `printf` from [http://www.cplusplus.com/reference/clibrary/cstdio/printf/](http://www.cplusplus.com/reference/clibrary/cstdio/printf/).

Fix this error, and commit your changes to your local repository.

### 7 Using `gdb`

To prepare a buggy code for debugging, change `stop` argument for `sum_array` in `main.c` to `ARRAY_SIZE * 9999`. After that, build and run the program. You will get a segmentation fault.

You should build a “debug” version of the code to debug. Modify Makefile to add compiler flags `-g` (add debug symbols) and `-O0` (do not optimize)

```
% CFLAGS := -Wall -g -O0
```
Then, you can start a debugging session in gdb.

```
$ gdb ./rollingsum
...
(gdb) r
...
Program received signal SIGSEGV, Segmentation fault.
0x000000000040095c in sum_array (array=0x7fffffffe1f0, size=10, start=0, stop=99990) at sum.c:9
9  sum += array[i];
(gdb) bt
#0 0x000000000040095c in sum_array (array=0x7fffffffe1f0, size=10, start=0, stop=99990) at sum.c:9
#1 0x00000000004007be in main () at main.c:33
(gdb) p i
$1 = 900
(gdb) p array
$2 = (int *) 0x7fffffffe1f0
```

You will learn more about gdb in project 0, and even more in another recitation.

### 8 Using assertions

The assert package is a useful tool for catching bugs before your program goes off into the weeds. Include assert.h, and add the following lines in sum_array.

```c
assert(start >= 0);
assert(start < stop);
assert(stop <= size);
```

Build and run the program. You will see an output about an assertion error. Since checking assertions is expensive, you do not want to do it in optimized build. You can add a compiler flag -DNDEBUG to define the NDEBUG macro, which disables assertion checks.

```
% CFLAGS := -Wall -O3 -DNDEBUG
```

Build and run the program again. You will get a segmentation fault, which means the assertions are not checked.

### 9 Using gprof

gprof is a tool for static instrumentations. You can use gprof to find out where your program spends the most time on. It inserts some code into the program to profile it. Thus, you have to add a compiler and a linker flag -pg to build a "gprof" version of the code.

```
% CFLAGS := -Wall -O3 -DNDEBUG -pg
LDFLAGS := -lrt -pg
```
Before you begin, you should modify rollingsum to do more computation. Increase ARRAY_SIZE to 10000, and write a for loop to run gen_array, print_array and sum_array 10000 times.

./rollingsum > /dev/null
mv gmon.out gmon.sum
./rollingsum > /dev/null
gprof -s ./rollingsum gmon.out gmon.sum
gprof ./rollingsum gmon.sum

10 Exercise

Modify Makefile such that

- ‘make’ builds an “optimized” version.
- ‘make DEBUG=1’ builds a “debug” version.
- ‘make PROFILE=1’ builds a “gprof” version, run the program twice, and write a human-readable profile in gprof-profile.

**Hint:** You can check for DEBUG=1 by

```bash
ifeq ($(DEBUG),1)
else
endif
```

Commit and push your changes back to the remote branch. Then, show your work to the TA to checkoff this recitation.

Before you leave, you should grant a permission to access the repository to your peer. You can do this by

```bash
% fsr sa /afs/csail.mit.edu/proj/courses/6.172/
student-repos/recitation1/yourusername peerusername rlidwa
```