Assignment 5. Low-level refactoring and performance

Introduction

This assignment is designed to give you some skills with low-level programming, which is used in later courses like the operating system class, as well as in real-world applications like the Internet of things (IoT). You'll start with a working program; you'll add a few features, and tune and refactor the program to make it better.

Useful pointers

- Ian Cooke, C for C++ Programmers (2004). This describes C89, also known as C90 and ANSI C. C17, the current version of C, supports `//` comments, declarations after statements, and (if you include `<stdbool.h>`) `bool`.
- Brian Kernighan and Dennis Ritchie, *The C Programming Language*, 2nd edition (1988), Prentice Hall, ISBN 0-13-110362-8. This classic book on C is still the language's best description, albeit limited to C89.
- Tom Plum, Introduction to C11 (2013). C11 is very close to C17, the current standard.
- Parlante, Zelenski, et. al, Unix Programming Tools (2001), section 3 — `gdb`.
- Valgrind Quick Start Guide (2020)
- Valgrind User Manual (2020)
- Richard Stallman, Roland Pesch, Stan Shebs, *et al.*, Debugging with GDB (2020)

Homework: Tuning and refactoring a C program

Keep a log in the file `notes.txt` of what you do in the homework so that you can reproduce the results later. This should not merely be a transcript of what you typed: it should be more like a true lab notebook, in which you briefly note down what you did and what happened.

You're trying to generate large quantities of random numbers for use in a machine-learning experiment. You have a program `randall` that can generate random byte streams, but it has problems. You want it to be (a) faster and (b) better-organized.

You can find a copy of the `randall` source code in the tarball `randall-assignment.txz`. It contains:

- A file `randall.c` that is a single main program, which you would like to modularize.
- A `Makefile` that can build the program `randall` and can build the tarball.

First, read and understand the code in `randall.c` and in `Makefile`.

Next, modify the `Makefile` so that the command '`make check`' tests your program. You can supply just a simple test, e.g., that the output is the correct length. You're doing this step first because you believe in test-driven development (TDD).

Next, split the `randall` implementation by copying its source code into the following modules, which you will need to likely need to modify to get everything to work:

- `options.c` with header `options.h`, which does command-line options processing. Initially there isn't very much of this.
- `output.c` with header `output.h`, which does the output.

- `rand64-hw.c` with header `rand64-hw.h`, which does the hardware-based random-number generation.
- `rand64-sw.c` with header `rand64-sw.h`, which does the software-based random-number generation.
- `randall.c` should contain the main program that glues together everything else. It should include the `.h` files mentioned above, and use their APIs to do its work.

You may add other modules if you like. Each module should include the minimal number of include files; for example, since `rand64-hw.c` doesn't need to do I/O, it shouldn't include `<stdio.h>`. Also, each module should keep as many symbols private as it can.

Next, modify the `Makefile` to compile and link your better-organized program.

Next, add some options to your program to help you try to improve its performance. Redo the program so that it has an option '`-i` *`input`*', where *`input`* is one of the following:

- `rdrand` – the hardware random-number generation supported by x86-64 processors if available; `randall` should issue an error message and fail if it is not available. This option is the default.
- `mrand48_r` – the `mrand48_r` function of the GNU C library.
- */`F`* (i.e., any argument beginning with `/`) – use the file */`F`* as a source of random data, instead of using `/dev/random`.

Also, redo the program so that it has an option `-o` *`output`*, where *`output`* is one of the following:

- `stdio` – Use `stdio` output, as before. This is the default.
- *`N`* (a positive decimal integer) – Output *`N`* bytes at a time, using the `write` system call. If the `write` call reports a successful write of fewer than *`N`* bytes, do not consider this to be an error; just add the successfully-written number of bytes to your count of bytes written. The last output block might be smaller than usual, if needed to get the total size to be correct. You can use the `malloc` function to allocate your output buffer.

You can use `getopt` to implement your option processing.

Add some '`make check`' tests to check your additions to `randall`.

When debugging, you may find the `valgrind` program useful. Also, the AddressSanitizer (asan) and the Undefined Behavior Sanitizer (ubsan) may be useful; these can be enabled with the GCC options `-fsanitize=address` and `-fsanitize=undefined`, respectively.

If the program encounters an error of any kind (including option, output and memory allocation failures), it should report the error to `stderr` and exit with status 1; otherwise, the program should succeed and exit with status 0. The program need not report `stderr` output errors.

Finally, time your implementation as follows ...

```
# This is a sanity check to test whether you're in the right ballpark.
time dd if=/dev/urandom ibs=8192 obs=8192 count=16384 >/dev/null

time ./randall 133562368 >/dev/null
time ./randall 133562368 | cat >/dev/null
time ./randall 133562368 >rand.data
```

... except that you may need different numbers if your implementation is faster or slower. Also you should try various combinations of the above options to see which gives you random data the fastest. One option that you should try is '`-i /dev/urandom`'.

Record your results (including your slow results) in `notes.txt`.

Use a private local Git repository to keep track of your work when you're modifying code, data, or `notes.txt`.

Submit

Submit two files:

1. The file `randall-submission.tgz`, which you can build by running the command `make submission`. Test your tarball before submitting it, by extracting from it into a fresh directory and by running '`make check`' there.
2. A gzipped copy of your private local Git repository, created by the command "`tar -czf randall-git.tgz .git`" in the parent of your Git repository.

All source files should be ASCII text files, with no carriage returns, and with no more than 100 columns per line. The shell command

```
expand Makefile notes.txt *.c *.h |
  awk '/\r/ || 100 < length'
```

should output nothing.

$Id: assign5.html,v 1.46 2021/02/17 02:49:25 eggert Exp $