Debugging and profiling

Matteo Cicuttin

October 19, 2021

Grab the code in the repository
https://gitlab.onelab.info/mcicuttin/snippets/info0939

Debugging & Profiling

The two activities are related to two questions:

- Debugging: Why my code does not work *as expected*?
- Profiling: Does my code perform well *on a given architecture*?

Organization of this class:

- Tips to avoid the need to debug
- Review of some concepts
- Debugging (hands-on)
- Profiling (hands-on)

About this class

I'll give you mostly a high-level overview of concepts & tools.
Details are too many. Only way to master them:

- study architectures
- countless sleepless debugging nights
- getting used to reason in non-conventional ways

Let's try to keep this class as interactive as possible!

Introduction

Avoiding debug

Rule #1 of debug:

Minimize the chance to end up debugging your code.

Said otherwise: Programmers **do fail** and **C is dangerous**: do your best to circumvent the most common failure modes or minimize their impact.

- Adopt an appropriate coding style
- Identify bad code patterns and habits and avoid them
- Use correctly the compiler
- Use ‘assert()‘
- Strive for a “correct by design” approach, not “correct because it passes some tests”
- ...

More generally, care about the quality of your code: cheap things end up being very expensive.

Coding style

Code: not something that somehow works, but the end product of a careful design process.

- Break down your problem in smaller subproblems, write corresponding functions. Functions should be small and fit on your screen.
- Use relevant and appropriate names for functions and variables ($\implies$ self-documenting code).
- Separate *state* from algorithms that modify your state.

Good coding style is good for you and for the others:

- For you: easier to track down problems in small, clearly separated modules.
- For the others: you won't earn much respect if you waste the time of your colleagues by writing unreadable code.

Bad patterns and habits

- Declaring variables without initializing them.
- Not checking return values of `open()`, `malloc()` and related calls.
- Long functions, functions with too many parameters
- Premature optimization
- Inconsistent naming and conventions
- Too many comments ($\Longrightarrow$ code not clear & comment rot)
- Add your own ...

Use correctly the compiler

The compiler has many facilities that can help you:

- Warnings: compiler warnings should be treated as errors. They are "just" warnings not because they are not important, but because there could be some legitimate use case of the problematic code.
- Increase warning level with `-Wall`, `-Wextra` (GCC), `-Weverything` (Clang) and `-pedantic`.
- In C++, exploit the typesystem to guarantee at compile-time that your program satisfies the properties you want (Take a look to Boost.Units).
- Use static analysis tools, as `scan-build` from the LLVM suite.

Use `assert()`

The macro `assert()` is used to assert that some condition must be true at a specific point.

- It is used in the debug builds.
- In release builds is disabled via `-DNDEBUG`.
- **DO NOT** use it to validate *user* input (if you do, you'll get zero in your project): `assert()` has to do with the *logic* of your program!

With `assert()` you should check

- Preconditions
- Invariants
- Postconditions

Example usage of assert() on an integer division program

Specification: given two integers $x \geq 0$ and $y > 0$, provide a program that returns their integer division $quo = x/y$ and $rem = x\%y$.

- Idea: subtract x from y while the remainder is greater than y.
- Let's see some code.

Example usage of assert() on an integer division program

Specification: given two integers $x \geq 0$ and $y > 0$, provide a program that returns their integer division $quo = x/y$ and $rem = x\%y$.

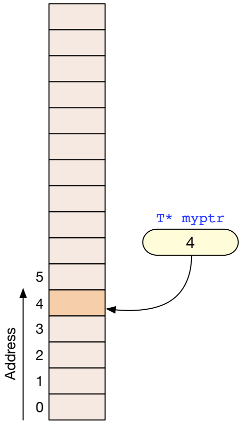
- Idea: subtract x from y while the remainder is greater than y.
- Let's see some code.

The precondition, invariant, postcondition and bound function I chose actually allow to formally prove the correctness and the termination of the program under the Hoare logic.

- `assert()` helps you to reason about your program and to document it.
- If you are interested in formal program verification, take a look at the classical book *Verification of Sequential and Concurrent Programs* by Apt, de Boer & Olderog.

Some basics

Memory & pointers



Memory is a box with many sequentially numbered places. The number of a place is called **address**.

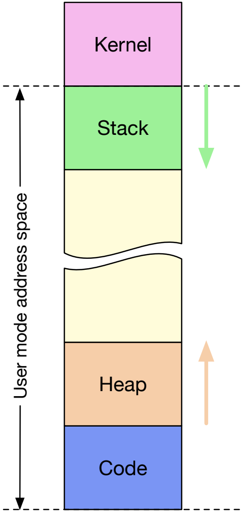
Pointers are unsigned integer variables that store addresses.

- Pointers **ARE NOT** vectors
- Pointers **ARE NOT** arrays
- Pointers **ARE NOT** structs

`T *myptr` tells you that there is a `T` at the address stored in `myptr`.

Beware of pointer arithmetic: `myptr+1` means that the address is increased by `sizeof(T)`.

A very approximate process' memory model

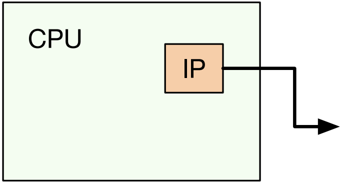


In timesharing operating systems all processes see a linear address space with 4 main segments:

- Code: fixed-size, contains your program.
- Heap: grows dynamically based on `malloc()` and `free()`.
- Stack: FIFO data structure, grows downwards on function calls and variable declarations.
- Kernel: area reserved to the operating system

Code segment

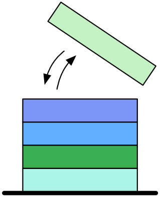
Your CPU fetches and executes instructions from this part of memory. The **Instruction Pointer** or **Program Counter** keeps track of the currently executing instruction.




```
push    rbp
mov     rbp, rsp
mov     dword ptr [rbp - 4], edi
mov     dword ptr [rbp - 8], esi
mov     eax, dword ptr [rbp - 4]
add     eax, dword ptr [rbp - 8]
pop     rbp
ret
```

Stack segment

The stack is a FIFO structure. The details change slightly between architectures, but in general it serves three main purposes



- Passing function parameters
- Saving the IP on function calls
- Storing local variables

Important concept: stack frame

Heap segment

Heap essentially contains the memory you get from `malloc()`.

- If there's enough memory on the heap, `malloc()` gives you a pointer to a slice of the size you asked.
- If there's not enough memory, `malloc()` asks the OS to increase the *program break*. If the OS has no memory, `malloc()` fails.
- `free()` marks some memory on the heap as free.
- After a while, the memory could get fragmented.

Debugging

Debug tools

We will take a look to `gdb` (a debugger)

- Backtrace (`bt`)
- Stepping (`step`, `next`, `continue`)
- Printing (`p`)
- Breakpoints (`break`)
- Moving between frames (`frame`)

and to AddressSanitizer (code instrumentation)

- General understanding of its output

We will focus on memory-related problems.

Profiling

Profiling

Profiling is about

- measuring your program ($\Longrightarrow$ tools & instrumentation)
- determining if your program running at maximum resource utilization ($\Longrightarrow$ architecture knowledge)

Both points are needed for optimization

If you do not measure or you don't know your architecture, you **can not** (decide to) optimize.

When you say you've optimized your program, be ready to give convincing arguments.

Profiling tools

- Quick look at `gprof`: where your program spends time
- Quick look at `cachegrind`: does your code use the cache correctly?
- Code instrumentation & the importance of knowing the architecture and choosing the right metric