# Code Review - Week 1: Kernel Sanders

**Reviewer: Spotlight (Jaime Sanchez, Karan Singhal, Stefan Dimitrov)**

**Reviewee: Kernel Sanders (Emmett Moore, Caleb Malchik, Dan Defossez)**

## Overview

We weren't able to compile the code as we do not have Plan9 at hand. All of the code in the repo is meant to run under Plan9 userspace, and the project itself is an attempt to rewrite parts of the OS. Thus we have no means to test and profile the code. Nevertheless, we were able to provide some general coding suggestions that would hopefully prevent maintainability and code quality problems down the road.

While unavoidable when tinkering and figuring things out, not leaving a README and commenting out code should be avoided at all costs. It not only makes it harder to understand the code but it makes it harder to manage the project as it grows in complexity. We suggest that you follow a clear and documented strategy when working on your scheduler. This will help you too when trying to understand the complexity of how everything fits together (and that's no small task for this project) Documenting the steps you took will likely help you stay on track.

Bellow we follow the Fog Creek Code Review Checklist.

## General

- **Does the code work? Does it perform its intended function, the logic is correct etc.** Provided that we didn't have the means to compile and run the code since it is meant to execute in Plan9 userspace, we weren't able to verify if the intended functionality was implemented. However, it appears that the simple hello world application Kernel Sanders used to familiarize themselves with the compiler should indeed be functional.
- **Is all the code easily understood?** The code in hello.c is straightforward as it is a simple hello world application:

```
#include <u.h>
#include <libc.h>

void main(void) {
    print("hello world!\n");
    exits(0);
}
```

The .h includes are unusual, clearly suggesting that this will not compile on a regular UN*X/windows box. We can safely assume that `u.h` contains the definitions of Plan9 system calls.

```
#include <u.h>
#include <libc.h>
```

A quick look at the Plan9 manpages tells us that the function `int print(char *format, ...)` is a system call that prints the string 'Hello World' to standard output:

```
void main(void) {
    print("hello world!\n");
    exits(0);
}
```

Pretty straightforward and nothing that's impossible to figure out even though comments are absent. Likewise, `void exits(char *msg)` terminates the process.

- **Does it conform to your agreed coding conventions? These will usually cover location of braces, variable and function names, line length, indentations, formatting, and comments.** Please note that the docs advise passing a useful message to `exits` which provides an 'explanation of the reason for exiting'. A null pointer or an empty string is also acceptable. Passing a `0` is perhaps a carryover from `exit(0);` which is fairly idiomatic in UN*X but perhaps not in Plan9. Likely `0` is equivalent to `NULL` and the program will terminate correctly, but from the standpoint of coding conventions and custom it should be avoided if we are to be nit-picky.

- **Is there any commented out code?** Extensive use of commenting out code in `qio.c` with no clear justification provided. Was this code commented out as part of an ongoing investigation into the working of the OS? Is it simply no longer needed? This is generally a really bad practice, as it adds to the confusion of reading someone else's code. It also indicates that a version control system is not being used to its full potential.

## Security

Since you are rewriting a part of the operating system it is important to decide how to avoiding adding regressions in terms of both the stability and the security of plan9. Watch out for buffer overflows, circumventing privileges and the like. You are working on the scheduler which will govern potentially malicious processes, so it would be important to consider if and how to guard against them (think about how it's ok for a process to spawn infinitely many children in Linux -- perhaps you want your scheduler to guard against such cases in order to, for example, improve the response time of a server being DDoS'ed)

## Documentation

- **Do comments exist and describe the intent of the code?** There seem to be two different compile scripts in both `9\hello` and `9\three-inch`, named `compile` & `run` and `compile` & `mkfile` respectively. It looks like the former ones seem to be older versions of

your compile scripts. It would've been nice to include a `README` file detailing how to build the code -- this is always a good practice. In addition any old code left from earlier stages should be removed from your repo. The name for `stuff.h` is not very approachable. It looks like the commented out parts are a rewrite of parts from `qio.c`. Why is that necessary? It is not clear what the exact role of `stuff.h` is. Even with a better name, a README here detailing your methodology and goals would have been very helpful.

- **Are all functions commented?** Functions is `qio.c` are well documented, even though more meaningful in the grand context of things. Someone with little knowledge of the big picture in Plan9 would not be able to start contributing right away but this is OK as long as the README provides the necessary detail.
- **Are data structures and units of measurement explained?** The data structures are well documented.
- **Is there any incomplete code? If so, should it be removed or flagged with a suitable marker like 'TODO'?** The repo contains a TODO list.

## Testing

- **Is the code testable? i.e. don't add too many or hide dependencies, unable to initialize objects, test frameworks can use methods etc.** It is hard to tell how you intend to test your code at this point, since you haven't gotten it to compile. Given the project it would be difficult to find suitable test frameworks. Moreover, you're attempting to isolate a part of the OS to cut on compilation time, but can you also reliably test in isolation? You'd have to think about injecting your scheduler into a build of the OS to run in a VM or on hardware.
- **Do unit tests actually test that the code is performing the intended functionality?** Unit tests here would be useful but as mentioned you should consider running your component as part of a testing build.
- **Are arrays checked for 'out-of-bound' errors?** This would be extremely important as you develop your algorithm: such errors will easily compromise the stability and security of the OS.