Computer Science 161: Operating Systems
How to write a Design Document

CS161 Course Staff
cs161@eecs.harvard.edu

January 22, 2013

1 Introduction

Assignments 2, 3, and 4 require that you write and submit a design document. Your design document will be reviewed in two phases. First, we will conduct in-class peer reviews. Your group will pair up with another group and you will spend half the class reviewing their document and half the class with them reviewing your document. Second, you will submit your (possibly revised) design document to your TF, who will review it, give you feedback, and assign a grade. Your grade will reflect both the quality of your design as well as your participation in the peer review. Peer review participation has many aspects. You will learn a great deal from doing a careful review of another design. Ideally you will be exposed to alternative thought processes and conclusions. You should ask good questions; if you do a good review, then you will help the other group improve their design document. Both teams will document the interaction from their vantage point and your full engagement in both sides of the review is expected. You will be asked to explain how your design evolved after the peer review. You are welcome to adopt ideas and mechanisms that you learn about through the peer review, but you must cite them. For example, if you adopt your peer team's approach to managing your file table, that's fine, but you have to tell us that that is where you got the idea.

The CS161 design documents are, in the opinions of the course staff, the single most important piece of work you will do in this class. (Corollary: They are the single most important piece of work you will do in your life.) Yes, other CS classes have required you to submit a "design" "document". But let's be honest, in such classes you tabled the design document and

hacked away at the assignment until it worked, only to go back and write the design document at the end.

This method, however, will not work in CS161. The motto here is "measure twice, cut once". We give you a week to complete design documents for a reason; we want you to spend a significant amount of time thinking about how the pieces will fit together (and what new pieces you might need), before you run off to hack them together. We give you two weeks after the design document due date to code, so there should be no need to rush this process.

To further motivate you, we remind you that the design is worth 30% of your final grade for the assignment. You do not want to spend four nights perfecting your system only to lose 10 points on the assignment because your design document wasn't of the right level of sophistication / completeness.

That brings us to an important question. *What is a design doc and how do I write one?* A good design document is as valuable as the code you will write. A design document should be a document that will allow a good programmer to write working code, even if that programmer doesn't know the internals of `os/161` too well. In other words, a design doc should reflect all the research and brainstorming you did before attempting the coding task. Do not shy away from details!

The rest of this document presents guidelines for writing a good design doc. Don't feel like you have to stick to them exactly, but if you do not know where to start, this should provide some guidance.

2 Introduction

In the introduction you should decompose the assignment into the major parts that you need to complete and what your approach will be to each part. Clearly, you will not go into detail about the different pieces, but after reading the introduction, the reader should be able to predict the outline of the rest of your document and some idea of what s/he will find in each section. The introduction is also the right place to identify any open problems you have not solved and about which you would welcome feedback. This doesn't mean that we'll tell you how to do the entire assignment, but if you know that you have some confusion that you've not been able to resolve, tell us about it now! We expect that you will not have foreseen every problem you will encounter, but the more you identify early, the better prepared you will be when you start coding.

3 Overview

In the introduction you've provided an outline of the rest of your design. The overview is where you dive into the design in more detail. For example, tell us how the different parts of the design fit together, describe the APIs between the different pieces. Introduce the data structures that you'll be using. If there are overarching algorithmic tasks, provide pseudo code for the algorithms you'll use. If you are going to be using global variables, this is the place to introduce them and tell us how they will be synchronized. If there is complex synchronization between different parts of the assignment, that is also a good topic for the overview. Finally, provide a brief explanation of *why* your design will work.

Here is how we think of the overview: In a large software project, there is usually a chief architect responsible for the overall vision of the project. The architect hands out individual tasks that require detailed design and implementation and decides what the pieces are, how they interact, what the important interfaces, data structures, and algorithms are. The overview is what the architect produces.

4 Topics

Finally, break up your design into appropriate topics and discuss the details of each topic as explained below. For example, in assignment 2, a good topic breakdown is as follows:

- Identifying processes
- File descriptors
- `fork`
- `execv`
- `waitpid/exit`
- Other system calls
- Scheduling
- Synchronization issues

In this more detailed discussion, you might come back to topics of interaction that appeared in the overview. However, when you do it here, you're

doing it in the context of your implementation and diving into the details. Using assignment 2 as an example once again, in the section on `fork` you will undoubtedly want to explain the implications for file descriptor management or how the various system calls will use process identification?

Sometimes it might be useful to write down which files you will need to modify for each part. This will force you to think about the actual implementation and how it fits in the system. You are not designing in isolation; you are designing in the context of os/161, and you need to make sure that your designs make sense in that environment. Spend more time reading through the os/161 code to see where these pieces should be placed.

4.1 Functions

Describe each function you have to implement. Discuss the algorithms you are going to use and why. Including pseudocode for key parts is a good idea. Work hard to figure out where the subtleties and hidden problems await. Every bug you find during design will save you hours of debugging. [1] If you can identify any helper or utility functions that you'll find useful, describe them here.

4.2 Plan of Action

Describe how you will divide up the work. There are two extremes that groups often try. One is, "We will do everything together." The other is, "I'll do half, my partner will do half, and we'll glue those two halfs together in a few minutes." Neither extreme works particularly well. You should divide work to allow you to work in parallel and to account for variations in working schedules. However, you should coordinate, sync up, and meet regularly. Review each other's code. You will be stunned both by how much you learn from reviewing code and by how many bugs you find by reviewing code.

When you divide up work, set milestones so you are each accountable to the other for completing your tasks in time for integration. If either or both of you start to slip, speak up. Do not hide from your partner. Do not keep saying, "Just a couple more hours." Speak up. Explain how far you are and what your strategy is for getting to the endline. Many a partnership

[1] This is perhaps something that doesn't quite resonate with you. In previous courses, debugging was relatively easy and you could play fast and loose and fix it all up during the debugging cycle. Such is not the case with an operating system. You are working in the context of a concurrent asynchronous system. By definition, thow two properties make things more difficult. Deep, thoughtful design *will* save you time.

has produced bad feelings, because individuals were too embarrassed to say, "Um, I'm confused and don't quite know what to do next." or "I partied too late last night and didn't get done what I needed to get done today." Be honest. When we review your design, we can also flag partitionings that we think won't work – we have a rough idea how long the various pieces take and if they don't look balanced, we can say something.

Former students frequently chuckle when they look back at their design documents, which outlined in detail what each person was going to do on which day, and compared their schedule with reality. For example, one assignment 2 initial design document featured the following "breakdown":

```
Friday        PIDs, getpid, descriptors,
More Friday   chdir, getcwd, scheduler
Saturday      i/o, exit, waitpid, execv
Sunday        fork, scheduler
Monday        test
Tuesday       test
Wednesday     test
Thursday      test
Friday        party
```

In fact, the actual schedule looked more like this:

```
Friday        nothing
Saturday      PIDs, getpid
Sunday        nothing
Monday        my partner and I panic
Tuesday       descriptors, chdir, getcwd
Wednesday     scheduler, we panic some more
Thursday      i/o, exit, waitpid
More Thursday (procrastinating before execv and fork)
Friday        execv, fork, test, test, test, test
Saturday      sleep
Sunday        sleep
Monday        sleep
```

5 Finishing up

After writing a design document, you should look at it and *convince* yourselves that you have explained every difficult detail. Often students write

things that translate into "I don't really know how I'm going to solve it, but somehow I will" or "We will implement `read`. We will also implement `write`." If you find yourself writing things like this, seek out assistance! Talk to your TF, head to office hours, post a question on Piazza. If you are still lost, then right there in black and white, expose that confusion in your design document, "I DON'T KNOW HOW TO DO THIS, HELP!!" Ideally your peer review partners will help you. If they can't help either, then they should have a similar line in their design document and you can ask for assistance during class, after class, on Piazza, or in the document you turn in.

As a rule of thumb, the more code-looking text you include in your design document, the better off you will be. However, code and prose should be well-balanced. No one should have to read your code to figure out *what* you are doing, but reading the code should tell them *how* you are doing it.