What I wish I knew for p3

Michael You

p3 is the giant of the 15-410 experience, and one of the largest projects you'll ever do at CMU in a class. While the assignment's individual parts aren't that hard, choosing the wrong design, and making poor choices throughout the project can lead to devastating outcomes, since large amounts of code is very difficult to fix and mistakes are amplified over time as you do the project.

The project handout from the 15-410 course staff compiles some warnings from previous semester, but I feel like there can be a bit more detail for certain parts. I will detail some parts of the handout I think deserved more attention, and hopefully it will help you as well when you undertake the p3 experience.

1 Relationship between p2 and p3

Maybe it's obvious, but when I started p3 I wasn't sure about the connection between p3 and p2. Of course, p2 is user space code, and p3 is kernel space code, but it seems like you might want to run user space code with kernel code, right?

In p2, you wrote user code that ran on top of a kernel that was already written for you. That way, you didn't have to worry about the kernel at all. In p3, you are implementing that kernel that ran under your user code. When you are running your kernel in p3, for the most part, you don't have to worry about your p2 code. The only thing is probably your `install_autostack` (see Plan of Attack 3), which is called on the entry of every program.

What this means, is things like the locks you wrote in p2 cannot be used by your p3 kernel. This means you will have to rewrite another `panic`, and rewrite locks for your p3 kernel.

When you start testing more advanced programs later on, you can run p2 code on top of your p3 kernel, and see if things still work the way you expect.

2 Error Checking

In class, they introduce why error checking is important and how you should go about handling errors in your code. The main lesson learned from class however, is that

`No error should be left unheard.` (1)

That makes sense, and what this means to a lot of people is just to implement the pass-the-buck error handling approach to everything.

However, the side that is not taught as much is how to actually handle these error, and how to write assertions throughout your code to guarantee that it is more safe.

Below, I'll outline the 3 most important error checking principles you should utilize throughout your project.

You should keep in mind that although Passing the Buck and Assertions are two strategies that you can actively implement in your code, `panic` is also a powerful error checking tool. This means that you should **actively consider all three methods of error checking**, without getting too hung up on a particular method, just because it's easy to use or you understand it best.

2.1 Passing the Buck

As mentioned earlier, this is usually the easiest to understand method of handling errors. The idea is that whenever something is wrong, you just return a negative error code. Since there can be many different errors,

you should have different negative error codes for different errors. One approach you can take is to have a bunch of defined error codes that are shared by functions throughout your kernel.

```
#define FUNCTION_ERRNO_NULL -1
#define FUNCTION_ERRNO_INIT -2
#define FUNCTION_ERRNO_ARGS -3
```

Listing 1: Negative error code defines

2.2 Assertions

While passing the buck is easy to implement, it can be weak since returning a negative error code means you won't be able to pinpoint where the error is until you have code that checks for a negative error code. For example, you could potentially have a mutex init that fails, which gets passed up to a pcb init, which then fails and is passed up to the kernel init function. You'll know by the kernel init function that something went wrong, but you'll have to track down some more to figure out when things *really* went wrong.

A more specific error check you can do is an assertion, which course staff sets up for you in `contracts.h`. Assertions are useful because when something goes wrong, the exact line number will be returned to you at the time of error. If you combined assertions with a good code tracing utility, you can debug your errors much faster. In addition, assertions are good when you have an error that you can't really recover from.

This is not to say an assertion is necessarily "better" than a passing the buck error. For example, sometimes you may want the caller of the current function to handle an error, which may be unrecoverable, in which case passing the buck is more appropriate than an assertion.

A good way to write your kernel is to write a bunch of assertions that describe how you expect your function to behave, and fix things when assertions start to break. This is actually a common practice in industry, known as TDD(Test Driven Development). While this method of development can be rather slow, it can help you write much more robust code, which is important for a project of this scale.

As an example, consider the following functions, one written entirely with passing the buck, and the other with a mix in Listing 2. The `assertStyle` code is preferred here, since unrecoverable errors will trigger the assertion error, helping you pinpoint errors, while errors that are not as severe, for example passing in a bad value for a argument, are not passed to the caller to handle.

```
function buckStyle (int *apples, int pineapple) {
  if (apples == NULL) {
    return FUNCTION_ERRNO_NULL;
  }

  if (valid_mem(apples)) {
    return FUNCTION_ERRNO_MEM;
  }

  if (pineapple == 0) {
    return FUNCTION_ERRNO_ARGS;
  }

  apples += pineapple;

  return pineapple;
}

function assertStyle (int *apples, int pineapple) {
  ASSERT(apples != NULL);
  ASSERT(valid_mem(apples));

  if (pineapple == 0) {
```

```
    return FUNCTION_ERRNO_ARGS;
  }

  apples += pineapple;

  return pineapple;
}
```

Listing 2: Comparing passing the buck and assertion styles

2.3 panic

I didn't do a whole lot of debugging in p2, so I didn't use panic. When I got to p3, I didn't really know what it was for. In the handout, we are told that `panic` is way to help us debug our code, and a good `panic` function will help us debug faster. When I first heard that, I had no idea what that meant...sounded like empty words to me.

The reason course staff is pushing you to write a `panic` function is because of the following – have you ever had the following scenario?

- You run your program, and there's a bug
- You stop the execution, and look around the context
- Let me look at `%esp, %eip, %cs, %ss`...
- Ok hmm...things look alright. Let me check the contents of the stack
- That seems fine...let me look at the parameters of the current TCB
- Alright, I'm curious what tasks are currently scheduled in the scheduler. Ugh...this is going to be a lot of `simics psym` commands.
- Oof, I also want to see what threads are current sleeping.
- Hmm...would also be nice to know how many free pages I current have.

The above debugging journey is not necessarily hard to do, but it is certainly *time consuming*. Typing in `simics` commands is quite unwieldy, and is also limited in what you can do. A `panic` function can help you do all these things in one go. Something goes wrong – `panic`! Everything you want to know in general can be printed out, from the current task, to the tasks in the scheduler, the current timer stats, the context on the surrounding stack area – literally anything you deem useful. Since your error-checking code, like `ASSERT`, will call `panic`, every time something goes wrong, you'll automatically be able to get a printout of what happens. You can even implement checks in your `panic` to automatically check certain sanity things for you, for example if your segment registers make sense or the program instruction pointers are reasonable.

Some features you might consider putting into your `panic`:

- Print out the current state
  - Registers
  - Any state variables stored in the kernel
  - Surrounding stack near the `%esp`
  - Surrounding stack of the user program, if applicable
- Print out the current TCB
- Print out the current PCB
- Print out some tasks in the scheduler's active, sleep queues
- Print out global variables

3 Locking

One trap I fell into while doing p3 was that I wanted to get everything working with disable_interrupts before continuing with locking. While this will definitely help you finish your project, it will probably not let you integrate locking into your project enough to the point where you can have a kernel with many interleaving threads with no race conditions.

My suggestion is, by the time you've coded up your TCB, PCB structures, you should strive to not disable interrupts on any syscall. Of course, it's ok to disable interrupts during debugging and other intermediary developments, but you should strive to be able to reach checkpoints where all your syscalls don't have to disable interrupts.

A good start for locking is to port all the fundamental locks you created in p2 – your mutex, cond, sem, rwlock – so this way it will encourage you to use them where you feel fit. You will likely have to slightly modify them a bit early on in your kernel, since you will likely not be able to do things like deschedule yet, but that is ok. If you don't port your locks over early on, locking will likely not be in your mind, so you'll neglect it.

I would like to acknowledge that there are steps in the "Plan of Attack" of the p3 handout that try to push for this, but I feel like they aren't explicit enough in what you should do to give you that push to put yourself in a better position with the locking situation. Porting over the p2 locks I think is a great way to put yourself on the right track, and also being extremely avoiding of disable_interrupts.

4 Plan of Attack Tips

There is a plan of attack section listed at the end of the handout that gives a nice overview about what steps you should approximately take to complete the project. Here are some additional tips, not written in any particular order, that you should consider while working on p3.

1. Strive to document your code as you go. At the very least, make doxygen headers on all the functions and files you create, so you don't have to go through the pain of adding them in later on.

2. One of the first things you should do is write all your system call and interrupt handlers. This includes exception handlers! They don't have to necessarily work yet (you can just return for now), However, if you're missing a handler and it gets triggered for whatever reason, you will get a mysterious General Protection Fault, since the handler is not installed in the IDT.

   You will probably find macros useful for installing all your interrupt handlers, since many handlers will share many parts. Early on though, you might want to just write it all out just because it's easier to do. There's not much documentation online about using macros in assembly, but using #define is a simple way to implement macros for assembly, just like how you do for C code. There are some extra precautions you need to take though when defining macros for assembly.

3. When they say to comment out some of your install_autostack code, what you should understand is that for any program, the first function that is called is the function in crt0.c, which just performs the install_autostack and then calls exit into the program you intended to execute. Therefore, since install_autostack will be called by every program, and the function requires syscalls you may not have written yet, i.e. new_pages, you will likely need to comment out most if not all of install_autostack.

4. They tell you to choose an error-handling approach early. When I originally read this, I thought they just meant to return various types of error codes. What I realized in retrospect was that you can use error checking to your advantage as you develop. See Section 2 for more details about how to use error checking as you write p3.

5. They also tell you to think about locking early on. As mentioned in Section 3, you should port over all your locking constructs from p2 early on, so you will think about locking early on in your project. This way, you'll be more encouraged to use locking when you find the chance, instead of feeling a big burden to port them over in the first place, and being distracted by trying to "finish" your kernel first with disable_interrupts everywhere.

6. If you are noticing that you are running a lot of the tests over and over again...especially the hurdle tests, it might be a good idea to write a program that runs suites of tests for you automatically. It'll save you time, and also encourage you to test more. One good practice when your kernel starts getting big is to run a set of tests every time you make a change, so if your change breaks something, you'll immediately know.